EXERCISE SOLUTIONS

CHAPTER 1

Exercise 1.1

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

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

Exercise 1.2

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

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

Exercise 1.3

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

Exercise 1.4

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

Exercise 1.5

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

Exercise 1.6

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

Exercise 1.7

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

Exercise 1.8

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

Exercise 1.9

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

Exercise 1.10

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

Exercise 1.11

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

Exercise 1.12

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

Exercise 1.13

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

Exercise 1.14

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

Exercise 1.15

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

Exercise 1.16

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

Exercise 1.17

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

Exercise 1.18

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

Exercise 1.19

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

Exercise 1.20

(a) 1001110; (b) 1111100; (c) 11101101001110 10; (d) 100 0000 0011 1111 1011 0000 0000 0001

Exercise 1.21

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

Exercise 1.22

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

Exercise 1.23

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

Exercise 1.24

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

Exercise 1.25

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

Exercise 1.26

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

Exercise 1.27

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

Exercise 1.28

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

Exercise 1.29

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

Exercise 1.30

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

Exercise 1.31

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

Exercise 1.32

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

Exercise 1.33

(a) 00000101; (b) 11111010

Exercise 1.34

(a) 00000111; (b) 11111001

Exercise 1.35

(a) 00000101; (b) 00001010

Exercise 1.36

(a) 00000111; (b) 00001001

Exercise 1.37

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

Exercise 1.38

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

Exercise 1.39

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

Exercise 1.40

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

Exercise 1.41

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

Exercise 1.42

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

Exercise 1.43

4, 8

Exercise 1.44

8

Exercise 1.45

5,760,000

Exercise 1.46

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

Exercise 1.47

46.566 gigabytes

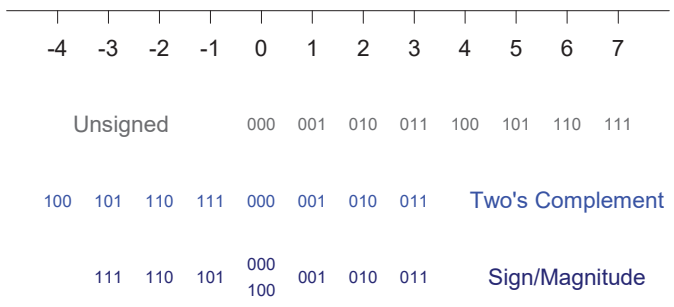
Exercise 1.48

2 billion

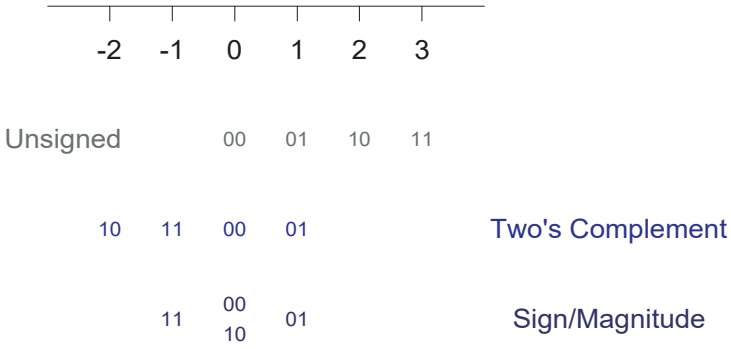
Exercise 1.49

128 kbits

Exercise 1.50



Exercise 1.51



Exercise 1.52

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

Exercise 1.53

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

Exercise 1.54

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

Exercise 1.55

(a) 11011101; (b) 110001000

Exercise 1.56

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

Exercise 1.57

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

Exercise 1.58

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

Exercise 1.59

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

Exercise 1.60

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

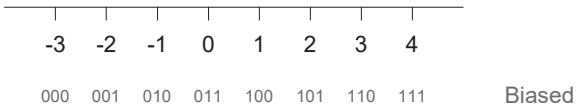
Exercise 1.61

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

Exercise 1.62

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

Exercise 1.63



Exercise 1.64

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

Exercise 1.65

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

Exercise 1.66

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

Exercise 1.67

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

Exercise 1.68

Both are right.

Exercise 1.69

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

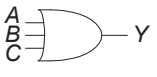
```
void main(void)
{
   char bin[80];
   int i = 0, dec = 0;

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

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

Exercise 1.70

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

#include <stdio.h>

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

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

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

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

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

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

Exercise 1.71

OR3

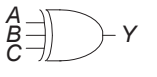


$Y = A+B+C$

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

(a)

XOR3



$Y = A \oplus B \oplus C$

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

(b)

XNOR4

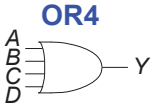
$Y = \overline{A \oplus B \oplus C \oplus D}$

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

(c)

Exercise 1.72

OR4

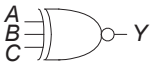


$Y = A+B+C+D$

(a)

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

XNOR3

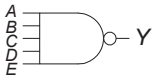


$Y = \overline{A \oplus B \oplus C}$

(b)

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

NAND5



$Y = \overline{ABCDE}$

(c)

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

Exercise 1.73

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

Exercise 1.74

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

Exercise 1.75

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

Exercise 1.76

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

Zero

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

A NOR B

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

$\overline{A}B$

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

NOT A

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

$A\overline{B}$

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

NOT B

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

XOR

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

NAND

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

AND

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

XNOR

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

B

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

$\overline{A}$ + B

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

A

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

A + $\overline{B}$

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

OR

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

One

Exercise 1.77

2^{2^N}

Exercise 1.78

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

Exercise 1.79

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

Exercise 1.80

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

Exercise 1.81

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

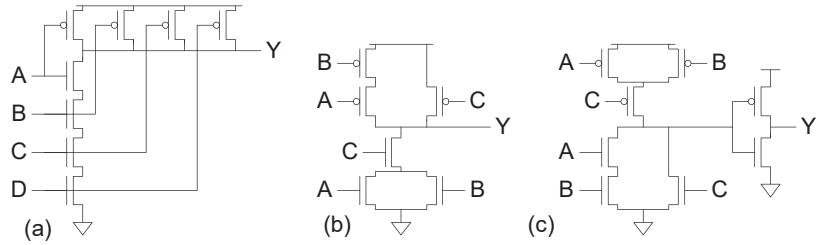
Exercise 1.82

(a) AND gate; (b) $V_{IL} = 1.5$; $V_{IH} = 2.25$; $V_{OL} = 0$; $V_{OH} = 3$

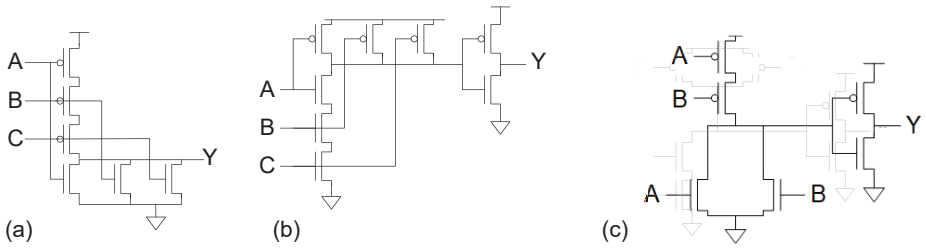
Exercise 1.83

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

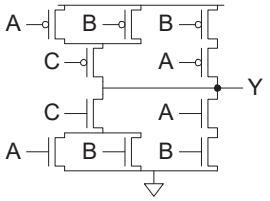
Exercise 1.84



Exercise 1.85



Exercise 1.86



Exercise 1.87

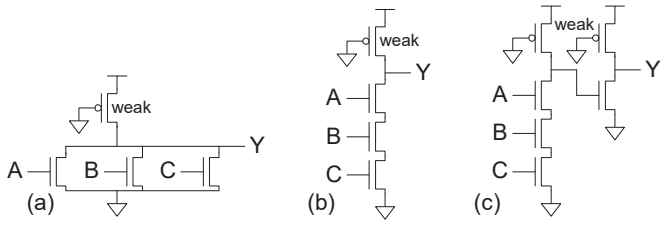
XOR

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

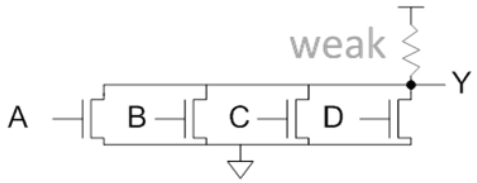
Exercise 1.88

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

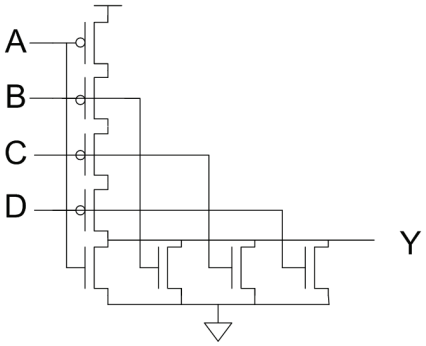
Exercise 1.89



Exercise 1.90



Question 1.1



Question 1.2

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

Question 1.3

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

CHAPTER 2

Exercise 2.1

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

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

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

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

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

Exercise 2.2

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

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

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

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

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

Exercise 2.3

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

(b)

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

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

(d)

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

(e)

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

Exercise 2.4

(a) $Y = A + B$

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

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

(d)

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

(e)

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

Exercise 2.5

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

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

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

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

(e)

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

This can also be expressed as:

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

Exercise 2.6

(a) $Y = A + B$

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

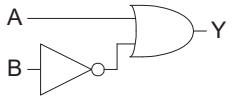
(c) $Y = AB + \overline{A}\overline{B}C$

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

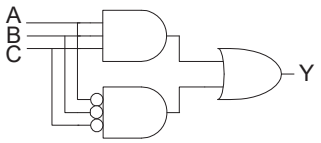
(e) $Y = A\overline{B} + \overline{A}BC + \overline{A}CD$ or $Y = A\overline{B} + \overline{A}BC + \overline{B}CD$

Exercise 2.7

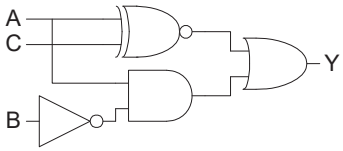
(a)



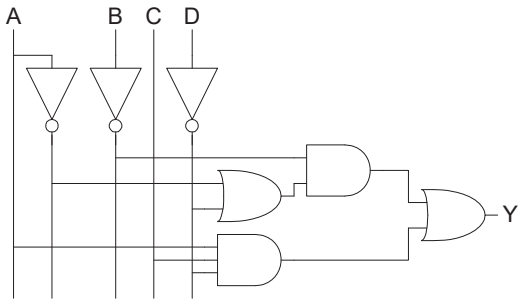
(b)



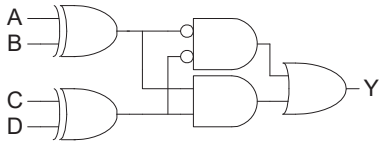
(c)



(d)

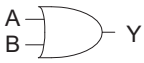


(e)

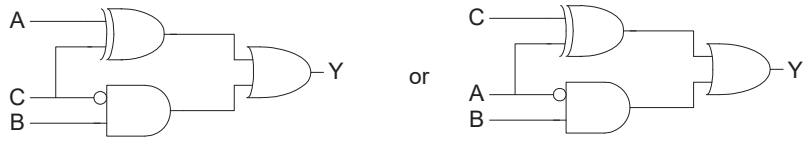


Exercise 2.8

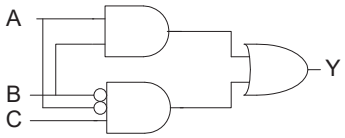
(a)



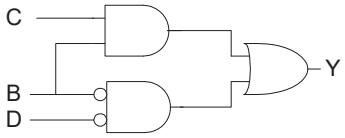
(b)



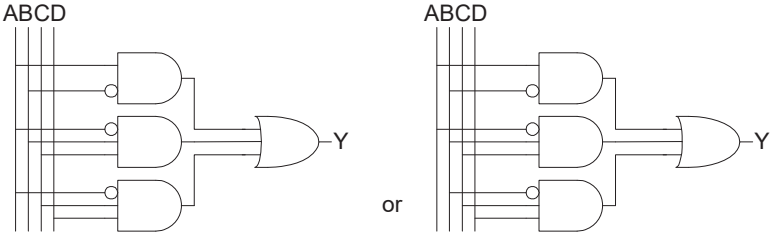
(c)



(d)

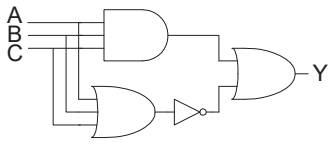


(e)

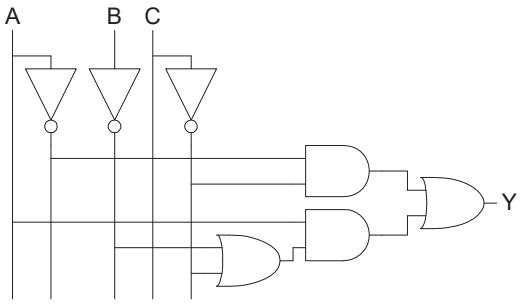


Exercise 2.9

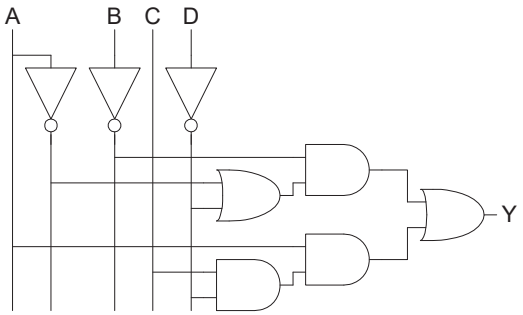
(a) Same as 2.7(a)
(b)



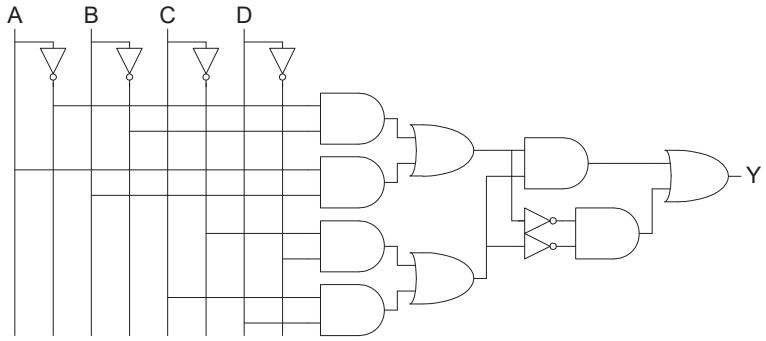
(c)



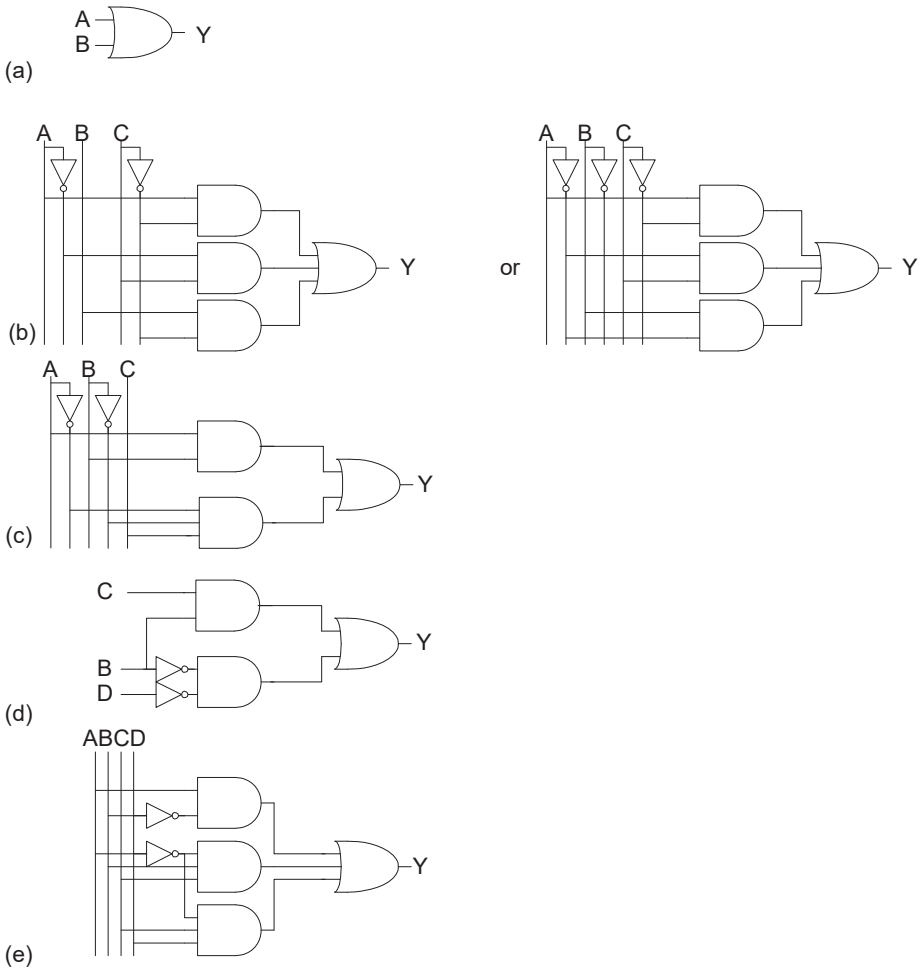
(d)



(e)

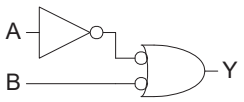


Exercise 2.10

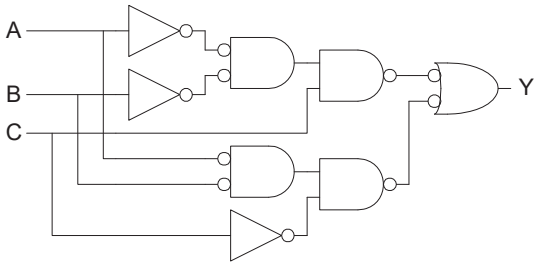


Exercise 2.11

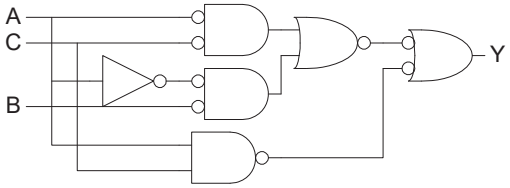
(a)



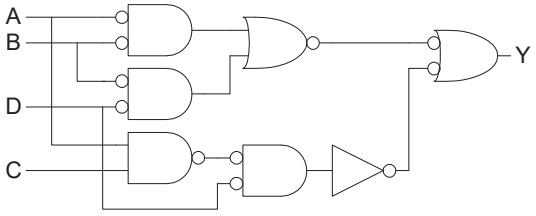
(b)



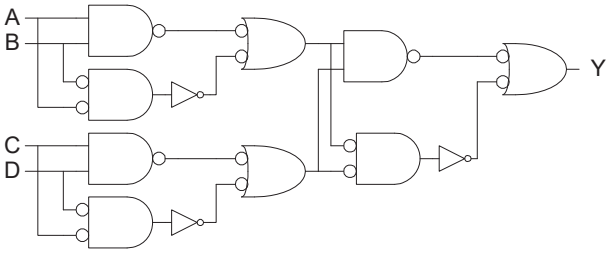
(c)



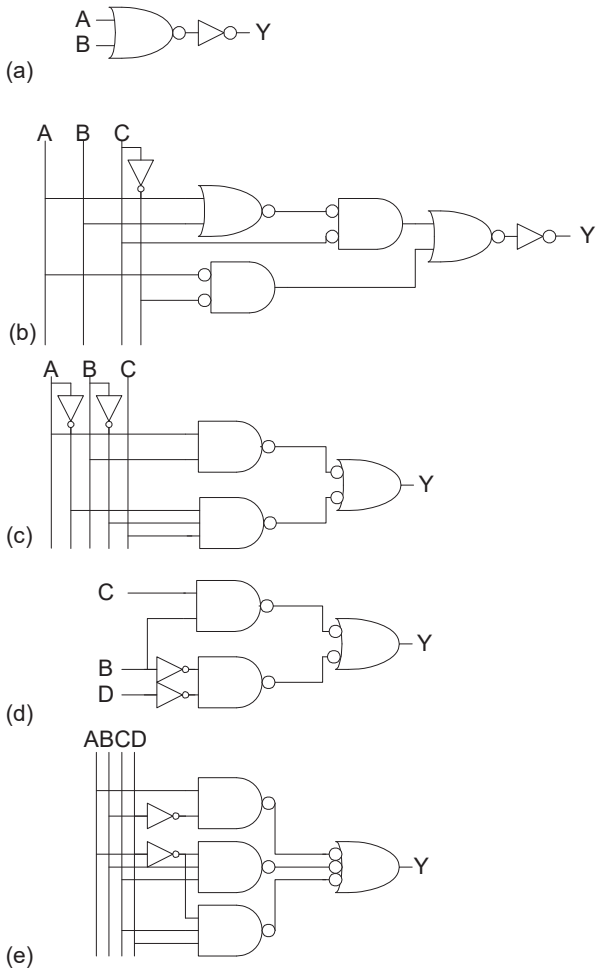
(d)



(e)



Exercise 2.12



Exercise 2.13

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

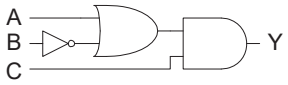
Exercise 2.14

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

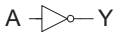
(c) $Y = A(\overline{B} + \overline{C} + \overline{D}) + \overline{B}\,\overline{C}\,\overline{D} = A\overline{BCD} + \overline{B}\,\overline{C}\,\overline{D}$

Exercise 2.15

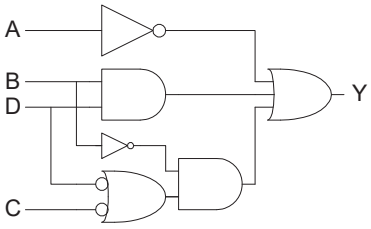
(a)



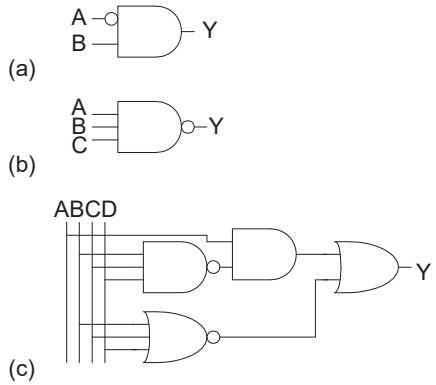
(b)



(c)



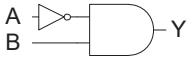
Exercise 2.16



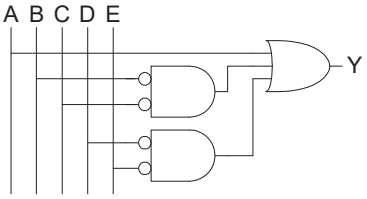
Exercise 2.17

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

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



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



Exercise 2.18

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

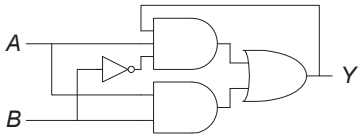
(b) $Y = (A + \overline{C})D + B$

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

Exercise 2.19

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

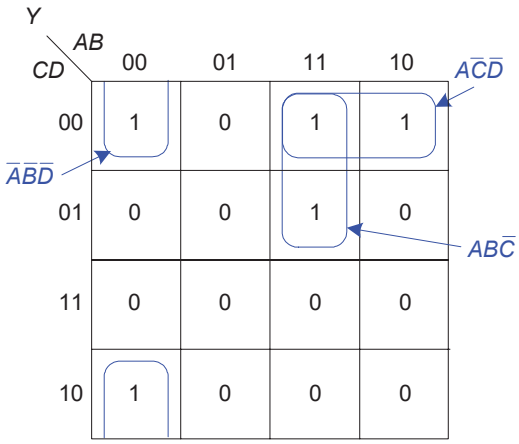
Exercise 2.20



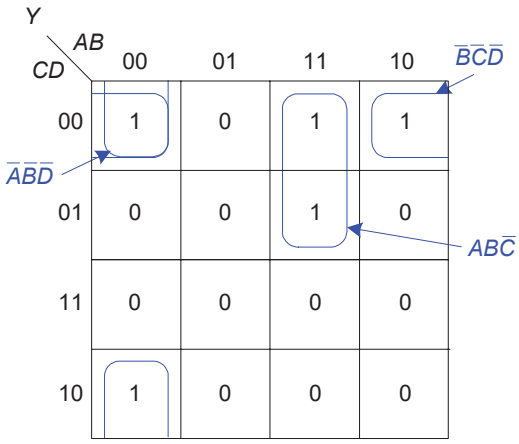
$Y = A$

Exercise 2.21

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



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



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

Exercise 2.22

(a)

B	$B \bullet B$
0	0
1	1

(b)

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

(c)

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

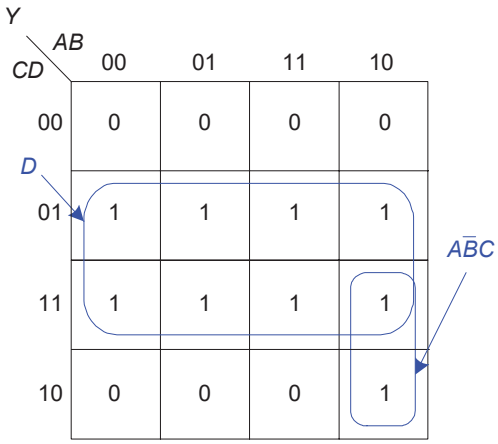
Exercise 2.23

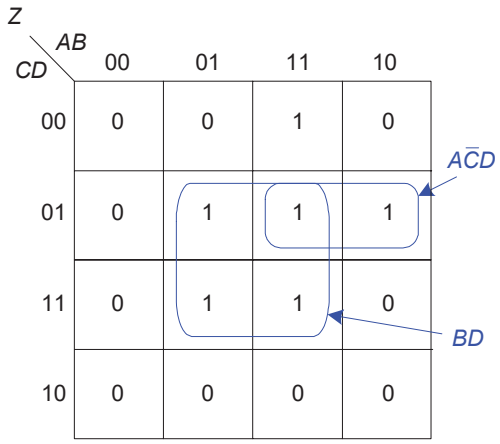
B_2	B_1	B_0	$\overline{B_2 \bullet B_1 \bullet B_0}$	$\overline{B}_2 + \overline{B}_1 + \overline{B}_0$
0	0	0	1	1
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

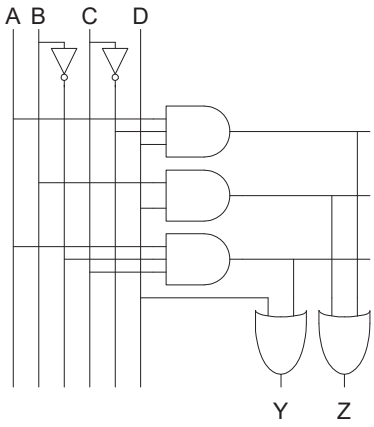
Exercise 2.24

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

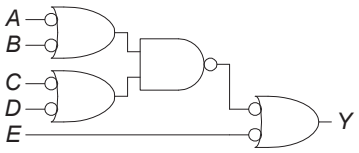
Exercise 2.25





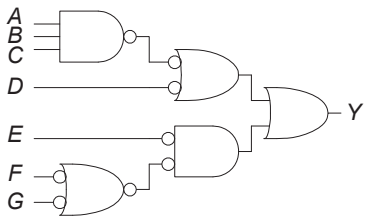


Exercise 2.26



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

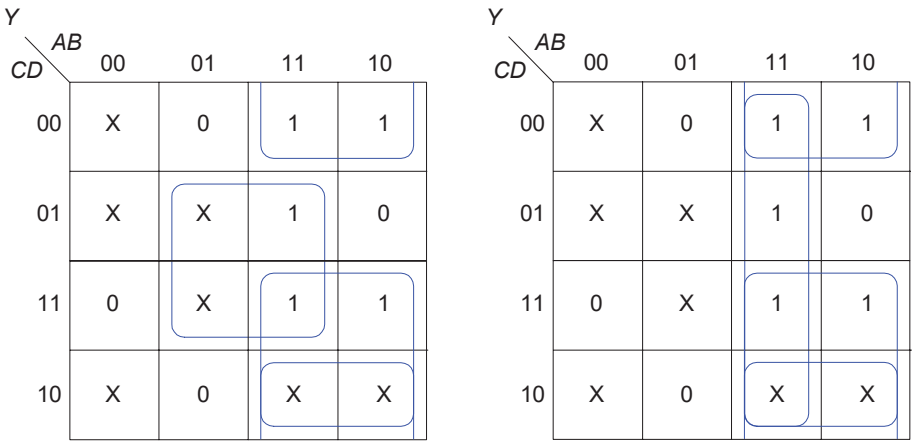
Exercise 2.27



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

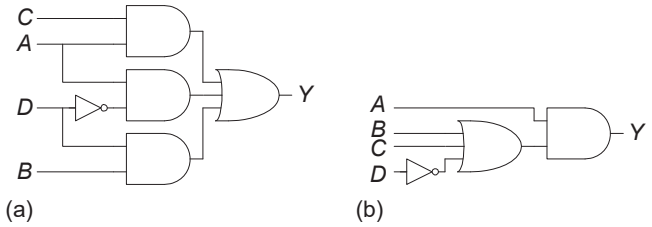
Exercise 2.28

Two possible options are shown below:



Exercise 2.29

Two possible options are shown below:



Exercise 2.30

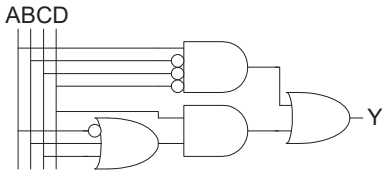
Option (a) could have a glitch when A=1, B=1, C=0, and D transitions from 1 to 0. The glitch could be removed by instead using the circuit in option (b).

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

Exercise 2.31

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

Exercise 2.32



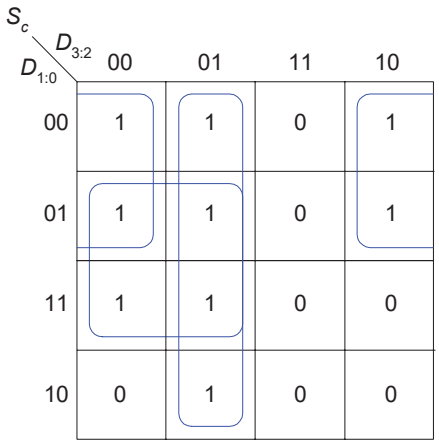
Exercise 2.33

The equation can be written directly from the description:

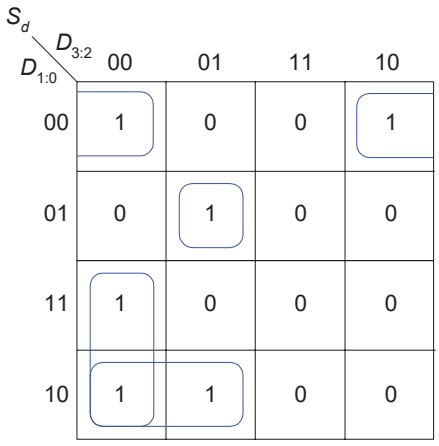
$$E = S\overline{A} + AL + H$$

Exercise 2.34

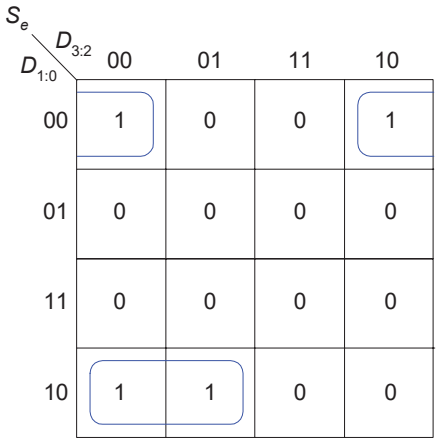
(a)



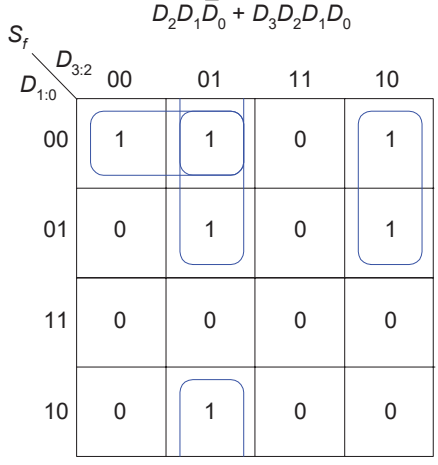
$S_c = \bar{D}_3 D_0 + \bar{D}_3 D_2 + \bar{D}_2 \bar{D}_1$



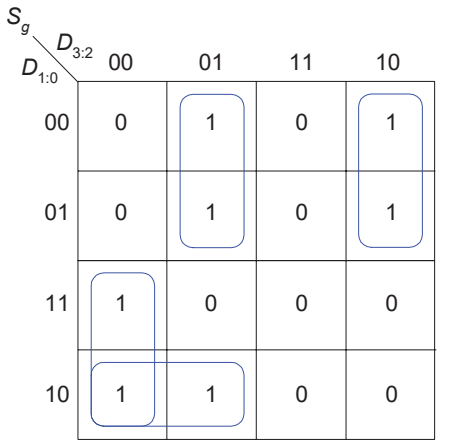
$S_d = \bar{D}_3 D_1 \bar{D}_0 + \bar{D}_3 \bar{D}_2 D_1 + \bar{D}_2 \bar{D}_1 \bar{D}_0 + \bar{D}_3 D_2 \bar{D}_1 D_0$



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

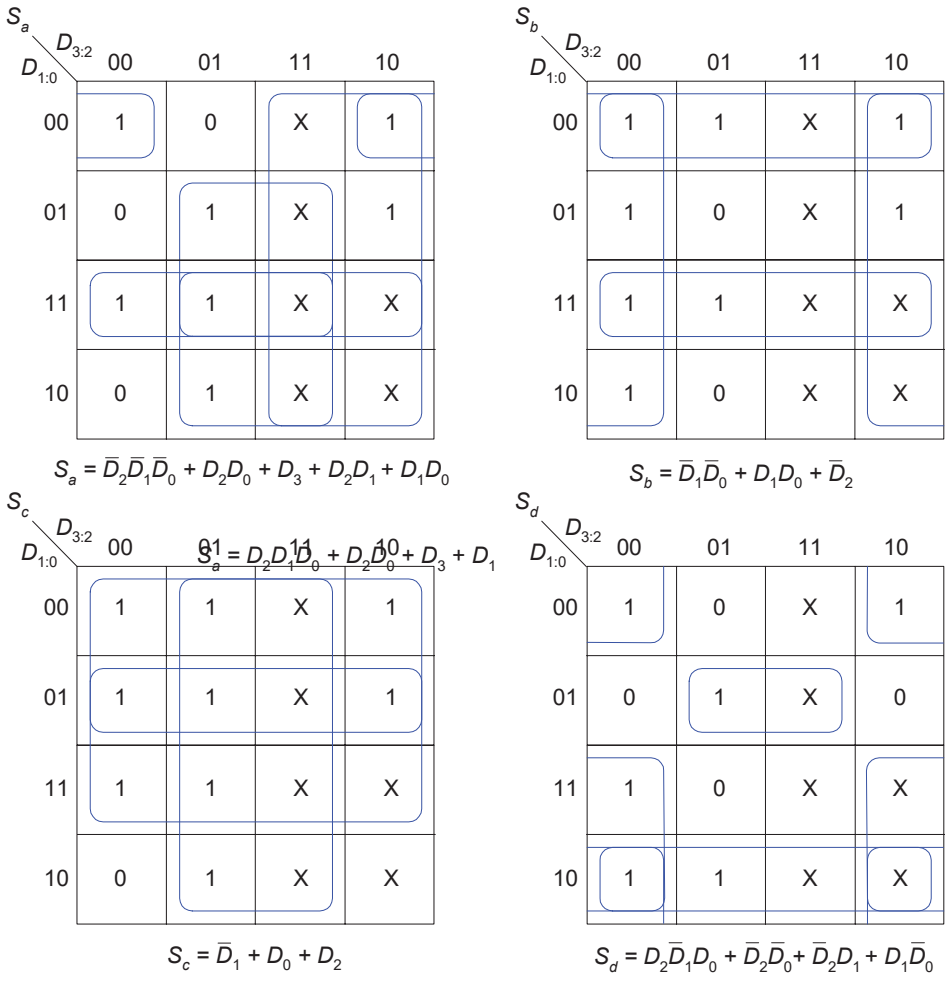


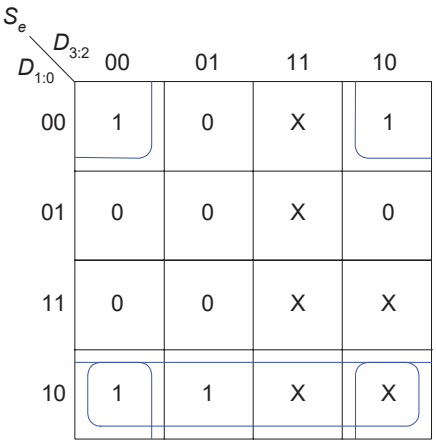
$S_f = \bar{D}_3 \bar{D}_1 \bar{D}_0 + \bar{D}_3 D_2 \bar{D}_1 + \bar{D}_3 D_2 \bar{D}_0 + D_3 \bar{D}_2 \bar{D}_1$



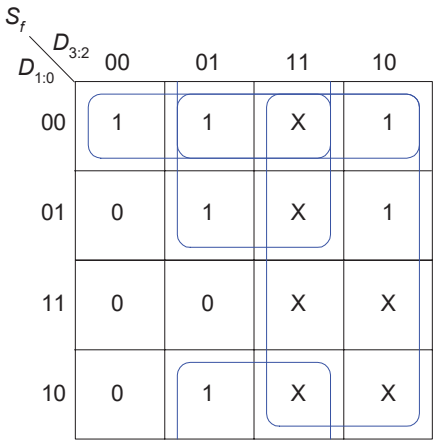
$$S_g = \bar{D}_3\bar{D}_2D_1 + \bar{D}_3D_1\bar{D}_0 + \bar{D}_3D_2\bar{D}_1 + D_3\bar{D}_2\bar{D}_1$$

(b)

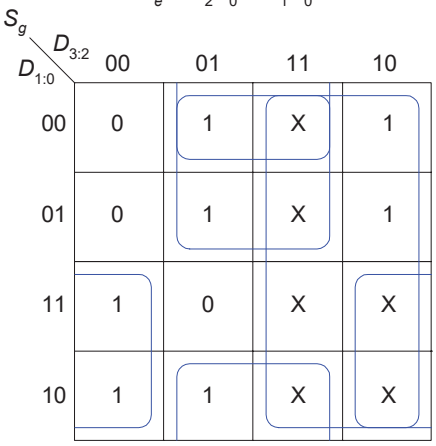

S_a
$D_{3:2}$
$D_{1:0}$
00 01 11 10
00 1 0 X 1
01 0 1 X 1
11 1 1 X X
10 0 1 X X
$S_a = \bar{D}_2\bar{D}_1\bar{D}_0 + D_2D_0 + D_3 + D_2D_1 + D_1D_0$
S_b
$D_{3:2}$
$D_{1:0}$
00 01 11 10
00 1 1 X 1
01 1 0 X 1
11 1 1 X X
10 1 0 X X
$S_b = \bar{D}_1\bar{D}_0 + D_1D_0 + \bar{D}_2$
S_c
$D_{3:2}$
$D_{1:0}$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



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

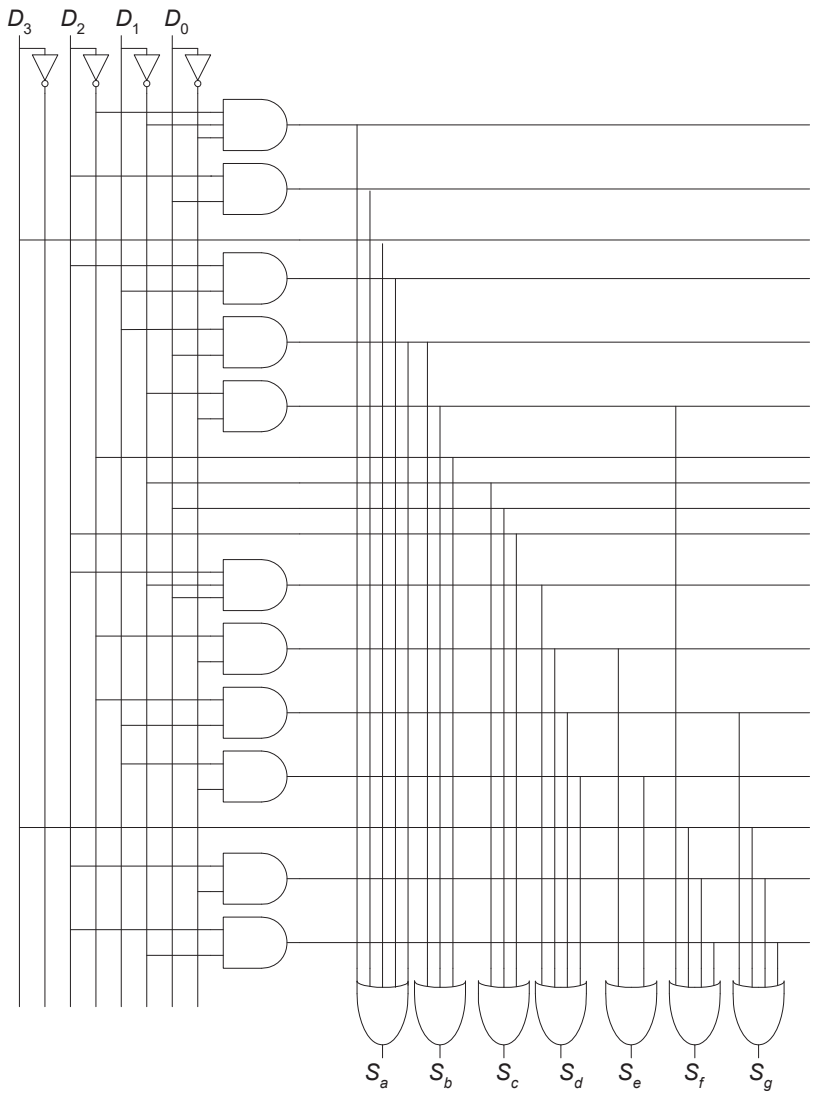


$S_f = \bar{D}_1\bar{D}_0 + D_2\bar{D}_1 + D_2\bar{D}_0 + D_3$



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

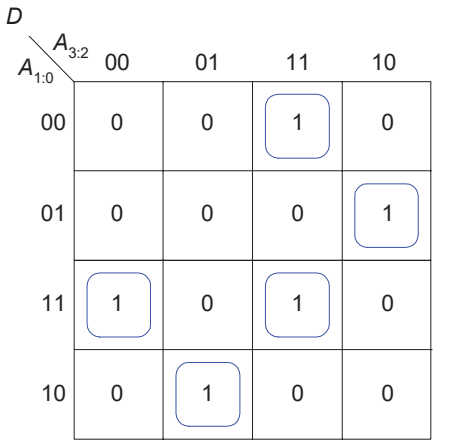
(c)



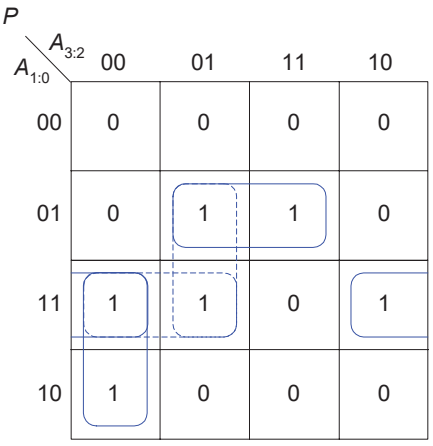
Exercise 2.35

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

P has two possible minimal solutions:



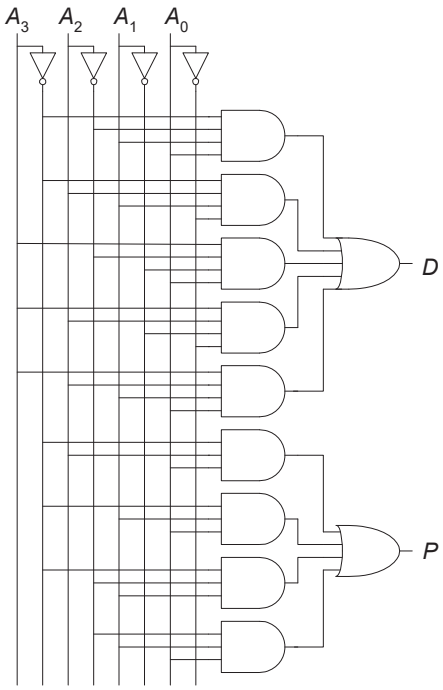
$$D = \bar{A}_3\bar{A}_2A_1A_0 + \bar{A}_3A_2A_1\bar{A}_0 + A_3\bar{A}_2\bar{A}_1A_0 + A_3A_2\bar{A}_1\bar{A}_0 + A_3A_2A_1A_0$$



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

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

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



Exercise 2.36

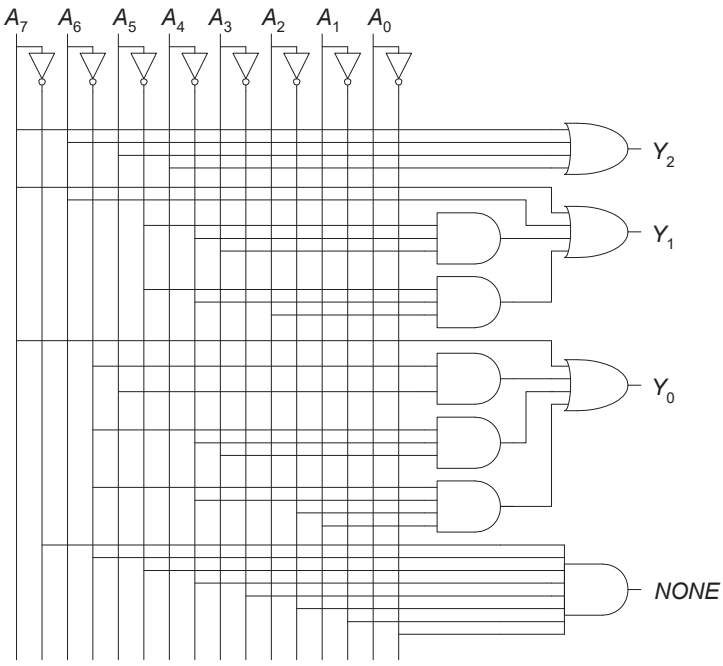
A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Y_2	Y_1	Y_0	*NONE*
0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	X	0	0	1	0
0	0	0	0	0	1	X	X	0	1	0	0
0	0	0	0	1	X	X	X	0	1	1	0
0	0	0	1	X	X	X	X	1	0	0	0
0	0	1	X	X	X	X	X	1	0	1	0
0	1	X	X	X	X	X	X	1	1	0	0
1	X	X	X	X	X	X	X	1	1	1	0

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

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

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

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

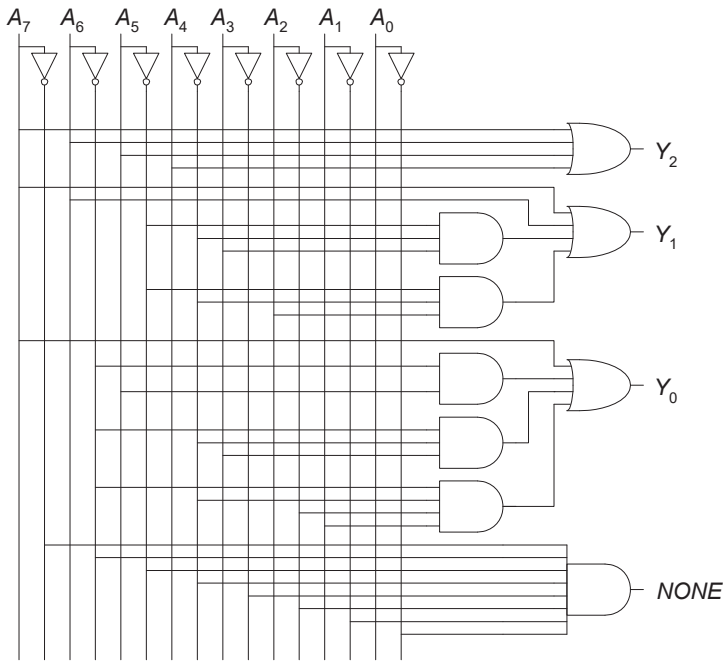


Exercise 2.37

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

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

$$Y_2 = A_7 + A_6 + A_5 + A_4$$
$$Y_1 = A_7 + A_6 + \overline{A}_5\overline{A}_4A_3 + \overline{A}_5\overline{A}_4A_2$$
$$Y_0 = A_7 + \overline{A}_6A_5 + \overline{A}_6\overline{A}_4A_3 + \overline{A}_6\overline{A}_4\overline{A}_2A_1$$


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

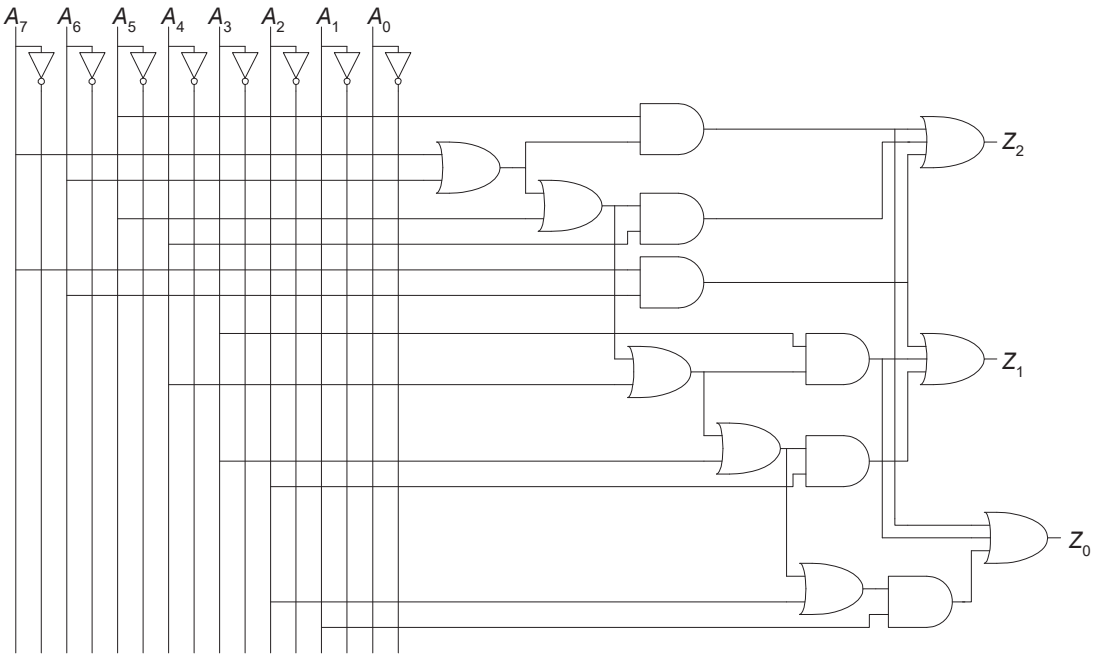
The truth table, equations, and circuit for $Z_{2:0}$ are as follows.

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

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

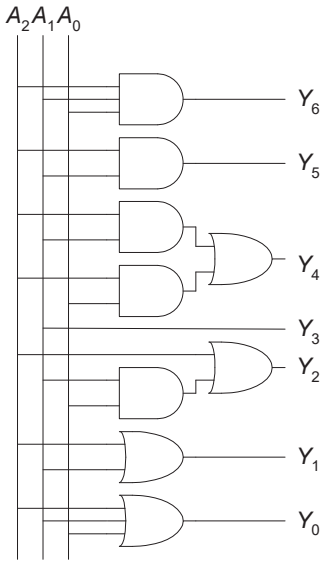
$$Z_1 = A_2(A_3 + A_4 + A_5 + A_6 + A_7) + A_3(A_4 + A_5 + A_6 + A_7) + A_6A_7$$

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



Exercise 2.38

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



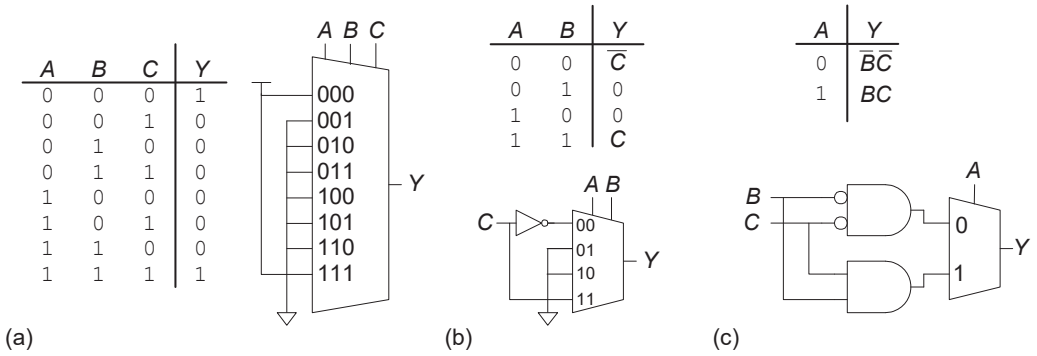
Exercise 2.39

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

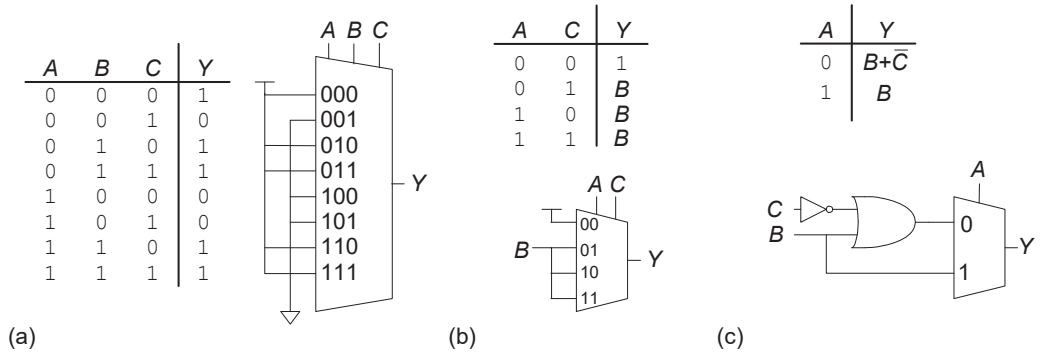
Exercise 2.40

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

Exercise 2.41



Exercise 2.42



Exercise 2.43

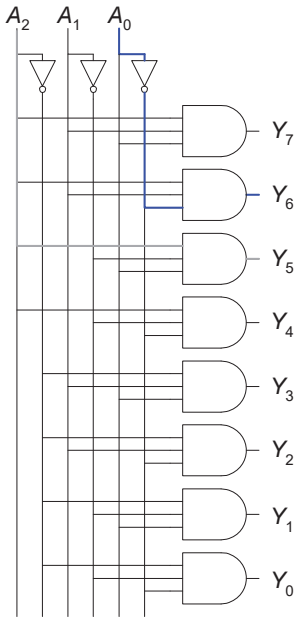
$t_{pd} = 3t_{pd\_NAND2} =$ **60 ps**
$t_{cd} = t_{cd\_NAND2} =$ **15 ps**

Exercise 2.44

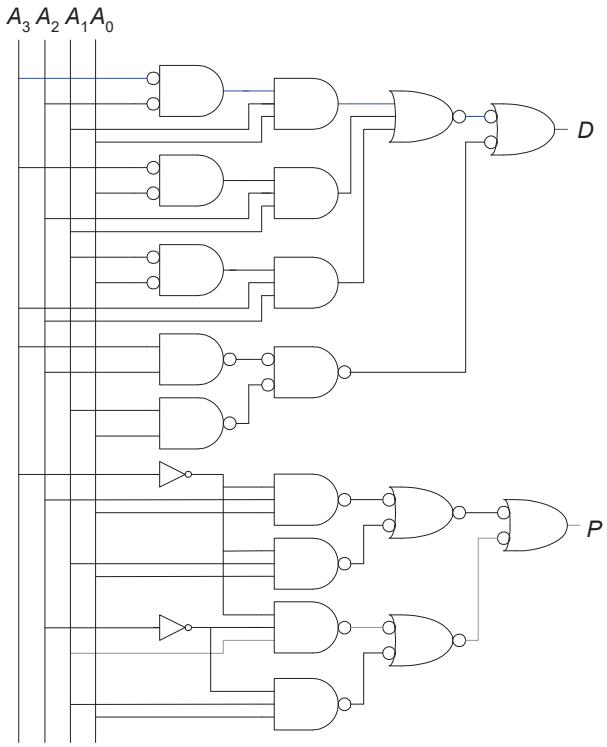
$t_{pd} = t_{pd\_AND2} + 2t_{pd\_NOR2} + t_{pd\_NAND2}$
$= [30 + 2\ (30) + 20]$ ps
$=$ **110 ps**
$t_{cd} = 2t_{cd\_NAND2} + t_{cd\_NOR2}$
$= [2\ (15) + 25]$ ps
$=$ **55 ps**

Exercise 2.45

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

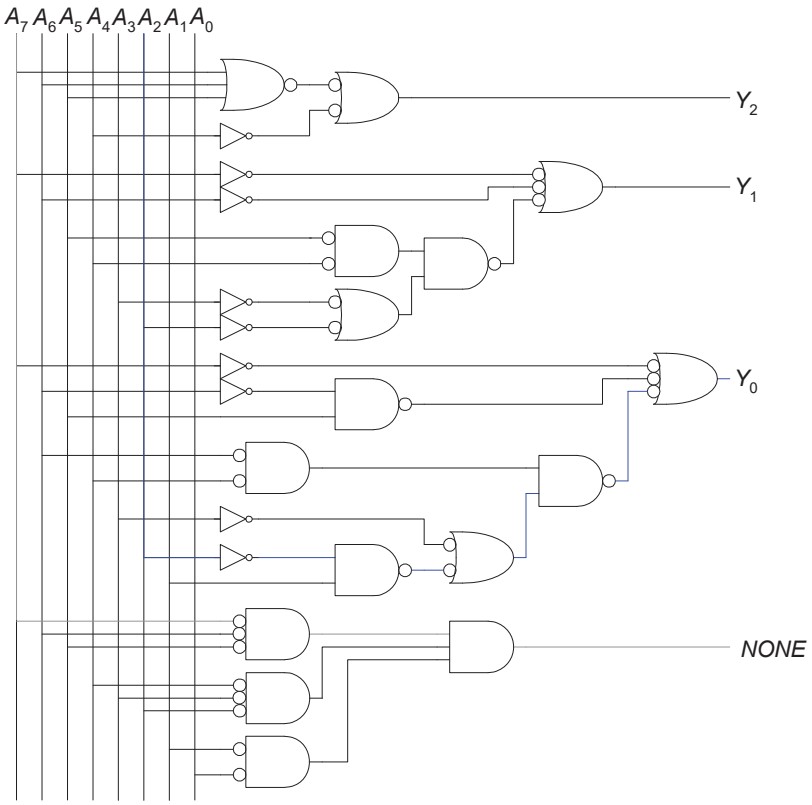


Exercise 2.46



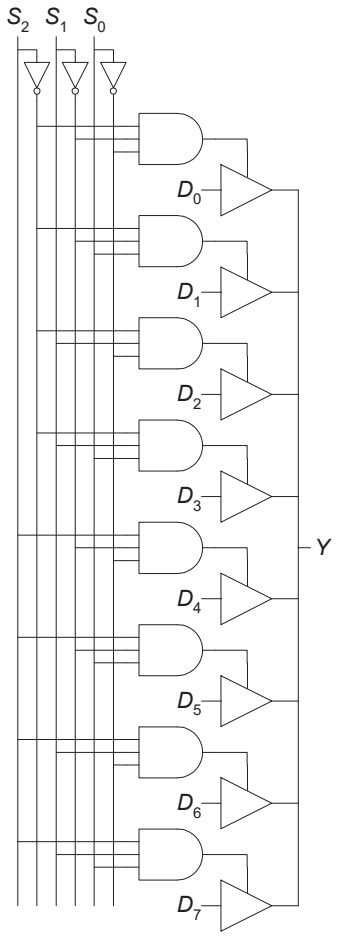
$t_{pd} = t_{pd\_NOR2} + t_{pd\_AND3} + t_{pd\_NOR3} + t_{pd\_NAND2}$
$= [30 + 40 + 45 + 20]$ ps
$=$ **135 ps**
$t_{cd} = 2t_{cd\_NAND2} + t_{cd\_OR2}$
$= [2\ (15) + 30]$ ps
$=$ **60 ps**

Exercise 2.47



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

Exercise 2.48



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

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

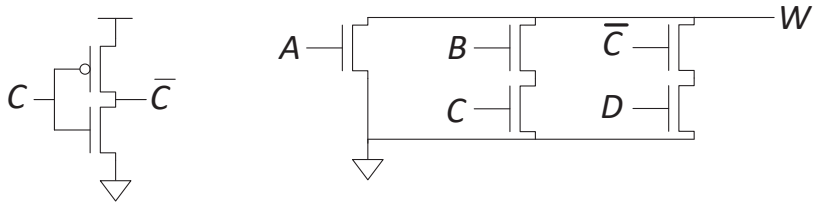
$t_{pd\_sy} = t_{pd\_\text{NOT}} + t_{pd\_\text{AND3}} + t_{pd\_\text{TRI\_SY}}$
$= [30 + 80 + 35]$ ps
$= \mathbf{145\ ps}$

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

Exercise 2.49

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

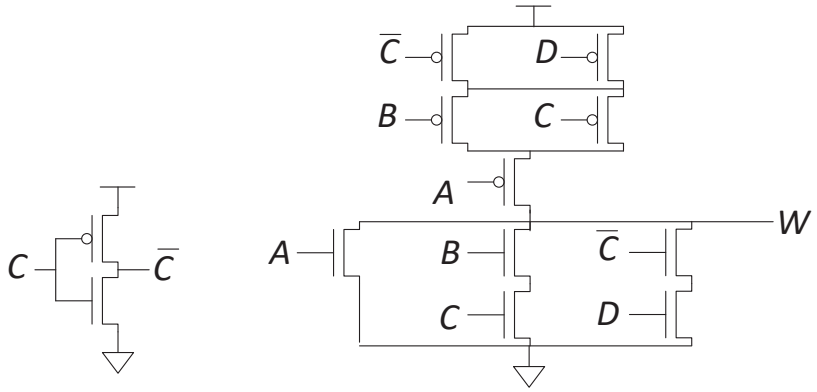
Pull-down network directly from equation:



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

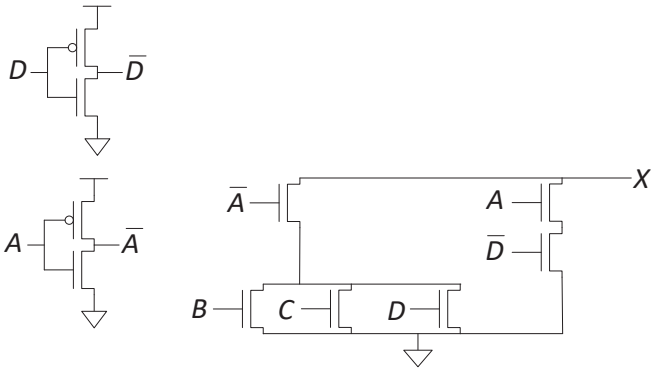
$W = \overline{A + BC + \bar{C}D}$
$W = \bar{A} * \overline{BC} * \overline{\bar{C}D}$
$W = \bar{A} * (\bar{B} + \bar{C}) * (C + \bar{D})$

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



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

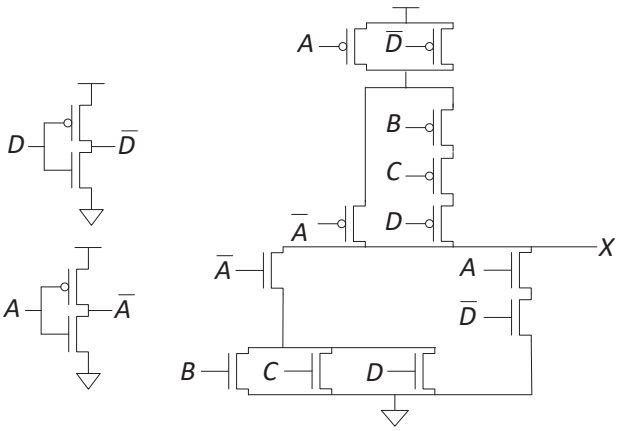
First, we build the pull-down network directly using the equation above:



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

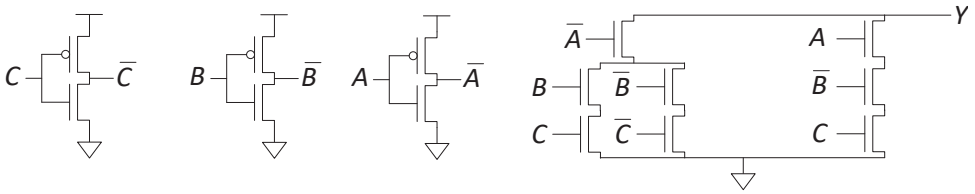
$X = \overline{\bar{A}(B + C + D) + A\bar{D}}$
$X = \overline{\bar{A}(B + C + D)} * \overline{A\bar{D}}$
$X = (A + (\overline{B + C + D})) * (\bar{A} + D)$
$X = (A + \bar{B}\bar{C}\bar{D}) * (\bar{A} + D)$
(Note that we could reduce it further by using distributivity, but that wouldn't reduce the number of transistors.)

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



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

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

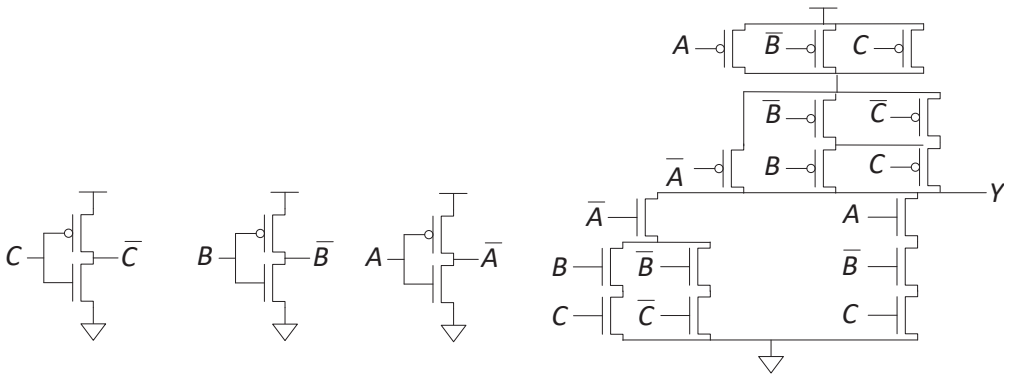


Now perform DeMorgan's on the equation to draw the pull-up network.

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

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

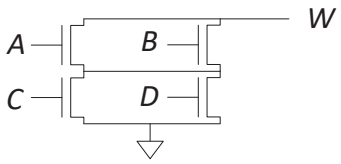
Now we add the pull-up network according to this equation. Note that the method we used in Chapter 1 of changing parallel (OR) connections to serial (AND) connections and vice versa is graphically applying DeMorgan's theorem.



Exercise 2.50

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

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



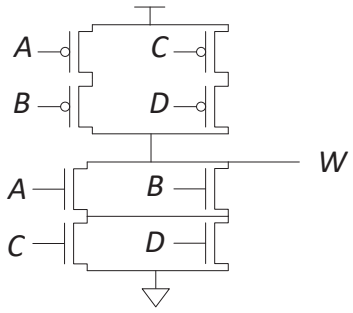
Now perform DeMorgan's on the equation to draw the pull-up network.

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

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

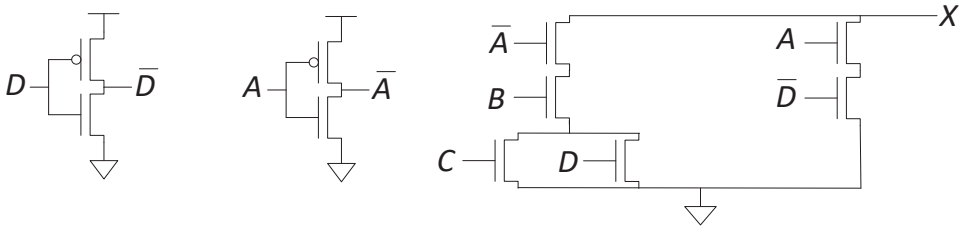
$Y = \bar{A}\bar{B} + \bar{C}\bar{D}$

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



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

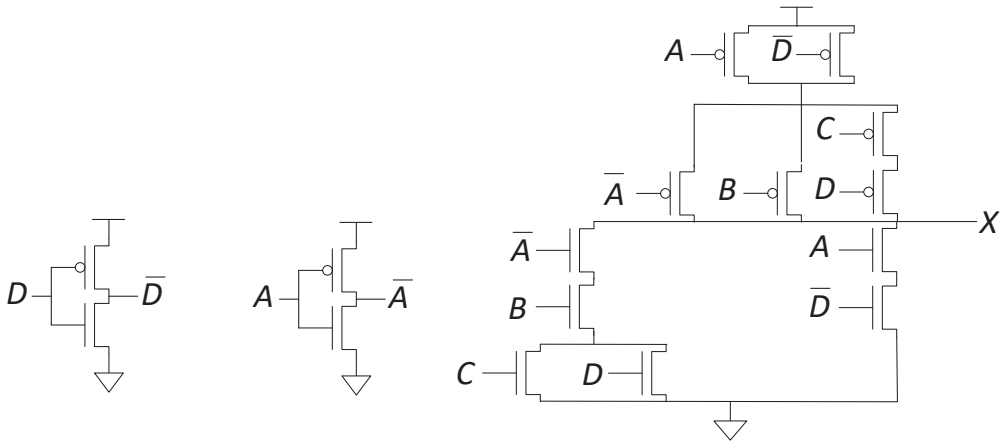
First, we build the pull-down network directly using the equation above:



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

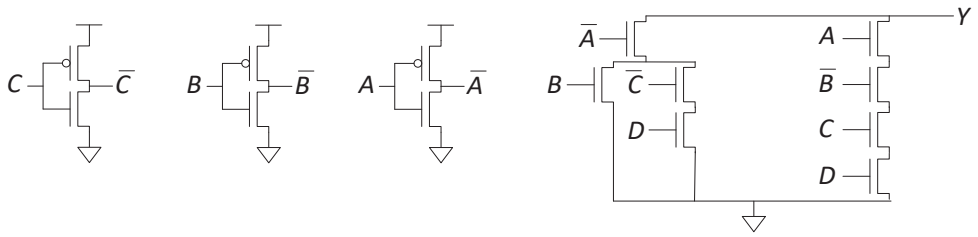
$Y = \overline{\bar{A}B(C+D) + A\bar{D}}$
$Y = \overline{\bar{A}B(C+D)} * \overline{A\bar{D}}$
$Y = (\overline{\bar{A}B} + \overline{(C+D)}) * (\bar{A} + D)$
$Y = (A + \bar{B} + \bar{C}\bar{D}) * (\bar{A} + D)$
$Y = (A + ((\bar{B} + \bar{C}) * (B + C)) * (\bar{A} + B + \bar{C})$

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



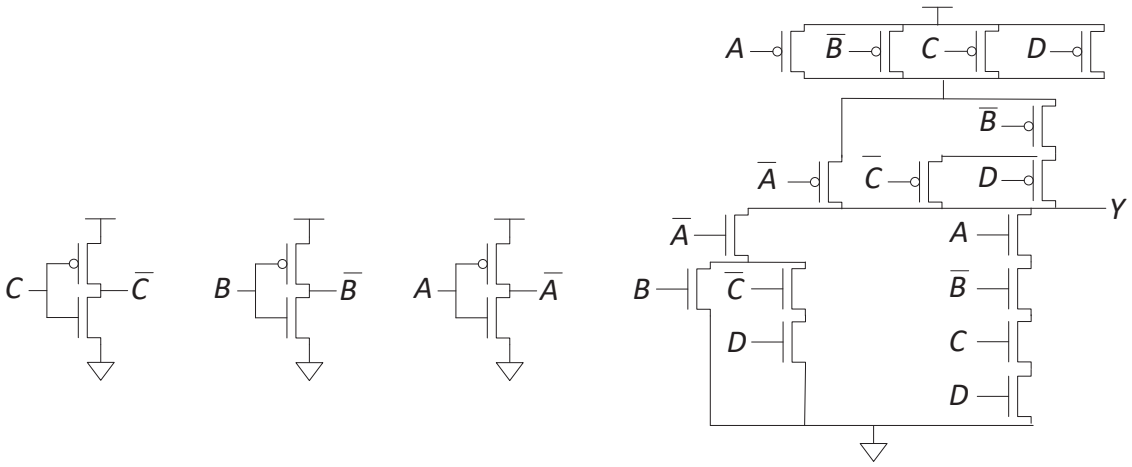
(c) $Y = \overline{\bar{A}(B + \bar{C}D) + A\bar{B}CD}$

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

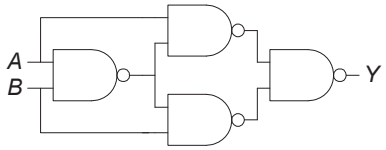


Now perform DeMorgan's on the equation to draw the pull-up network.

$Y = \overline{\bar{A}(B + \bar{C}D) + A\bar{B}CD}$
$Y = \overline{\bar{A}(B + \bar{C}D)} * \overline{A\bar{B}CD}$
$Y = \left[A + \overline{(B + \bar{C}D)}\right] * (\bar{A} + B + \bar{C} + \bar{D})$
$Y = \left[A + \bar{B} * \overline{\bar{C}D}\right] * (\bar{A} + B + \bar{C} + \bar{D})$
$Y = [A + \bar{B} * (C + \bar{D})] * (\bar{A} + B + \bar{C} + \bar{D})$

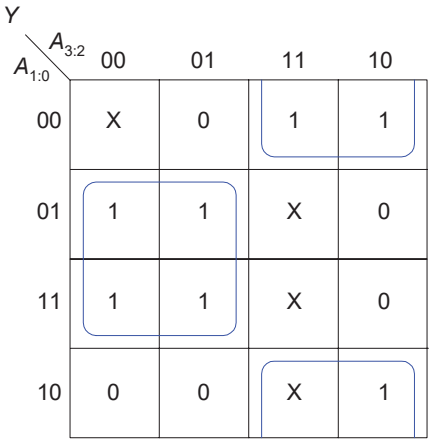


Question 2.1

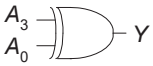


Question 2.2

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



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



Question 2.3

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

Question 2.4

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

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

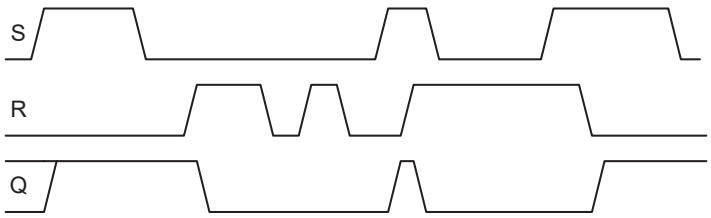
(c) The NAND gate by itself is universal. A NAND gate with its inputs tied together performs the NOT operation. A NAND gate with a NOT gate on its output performs AND. And a NAND gate with NOT gates on its inputs performs OR. Thus, a NAND gate is equivalent to the set {AND, OR, NOT} and is universal.

Question 2.5

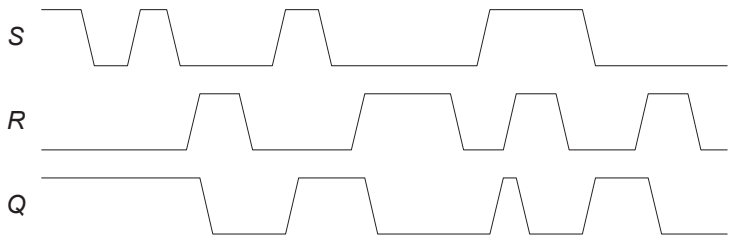
A circuit's contamination delay might be less than its propagation delay because the circuit may operate over a range of temperatures and supply voltages, for example, 3-3.6 V for LVCMOS (low voltage CMOS) chips. As temperature increases and voltage decreases, circuit delay increases. Also, the circuit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

CHAPTER 3

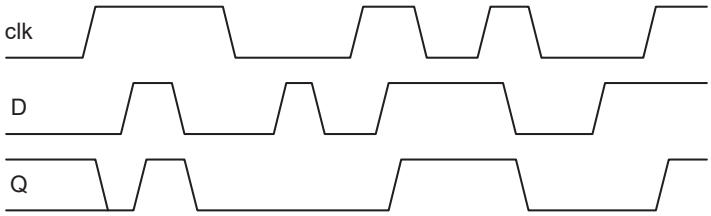
Exercise 3.1



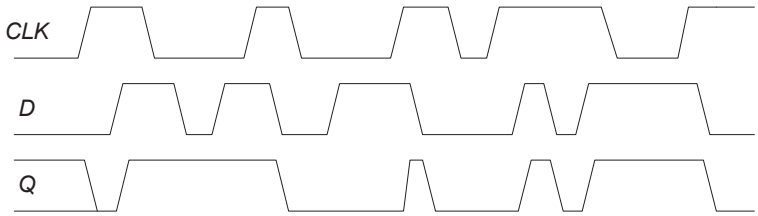
Exercise 3.2



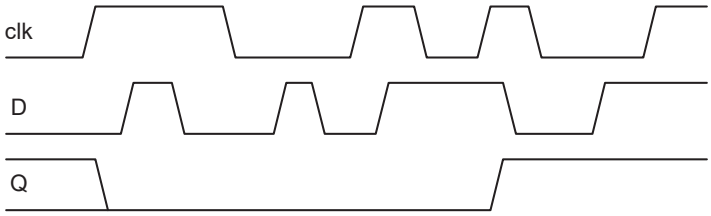
Exercise 3.3



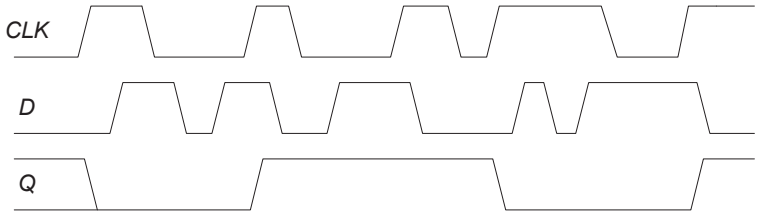
Exercise 3.4



Exercise 3.5



Exercise 3.6



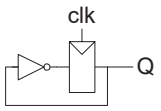
Exercise 3.7

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

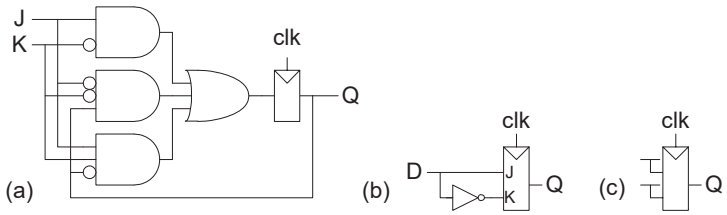
Exercise 3.8

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

Exercise 3.9



Exercise 3.10

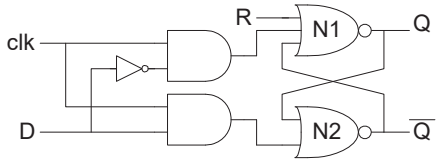


Exercise 3.11

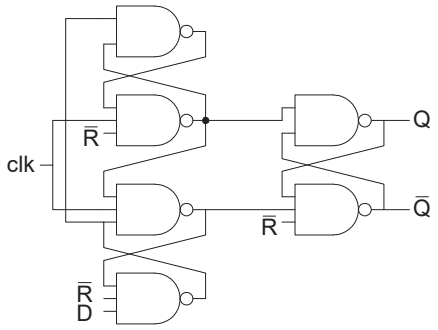
If A and B have the same value, C takes on that value. Otherwise, C retains its old value.

Exercise 3.12

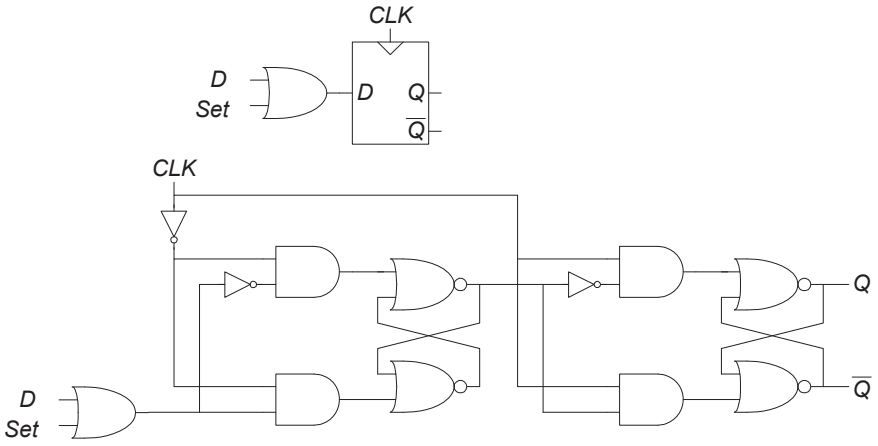
Make sure these next ones are correct too.



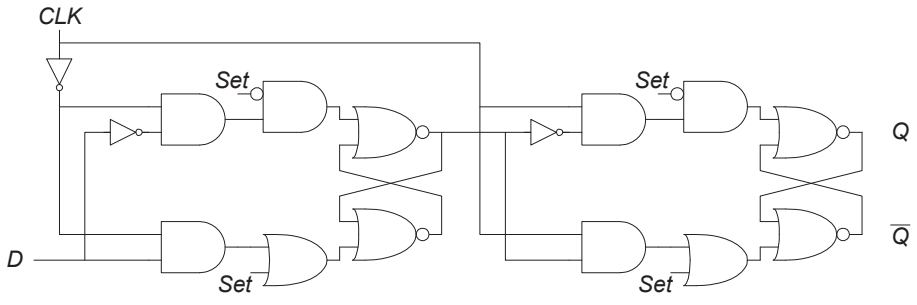
Exercise 3.13



Exercise 3.14



Exercise 3.15



Exercise 3.16

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

Exercise 3.17

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

Exercise 3.18

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

Exercise 3.19

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

Exercise 3.20

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

Exercise 3.21

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

Exercise 3.22

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

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

TABLE 3.1 State encoding for Exercise 3.22

.

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

TABLE 3.2 State transition table with binary encodings for Exercise 3.22

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

TABLE 3.2 State transition table with binary encodings for Exercise 3.22

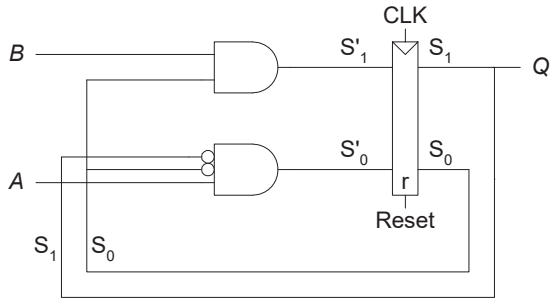
.

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

TABLE 3.3 Output table with binary encodings for Exercise 3.22

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

$$Q = S_1$$



Exercise 3.23

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

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

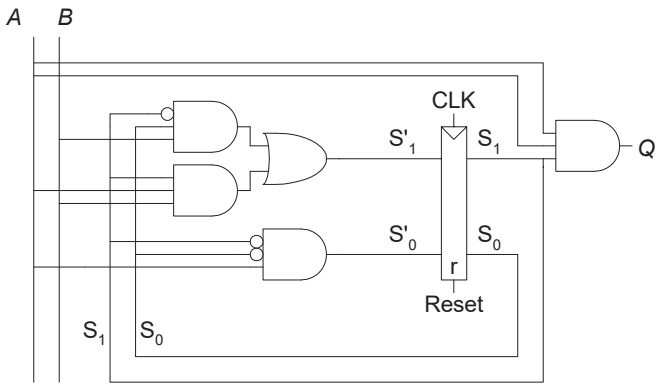
TABLE 3.4 State encoding for Exercise 3.23

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

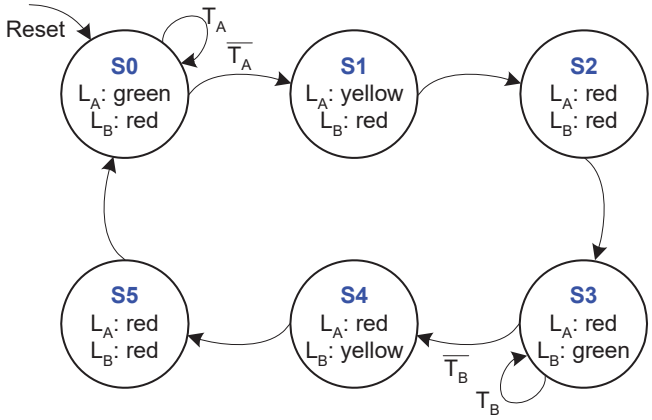
TABLE 3.5 Combined state transition and output table with binary encodings for Exercise 3.23

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

$$Q' = S_1 A B$$



Exercise 3.24



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

TABLE 3.6 State encoding for Exercise 3.24

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

TABLE 3.6 State encoding for Exercise 3.24

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

TABLE 3.7 State transition table with binary encodings for Exercise 3.24

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

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

TABLE 3.8 Output table for Exercise 3.24

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

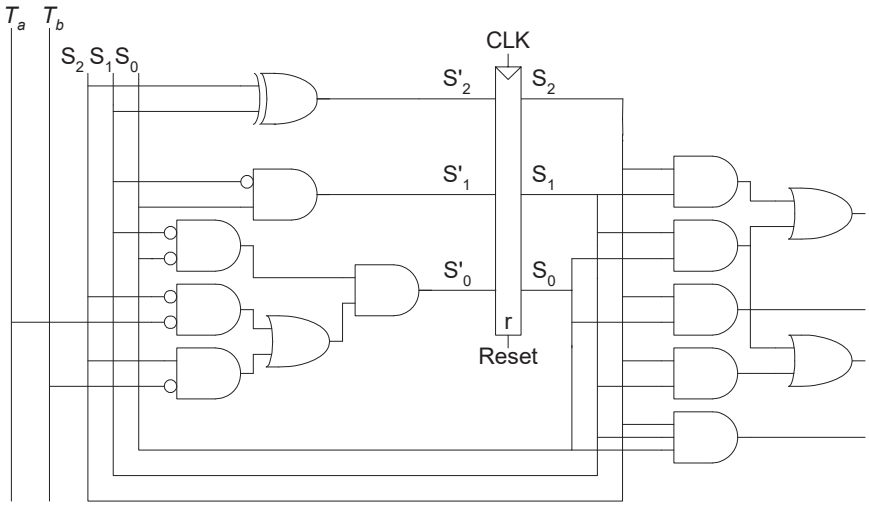
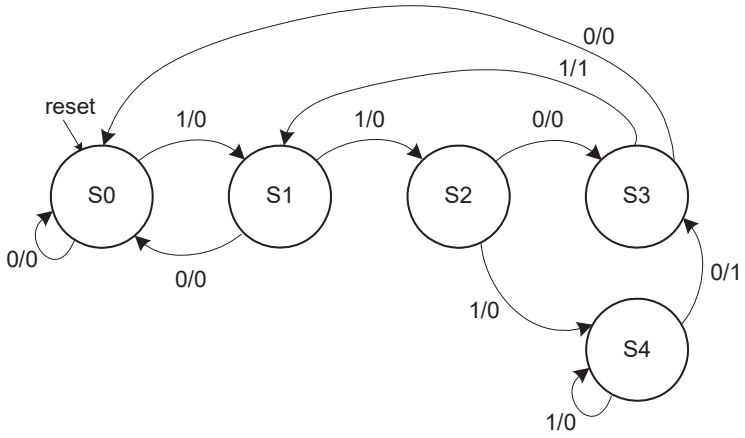


FIGURE 3.1 State machine circuit for traffic light controller for Exercise 3.21

Exercise 3.25



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

TABLE 3.9 State encoding for Exercise 3.25

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

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

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

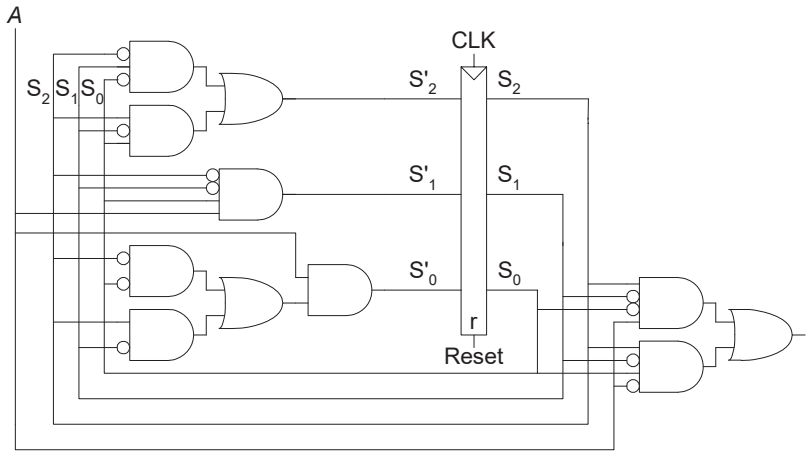
TABLE 3.10 Combined state transition and output table with binary encodings for Exercise 3.25

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

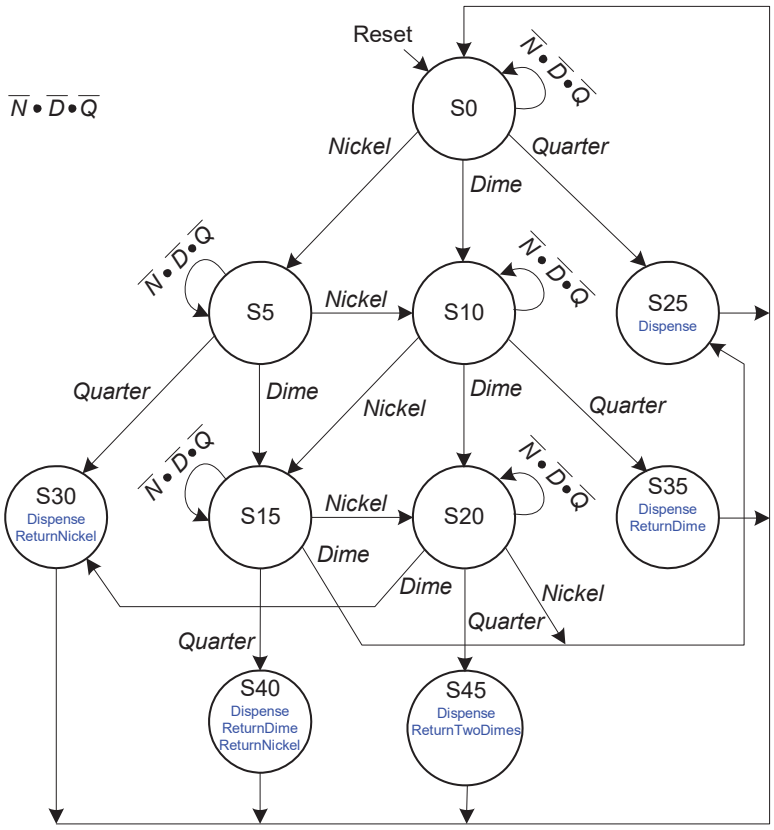
$$S'_1 = \overline{S_2} \overline{S_1} S_0 A$$

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

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



Exercise 3.26



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

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

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

FIGURE 3.3 State Encodings for Exercise 3.26

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

TABLE 3.11 State transition table for Exercise 3.26

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

TABLE 3.11 State transition table for Exercise 3.26

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

TABLE 3.12 State transition table for Exercise 3.26

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

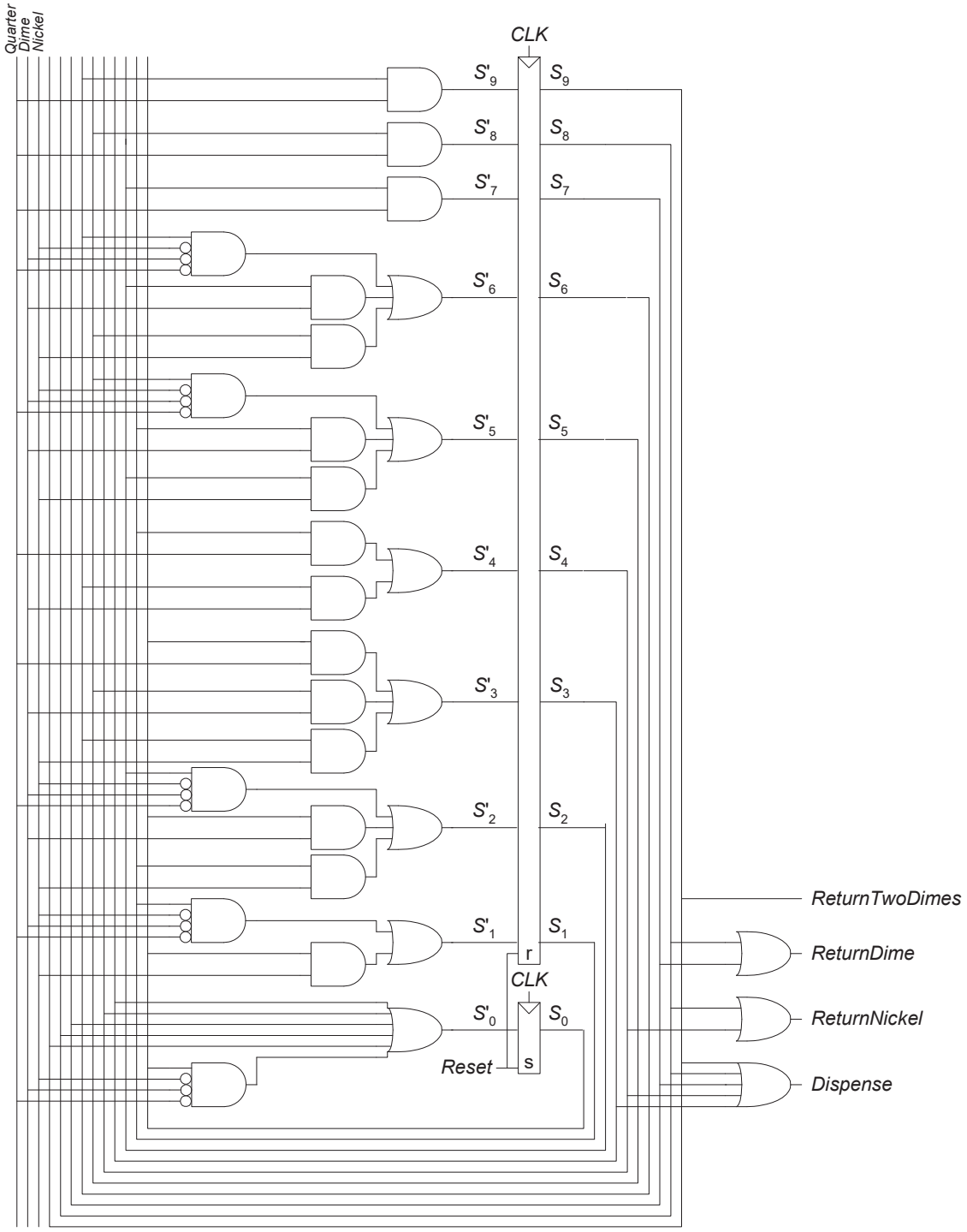
TABLE 3.12 State transition table for Exercise 3.26

$$S'_9 = S_6 Q$$

$$S'_8 = S_5 Q$$

$$S'_7 = S_2 Q$$
$$S'_6 = S_2 D + S_5 N + S_6 \overline{N}\,\overline{D}\,\overline{Q}$$
$$S'_5 = S_1 D + S_2 N + S_5 NDQ$$
$$S'_4 = S_1 Q + S_6 D$$
$$S'_3 = S_0 Q + S_5 D + S_6 N$$
$$S'_2 = S_0 D + S_1 N + S_2 \overline{N}\,\overline{D}\,\overline{Q}$$
$$S'_1 = S_0 N + S_1 NDQ$$
$$S'_0 = S_0 \overline{N}\,\overline{D}\,\overline{Q} + S_3 + S_4 + S_7 + S_8 + S_9$$

$$Dispense = S_3 + S_4 + S_7 + S_8 + S_9$$
$$ReturnNickel = S_4 + S_8$$
$$ReturnDime = S_7 + S_8$$
$$ReturnTwoDimes = S_9$$


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

Exercise 3.27

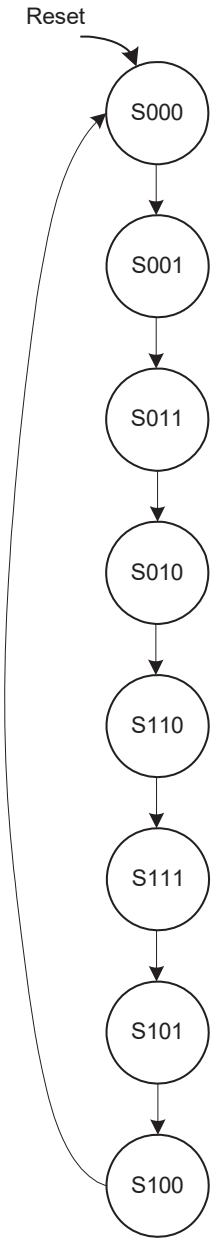


FIGURE 3.4 State transition diagram for Exercise 3.27

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

TABLE 3.13 State transition table for Exercise 3.27

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

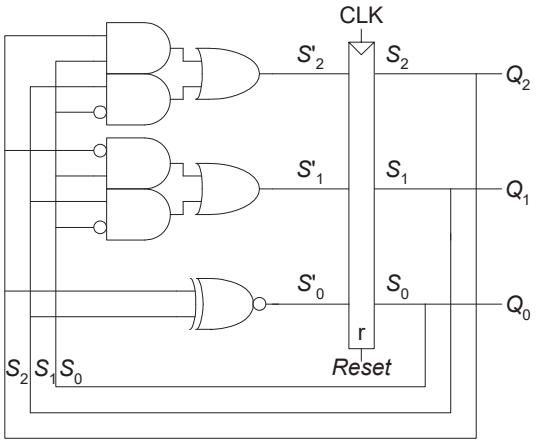


FIGURE 3.5 Hardware for Gray code counter FSM for Exercise 3.27

Exercise 3.28

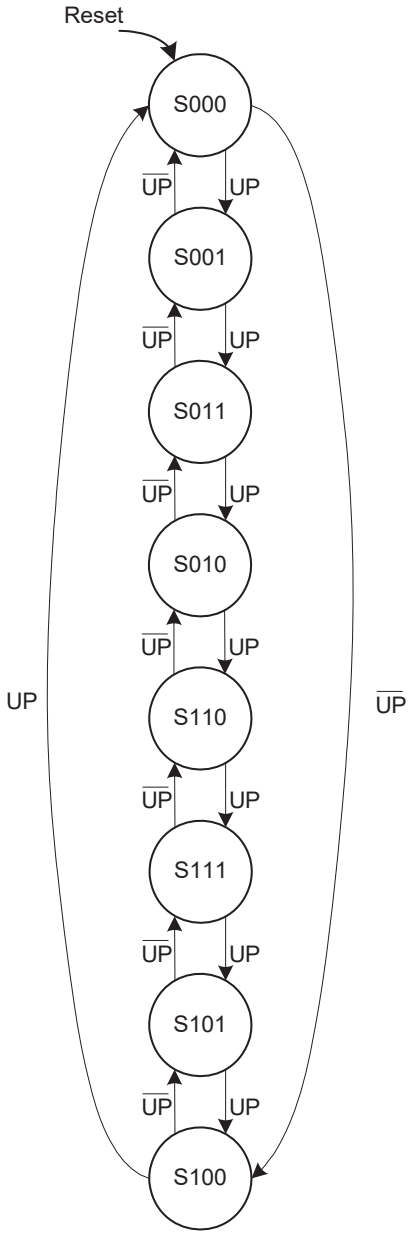


FIGURE 3.6 State transition diagram for Exercise 3.28

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

TABLE 3.14 State transition table for Exercise 3.28

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

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

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

$$Q_2 = S_2$$

$$Q_1 = S_1$$

$$Q_0 = S_0$$

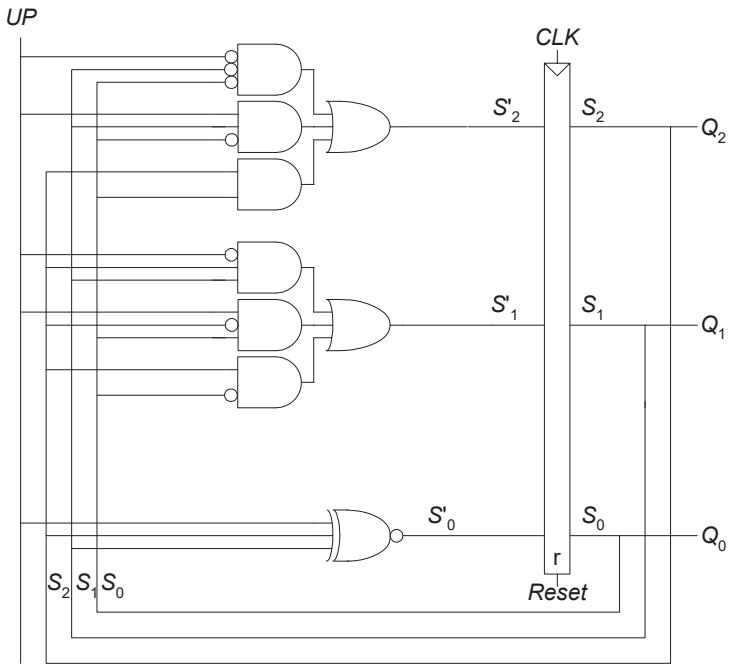


FIGURE 3.7 Finite state machine hardware for Exercise 3.28

Exercise 3.29

(a)

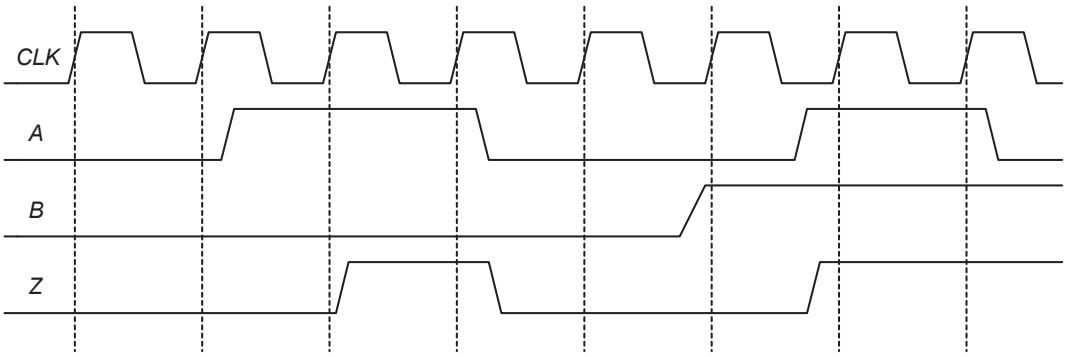


FIGURE 3.8 Waveform showing *Z* output for Exercise 3.29

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

(c)

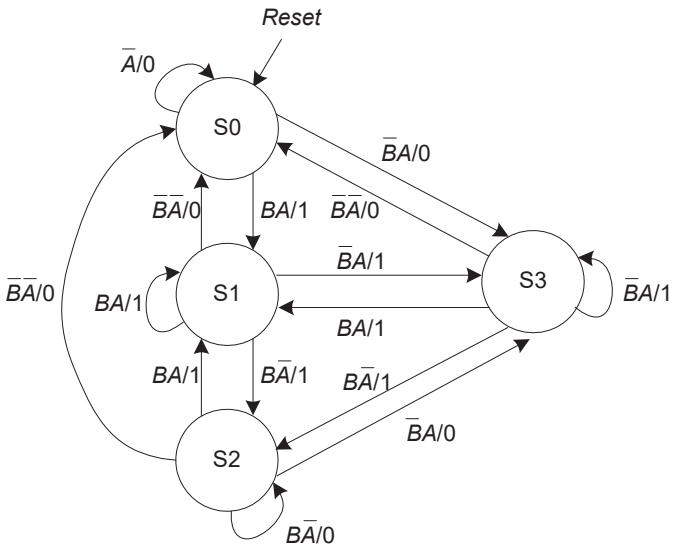


FIGURE 3.9 State transition diagram for Exercise 3.29

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

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

TABLE 3.15 State transition table for Exercise 3.29

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

TABLE 3.15 State transition table for Exercise 3.29

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

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

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

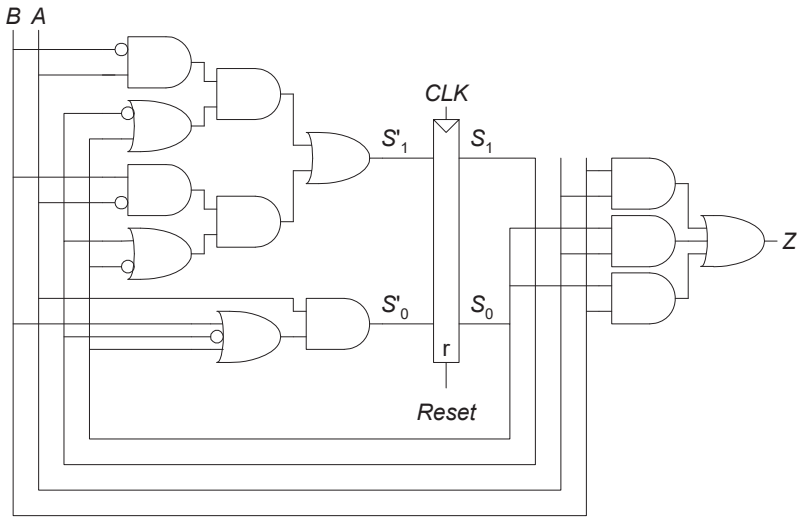


FIGURE 3.10 Hardware for FSM of Exercise 3.26

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

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

Exercise 3.30

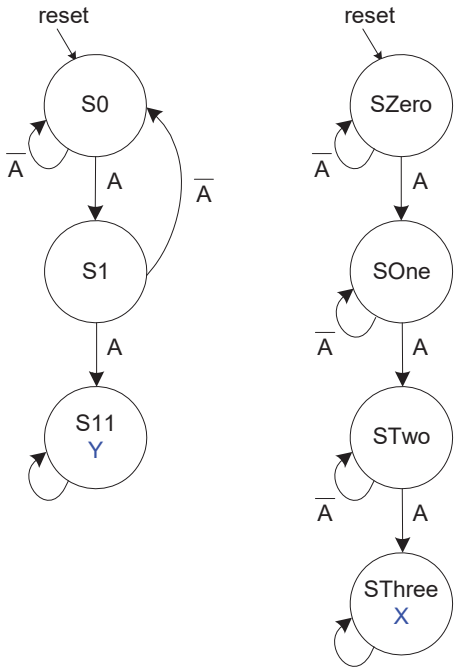


FIGURE 3.11 Factored state transition diagram for Exercise 3.30

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

TABLE 3.16 State transition table for output Y for Exercise 3.30

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

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

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

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

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

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

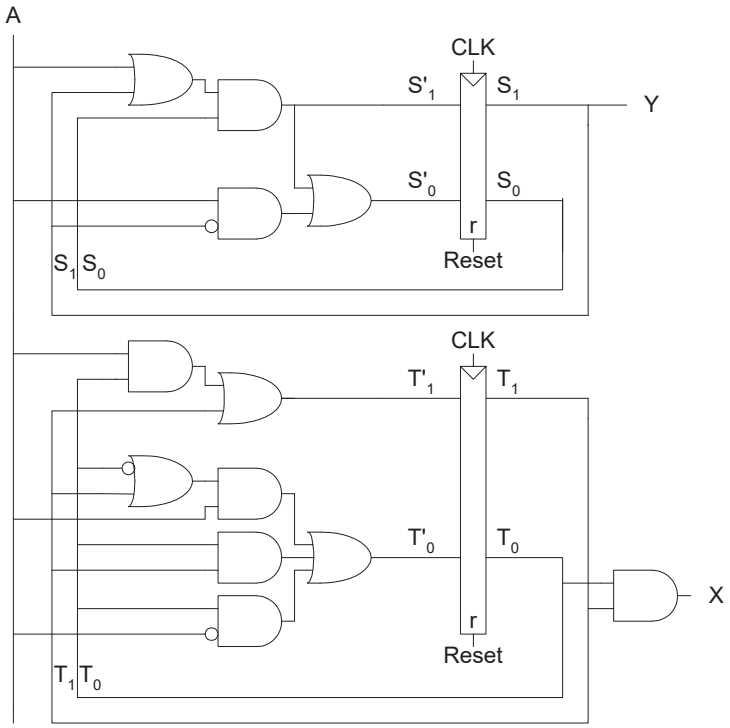


FIGURE 3.12 Finite state machine hardware for Exercise 3.30

Exercise 3.31

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

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

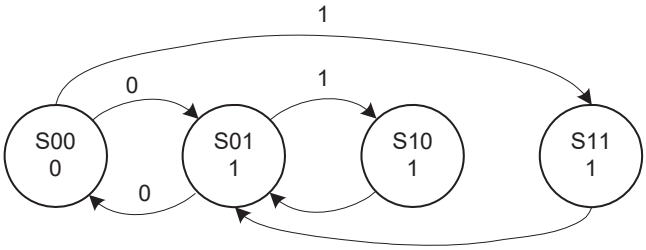
TABLE 3.18 State transition table with binary encodings for Exercise 3.31

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

TABLE 3.18 State transition table with binary encodings for Exercise 3.31

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

TABLE 3.19 Output table for Exercise 3.31



Exercise 3.32

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

TABLE 3.20 State transition table with binary encodings for Exercise 3.32

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

TABLE 3.20 State transition table with binary encodings for Exercise 3.32

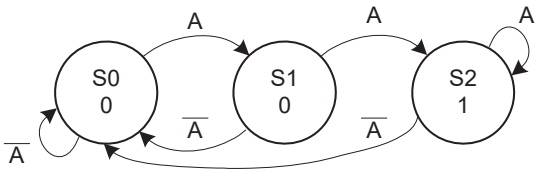


FIGURE 3.13 State transition diagram for Exercise 3.32

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

Exercise 3.33

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

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

Next, we calculate the cycle time:

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

(b)

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

Thus,

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

(c)

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

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

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

(d)

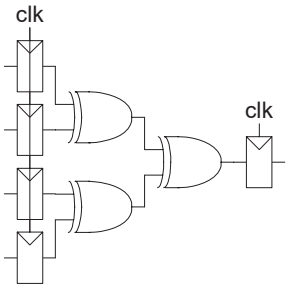


FIGURE 3.14 Alyssa's improved circuit for Exercise 3.33

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

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

Next, we calculate the cycle time:

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

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

Exercise 3.34

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

Exercise 3.35

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

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

Exercise 3.36

1.138 ns

Exercise 3.37

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

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

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

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

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

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

Exercise 3.38

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

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

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

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

Thus,

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

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

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

Exercise 3.39

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

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

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

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

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

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

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

Exercise 3.40

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

Question 3.1

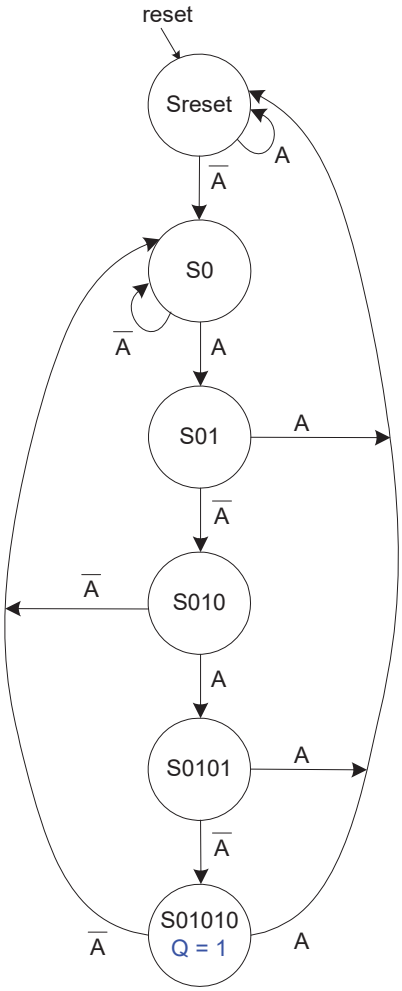


FIGURE 3.15 State transition diagram for Question 3.1

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

TABLE 3.21 State transition table for Question 3.1

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

TABLE 3.21 State transition table for Question 3.1

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

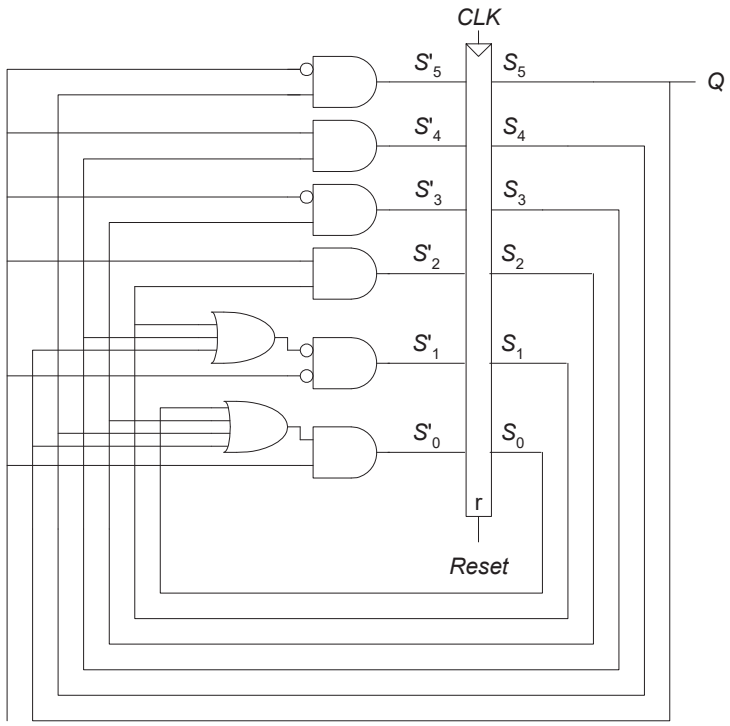


FIGURE 3.16 Finite state machine hardware for Question 3.1

Question 3.2

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

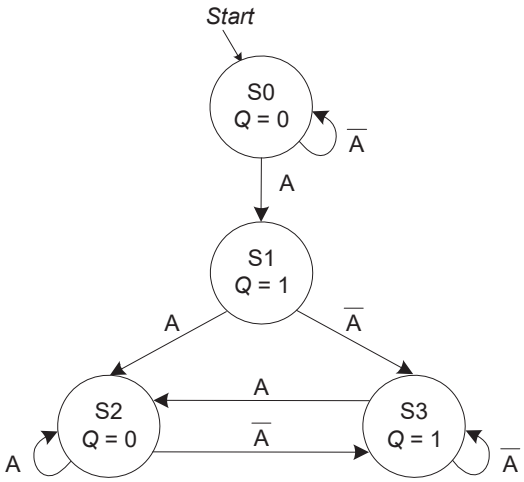


FIGURE 3.17 State transition diagram for Question 3.2

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

TABLE 3.22 State transition table for Question 3.2

$$S'_1 = S_1 + S_0$$

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

$$Q = S_0$$

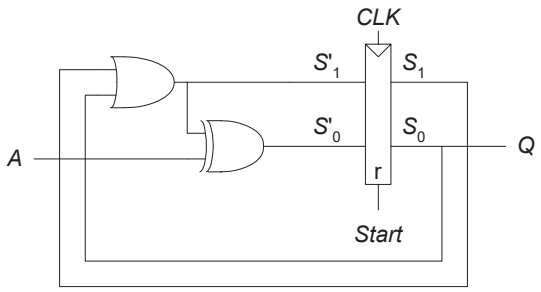


FIGURE 3.18 Finite state machine hardware for Question 3.2

Question 3.3

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

Question 3.4

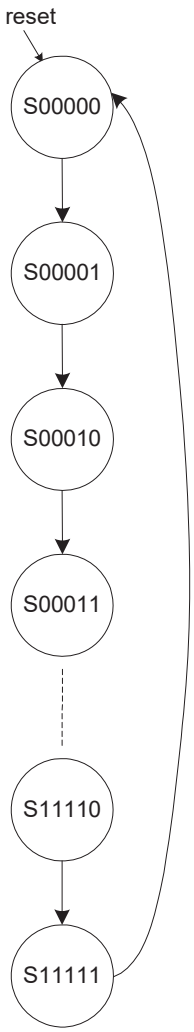


FIGURE 3.19 State transition diagram for Question 3.4

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

TABLE 3.23 State transition table for Question 3.4

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

TABLE 3.23 State transition table for Question 3.4

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

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

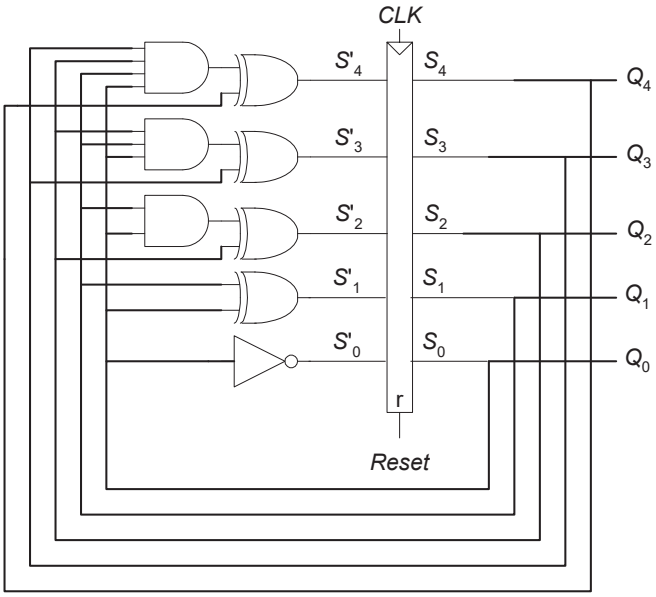


FIGURE 3.20 Finite state machine hardware for Question 3.4

Question 3.5

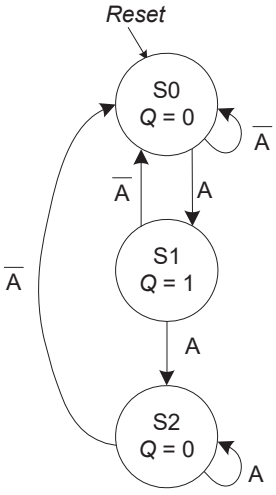


FIGURE 3.21 State transition diagram for edge detector circuit of Question 3.5

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

TABLE 3.24 State transition table for Question 3.5

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

$$Q = S_1$$

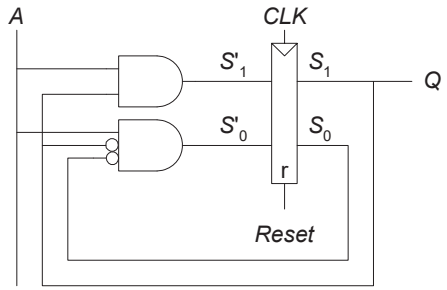


FIGURE 3.22 Finite state machine hardware for Question 3.5

Question 3.6

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

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

Question 3.7

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

Question 3.8

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

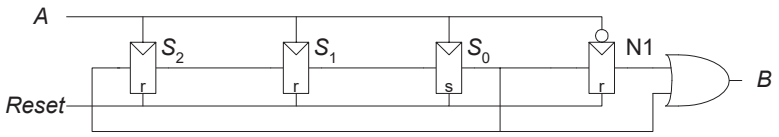


FIGURE 3.23 Hardware for Question 3.8

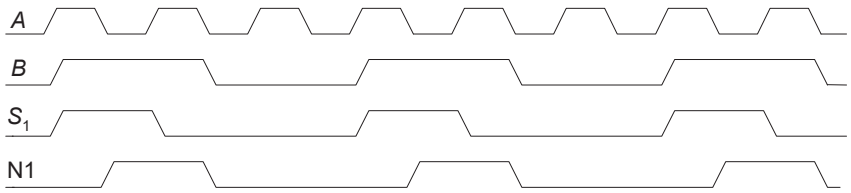


FIGURE 3.24 Waveforms for Question 3.8

Question 3.9

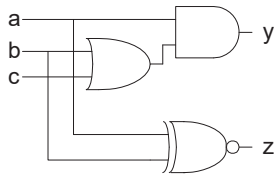
Without the added buffer, the propagation delay through the logic, t_{pd}, must be less than or equal to T_c - (t_{pcq} + t_{setup}). However, if you add a buffer to the clock input of the receiver, the clock arrives at the receiver later. The earliest that the clock edge arrives at the receiver is $t_{\text{cd\_BUF}}$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

CHAPTER 4

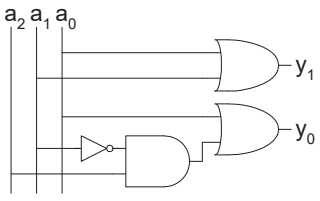
Note: the HDL files given in the following solutions are available on the textbook's companion website at:

http://textbooks.elsevier.com/9780123704979

Exercise 4.1



Exercise 4.2



Exercise 4.3

SystemVerilog

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

   assign y = ^a;
endmodule
```

VHDL

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

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

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

Exercise 4.4

ex4_4.tv file:

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

SystemVerilog

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

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

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

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

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

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

VHDL

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

entity ex4_4_testbench is -- no inputs or outputs
end;

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

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

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

(*VHDL continued on next page*)

(*continued from previous page*)

VHDL

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

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

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

Exercise 4.5

SystemVerilog

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

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

VHDL

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

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

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

Exercise 4.6

SystemVerilog

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

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

VHDL

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

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

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

Exercise 4.7

ex4_7.tv file:

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

Option 1:

SystemVerilog

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

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

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

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

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

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

VHDL

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

entity ex4_7_testbench is -- no inputs or outputs
end;

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

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

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

(*VHDL continued on next page*)

(*continued from previous page*)

VHDL

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

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

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

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

Option 2 (VHDL only):

VHDL

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

entity ex4_7_testbench is -- no inputs or outputs
end;

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

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

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

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

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

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

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

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

Exercise 4.8

SystemVerilog

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

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

VHDL

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

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

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

Exercise 4.9

SystemVerilog

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

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

VHDL

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

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

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

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

Exercise 4.10

SystemVerilog

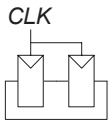
```
module ex4_10
   (input  logic a, b, c,
    output logic y);

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

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

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

VHDL

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

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

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

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

Exercise 4.11

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



Exercise 4.12

SystemVerilog

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

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

VHDL

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

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

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

Exercise 4.13

SystemVerilog

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

VHDL

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

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

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

Exercise 4.14

SystemVerilog

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

  logic [11:0] y2_4;

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

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

VHDL

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

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

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

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

(continued on next page)

(continued from previous page)

SystemVerilog

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

VHDL

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

Exercise 4.15

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

SystemVerilog

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

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

VHDL

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

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

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

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

SystemVerilog

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

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

VHDL

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

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

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

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

SystemVerilog

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

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

VHDL

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

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

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

Exercise 4.16

SystemVerilog

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

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

VHDL

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

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

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

end;
```

Exercise 4.17

SystemVerilog

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

  logic n1, n2, n3, n4, n5;

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

VHDL

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

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

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

Exercise 4.18

Verilog

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

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

VHDL

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

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

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

Exercise 4.19

SystemVerilog

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

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

VHDL

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

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

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

Exercise 4.20

SystemVerilog

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

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

VHDL

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

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

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

Exercise 4.21

SystemVerilog

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

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

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

VHDL

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

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

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

Exercise 4.22

SystemVerilog

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

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

VHDL

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

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

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

Exercise 4.23

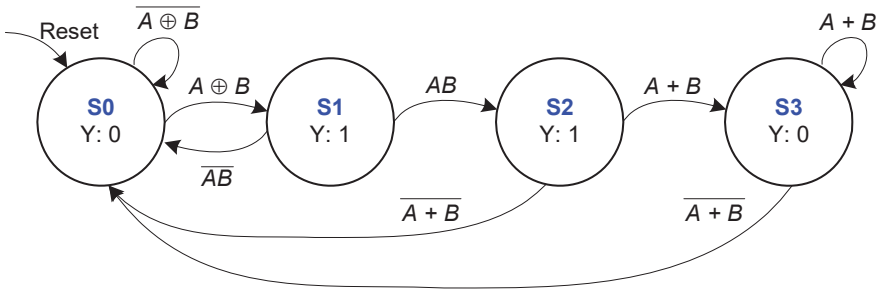
SystemVerilog

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

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

VHDL

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

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

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

Exercise 4.24



Exercise 4.25

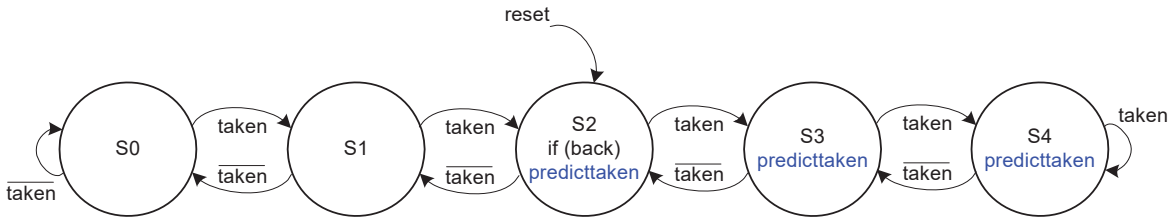


FIGURE 4.1 State transition diagram for Exercise 4.25

Exercise 4.26

SystemVerilog

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

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

VHDL

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

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

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

Exercise 4.27

SystemVerilog

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

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

VHDL

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

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

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

Exercise 4.28

SystemVerilog

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

  logic n1, n2, clk_b;

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

VHDL

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

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

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

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

Exercise 4.29

SystemVerilog

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

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

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

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

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

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

VHDL

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

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

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

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

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

Exercise 4.30

Mode Module

SystemVerilog

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

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

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

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

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

VHDL

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

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

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

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

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

(continued on next page)

Lights Module

SystemVerilog

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

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

  statetype [1:0] state, nextstate;

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

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

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

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

VHDL

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

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

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

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

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

(continued on next page)

Controller Module

SystemVerilog

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

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

VHDL

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

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

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

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

Exercise 4.31

SystemVerilog

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

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

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

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

VHDL

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

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

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

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

Exercise 4.32

SystemVerilog

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

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

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

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

VHDL

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

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

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

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

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

Exercise 4.33

SystemVerilog

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

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

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

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

VHDL

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

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

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

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

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

Exercise 4.34

SystemVerilog

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

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

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

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

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

VHDL

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

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

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

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

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

Exercise 4.35

SystemVerilog

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

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

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

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

VHDL

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

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

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

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

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

Exercise 4.36

(starting on next page)

SystemVerilog

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

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

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

VHDL

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

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

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

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

(continued from previous page)

SystemVerilog

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

VHDL

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

Exercise 4.37

SystemVerilog

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

  statetype [2:0] state, nextstate;

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

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

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

VHDL

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

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

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

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

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

Exercise 4.38

SystemVerilog

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

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

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

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

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

(continued on next page)

VHDL

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

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

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

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

(continued from previous page)

VHDL

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

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

Exercise 4.39

Option 1

SystemVerilog

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

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

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

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

VHDL

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

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

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

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

(continued from previous page)

VHDL

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

Option 2

SystemVerilog

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

  logic aprev;

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

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

VHDL

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

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

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


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

Exercise 4.40

SystemVerilog

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

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

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

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

VHDL

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

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

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

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

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

(continued on next page)

(continued from previous page)

SystemVerilog

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

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

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

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

VHDL

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

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

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

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

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

Exercise 4.41

SystemVerilog

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

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

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

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

VHDL

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

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

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

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

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

Exercise 4.42

SystemVerilog

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

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

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

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

VHDL

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

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

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

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

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

Exercise 4.43

SystemVerilog

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

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

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

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

VHDL

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

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

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

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

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

Exercise 4.44

(a)

SystemVerilog

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

  logic areg, breg, creg, dreg;

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

VHDL

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

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

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

(d)

SystemVerilog

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

  logic areg, breg, creg, dreg;

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

VHDL

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

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

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

Exercise 4.45

SystemVerilog

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

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

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

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

VHDL

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

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

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

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

Exercise 4.46

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

Exercise 4.47

SystemVerilog

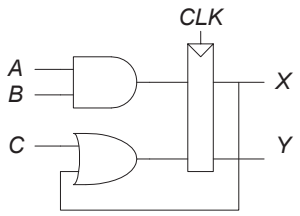
```
module syncbad(input  logic clk,
               input  logic d,
               output logic q);

  logic n1;

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

VHDL

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

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

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

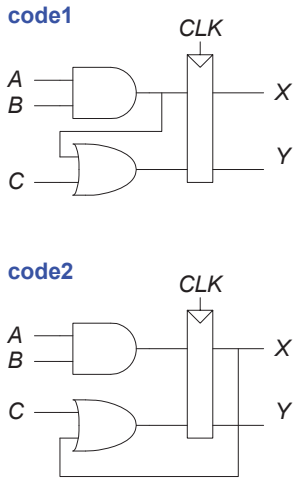
Exercise 4.48

They have the same function.



Exercise 4.49

They do not have the same function.



Exercise 4.50

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

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

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

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

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

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

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

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

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

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

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

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

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

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

   logic  state, nextstate;

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

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

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

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

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

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

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

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

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

   logic  [1:0] state, nextstate;

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

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

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

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

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

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

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

endmodule
```

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

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

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

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

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

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

   logic tmp;

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

Exercise 4.51

It is necessary to write

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

rather than simply

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

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

Exercise 4.52

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Question 4.1

SystemVerilog

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

VHDL

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

Question 4.2

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

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

SystemVerilog

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

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

VHDL

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

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

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

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

Question 4.3

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

CHAPTER 5

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

Exercise 5.1

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

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

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

$$t_{CLA} = t_{\text{pg}} + t_{\text{pg\_block}} + \left(\frac{N}{k} - 1\right) t_{\text{AND\_OR}} + kt_{\text{FA}} \qquad 150$$

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

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

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

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

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

Exercise 5.2

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

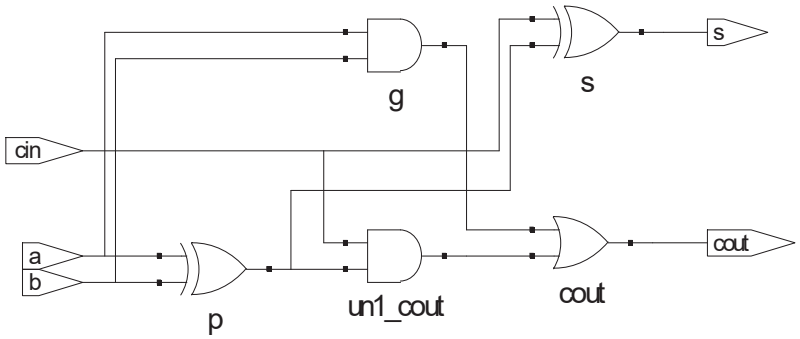


FIGURE 5.1 Full adder implementation

The full adder delay is three two-input gates.

$$t_{FA} = 3(50)\text{ ps} = 150\text{ ps}$$

The full adder area is five two-input gates.

$$A_{FA} = 5(15\ \mu m^2) = 75\ \mu m^2$$

The full adder capacitance is five two-input gates.

$$C_{FA} = 5(20\ \text{fF}) = 100\text{ fF}$$

Thus, the ripple-carry adder delay, area, and capacitance are:

$$t_{\text{ripple}} = Nt_{\text{FA}} = 64(150\text{ ps}) = 9.6\text{ ns}$$

$$A_{\text{ripple}} = NA_{\text{FA}} = 64(75\ \mu m^2) = 4800\ \mu m^2$$

$$C_{\text{ripple}} = NC_{\text{FA}} = 64(100\text{ fF}) = 6.4\text{ pF}$$

Using the carry-lookahead adder from Figure 5.6, we can calculate delay, area, and capacitance. Using Equation 5.6:

$$t_{CLA} = [50 + 6(50) + 15(100) + 4(150)]\text{ ps} = 2.45\text{ ns}$$

(The actual delay is only 2.4 ns because the first AND_OR gate only contributes one gate delay.)

For each 4-bit block of the 64-bit carry-lookahead adder, there are 4 full adders, 8 two-input gates to generate P_i and G_i, and 11 two-input gates to generate $P_{i:j}$ and $G_{i:j}$. Thus, the area and capacitance are:

$$A_{CLAblock} = [4(75) + 19(15)]\ \mu m^2 = 585\ \mu m^2$$

$$A_{CLA} = 16(585)\ \mu m^2 = 9360\ \mu m^2$$

$$C_{CLAblock} = [4(100) + 19(20)]\ \text{fF} = 780\ \text{fF}$$

$$C_{CLA} = 16(780)\ \text{fF} = 12.48\ \text{pF}$$

Now solving for power using Equation 1.4,

$$P_{\text{dynamic\_ripple}} = \frac{1}{2}CV_{DD}^2 f = \frac{1}{2}(6.4\ \text{pF})(1.2\ \text{V})^2(100\text{MHz}) = 0.461\ mW$$

$$P_{\text{dynamic\_CLA}} = \frac{1}{2}CV_{DD}^2 f = \frac{1}{2}(12.48\ \text{pF})(1.2\ \text{V})^2(100\text{MHz}) = 0.899\ mW$$

.

	ripple-carry	carry-lookahead	cla/ripple
Area (μm^2)	4800	9360	1.95
Delay (ns)	9.6	2.45	0.26
Power (mW)	0.461	0.899	1.95

TABLE 5.1 CLA and ripple-carry adder comparison

(b) Compared to the ripple-carry adder, the carry-lookahead adder is almost twice as large and uses almost twice the power, but is almost four times as fast. Thus for performance-limited designs where area and power are not constraints, the carry-lookahead adder is the clear choice. On the other hand, if either area or power are the limiting constraints, one would choose a ripple-carry adder if performance were not a constraint.

Exercise 5.3

A designer might choose to use a ripple-carry adder instead of a carry-lookahead adder if chip area is the critical resource and delay is not the critical constraint.

Exercise 5.4

SystemVerilog

```
module prefixadd16(input  logic [15:0] a, b,
                   input  logic        cin,
                   output logic [15:0] s,
                   output logic        cout);

  logic [14:0] p, g;
  logic [7:0]  pij_0, gij_0, pij_1, gij_1,
               pij_2, gij_2, pij_3, gij_3;
  logic [15:0] gen;


  pgblock pgblock_top(a[14:0], b[14:0], p, g);
  pgblackblock pgblackblock_0({p[14], p[12], p[10],
   p[8], p[6], p[4], p[2], p[0]},
  {g[14], g[12], g[10], g[8], g[6], g[4], g[2], g[0]},
  {p[13], p[11], p[9], p[7], p[5], p[3], p[1], 1'b0},
  {g[13], g[11], g[9], g[7], g[5], g[3], g[1], cin},
                              pij_0, gij_0);

  pgblackblock pgblackblock_1({pij_0[7], p[13],
    pij_0[5], p[9], pij_0[3], p[5], pij_0[1], p[1]},
   {gij_0[7], g[13], gij_0[5], g[9], gij_0[3],
    g[5], gij_0[1], g[1]},
   { {2{pij_0[6]}}, {2{pij_0[4]}}, {2{pij_0[2]}},
     {2{pij_0[0]}} },
   { {2{gij_0[6]}}, {2{gij_0[4]}}, {2{gij_0[2]}},
     {2{gij_0[0]}} },
                              pij_1, gij_1);

  pgblackblock pgblackblock_2({pij_1[7], pij_1[6],
pij_0[6], p[11], pij_1[3], pij_1[2], pij_0[2], p[3]},
{gij_1[7], gij_1[6], gij_0[6], g[11], gij_1[3],
gij_1[2], gij_0[2], g[3]},
{ {4{pij_1[5]}}, {4{pij_1[1]}} },
{ {4{gij_1[5]}}, {4{gij_1[1]}} },
 pij_2, gij_2);

  pgblackblock pgblackblock_3({pij_2[7], pij_2[6],
    pij_2[5], pij_2[4], pij_1[5], pij_1[4],
    pij_0[4], p[7]},
   {gij_2[7], gij_2[6], gij_2[5],
    gij_2[4], gij_1[5], gij_1[4], gij_0[4], g[7]},
  { 8{pij_2[3]} },{ 8{gij_2[3]} }, pij_3, gij_3);

  sumblock sum_out(a, b, gen, s);

  assign gen  = {gij_3, gij_2[3:0],
                 gij_1[1:0], gij_0[0], cin};
  assign cout = (a[15] & b[15]) |
                (gen[15] & (a[15] | b[15]));

endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity prefixadd16 is
  port(a, b: in  STD_LOGIC_VECTOR(15 downto 0);
       cin:  in  STD_LOGIC;
       s:    out STD_LOGIC_VECTOR(15 downto 0);
       cout: out STD_LOGIC);
end;

architecture synth of prefixadd16 is
  component pgblock
    port(a, b: in  STD_LOGIC_VECTOR(14 downto 0);
         p, g: out STD_LOGIC_VECTOR(14 downto 0));
  end component;

  component pgblackblock is
    port (pik, gik: in STD_LOGIC_VECTOR(7 downto 0);
          pkj, gkj: in STD_LOGIC_VECTOR(7 downto 0);
          pij: out STD_LOGIC_VECTOR(7 downto 0);
          gij: out STD_LOGIC_VECTOR(7 downto 0));
  end component;

  component sumblock is
    port (a, b, g: in  STD_LOGIC_VECTOR(15 downto 0);
          s:       out STD_LOGIC_VECTOR(15 downto 0));
  end component;

  signal p, g: STD_LOGIC_VECTOR(14 downto 0);
  signal pij_0, gij_0, pij_1, gij_1,
         pij_2, gij_2, gij_3:
                STD_LOGIC_VECTOR(7 downto 0);
  signal gen:  STD_LOGIC_VECTOR(15 downto 0);
  signal pik_0, pik_1, pik_2, pik_3,
         gik_0, gik_1, gik_2, gik_3,
         pkj_0, pkj_1, pkj_2, pkj_3,
         gkj_0, gkj_1, gkj_2, gkj_3, dummy:
                STD_LOGIC_VECTOR(7 downto 0);
begin
  pgblock_top: pgblock
    port map(a(14 downto 0), b(14 downto 0), p, g);

  pik_0 <=
    (p(14)&p(12)&p(10)&p(8)&p(6)&p(4)&p(2)&p(0));
  gik_0 <=
    (g(14)&g(12)&g(10)&g(8)&g(6)&g(4)&g(2)&g(0));
  pkj_0 <=
    (p(13)&p(11)&p(9)&p(7)&p(5)& p(3)& p(1)&'0');
  gkj_0 <=
    (g(13)&g(11)&g(9)&g(7)&g(5)& g(3)& g(1)& cin);

  pgblackblock_0: pgblackblock
         port map(pik_0, gik_0, pkj_0, gkj_0,
         pij_0, gij_0);
```

(*continued from previouspage*)

Verilog

VHDL

```
  pik_1 <= (pij_0(7)&p(13)&pij_0(5)&p(9)&
            pij_0(3)&p(5)&pij_0(1)&p(1));
  gik_1 <= (gij_0(7)&g(13)&gij_0(5)&g(9)&
            gij_0(3)&g(5)&gij_0(1)&g(1));
  pkj_1 <= (pij_0(6)&pij_0(6)&pij_0(4)&pij_0(4)&
            pij_0(2)&pij_0(2)&pij_0(0)&pij_0(0));
  gkj_1 <= (gij_0(6)&gij_0(6)&gij_0(4)&gij_0(4)&
            gij_0(2)&gij_0(2)&gij_0(0)&gij_0(0));

  pgblackblock_1: pgblackblock
        port map(pik_1, gik_1, pkj_1, gkj_1,
              pij_1, gij_1);

  pik_2 <= (pij_1(7)&pij_1(6)&pij_0(6)&
                    p(11)&pij_1(3)&pij_1(2)&
                    pij_0(2)&p(3));
  gik_2 <= (gij_1(7)&gij_1(6)&gij_0(6)&
                    g(11)&gij_1(3)&gij_1(2)&
                    gij_0(2)&g(3));
  pkj_2 <= (pij_1(5)&pij_1(5)&pij_1(5)&pij_1(5)&
     pij_1(1)&pij_1(1)&pij_1(1)&pij_1(1));
  gkj_2 <= (gij_1(5)&gij_1(5)&gij_1(5)&gij_1(5)&
     gij_1(1)&gij_1(1)&gij_1(1)&gij_1(1));

  pgblackblock_2: pgblackblock
   port map(pik_2, gik_2, pkj_2, gkj_2, pij_2, gij_2);

  pik_3 <= (pij_2(7)&pij_2(6)&pij_2(5)&
                   pij_2(4)&pij_1(5)&pij_1(4)&
                   pij_0(4)&p(7));
  gik_3 <= (gij_2(7)&gij_2(6)&gij_2(5)&
                   gij_2(4)&gij_1(5)&gij_1(4)&
                   gij_0(4)&g(7));
  pkj_3 <= (pij_2(3),pij_2(3),pij_2(3),pij_2(3),
    pij_2(3),pij_2(3),pij_2(3),pij_2(3));
  gkj_3 <= (gij_2(3),gij_2(3),gij_2(3),gij_2(3),
    gij_2(3),gij_2(3),gij_2(3),gij_2(3));

  pgblackblock_3: pgblackblock
         port map(pik_3, gik_3, pkj_3, gkj_3, dummy,
gij_3);

  sum_out: sumblock
        port map(a, b, gen, s);

  gen  <= (gij_3&gij_2(3 downto 0)&gij_1(1 downto 0)&
                  gij_0(0)&cin);
  cout <= (a(15) and b(15)) or
          (gen(15) and (a(15) or b(15)));
end;
```

(*continued from previous page*)

SystemVerilog

```
module pgblock(input  logic [14:0] a, b,
               output logic [14:0] p, g);

  assign p = a | b;
  assign g = a & b;

endmodule

module pgblackblock(input  logic [7:0] pik, gik,
                                       pkj, gkj,
                    output logic [7:0] pij, gij);

  assign pij = pik & pkj;
  assign gij = gik | (pik & gkj);

endmodule

module sumblock(input  logic [15:0] a, b, g,
                output logic [15:0] s);

  assign s = a ^ b ^ g;

endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity pgblock is
  port(a, b: in  STD_LOGIC_VECTOR(14 downto 0);
       p, g: out STD_LOGIC_VECTOR(14 downto 0));
end;

architecture synth of pgblock is
begin
  p <= a or b;
  g <= a and b;
end;


library IEEE; use IEEE.STD_LOGIC_1164.all;

entity pgblackblock is
  port(pik, gik, pkj, gkj:
         in  STD_LOGIC_VECTOR(7 downto 0);
       pij, gij:
         out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of pgblackblock is
begin
  pij <= pik and pkj;
  gij <= gik or (pik and gkj);
end;


library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sumblock is
  port(a, b, g: in  STD_LOGIC_VECTOR(15 downto 0);
       s:       out STD_LOGIC_VECTOR(15 downto 0));
end;

architecture synth of sumblock is
begin
  s <= a xor b xor g;
end;
```

Exercise 5.5

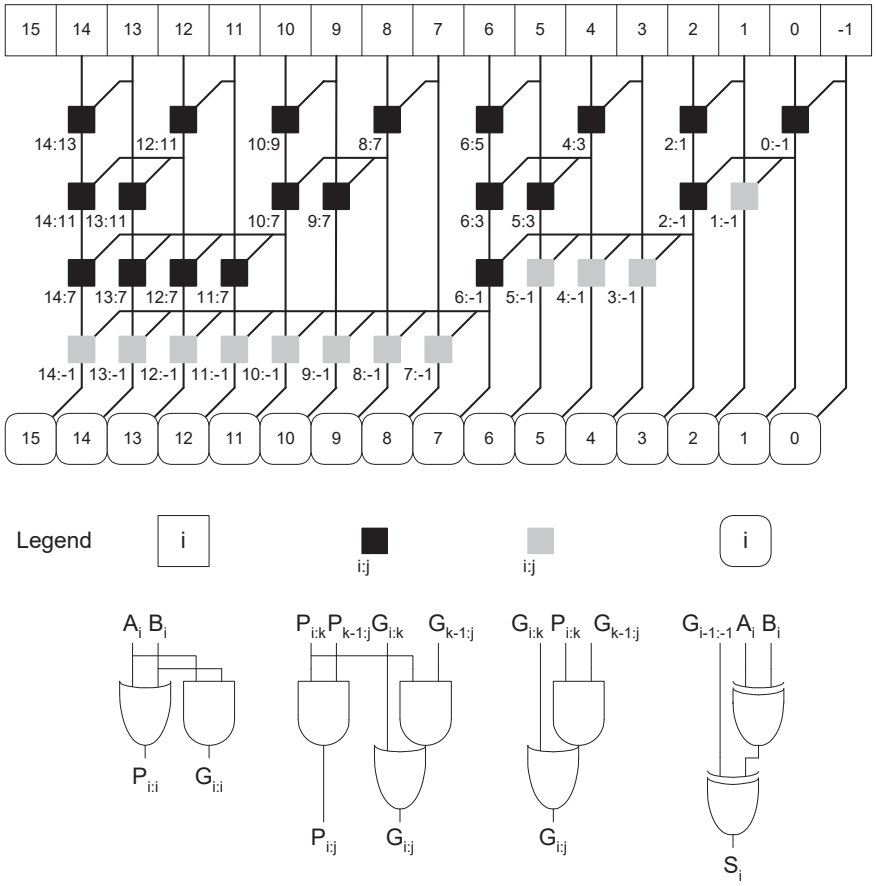


FIGURE 5.2 16-bit prefix adder with “gray cells”

Exercise 5.6

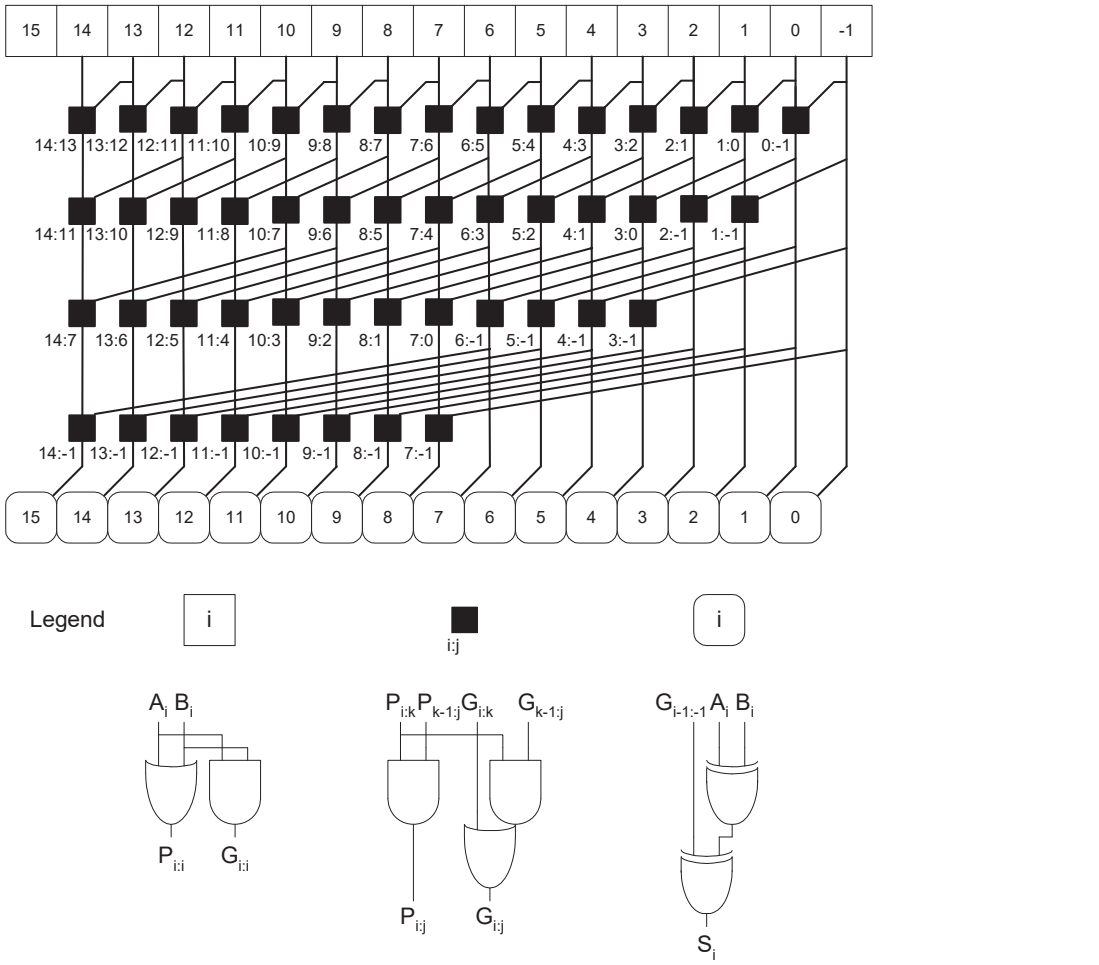


FIGURE 5.3 Schematic of a 16-bit Kogge-Stone adder

Exercise 5.7

(a) We show an 8-bit priority circuit in Figure 5.4. In the figure $X_7 = \overline{A}_7$, $X_{7:6} = \overline{A}_7\overline{A}_6$, $X_{7:5} = \overline{A}_7\overline{A}_6\overline{A}_5$, and so on. The priority encoder's delay is $\log_2 N$ 2-input AND gates followed by a final row of 2-input AND gates. The final stage is an (N/2)-input OR gate. Thus, in general, the delay of an N-input priority encoder is:

$t_{pd\_\text{priority}} = (\log_2 N + 1)t_{pd\_\text{AND2}} + t_{pd\_\text{OR}N/2}$

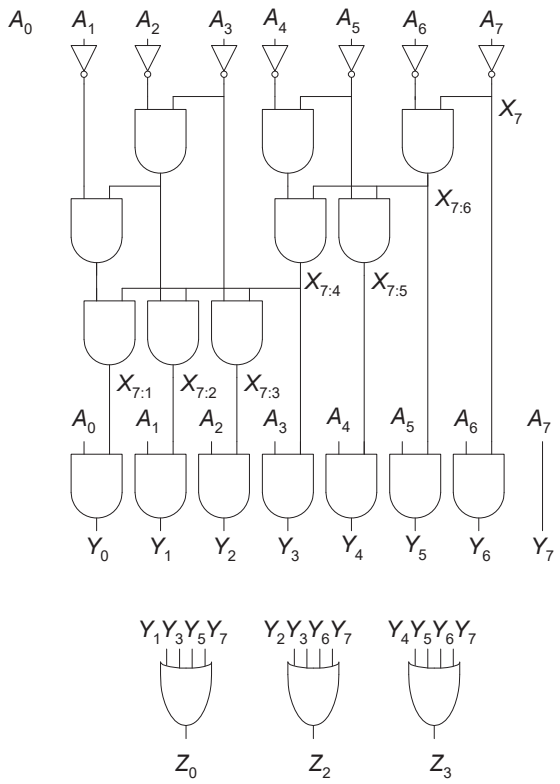


FIGURE 5.4 8-input priority encoder

SystemVerilog

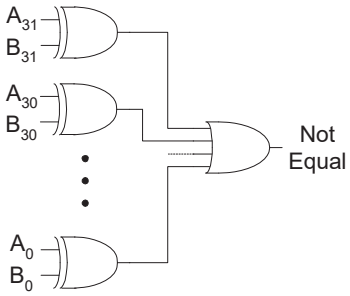
```
module priorityckt(input  logic [7:0] a,
                   output logic [2:0] z);
  logic [7:0] y;
  logic       x7, x76, x75, x74, x73, x72, x71;
  logic       x32, x54, x31;
  logic [7:0] abar;

  // row of inverters
  assign abar = ~a;

  // first row of AND gates
  assign x7 = abar[7];
  assign x76 = abar[6] & x7;
  assign x54 = abar[4] & abar[5];
  assign x32 = abar[2] & abar[3];

  // second row of AND gates
  assign x75 = abar[5] & x76;
  assign x74 = x54 & x76;
  assign x31 = abar[1] & x32;

  // third row of AND gates
  assign x73 = abar[3] & x74;
  assign x72 = x32 & x74;
  assign x71 = x31 & x74;

  // fourth row of AND gates
  assign y = {a[7],       a[6] & x7,  a[5] & x76,
              a[4] & x75, a[3] & x74, a[2] & x73,
              a[1] & x72, a[0] & x71};

  // row of OR gates
  assign z = { |{y[7:4]},
               |{y[7:6], y[3:2]},
               |{y[1], y[3], y[5], y[7]} };
endmodule
```

VHDL

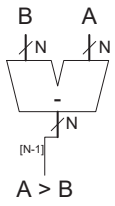
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priorityckt is
  port(a: in  STD_LOGIC_VECTOR(7 downto 0);
       z: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture synth of priorityckt is
  signal y, abar:      STD_LOGIC_VECTOR(7 downto 0);
  signal x7, x76, x75, x74, x73, x72, x71,
         x32, x54, x31: STD_LOGIC;
begin
  -- row of inverters
  abar <= not a;

  -- first row of AND gates
  x7 <= abar(7);
  x76 <= abar(6) and x7;
  x54 <= abar(4) and abar(5);
  x32 <= abar(2) and abar(3);

  -- second row of AND gates
  x75 <= abar(5) and x76;
  x74 <= x54 and x76;
  x31 <= abar(1) and x32;

  -- third row of AND gates
  x73 <= abar(3) and x74;
  x72 <= x32 and x74;
  x71 <= x31 and x74;

  -- fourth row of AND gates
  y <= (a(7) & (a(6) and x7) & (a(5) and x76) &
        (a(4) and x75) & (a(3) and x74) & (a(2) and
x73) &
       (a(1) and x72) & (a(0) and x71));

  -- row of OR gates
  z <= ( (y(7) or y(6) or y(5) or y(4)) &
         (y(7) or y(6) or y(3) or y(2)) &
         (y(1) or y(3) or y(5) or y(7)) );

end;
```

Exercise 5.8

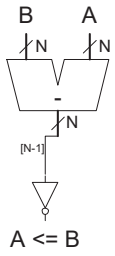
(a)



(b)



(c)



Exercise 5.9

(a) Answers will vary.
3 and 5: $3 - 5 = 0011_2 - 0101_2 = 0011_2 + 1010_2 + 1 = 1110_2$ (= -2_{10}). The sign bit (most significant bit) is 1, so the 4-bit signed comparator of Figure 5.12 correctly computes that 3 is less than 5.

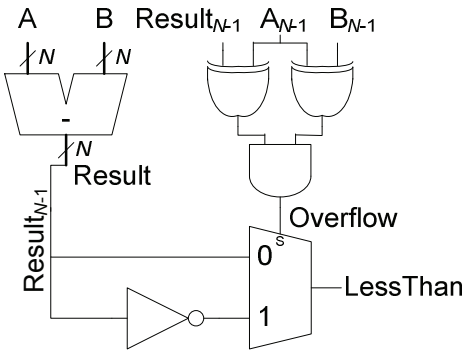
(b) Answers will vary.
-3 and 6: -3 – 6 = 1101 – 0110 = 1101 + 1001 + 1 = 01112 (= -7, but overflow occurred – the result should be -9). The sign bit (most significant bit) is 0, so the 4-bit signed comparator of Figure 5.12 **incorrectly** computes that -3 is **not** less than 6.

(c) In the general, the *N*-bit signed comparator of Figure 5.12 operates incorrectly upon overflow.

Exercise 5.10

If no overflow occurs, connect the sign bit (i.e., most significant bit) of the result to the LessThan output.
If overflow occurs, invert the sign bit of the result and connect it to the *LessThan* output.

Overflow occurs when (1) the two inputs have different signs, AND (2) the sign of the subtraction result has a different sign than the *A* input, as shown in the figure below.



We could also have built this as: *LessThan* = $N \oplus V$, where *N* is $Result_{N-1}$ and *V* is the *Overflow* signal.

Exercise 5.11

SystemVerilog

```
module alu(input  logic [31:0] a,
           input  logic [31:0] b,
           input  logic [1:0]  alucontrol,
           output logic [31:0] result);

  logic [31:0] condinvb, sum;

  assign condinvb = alucontrol[0] ? ~b : b;
  assign sum = a + condinvb + alucontrol[0];

  always_comb
    case (alucontrol)
      2'b00:    result = sum;                    // add
      2'b01:    result = sum;                    // subtract
      2'b10:    result = a & b;                  // and
      2'b11:    result = a | b;                  // or
      default:  result = 32'bx;
    endcase
endmodule
```

Exercise 5.12

SystemVerilog

```
module alu(input  logic [31:0] a,
           input  logic [31:0] b,
           input  logic [1:0]  alucontrol,
           output logic [31:0] result,
           output logic [3:0]  flags);

  logic [31:0] condinvb, sum;
  logic        v, c, n, z;     // flags: overflow, carry out, negative, zero
  logic        cout;           // carry out of adder
  logic        isAddSub;       // true if is an add or subtract operation

  assign flags = {v, c, n, z};
  assign condinvb = alucontrol[0] ? ~b : b;
  assign {cout, sum} = a + condinvb + alucontrol[0];
  assign isAddSub = ~alucontrol[1];

  always_comb
    case (alucontrol)
      2'b00:    result = sum;                    // add
      2'b01:    result = sum;                    // subtract
      2'b10:    result = a & b;                  // and
      2'b11:    result = a | b;                  // or
      default:  result = 32'bx;
    endcase

  assign z = (result == 32'b0);
  assign n = result[31];
```

```
  assign c = cout & isAddSub;
  assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;
endmodule
```

Exercise 5.13

SystemVerilog

```
module alu(input  logic [31:0] a,
           input  logic [31:0] b,
           input  logic [2:0]  alucontrol,
           output logic [31:0] result);

  logic [31:0] condinvb, sum;
  logic        cout;           // carry out of adder

  assign condinvb = alucontrol[0] ? ~b : b;
  assign {cout, sum} = a + condinvb + alucontrol[0];

  always_comb
    case (alucontrol)
      3'b000:   result = sum;                  // add
      3'b001:   result = sum;                  // subtract
      3'b010:   result = a & b;                // and
      3'b011:   result = a | b;                // or
      3'b101:   result = sum[31];              // slt
      default:  result = 32'bx;
    endcase
endmodule
```

Exercise 5.14

SystemVerilog

```
module alu(input  logic [31:0] a,
           input  logic [31:0] b,
           input  logic [2:0]  alucontrol,
           output logic [31:0] result,
           output logic [3:0]  flags);

  logic [31:0] condinvb, sum;
  logic        v, c, n, z;     // flags: overflow, carry out, negative, zero
  logic        cout;           // carry out of adder
  logic        isAddSub;       // true if is an add or subtract operation

  assign flags = {v, c, n, z};
  assign condinvb = alucontrol[0] ? ~b : b;
  assign {cout, sum} = a + condinvb + alucontrol[0];
  assign isAddSub = ~alucontrol[1];

  always_comb
    case (alucontrol)
      3'b000:   result  = sum;                  // add
      3'b001:   result  = sum;                  // subtract
```

```
      3'b010:   result  = a & b;                  // and
      3'b011:   result  = a | b;                  // or
      3'b101:   result  = sum[31] ^ v;            // slt
      default:  result  = 32'bx;
    endcase

  // added for blt and other branches
  assign z = (result == 32'b0);
  assign n = result[31];
  assign c = cout & isAddSub;
  assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;
endmodule
```

Exercise 5.15

SystemVerilog

```
module testbench();
  logic        clk, reset;
  logic [31:0] a, b, result, resultexpected;
  logic [1:0]  alucontrol;
  logic [31:0] vectornum, errors;
  logic [97:0] testvectors[10000:0];

  // instantiate device under test
  alu dut(a, b, alucontrol, result);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemh("example.txt", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #22; reset = 0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {alucontrol, a, b, resultexpected} = testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
    if (~reset) begin // skip during reset
      if (result !== resultexpected) begin  // check result
        $display("Error: inputs: a = %h, b = %h, alucontrol = %h,", a, b,
                 alucontrol);
        $display("  outputs: result = %h (%h expected)", result, resultexpected);
        errors = errors + 1;
      end
      vectornum = vectornum + 1;
```

```
      if (testvectors[vectornum] === 98'bx) begin
        $display("%d tests completed with %d errors",
                 vectornum, errors);
        $stop;
      end
    end
endmodule
```

Testvectors:

```
// alucontrol_a_b_resultexpected
0_00000007_00000005_0000000C  // add
0_AABBCCDD_00000005_AABBCCE2  // add
0_FF123456_FABCDEF1_F9CF1347  // add
1_00000007_00000005_00000002  // sub
1_00000005_00000007_FFFFFFFE  // sub
1_AABBCCDD_11445588_99777755  // sub
2_FFFF0123_ABCDEFAA_ABCD0122  // and
2_AABBCCDD_00000008_00000008  // and
3_FFFF0123_ABCDEFAA_FFFFEFAB  // or
3_AABBCCDD_00000008_AABBCCDD  // or
```

Exercise 5.16

SystemVerilog

```
module testbench();
  logic        clk, reset;
  logic [31:0] a, b, result, resultexpected;
  logic [1:0]  alucontrol;
  logic [3:0]  flags, flagsexpected;
  logic [31:0] vectornum, errors;
  logic [101:0] testvectors[10000:0];

  // instantiate device under test
  alu dut(a, b, alucontrol, result, flags);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemh("example.txt", testvectors);

      vectornum = 0; errors = 0;
      reset = 1; #22; reset = 0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
```

```
      #1; {alucontrol, a, b, resultexpected, flagsexpected} = testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
    if (~reset) begin // skip during reset
      if ((result !== resultexpected) | (flags !== flagsexpected)) begin  // check
result
        $display("Error: inputs: a = %h, b = %h, alucontrol = %h,", a, b,
                 alucontrol);
        $display("  outputs: result = %h (%h expected), flags = %h (%h expected)",
                 result, resultexpected, flags, flagsexpected);
        errors = errors + 1;
      end
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 102'bx) begin
        $display("%d tests completed with %d errors",
                 vectornum, errors);
        $stop;
      end
    end
endmodule
```

Testvectors:

```
// alucontrol_a_b_resultexpected_flags
0_00000007_00000005_0000000C_0  // add
0_AABBCCDD_00000005_AABBCCE2_2  // add
0_FF123456_FABCDEF1_F9CF1347_6  // add
0_7FFFFFFF_00000002_80000001_A  // add
0_80000000_81234567_01234567_C  // add
1_80000000_81234567_FEDCBA99_2  // sub
1_7FFFFFFF_FFFFFFFE_80000001_A  // sub
1_00000007_00000005_00000002_4  // sub
1_00000005_00000007_FFFFFFFE_2  // sub
1_AABBCCDD_11445588_99777755_6  // sub
1_7FFFFFFF_FFFFFFFF_80000000_A  // sub
2_FFFF0123_ABCDEFAA_ABCD0122_2  // and
2_AABBCCDD_00000008_00000008_0  // and
3_FFFF0123_ABCDEFAA_FFFFEFAB_2  // or
3_AABBCCDD_00000008_AABBCCDD_2  // or
```

Exercise 5.17

SystemVerilog

```
module testbench();
  logic        clk, reset;
  logic [31:0] a, b, result, resultexpected;
  logic [2:0]  alucontrol;
  logic [31:0] vectornum, errors;
  logic [98:0] testvectors[10000:0];

  // instantiate device under test
  alu dut(a, b, alucontrol, result);

  // generate clock
```

```
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemh("example.txt", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #22; reset = 0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {alucontrol, a, b, resultexpected} = testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
    if (~reset) begin // skip during reset
      if (result !== resultexpected) begin  // check result
        $display("Error: inputs: a = %h, b = %h, alucontrol = %h,", a, b,
                 alucontrol);
        $display("  outputs: result = %h (%h expected)", result, resultexpected);
        errors = errors + 1;
      end
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 99'bx) begin
        $display("%d tests completed with %d errors",
                 vectornum, errors);
        $stop;
      end
    end
endmodule
```

Testvectors:

```
// alucontrol_a_b_resultexpected_flags
0_00000007_00000005_0000000C  // add
0_AABBCCDD_00000005_AABBCCE2  // add
0_FF123456_FABCDEF1_F9CF1347  // add
0_7FFFFFFF_00000002_80000001  // add
0_80000000_81234567_01234567  // add
1_80000000_81234567_FEDCBA99  // sub
1_7FFFFFFF_FFFFFFFE_80000001  // sub
1_00000007_00000005_00000002  // sub
1_00000005_00000007_FFFFFFFE  // sub
1_AABBCCDD_11445588_99777755  // sub
1_7FFFFFFF_FFFFFFFF_80000000  // sub
2_FFFF0123_ABCDEFAA_ABCD0122  // and
2_AABBCCDD_00000008_00000008  // and
3_FFFF0123_ABCDEFAA_FFFFEFAB  // or
3_AABBCCDD_00000008_AABBCCDD  // or
5_00000007_00000005_00000000  // slt
5_80000000_81234567_00000001  // slt
5_80000000_00000001_00000000  // slt - wrong result due to overflow
```

```
5_7FFFFFFF_FFFFFFFE_00000001  // slt - wrong result due to overflow
5_0000FFF1_000FFFF1_00000001  // slt
```

Exercise 5.18

SystemVerilog

```
module testbench();
  logic        clk, reset;
  logic [31:0] a, b, result, resultexpected;
  logic [2:0]  alucontrol;
  logic [3:0]  flags, flagsexpected;
  logic [31:0] vectornum, errors;
  logic [102:0] testvectors[10000:0];

  // instantiate device under test
  alu dut(a, b, alucontrol, result, flags);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemh("example.txt", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #22; reset = 0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {alucontrol, a, b, resultexpected, flagsexpected} = testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
    if (~reset) begin // skip during reset
      if ((result !== resultexpected) | (flags !== flagsexpected)) begin
        // check result
        $display("Error: inputs: a = %h, b = %h, alucontrol = %h,", a, b,
                 alucontrol);
        $display("  outputs: result = %h (%h expected), flags = %h (%h expected)",
                  result, resultexpected, flags, flagsexpected);
        errors = errors + 1;
      end
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 103'bx) begin
        $display("%d tests completed with %d errors",
                 vectornum, errors);
        $stop;
      end
    end
endmodule
```

Testvectors:

```
// alucontrol_a_b_resultexpected_flags
0_00000007_00000005_0000000C_0  // add
0_AABBCCDD_00000005_AABBCCE2_2  // add
0_FF123456_FABCDEF1_F9CF1347_6  // add
0_7FFFFFFF_00000002_80000001_A  // add
0_80000000_81234567_01234567_C  // add
1_80000000_81234567_FEDCBA99_2  // sub
1_7FFFFFFF_FFFFFFFE_80000001_A  // sub
1_00000007_00000005_00000002_4  // sub
1_00000005_00000007_FFFFFFFE_2  // sub
1_AABBCCDD_11445588_99777755_6  // sub
1_7FFFFFFF_FFFFFFFF_80000000_A  // sub
2_FFFF0123_ABCDEFAA_ABCD0122_2  // and
2_AABBCCDD_00000008_00000008_0  // and
3_FFFF0123_ABCDEFAA_FFFFEFAB_2  // or
3_AABBCCDD_00000008_AABBCCDD_2  // or
5_00000007_00000005_00000000_5  // slt
5_80000000_81234567_00000001_0  // slt
5_80000000_00000001_00000001_C  // slt - corrected result vs. Figure 5.18a
5_7FFFFFFF_FFFFFFFE_00000000_9  // slt - corrected result vs. Figure 5.18a
5_0000FFF1_000FFFF1_00000001_0  // slt
```

Exercise 5.19

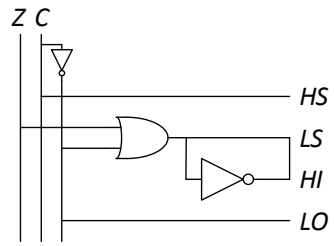
(a) $HS = C$

$LS = Z + \bar{C}$

$HI = \bar{Z}C = \overline{LS}$

$LO = \bar{C} = \overline{HS}$

(b)



Exercise 5.20

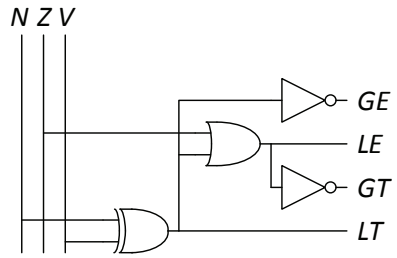
(a) $GE = \overline{N \oplus V}$

$LE = Z + (N \oplus V)$

$GT = \overline{LE} = \overline{Z}(\overline{N \oplus V})$

$LT = \overline{GE} = N \oplus V$

(b)



Exercise 5.21

A 2-bit left shifter creates the output by appending two zeros to the least significant bits of the input and dropping the two most significant bits.

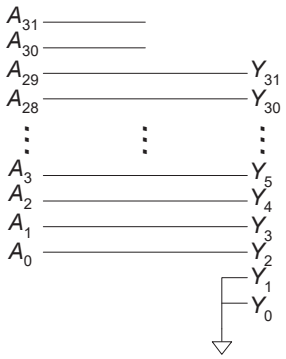


FIGURE 5.6 2-bit left shifter, 32-bit input and output

2-bit Left Shifter

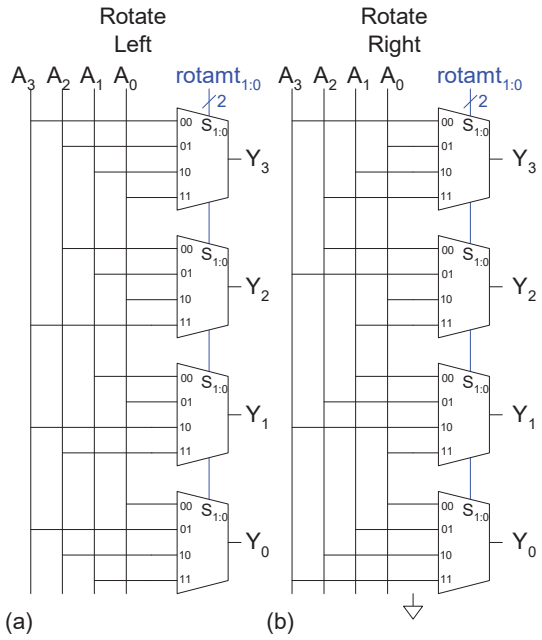
SystemVerilog

```
module leftshift2_32(input  logic [31:0] a,
                     output logic [31:0] y);
  assign y = {a[29:0], 2'b0};

endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity leftshift2_32 is
  port(a: in  STD_LOGIC_VECTOR(31 downto 0);
       y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of leftshift2_32 is
begin
  y <= a(29 downto 0) & "00";
end;
```

Exercise 5.22


Rotate
Left
A3 A2 A1 A0
rotamt1:0
2
00
01
10
11
S1:0
Y3
Y2
Y1
Y0
(a)
Rotate
Right
A3 A2 A1 A0
rotamt1:0
2
(b)

4-bit Left and Right Rotator

SystemVerilog

```
module   ex5_14(a,   right_rotated,   left_rotated,
shamt);
    input  logic [3:0] a;
    output logic [3:0] right_rotated;
    output logic [3:0] left_rotated;
    input  logic [1:0] shamt;

 // right rotated
 always_comb
   case(shamt)
     2'b00: right_rotated = a;
     2'b01: right_rotated =
       {a[0], a[3], a[2], a[1]};
     2'b10: right_rotated =
       {a[1], a[0], a[3], a[2]};
     2'b11: right_rotated =
       {a[2], a[1], a[0], a[3]};
     default: right_rotated = 4'bxxxx;
   endcase

 // left rotated
 always_comb
   case(shamt)
     2'b00: left_rotated = a;
     2'b01: left_rotated =
       {a[2], a[1], a[0], a[3]};
     2'b10: left_rotated =
       {a[1], a[0], a[3], a[2]};
     2'b11: left_rotated =
       {a[0], a[3], a[2], a[1]};
    default: left_rotated = 4'bxxxx;
   endcase

endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ex5_14 is
  port(a:     in STD_LOGIC_VECTOR(3 downto 0);
       right_rotated, left_rotated: out
                 STD_LOGIC_VECTOR(3 downto 0);
       shamt: in STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of ex5_14 is
begin

-- right-rotated
  process(all) begin
    case shamt is
      when "00" => right_rotated <= a;
      when "01" => right_rotated <=
                   (a(0), a(3), a(2), a(1));
      when "10" => right_rotated <=
                   (a(1), a(0), a(3), a(2));
      when "11" => right_rotated <=
                   (a(2), a(1), a(0), a(3));
      when others => right_rotated <= "XXXX";
    end case;
  end process;

-- left-rotated
  process(all) begin
    case shamt is
      when "00" => left_rotated <= a;
      when "01" => left_rotated <=
                   (a(2), a(1), a(0), a(3));
      when "10" => left_rotated <=
                   (a(1), a(0), a(3), a(2));
      when "11" => left_rotated <=
                   (a(0), a(3), a(2), a(1));
      when others => left_rotated <= "XXXX";
    end case;
  end process;
end;
```

Exercise 5.23

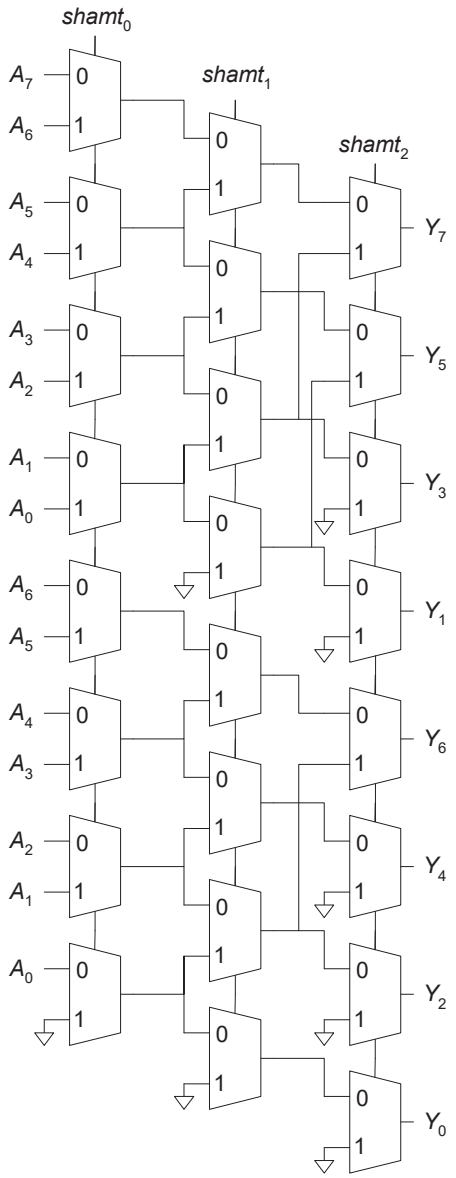


FIGURE 5.7 8-bit left shifter using 24 2:1 multiplexers

Exercise 5.24

Any N-bit shifter can be built by using $\log_2 N$ columns of 2-bit shifters. The first column of multiplexers shifts or rotates 0 to 1 bit, the second column shifts or rotates 0 to 3 bits, the following 0 to 7 bits, etc. until the final column shifts or rotates 0 to N-1 bits. The second column of multiplexers takes its inputs from the first column of multiplexers, the third column takes its input from the second column, and so forth. The 1-bit select input of each column is a single bit of the *shamt* (shift amount) control signal, with the least significant bit for the left-most column and the most significant bit for the right-most column.

Exercise 5.25

(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) $B = A$, $C = A$, $k = N - shamt$

Exercise 5.26

$t_{pd\_\mathrm{MULT4}} = t_{\mathrm{AND}} + 8t_{FA}$

For $N = 1$, the delay is t_{AND}. For $N > 1$, an $N \times N$ multiplier has N-bit operands, N partial products, and N-1 stages of 1-bit adders. The delay is through the AND gate, then through all N adders in the first stage, and finally through 2 adder delays for each of the remaining stages. So the propagation is:

$$t_{pd\_\mathrm{MULT}N} = t_{\mathrm{AND}} + [N + 2(N\text{-}1)]t_{FA}$$

Exercise 5.27

$t_{pd\_\mathrm{DIV4}} = 4\,(4t_{FA} + t_{\mathrm{MUX}}) = 16t_{FA} + 4t_{\mathrm{MUX}}$

$t_{pd\_\mathrm{DIV}N} = N^2 t_{FA} + Nt_{\mathrm{MUX}}$

Exercise 5.28

Recall that a two's complement number has the same weights for the least significant N-1 bits, regardless of the sign. The sign bit has a weight of -2^{N-1}. Thus, the product of two N-bit complement numbers, y and x is:

$$P = \left(-y_{N-1}2^{N-1} + \sum_{j=0}^{N-2} y_j 2^j\right)\left(-x_{N-1}2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i\right)$$

Thus,

$$\sum_{i=0}^{N-2}\sum_{j=0}^{N-2} x_i y_j 2^{i+j} + x_{N-1}y_{N-1}2^{2N-2} - \sum_{i=0}^{N-2} x_i y_{N-1} 2^{i+N-1} - \sum_{j=0}^{N-2} x_{N-1} y_j 2^{j+N-1}$$

The two negative partial products are formed by taking the two's complement (inverting the bits and adding 1). Figure 5.8 shows a 4 x 4 multiplier. Figure 5.8 (b) shows the partial products using the above equation. Figure 5.8 (c) shows a simplified version, pushing through the 1's. This is known as a *modified Baugh-Wooley multiplier*. It can be built using a hierarchy of adders.

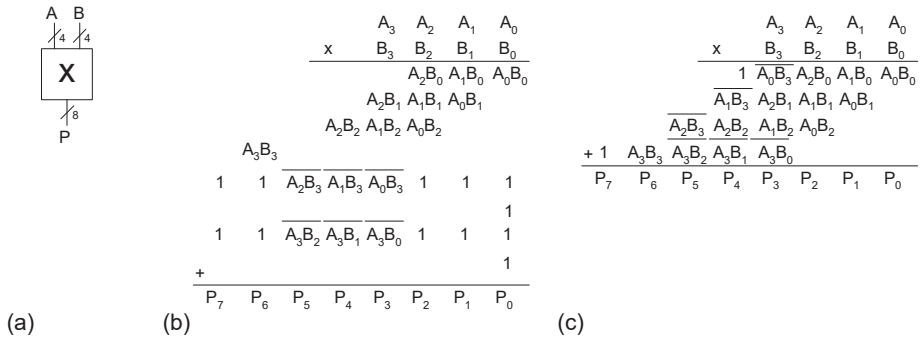


FIGURE 5.8 Multiplier: (a) symbol, (b) function, (c) simplified function

Exercise 5.29

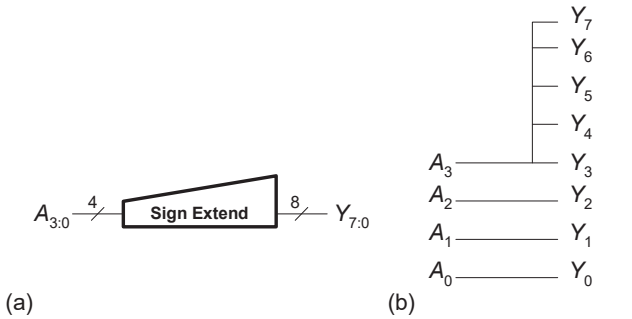


FIGURE 5.9 Sign extension unit (a) symbol, (b) underlying hardware

SystemVerilog

```
module signext4_8(input  logic [3:0] a,
                  output logic [7:0] y);

  assign y = { {4{a[3]}}, a};

endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity signext4_8 is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of signext4_8 is
begin
```

Exercise 5.30

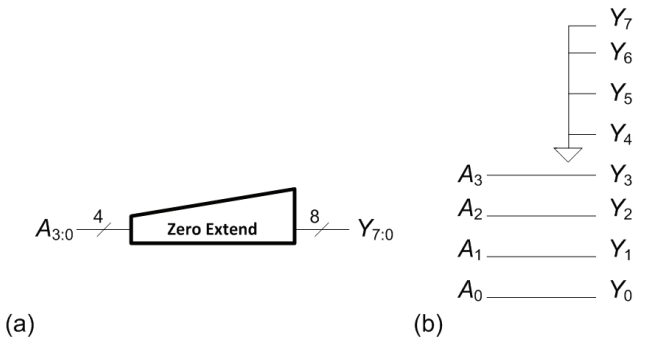


FIGURE 5.10 Zero extension unit (a) symbol, (b) underlying hardware

SystemVerilog

```
module zeroext4_8(input  logic [3:0] a,
                  output logic [7:0] y);

  assign y = {4'b0, a};

endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity zeroext4_8 is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of zeroext4_8 is
begin
  y <= "0000" & a(3 downto 0);
end;
```

Exercise 5.31

```
          100.110
1100 | 111001.000
     -1100
      001001 0
     -   110 0
          11 00
     -    11 00
               0
```

Exercise 5.32

(a) $\left[0, \left(2^{12} - 1 + \frac{2^{12} - 1}{2^{12}}\right)\right]$

(b) $\left[-\left(2^{11} - 1 + \frac{2^{12} - 1}{2^{12}}\right), \left(2^{11} - 1 + \frac{2^{12} - 1}{2^{12}}\right)\right]$

(c) $\left[-\left(2^{11} + \frac{2^{12} - 1}{2^{12}}\right), \left(2^{11} - 1 + \frac{2^{12} - 1}{2^{12}}\right)\right]$

Exercise 5.33

(a) 1000 1101 . 1001 0000 = 0x8D90
(b) 0010 1010 . 0101 0000 = 0x2A50
(c) 1001 0001 . 0010 1000 = 0x9128

Exercise 5.34

(a) 111110.100000 = 0xFA0
(b) 010000.010000 = 0x410
(c) 101000.000101 = 0xA05

Exercise 5.35

(a) 1111 0010 . 0111 0000 = 0xF270
(b) 0010 1010 . 0101 0000 = 0x2A50
(c) 1110 1110 . 1101 1000 = 0xEED8

Exercise 5.36

(a) 100001.100000 = 0x860
(b) 010000.010000 = 0x410
(c) 110111.111011 = 0xDFB

Exercise 5.37

(a) $-1101.1001 = -1.1011001 \times 2^3$
Thus, the biased exponent = $127 + 3 = 130 = 1000\ 0010_2$
In IEEE 754 single-precision floating-point format:
1 1000 0010 101 1001 0000 0000 0000 0000 = **0xC1590000**

(b) $101010.0101 = 1.010100101 \times 2^5$
Thus, the biased exponent = $127 + 5 = 132 = 1000\ 0100_2$
In IEEE 754 single-precision floating-point format:
0 1000 0100 010 1001 0100 0000 0000 0000 = **0x42294000**

(c) $-10001.00101 = -1.000100101 \times 2^4$
Thus, the biased exponent = $127 + 4 = 131 = 1000\ 0011_2$
In IEEE 754 single-precision floating-point format:
1 1000 0011 000 1001 0100 0000 0000 0000 = **0xC1894000**

Exercise 5.38

(a) -11110.1 = -1.111101×2^4
Thus, the biased exponent = 127 + 4 = 131 = $1000\ 0011_2$
In IEEE 754 single-precision floating-point format:
1 1000 0011 111 1010 0000 0000 0000 0000 = **0xC1F40000**

(b) 10000.01 = 1.000001×2^4
Thus, the biased exponent = 127 + 4 = 131 = $1000\ 0011_2$
In IEEE 754 single-precision floating-point format:
0 1000 0011 000 0010 0100 0000 0000 0000 = **0x41820000**

(c) -1000.000101 = -1.000000101×2^3
Thus, the biased exponent = 127 + 3 = 130 = $1000\ 0010_2$
In IEEE 754 single-precision floating-point format:
1 1000 0010 000 0001 0100 0000 0000 0000 = **0xC1014000**

Exercise 5.39

(a) 5.5
(b) $-0000.0001_2 = -0.0625$
(c) -8

Exercise 5.40

(a) 29.65625
(b) -25.1875
(c) -23.875

Exercise 5.41

When adding two floating point numbers, the number with the smaller exponent is shifted to preserve the most significant bits. For example, suppose we were adding the two floating point numbers 1.0×2^0 and 1.0×2^{-27}. We make the two exponents equal by shifting the second number right by 27 bits. Because the mantissa is limited to 24 bits, the second number ($1.000\ 0000\ 0000\ 0000\ 0000 \times 2^{-27}$) becomes $0.000\ 0000\ 0000\ 0000\ 0000 \times 2^0$, because the 1 is shifted off to the right. If we had shifted the number with the larger exponent (1.0×2^0) to the left, we would have shifted off the more significant bits (on the order of 2^0 instead of on the order of 2^{-27}).

Exercise 5.42

(a) C0123456
(b) D1E072C3
(c) 5F19659A

Exercise 5.43

(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 × 2^{64}
0x3FF80000 = 0 0111 1111 111 1000 0000 0000 0000 0000
= 1.111 1 × 2^0
0xDFDE4000 = 1 1011 1111 101 1110 0100 0000 0000 0000 0000
= - 1.101 1110 01 × 2^{64}

Thus, $(1.011\ 1110\ 01 \times 2^{64} + 1.111\ 1 \times 2^{0}) = 1.011\ 1110\ 01 \times 2^{64}$

And, $(1.011\ 1110\ 01 \times 2^{64} + 1.111\ 1 \times 2^{0}) - 1.101\ 1110\ 01 \times 2^{64} =$
$- 0.01 \times 2^{64} = -1.0 \times 2^{64}$
$= 1\ 1011\ 1101\ 000\ 0000\ 0000\ 0000\ 0000\ 0000$
= 0xDE800000

This is counterintuitive because the second number (0x3FF80000) does not affect the result because its order of magnitude is less than 2^{23} of the other numbers. This second number's significant bits are shifted off when the exponents are made equal.

Exercise 5.44

We only need to change step 5.

1. Extract exponent and fraction bits.
2. Prepend leading 1 to form the mantissa.
3. Compare exponents.
4. Shift smaller mantissa if necessary.
5. If one number is negative: Subtract it from the other number. If the result is negative, take the absolute value of the result and make the sign bit 1.

   If both numbers are negative: Add the numbers and make the sign bit 1.
   If both numbers are positive: Add the numbers and make the sign bit 0.
6. Normalize mantissa and adjust exponent if necessary.
7. Round result
8. Assemble exponent and fraction back into floating-point number

Exercise 5.45

(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) $\infty\pm$ and NaN are given special representations because they are often used in calculations and in representing results. These values also give useful information to the user as return values, instead of returning garbage upon overflow, underflow, or divide by zero.

Exercise 5.46

(a) $245 = 11110101 = 1.1110101 \times 2^7$
$= 0\ 1000\ 0110\ 111\ 0101\ 0000\ 0000\ 0000\ 0000$
$=$ 0x43750000
$0.0625 = 0.0001 = 1.0 \times 2^{-4}$
$= 0\ 0111\ 1011\ 000\ 0000\ 0000\ 0000\ 0000\ 0000$
$=$ 0x3D800000

(b) 0x43750000 is greater than 0x3D800000, so magnitude comparison gives the correct result.

(c)
$1.1110101 \times 2^7 = 0\ 0000\ 0111\ 111\ 0101\ 0000\ 0000\ 0000\ 0000$
$=$ 0x03F50000
$1.0 \times 2^{-4} = 0\ 1111\ 1100\ 000\ 0000\ 0000\ 0000\ 0000\ 0000$
$=$ 0x7E000000

(d) No, integer comparison no longer works. 7E000000 > 03F50000 (indicating that 1.0×2^{-4} is greater than 1.1110101×2^7, which is incorrect.)

(e) It is convenient for integer comparison to work with floating-point numbers because then the computer can compare numbers without needing to ex-tract the mantissa, exponent, and sign.

Exercise 5.47

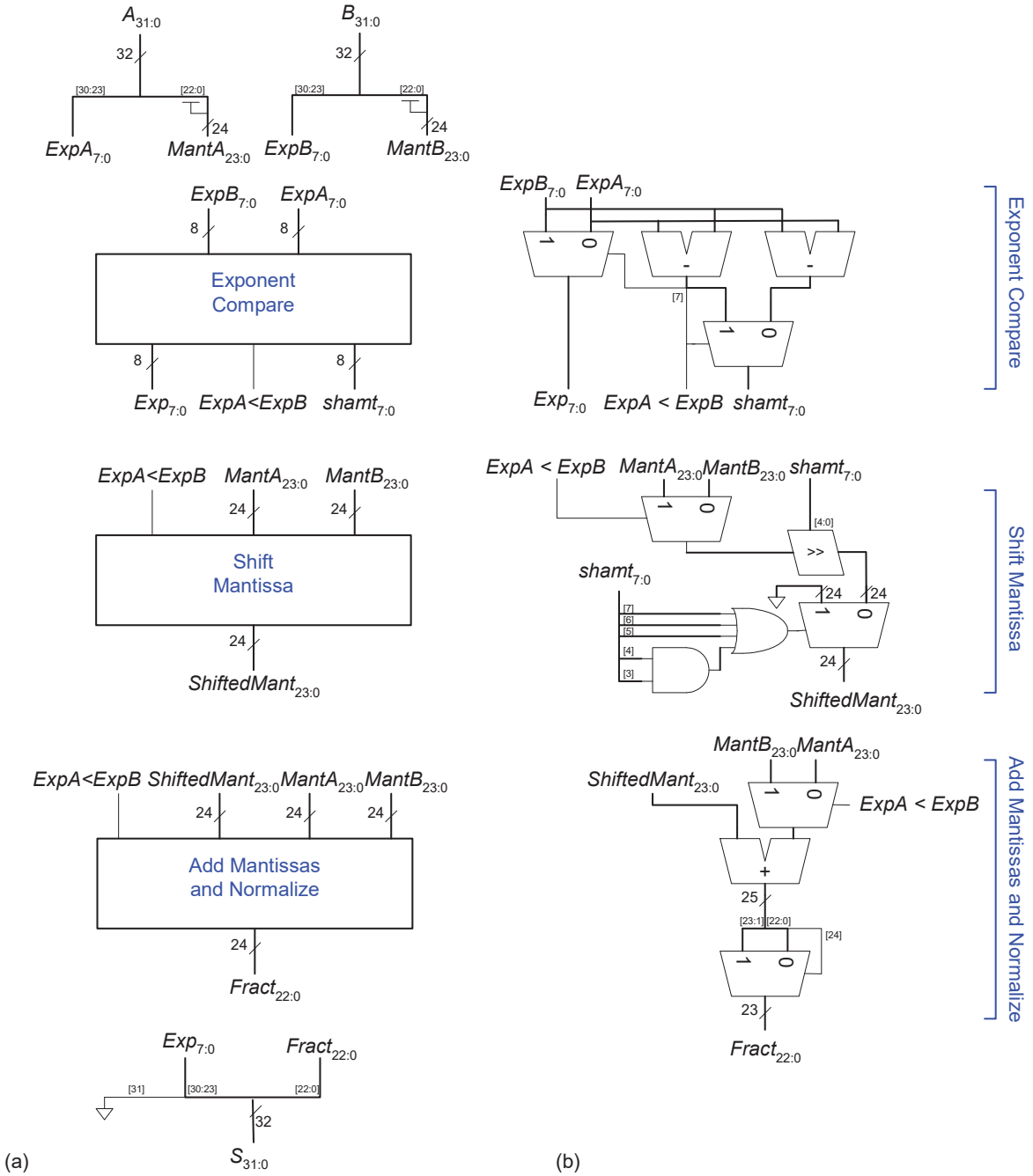


FIGURE 5.11 Floating-point adder hardware: (a) block diagram, (b) underlying hardware

SystemVerilog

```
module fpadd(input  logic [31:0] a, b,
             output logic [31:0] s);

  logic [7:0]  expa, expb, exp_pre, exp, shamt;
  logic        alessb;
  logic [23:0] manta, mantb, shmant;
  logic [22:0] fract;

  assign {expa, manta} = {a[30:23], 1'b1, a[22:0]};
  assign {expb, mantb} = {b[30:23], 1'b1, b[22:0]};
  assign s             = {1'b0, exp, fract};

  expcomp   expcomp1(expa, expb, alessb, exp_pre,
                     shamt);
  shiftmant shiftmant1(alessb, manta, mantb,
                       shamt, shmant);
  addmant   addmant1(alessb, manta, mantb,
                     shmant, exp_pre, fract, exp);

endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity fpadd is
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       s:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of fpadd is
  component expcomp
   port(expa, expb: in  STD_LOGIC_VECTOR(7 downto 0);
        alessb:     inout STD_LOGIC;
       exp,shamt:  out STD_LOGIC_VECTOR(7 downto 0));
  end component;

  component shiftmant
    port(alessb:  in  STD_LOGIC;
         manta:   in  STD_LOGIC_VECTOR(23 downto 0);
         mantb:   in  STD_LOGIC_VECTOR(23 downto 0);
         shamt:   in  STD_LOGIC_VECTOR(7 downto 0);
        shmant:  out STD_LOGIC_VECTOR(23 downto 0));
  end component;

  component addmant
    port(alessb:  in  STD_LOGIC;
         manta:   in  STD_LOGIC_VECTOR(23 downto 0);
         mantb:   in  STD_LOGIC_VECTOR(23 downto 0);
         shmant:  in  STD_LOGIC_VECTOR(23 downto 0);
         exp_pre: in  STD_LOGIC_VECTOR(7 downto 0);
         fract:   out STD_LOGIC_VECTOR(22 downto 0);
         exp:     out STD_LOGIC_VECTOR(7 downto 0));
  end component;

  signal expa, expb: STD_LOGIC_VECTOR(7 downto 0);
  signal exp_pre, exp: STD_LOGIC_VECTOR(7 downto 0);
  signal shamt: STD_LOGIC_VECTOR(7 downto 0);
  signal alessb: STD_LOGIC;
  signal manta: STD_LOGIC_VECTOR(23 downto 0);
  signal mantb: STD_LOGIC_VECTOR(23 downto 0);
  signal shmant: STD_LOGIC_VECTOR(23 downto 0);
  signal fract: STD_LOGIC_VECTOR(22 downto 0);
begin

  expa  <= a(30 downto 23);
  manta <= '1' & a(22 downto 0);
  expb  <= b(30 downto 23);
  mantb <= '1' & b(22 downto 0);

  s     <= '0' & exp & fract;

  expcomp1:  expcomp
    port map(expa, expb, alessb, exp_pre, shamt);
  shiftmant1: shiftmant
    port map(alessb, manta, mantb, shamt, shmant);
  addmant1: addmant
    port map(alessb, manta, mantb, shmant,
             exp_pre, fract, exp);

end;
```

(continued from previous page)

SystemVerilog

```
module expcomp(input  logic [7:0] expa, expb,
               output logic       alessb,
               output logic [7:0] exp, shamt);
  logic [7:0] aminusb, bminusa;

  assign aminusb = expa - expb;
  assign bminusa = expb - expa;
  assign alessb  = aminusb[7];

  always_comb
    if (alessb) begin
      exp = expb;
      shamt = bminusa;
    end
    else begin
      exp = expa;
      shamt = aminusb;
    end
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity expcomp is
  port(expa, expb: in  STD_LOGIC_VECTOR(7 downto 0);
       alessb:     inout STD_LOGIC;
      exp,shamt:  out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of expcomp is
  signal aminusb: STD_LOGIC_VECTOR(7 downto 0);
  signal bminusa: STD_LOGIC_VECTOR(7 downto 0);
begin
  aminusb <= expa - expb;
  bminusa <= expb - expa;
  alessb <= aminusb(7);

  exp <= expb when alessb = '1' else expa;
  shamt <= bminusa when alessb = '1' else aminusb;

end;
```

(continued on next page)

(continued from previous page)

SystemVerilog

```
module shiftmant(input  logic alessb,
                 input  logic [23:0] manta, mantb,
                 input  logic [7:0] shamt,
                 output logic [23:0] shmant);

  logic [23:0] shiftedval;

  assign shiftedval = alessb ?
    (manta >> shamt) : (mantb >> shamt);

  always_comb
    if (shamt[7] | shamt[6] | shamt[5] |
        (shamt[4] & shamt[3]))
        shmant = 24'b0;
    else
        shmant = shiftedval;

endmodule




module addmant(input  logic        alessb,
               input  logic [23:0] manta,
                                   mantb, shmant,
               input  logic [7:0]  exp_pre,
               output logic [22:0] fract,
               output logic [7:0]  exp);

  logic [24:0] addresult;
  logic [23:0] addval;

  assign addval    = alessb ? mantb : manta;
  assign addresult = shmant + addval;
  assign fract     = addresult[24] ?
                     addresult[23:1] :
                     addresult[22:0];

  assign exp       = addresult[24] ?
                     (exp_pre + 1) :
                     exp_pre;

endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity shiftmant is
    port(alessb:  in  STD_LOGIC;
         manta:   in  STD_LOGIC_VECTOR(23 downto 0);
         mantb:   in  STD_LOGIC_VECTOR(23 downto 0);
         shamt:   in  STD_LOGIC_VECTOR(7 downto 0);
        shmant:  out STD_LOGIC_VECTOR(23 downto 0));
end;

architecture synth of shiftmant is
  signal shiftedval: unsigned (23 downto 0);
  signal shiftamt_vector: STD_LOGIC_VECTOR (7 downto
0);
begin

  shiftedval <= SHIFT_RIGHT( unsigned(manta), to_in-
teger(unsigned(shamt))) when alessb = '1'
           else SHIFT_RIGHT( unsigned(mantb), to_in-
teger(unsigned(shamt)));

  shmant <= X"000000" when (shamt > 22)
         else STD_LOGIC_VECTOR(shiftedval);
end;


library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity addmant is
  port(alessb: in  STD_LOGIC;
       manta:  in  STD_LOGIC_VECTOR(23 downto 0);
       mantb:  in  STD_LOGIC_VECTOR(23 downto 0);
       shmant:  in  STD_LOGIC_VECTOR(23 downto 0);
       exp_pre:  in  STD_LOGIC_VECTOR(7 downto 0);
       fract:  out STD_LOGIC_VECTOR(22 downto 0);
       exp:  out  STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of addmant is
  signal addresult: STD_LOGIC_VECTOR(24 downto 0);
  signal addval:    STD_LOGIC_VECTOR(23 downto 0);
begin
  addval <= mantb when alessb = '1' else manta;
  addresult <= ('0'&shmant) + addval;
  fract <= addresult(23 downto 1)
           when addresult(24) = '1'
           else addresult(22 downto 0);
  exp   <= (exp_pre + 1)
           when addresult(24) = '1'
           else exp_pre;

end;
```

Exercise 5.48

(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

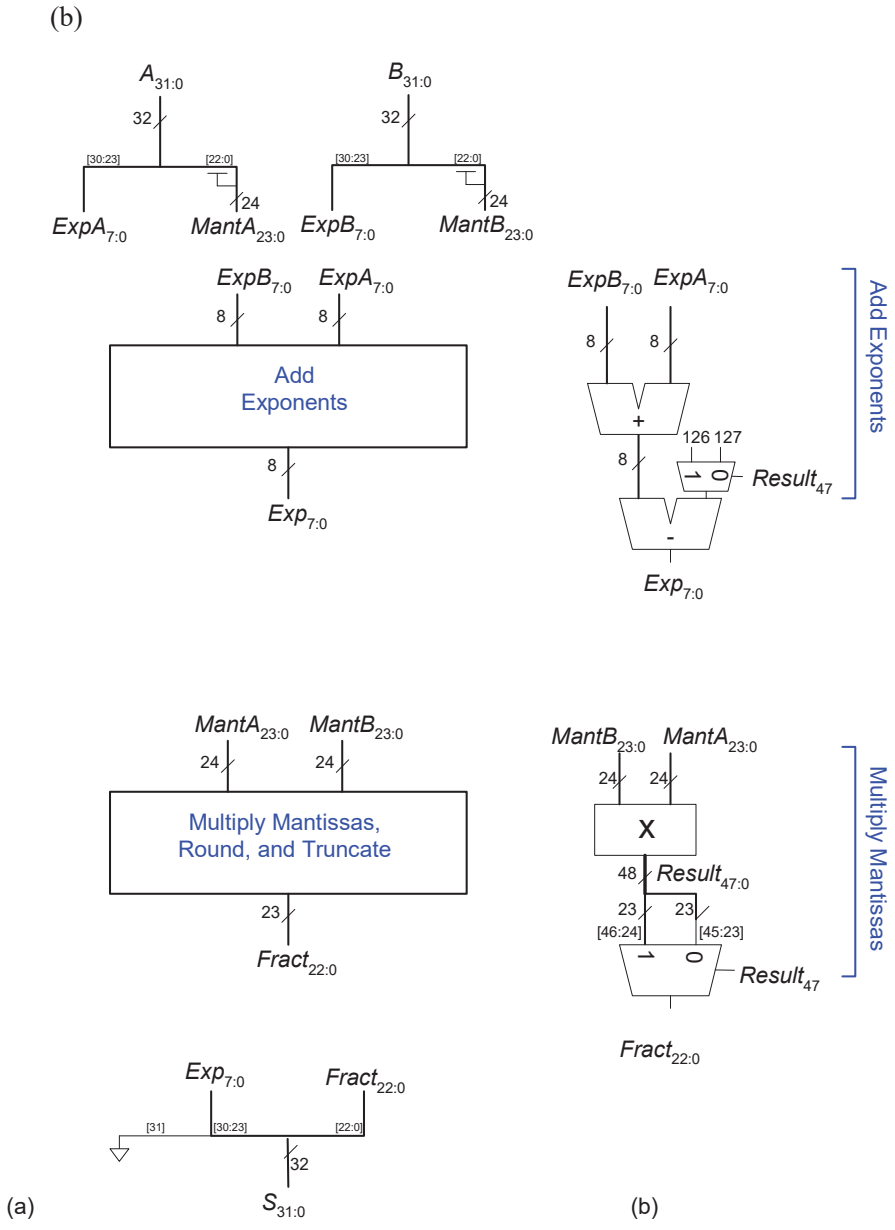


FIGURE 5.12 Floating-point multiplier block diagram

(c)

SystemVerilog

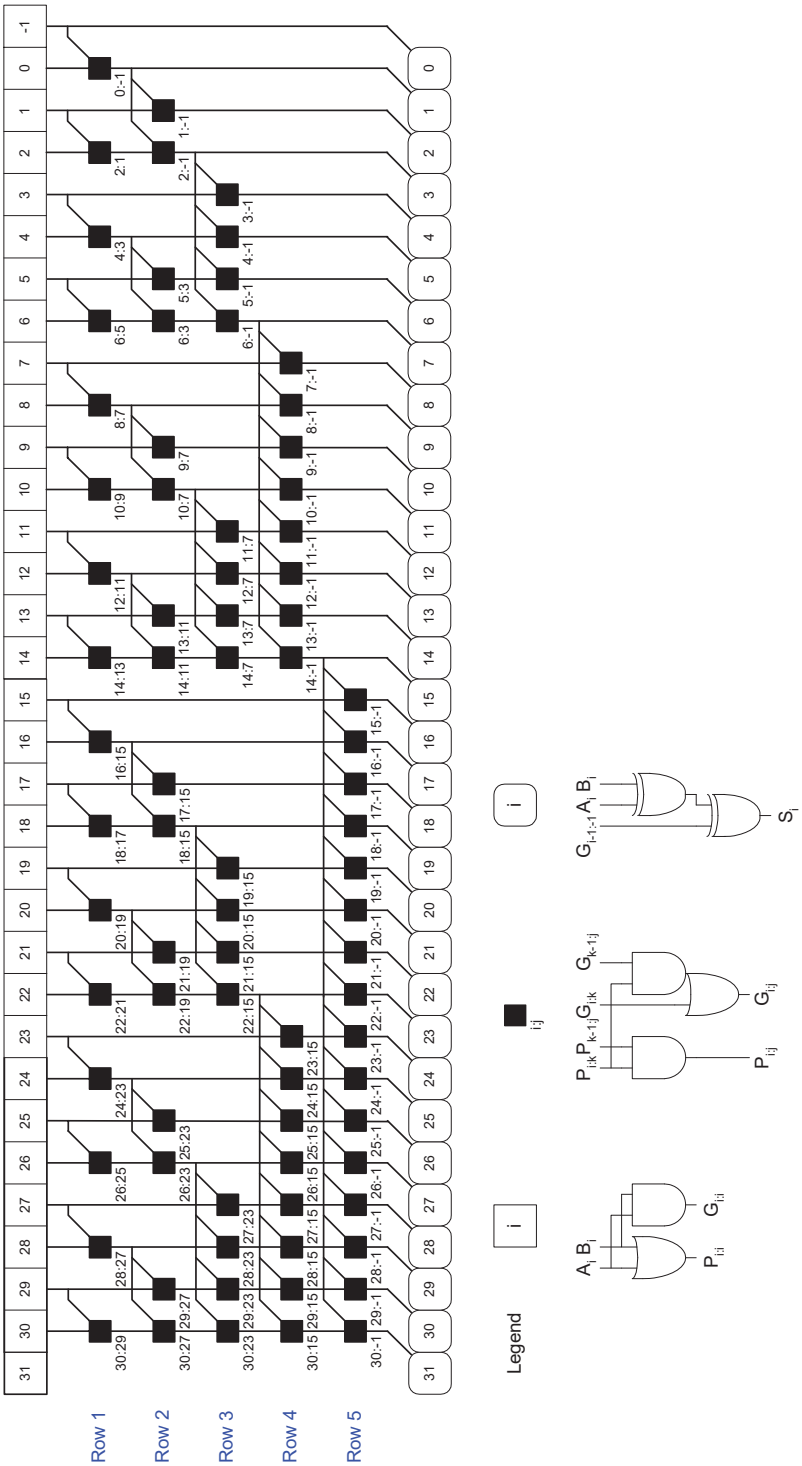
```
module fpmult(input  logic [31:0] a, b,
              output logic [31:0] m);

  logic [7:0]  expa, expb, exp;
  logic [23:0] manta, mantb;
  logic [22:0] fract;
  logic [47:0] result;

  assign {expa, manta} = {a[30:23], 1'b1, a[22:0]};
  assign {expb, mantb} = {b[30:23], 1'b1, b[22:0]};
  assign m             = {1'b0, exp, fract};

  assign result = manta * mantb;
  assign fract = result[47] ?
                 result[46:24] :
                 result[45:23];

  assign exp = result[47] ?
               (expa + expb - 126) :
               (expa + expb - 127);

endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity fpmult is
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       m:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of fpmult is
  signal expa, expb, exp:
    STD_LOGIC_VECTOR(7 downto 0);
  signal manta, mantb:
    STD_LOGIC_VECTOR(23 downto 0);
  signal fract:
    STD_LOGIC_VECTOR(22 downto 0);
  signal result:
    STD_LOGIC_VECTOR(47 downto 0);
begin
  expa   <= a(30 downto 23);
  manta  <= '1' & a(22 downto 0);
  expb   <= b(30 downto 23);
  mantb  <= '1' & b(22 downto 0);

  m      <= '0' & exp & fract;
  result <= manta * mantb;
  fract  <= result(46 downto 24)
              when (result(47) = '1')
            else result(45 downto 23);
  exp    <= (expa + expb - 126)
              when (result(47) = '1')
            else (expa + expb - 127);

end;
```

Exercise 5.49

(a) Figure on next page


Row 1
Row 2
Row 3
Row 4
Row 5
Legend
i
i:j
$A_i B_i$
P_{ij}
G_{ij}
$P_{ik} P_{k-1:j} G_{ik}$
$G_{k-1:j}$
$G_{i-1:-1} A_i B_i$
S_i

5.49 (b)

SystemVerilog

```
module prefixadd(input  logic [31:0] a, b,
                 input  logic        cin,
                 output logic [31:0] s,
                 output logic        cout);

  logic [30:0] p, g;
  // p and g prefixes for rows 1 - 5
  logic [15:0] p1, p2, p3, p4, p5;
  logic [15:0] g1, g2, g3, g4, g5;

  pandg row0(a, b, p, g);
  blackbox row1({p[30],p[28],p[26],p[24],p[22],
                 p[20],p[18],p[16],p[14],p[12],
                 p[10],p[8],p[6],p[4],p[2],p[0]},
                {p[29],p[27],p[25],p[23],p[21],
                 p[19],p[17],p[15],p[13],p[11],
                 p[9],p[7],p[5],p[3],p[1],1'b0},
                {g[30],g[28],g[26],g[24],g[22],
                 g[20],g[18],g[16],g[14],g[12],
                 g[10],g[8],g[6],g[4],g[2],g[0]},
                {g[29],g[27],g[25],g[23],g[21],
                 g[19],g[17],g[15],g[13],g[11],
                 g[9],g[7],g[5],g[3],g[1],cin},
                 p1, g1);
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity prefixadd is
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       cin:  in  STD_LOGIC;
       s:    out STD_LOGIC_VECTOR(31 downto 0);
       cout: out STD_LOGIC);
end;

architecture synth of prefixadd is
  component pgblock
    port(a, b: in  STD_LOGIC_VECTOR(30 downto 0);
         p, g: out STD_LOGIC_VECTOR(30 downto 0));
  end component;

  component pgblackblock is
    port (pik, gik: in STD_LOGIC_VECTOR(15 downto 0);
          pkj, gkj: in STD_LOGIC_VECTOR(15 downto 0);
           pij: out STD_LOGIC_VECTOR(15 downto 0);
           gij: out STD_LOGIC_VECTOR(15 downto 0));
  end component;

  component sumblock is
    port (a, b, g: in  STD_LOGIC_VECTOR(31 downto 0);
          s:      out STD_LOGIC_VECTOR(31 downto 0));
  end component;

  signal p, g: STD_LOGIC_VECTOR(30 downto 0);
  signal pik_1, pik_2, pik_3, pik_4, pik_5,
         gik_1, gik_2, gik_3, gik_4, gik_5,
         pkj_1, pkj_2, pkj_3, pkj_4, pkj_5,
         gkj_1, gkj_2, gkj_3, gkj_4, gkj_5,
         p1, p2, p3, p4, p5,
         g1, g2, g3, g4, g5:
               STD_LOGIC_VECTOR(15 downto 0);
  signal g6:   STD_LOGIC_VECTOR(31 downto 0);

begin
  row0: pgblock
    port map(a(30 downto 0), b(30 downto 0), p, g);

  pik_1 <=
   (p(30)&p(28)&p(26)&p(24)&p(22)&p(20)&p(18)&p(16)&
    p(14)&p(12)&p(10)&p(8)&p(6)&p(4)&p(2)&p(0));
  gik_1 <=
   (g(30)&g(28)&g(26)&g(24)&g(22)&g(20)&g(18)&g(16)&
    g(14)&g(12)&g(10)&g(8)&g(6)&g(4)&g(2)&g(0));
  pkj_1 <=
   (p(29)&p(27)&p(25)&p(23)&p(21)&p(19)&p(17)&p(15)&
    p(13)&p(11)&p(9)&p(7)&p(5)&p(3)&p(1)&'0');
  gkj_1 <=
   (g(29)&g(27)&g(25)&g(23)&g(21)&g(19)&g(17)&g(15)&
    g(13)&g(11)&g(9)&g(7)&g(5)& g(3)& g(1)& cin);

  row1: pgblackblock
        port map(pik_1, gik_1, pkj_1, gkj_1,
                 p1, g1);
```

(continued on next page)
(continued from previous page)

SystemVerilog

```
blackbox row2({p1[15],p[29],p1[13],p[25],p1[11],
               p[21],p1[9],p[17],p1[7],p[13],
               p1[5],p[9],p1[3],p[5],p1[1],p[1]},
              {{2{p1[14]}},{2{p1[12]}},{2{p1[10]}},
               {2{p1[8]}},{2{p1[6]}},{2{p1[4]}},
               {2{p1[2]}},{2{p1[0]}}},
              {g1[15],g[29],g1[13],g[25],g1[11],
               g[21],g1[9],g[17],g1[7],g[13],
               g1[5],g[9],g1[3],g[5],g1[1],g[1]},
              {{2{g1[14]}},{2{g1[12]}},{2{g1[10]}},
               {2{g1[8]}},{2{g1[6]}},{2{g1[4]}},
               {2{g1[2]}},{2{g1[0]}}},
               p2, g2);

blackbox row3({p2[15],p2[14],p1[14],p[27],p2[11],
               p2[10],p1[10],p[19],p2[7],p2[6],
              p1[6],p[11],p2[3],p2[2],p1[2],p[3]},
              {{4{p2[13]}},{4{p2[9]}},{4{p2[5]}},
               {4{p2[1]}}},
               {g2[15],g2[14],g1[14],g[27],g2[11],
               g2[10],g1[10],g[19],g2[7],g2[6],
              g1[6],g[11],g2[3],g2[2],g1[2],g[3]},
               {{4{g2[13]}},{4{g2[9]}},{4{g2[5]}},
               {4{g2[1]}}},
               p3, g3);
```

VHDL

```
  pik_2 <= p1(15)&p(29)&p1(13)&p(25)&p1(11)&
           p(21)&p1(9)&p(17)&p1(7)&p(13)&
           p1(5)&p(9)&p1(3)&p(5)&p1(1)&p(1);

gik_2 <= g1(15)&g(29)&g1(13)&g(25)&g1(11)&
         g(21)&g1(9)&g(17)&g1(7)&g(13)&
         g1(5)&g(9)&g1(3)&g(5)&g1(1)&g(1);

pkj_2 <=
        p1(14)&p1(14)&p1(12)&p1(12)&p1(10)&p1(10)&
         p1(8)&p1(8)&p1(6)&p1(6)&p1(4)&p1(4)&
         p1(2)&p1(2)&p1(0)&p1(0);

gkj_2 <=
        g1(14)&g1(14)&g1(12)&g1(12)&g1(10)&g1(10)&
         g1(8)&g1(8)&g1(6)&g1(6)&g1(4)&g1(4)&
         g1(2)&g1(2)&g1(0)&g1(0);

row2: pgblackblock
      port map(pik_2, gik_2, pkj_2, gkj_2,
               p2, g2);

pik_3 <= p2(15)&p2(14)&p1(14)&p(27)&p2(11)&
         p2(10)&p1(10)&p(19)&p2(7)&p2(6)&
         p1(6)&p(11)&p2(3)&p2(2)&p1(2)&p(3);
gik_3 <= g2(15)&g2(14)&g1(14)&g(27)&g2(11)&
         g2(10)&g1(10)&g(19)&g2(7)&g2(6)&
         g1(6)&g(11)&g2(3)&g2(2)&g1(2)&g(3);
pkj_3 <= p2(13)&p2(13)&p2(13)&p2(13)&
         p2(9)&p2(9)&p2(9)&p2(9)&
         p2(5)&p2(5)&p2(5)&p2(5)&
         p2(1)&p2(1)&p2(1)&p2(1);
gkj_3 <= g2(13)&g2(13)&g2(13)&g2(13)&
         g2(9)&g2(9)&g2(9)&g2(9)&
         g2(5)&g2(5)&g2(5)&g2(5)&
         g2(1)&g2(1)&g2(1)&g2(1);

row3: pgblackblock
     port map(pik_3, gik_3, pkj_3, gkj_3, p3, g3);
```

(continued on next page)

SystemVerilog

```
  blackbox row4({p3[15:12],p2[13:12],
                 p1[12],p[23],p3[7:4],
                 p2[5:4],p1[4],p[7]},
                 {{8{p3[11]}},{8{p3[3]}}},
                 {g3[15:12],g2[13:12],
                 g1[12],g[23],g3[7:4],
                 g2[5:4],g1[4],g[7]},
                 {{8{g3[11]}},{8{g3[3]}}},
                 p4, g4);

  blackbox row5({p4[15:8],p3[11:8],p2[9:8],
                 p1[8],p[15]},
                 {{16{p4[7]}}},
                 {g4[15:8],g3[11:8],g2[9:8],
                 g1[8],g[15]},
                 {{16{g4[7]}}},
                 p5,g5);

  sum row6({g5,g4[7:0],g3[3:0],g2[1:0],g1[0],cin},
           a, b, s);

  // generate cout
  assign cout = (a[31] & b[31]) |
                (g5[15] & (a[31] | b[31]));

endmodule
```

VHDL

```
  pik_4 <= p3(15 downto 12)&p2(13 downto 12)&
           p1(12)&p(23)&p3(7 downto 4)&
           p2(5 downto 4)&p1(4)&p(7);
  gik_4 <= g3(15 downto 12)&g2(13 downto 12)&
           g1(12)&g(23)&g3(7 downto 4)&
           g2(5 downto 4)&g1(4)&g(7);
  pkj_4 <= p3(11)&p3(11)&p3(11)&p3(11)&
           p3(11)&p3(11)&p3(11)&p3(11)&
           p3(3)&p3(3)&p3(3)&p3(3)&
           p3(3)&p3(3)&p3(3)&p3(3);
  gkj_4 <= g3(11)&g3(11)&g3(11)&g3(11)&
           g3(11)&g3(11)&g3(11)&g3(11)&
           g3(3)&g3(3)&g3(3)&g3(3)&
           g3(3)&g3(3)&g3(3)&g3(3);

  row4: pgblackblock
        port map(pik_4, gik_4, pkj_4, gkj_4, p4, g4);

  pik_5 <= p4(15 downto 8)&p3(11 downto 8)&
           p2(9 downto 8)&p1(8)&p(15);
  gik_5 <= g4(15 downto 8)&g3(11 downto 8)&
           g2(9 downto 8)&g1(8)&g(15);
  pkj_5 <= p4(7)&p4(7)&p4(7)&p4(7)&
           p4(7)&p4(7)&p4(7)&p4(7)&
           p4(7)&p4(7)&p4(7)&p4(7)&
           p4(7)&p4(7)&p4(7)&p4(7);
           gkj_5 <= g4(7)&g4(7)&g4(7)&g4(7)&
           g4(7)&g4(7)&g4(7)&g4(7)&
           g4(7)&g4(7)&g4(7)&g4(7)&
           g4(7)&g4(7)&g4(7)&g4(7);

  row5: pgblackblock
        port map(pik_5, gik_5, pkj_5, gkj_5, p5, g5);

  g6 <= (g5 & g4(7 downto 0) & g3(3 downto 0) &
         g2(1 downto 0) & g1(0) & cin);

  row6: sumblock
        port map(g6, a, b, s);

  -- generate cout
  cout <= (a(31) and b(31)) or
          (g6(31) and (a(31) or b(31)));

end;
```

(continued on next page)

(continued from previous page)

SystemVerilog

```
module pandg(input  logic [30:0] a, b,
             output logic [30:0] p, g);

  assign p = a | b;
  assign g = a & b;

endmodule

module blackbox(input  logic [15:0] pleft, pright,
                             gleft, gright,
                output logic [15:0] pnext, gnext);

  assign pnext = pleft & pright;
  assign gnext = pleft & gright | gleft;
endmodule


module sum(input  logic [31:0] g, a, b,
           output logic [31:0] s);

  assign s = a ^ b ^ g;

endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity pgblock is
  port(a, b: in  STD_LOGIC_VECTOR(30 downto 0);
       p, g: out STD_LOGIC_VECTOR(30 downto 0));
end;

architecture synth of pgblock is
begin
  p <= a or b;
  g <= a and b;
end;


library IEEE; use IEEE.STD_LOGIC_1164.all;

entity pgblackblock is
  port(pik, gik, pkj, gkj:
         in  STD_LOGIC_VECTOR(15 downto 0);
       pij, gij:
         out STD_LOGIC_VECTOR(15 downto 0));
end;

architecture synth of pgblackblock is
begin
  pij <= pik and pkj;
  gij <= gik or (pik and gkj);
end;


library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sumblock is
  port(g, a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       s:       out STD_LOGIC_VECTOR(31 downto 0));
end;


architecture synth of sumblock is
begin
  s <= a xor b xor g;
end;
```

5.49 (c) Using Equation 5.11 to find the delay of the prefix adder:

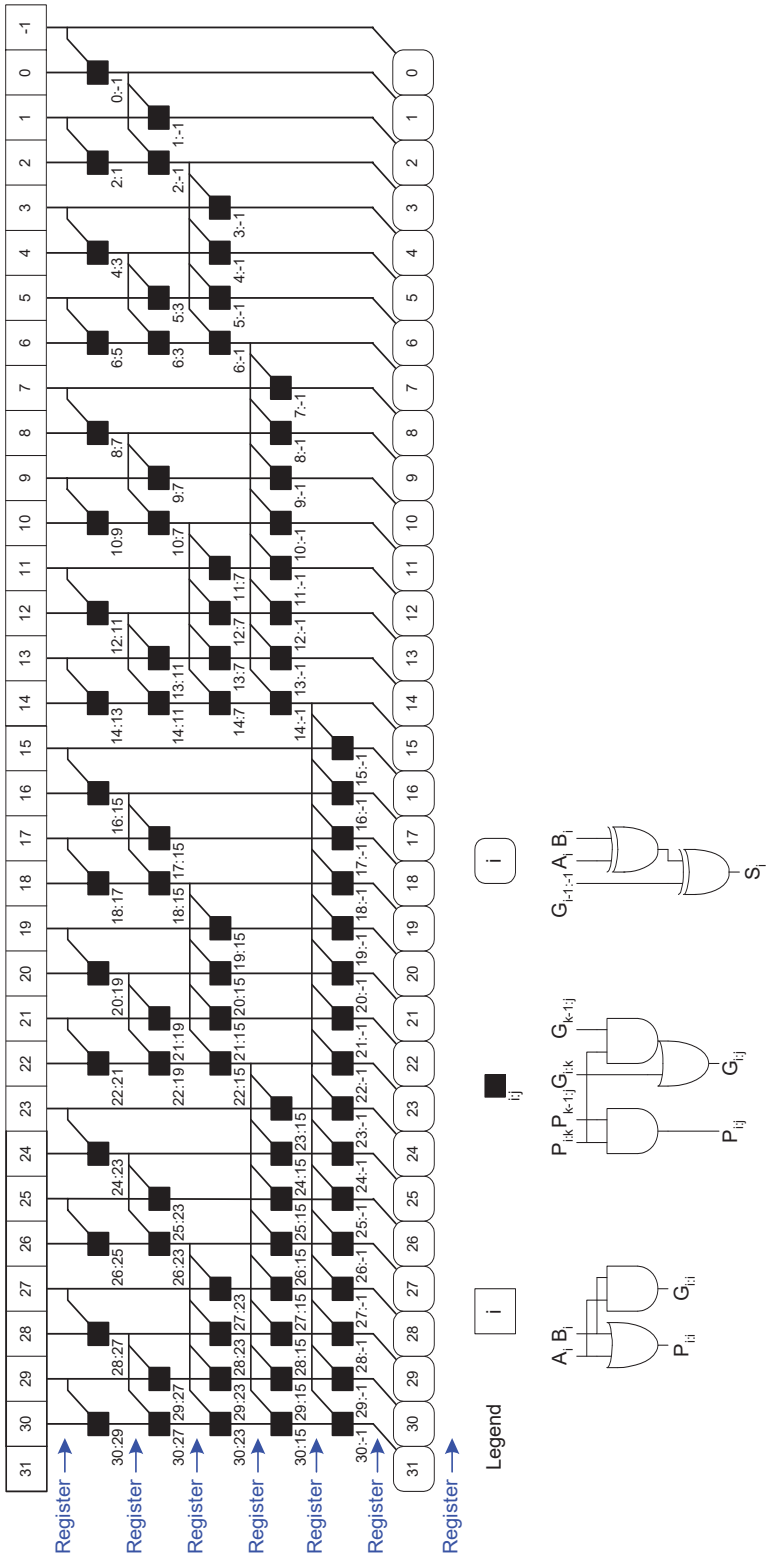
$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{XOR}$$

We find the delays for each block:
$t_{pg} = 100$ ps
$t_{pg\_prefix} = 200$ ps
$t_{XOR} = 100$ ps

Thus,
$t_{PA} = [100 + 5(200) + 100]$ ps = 1200 ps = **1.2 ns**

5.41 (d) To make a pipelined prefix adder, add pipeline registers between each of the rows of the prefix adder. Now each stage will take 200 ps plus the

sequencing overhead, $t_{pq} + t_{\text{setup}} = 80$ps. Thus each cycle is 280 ps and the design can run at 3.57 GHz.


Register
Legend
i
i:j
A_i B_i
$P_{i:i}$ $G_{i:i}$
$P_{i:k}$ $P_{k-1:j}$ $G_{i:k}$ $G_{k-1:j}$
$P_{i:j}$ $G_{i:j}$
$G_{i-1:-1}$ A_i B_i
S_i

5.49 (e)

SystemVerilog

```
module prefixaddpipe(input  logic        clk, cin,
                     input  logic [31:0] a, b,
                     output logic [31:0] s, output cout);

  // p and g prefixes for rows 0 - 5
  logic [30:0] p0, p1, p2, p3, p4, p5;
  logic [30:0] g0, g1, g2, g3, g4, g5;
  logic p_1_0, p_1_1, p_1_2, p_1_3, p_1_4, p_1_5,
       g_1_0, g_1_1, g_1_2, g_1_3, g_1_4, g_1_5;

  // pipeline values for a and b
  logic [31:0] a0, a1, a2, a3, a4, a5,
               b0, b1, b2, b3, b4, b5;

  // row 0
  flop #(2) flop0_pg_1(clk, {1'b0,cin}, {p_1_0,g_1_0});
  pandg row0(clk, a[30:0], b[30:0], p0, g0);

  // row 1
  flop #(2)  flop1_pg_1(clk, {p_1_0,g_1_0}, {p_1_1,g_1_1});
                           flop             #(30)             flop1_pg(clk,
{p0[29],p0[27],p0[25],p0[23],p0[21],p0[19],p0[17],p0[15],
                            p0[13],p0[11],p0[9],p0[7],p0[5],p0[3],p0[1],

g0[29],g0[27],g0[25],g0[23],g0[21],g0[19],g0[17],g0[15],
                            g0[13],g0[11],g0[9],g0[7],g0[5],g0[3],g0[1]},
{p1[29],p1[27],p1[25],p1[23],p1[21],p1[19],p1[17],p1[15],
                            p1[13],p1[11],p1[9],p1[7],p1[5],p1[3],p1[1],

g1[29],g1[27],g1[25],g1[23],g1[21],g1[19],g1[17],g1[15],
                            g1[13],g1[11],g1[9],g1[7],g1[5],g1[3],g1[1]});

  blackbox row1(clk,
 {p0[30],p0[28],p0[26],p0[24],p0[22],
                 p0[20],p0[18],p0[16],p0[14],p0[12],
                 p0[10],p0[8],p0[6],p0[4],p0[2],p0[0]},

                {p0[29],p0[27],p0[25],p0[23],p0[21],
                 p0[19],p0[17],p0[15],p0[13],p0[11],
                 p0[9],p0[7],p0[5],p0[3],p0[1],1'b0},

                {g0[30],g0[28],g0[26],g0[24],g0[22],
                 g0[20],g0[18],g0[16],g0[14],g0[12],
                 g0[10],g0[8],g0[6],g0[4],g0[2],g0[0]},

                {g0[29],g0[27],g0[25],g0[23],g0[21],
                 g0[19],g0[17],g0[15],g0[13],g0[11],
                 g0[9],g0[7],g0[5],g0[3],g0[1],g_1_0},

                {p1[30],p1[28],p1[26],p1[24],p1[22],p1[20],
 p1[18],p1[16],p1[14],p1[12],p1[10],p1[8],
 p1[6],p1[4],p1[2],p1[0]},

                {g1[30],g1[28],g1[26],g1[24],g1[22],g1[20],
 g1[18],g1[16],g1[14],g1[12],g1[10],g1[8],
 g1[6],g1[4],g1[2],g1[0]});

  // row 2
  flop #(2)  flop2_pg_1(clk, {p_1_1,g_1_1}, {p_1_2,g_1_2});
                           flop             #(30)             flop2_pg(clk,
{p1[28:27],p1[24:23],p1[20:19],p1[16:15],p1[12:11],
```

```
                           p1[8:7],p1[4:3],p1[0],
g1[28:27],g1[24:23],g1[20:19],g1[16:15],g1[12:11],
g1[8:7],g1[4:3],g1[0]},
 {p2[28:27],p2[24:23],p2[20:19],p2[16:15],p2[12:11],
                           p2[8:7],p2[4:3],p2[0],
 g2[28:27],g2[24:23],g2[20:19],g2[16:15],g2[12:11],
 g2[8:7],g2[4:3],g2[0]});
  blackbox row2(clk,

{p1[30:29],p1[26:25],p1[22:21],p1[18:17],p1[14:13],p1[10:9],p1[6:5],p1[2:1]
},

  {  {2{p1[28]}},  {2{p1[24]}},  {2{p1[20]}},  {2{p1[16]}},  {2{p1[12]}},
{2{p1[8]}},
   {2{p1[4]}}, {2{p1[0]}} },


{g1[30:29],g1[26:25],g1[22:21],g1[18:17],g1[14:13],g1[10:9],g1[6:5],g1[2:1]
},

  {  {2{g1[28]}},  {2{g1[24]}},  {2{g1[20]}},  {2{g1[16]}},  {2{g1[12]}},
{2{g1[8]}},
   {2{g1[4]}}, {2{g1[0]}} },


{p2[30:29],p2[26:25],p2[22:21],p2[18:17],p2[14:13],p2[10:9],p2[6:5],p2[2:1]
},


{g2[30:29],g2[26:25],g2[22:21],g2[18:17],g2[14:13],g2[10:9],g2[6:5],g2[2:1]
} );

  // row 3
  flop #(2)  flop3_pg_1(clk, {p_1_2,g_1_2}, {p_1_3,g_1_3});
  flop #(30) flop3_pg(clk, {p2[26:23],p2[18:15],p2[10:7],p2[2:0],
 g2[26:23],g2[18:15],g2[10:7],g2[2:0]},
{p3[26:23],p3[18:15],p3[10:7],p3[2:0],
 g3[26:23],g3[18:15],g3[10:7],g3[2:0]});
  blackbox row3(clk,
                {p2[30:27],p2[22:19],p2[14:11],p2[6:3]},
 { {4{p2[26]}}, {4{p2[18]}}, {4{p2[10]}}, {4{p2[2]}} },
 {g2[30:27],g2[22:19],g2[14:11],g2[6:3]},
 { {4{g2[26]}}, {4{g2[18]}}, {4{g2[10]}}, {4{g2[2]}} },
 {p3[30:27],p3[22:19],p3[14:11],p3[6:3]},
 {g3[30:27],g3[22:19],g3[14:11],g3[6:3]});

  // row 4
  flop #(2)  flop4_pg_1(clk, {p_1_3,g_1_3}, {p_1_4,g_1_4});
  flop #(30) flop4_pg(clk, {p3[22:15],p3[6:0],
 g3[22:15],g3[6:0]},
                           {p4[22:15],p4[6:0],
 g4[22:15],g4[6:0]});

  blackbox row4(clk,
                {p3[30:23],p3[14:7]},
 { {8{p3[22]}}, {8{p3[6]}} },
                {g3[30:23],g3[14:7]},
 { {8{g3[22]}}, {8{g3[6]}} },
 {p4[30:23],p4[14:7]},
 {g4[30:23],g4[14:7]});

  // row 5
  flop #(2)  flop5_pg_1(clk, {p_1_4,g_1_4}, {p_1_5,g_1_5});
  flop #(30) flop5_pg(clk, {p4[14:0],g4[14:0]},
                           {p5[14:0],g5[14:0]});
```

```
  blackbox row5(clk,
                p4[30:15],
 {16{p4[14]}},
 g4[30:15],
 {16{g4[14]}},
 p5[30:15], g5[30:15]);

  // pipeline registers for a and b
  flop #(64) flop0_ab(clk, {a,b}, {a0,b0});
  flop #(64) flop1_ab(clk, {a0,b0}, {a1,b1});
  flop #(64) flop2_ab(clk, {a1,b1}, {a2,b2});
  flop #(64) flop3_ab(clk, {a2,b2}, {a3,b3});
  flop #(64) flop4_ab(clk, {a3,b3}, {a4,b4});
  flop #(64) flop5_ab(clk, {a4,b4}, {a5,b5});

  sum row6(clk, {g5,g_1_5}, a5, b5, s);
  // generate cout
  assign cout = (a5[31] & b5[31]) | (g5[30] & (a5[31] | b5[31]));
endmodule

// submodules
module pandg(input  logic        clk,
             input  logic [30:0] a, b,
             output logic [30:0] p, g);

  always_ff @(posedge clk)
  begin
    p <= a | b;
    g <= a & b;
  end

endmodule

module blackbox(input  logic clk,
                input  logic [15:0] pleft, pright, gleft, gright,
                output logic [15:0] pnext, gnext);

  always_ff @(posedge clk)
  begin
    pnext <= pleft & pright;
    gnext <= pleft & gright | gleft;
  end
endmodule


module sum(input  logic        clk,
           input  logic [31:0] g, a, b,
           output logic [31:0] s);

  always_ff @(posedge clk)
    s <= a ^ b ^ g;
endmodule

module flop
  #(parameter width = 8)
  (input  logic             clk,
   input  logic [width-1:0] d,
   output logic [width-1:0] q);

  always_ff @(posedge clk)
    q <= d;
endmodule
```

5.49 (e)

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity prefixaddpipe is
  port(clk:  in  STD_LOGIC;
       a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       cin:  in  STD_LOGIC;
       s:    out STD_LOGIC_VECTOR(31 downto 0);
       cout: out STD_LOGIC);
end;

architecture synth of prefixaddpipe is
  component pgblock
    port(clk:  in  STD_LOGIC;
         a, b: in  STD_LOGIC_VECTOR(30 downto 0);
         p, g: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component sumblock is
    port (clk:     in  STD_LOGIC;
          a, b, g: in  STD_LOGIC_VECTOR(31 downto 0);
          s:       out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component flop is generic(width: integer);
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:   out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component flop1 is
    port(clk:        in  STD_LOGIC;
         d:          in  STD_LOGIC;
         q:          out STD_LOGIC);
  end component;
  component row1 is
    port(clk:  in  STD_LOGIC;
         p0, g0: in  STD_LOGIC_VECTOR(30 downto 0);
         p_1_0, g_1_0: in STD_LOGIC;
         p1, g1: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component row2 is
    port(clk:  in  STD_LOGIC;
         p1, g1: in  STD_LOGIC_VECTOR(30 downto 0);
         p2, g2: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component row3 is
    port(clk:  in  STD_LOGIC;
         p2, g2: in  STD_LOGIC_VECTOR(30 downto 0);
         p3, g3: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component row4 is
    port(clk:  in  STD_LOGIC;
         p3, g3: in  STD_LOGIC_VECTOR(30 downto 0);
         p4, g4: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
  component row5 is
    port(clk:  in  STD_LOGIC;
         p4, g4: in  STD_LOGIC_VECTOR(30 downto 0);
         p5, g5: out STD_LOGIC_VECTOR(30 downto 0));
  end component;
```

```
  -- p and g prefixes for rows 0 - 5
  signal p0, p1, p2, p3, p4, p5: STD_LOGIC_VECTOR(30 downto 0);
  signal g0, g1, g2, g3, g4, g5: STD_LOGIC_VECTOR(30 downto 0);

  -- p and g prefixes for column -1, rows 0 - 5
  signal p_1_0, p_1_1, p_1_2, p_1_3, p_1_4, p_1_5,
         g_1_0, g_1_1, g_1_2, g_1_3, g_1_4, g_1_5: STD_LOGIC;

  -- pipeline values for a and b
  signal a0, a1, a2, a3, a4, a5,
         b0, b1, b2, b3, b4, b5: STD_LOGIC_VECTOR(31 downto 0);

  -- final generate signal
  signal g5_all: STD_LOGIC_VECTOR(31 downto 0);

begin

  -- p and g calculations
  row0_reg: pgblock port map(clk, a(30 downto 0), b(30 downto 0), p0, g0);
  row1_reg: row1 port map(clk, p0, g0, p_1_0, g_1_0, p1, g1);
  row2_reg: row2 port map(clk, p1, g1, p2, g2);
  row3_reg: row3 port map(clk, p2, g2, p3, g3);
  row4_reg: row4 port map(clk, p3, g3, p4, g4);
  row5_reg: row5 port map(clk, p4, g4, p5, g5);


  -- pipeline registers for a and b
  flop0_a: flop generic map(32) port map (clk, a, a0);
  flop0_b: flop generic map(32) port map (clk, b, b0);
  flop1_a: flop generic map(32) port map (clk, a0, a1);
  flop1_b: flop generic map(32) port map (clk, b0, b1);
  flop2_a: flop generic map(32) port map (clk, a1, a2);
  flop2_b: flop generic map(32) port map (clk, b1, b2);
  flop3_a: flop generic map(32) port map (clk, a2, a3);
  flop3_b: flop generic map(32) port map (clk, b2, b3);
  flop4_a: flop generic map(32) port map (clk, a3, a4);
  flop4_b: flop generic map(32) port map (clk, b3, b4);
  flop5_a: flop generic map(32) port map (clk, a4, a5);
  flop5_b: flop generic map(32) port map (clk, b4, b5);

  -- pipeline p and g for column -1
  p_1_0 <= '0'; flop_1_g0: flop1 port map (clk, cin, g_1_0);
  flop_1_p1: flop1 port map (clk, p_1_0, p_1_1);
  flop_1_g1: flop1 port map (clk, g_1_0, g_1_1);
  flop_1_p2: flop1 port map (clk, p_1_1, p_1_2);
  flop_1_g2: flop1 port map (clk, g_1_1, g_1_2);
  flop_1_p3: flop1 port map (clk, p_1_2, p_1_3); flop_1_g3:
  flop1 port map (clk, g_1_2, g_1_3);
  flop_1_p4: flop1 port map (clk, p_1_3, p_1_4);
  flop_1_g4: flop1 port map (clk, g_1_3, g_1_4);
  flop_1_p5: flop1 port map (clk, p_1_4, p_1_5);
  flop_1_g5: flop1 port map (clk, g_1_4, g_1_5);

  -- generate sum and cout
  g5_all <= (g5&g_1_5);
  row6: sumblock port map(clk, g5_all, a5, b5, s);

  -- generate cout
  cout <= (a5(31) and b5(31)) or (g5(30) and (a5(31) or b5(31)));
end;


library IEEE; use IEEE.STD_LOGIC_1164.all;
entity pgblock is
  port(clk:  in  STD_LOGIC;
```

```
       a, b: in  STD_LOGIC_VECTOR(30 downto 0);
       p, g: out STD_LOGIC_VECTOR(30 downto 0));
end;

architecture synth of pgblock is
begin
  process(clk) begin
    if rising_edge(clk) then
        p <= a or b;
        g <= a and b;
    end if;
  end process;
end;


library IEEE; use IEEE.STD_LOGIC_1164.all;

entity blackbox is
  port(clk:  in  STD_LOGIC;
       pik, pkj, gik, gkj:
             in  STD_LOGIC_VECTOR(15 downto 0);
       pij, gij:
             out STD_LOGIC_VECTOR(15 downto 0));
end;

architecture synth of blackbox is
begin
  process(clk) begin
    if rising_edge(clk) then
      pij <= pik and pkj;
      gij <= gik or (pik and gkj);
    end if;
  end process;
end;


library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sumblock is
  port(clk:  in  STD_LOGIC;
       g, a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       s:       out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of sumblock is
begin
  process(clk) begin
    if rising_edge(clk) then
      s <= a xor b xor g;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;  use IEEE.STD_LOGIC_ARITH.all;
entity flop is -- parameterizable flip flop
  generic(width: integer);
  port(clk:        in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:          out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of flop is
begin
  process(clk) begin
    if rising_edge(clk) then
      q <= d;
```

```
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;  use IEEE.STD_LOGIC_ARITH.all;
entity flop1 is -- 1-bit flip flop
  port(clk:        in  STD_LOGIC;
       d:          in  STD_LOGIC;
       q:          out STD_LOGIC);
end;

architecture synth of flop1 is
begin
  process(clk) begin
    if rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity row1 is
  port(clk:    in STD_LOGIC;
       p0, g0: in  STD_LOGIC_VECTOR(30 downto 0);
       p_1_0, g_1_0: in STD_LOGIC;
       p1, g1: out STD_LOGIC_VECTOR(30 downto 0));
end;

architecture synth of row1 is
  component blackbox is
    port (clk:      in  STD_LOGIC;
          pik, pkj: in STD_LOGIC_VECTOR(15 downto 0);
          gik, gkj: in STD_LOGIC_VECTOR(15 downto 0);
          pij:      out STD_LOGIC_VECTOR(15 downto 0);
          gij:      out STD_LOGIC_VECTOR(15 downto 0));
  end component;
  component flop is generic(width: integer);
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:   out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;

  -- internal signals for calculating p, g
  signal pik_0, gik_0, pkj_0, gkj_0,
         pij_0, gij_0: STD_LOGIC_VECTOR(15 downto 0);

  -- internal signals for pipeline registers
  signal pg0_in, pg1_out: STD_LOGIC_VECTOR(29 downto 0);

begin
  pg0_in <= (p0(29)&p0(27)&p0(25)&p0(23)&p0(21)&p0(19)&p0(17)&p0(15)&
                 p0(13)&p0(11)&p0(9)&p0(7)&p0(5)&p0(3)&p0(1)&
                 g0(29)&g0(27)&g0(25)&g0(23)&g0(21)&g0(19)&g0(17)&g0(15)&
                 g0(13)&g0(11)&g0(9)&g0(7)&g0(5)&g0(3)&g0(1));
  flop1_pg: flop generic map(30) port map (clk, pg0_in, pg1_out);

  p1(29) <= pg1_out(29); p1(27)<= pg1_out(28); p1(25)<= pg1_out(27);
  p1(23) <= pg1_out(26);
  p1(21) <= pg1_out(25); p1(19) <= pg1_out(24); p1(17) <= pg1_out(23);
  p1(15) <= pg1_out(22); p1(13) <= pg1_out(21); p1(11) <= pg1_out(20);
  p1(9) <= pg1_out(19); p1(7) <= pg1_out(18); p1(5) <= pg1_out(17);
  p1(3) <= pg1_out(16); p1(1) <= pg1_out(15);
  g1(29) <= pg1_out(14); g1(27) <= pg1_out(13); g1(25) <= pg1_out(12);
  g1(23) <= pg1_out(11); g1(21) <= pg1_out(10); g1(19) <= pg1_out(9);
  g1(17) <= pg1_out(8); g1(15) <= pg1_out(7); g1(13) <= pg1_out(6);
```

```
  g1(11) <= pg1_out(5); g1(9) <= pg1_out(4); g1(7) <= pg1_out(3);
  g1(5) <= pg1_out(2); g1(3) <= pg1_out(1); g1(1) <= pg1_out(0);

  -- pg calculations
  pik_0 <= (p0(30)&p0(28)&p0(26)&p0(24)&p0(22)&p0(20)&p0(18)&p0(16)&
            p0(14)&p0(12)&p0(10)&p0(8)&p0(6)&p0(4)&p0(2)&p0(0));
  gik_0 <= (g0(30)&g0(28)&g0(26)&g0(24)&g0(22)&g0(20)&g0(18)&g0(16)&
            g0(14)&g0(12)&g0(10)&g0(8)&g0(6)&g0(4)&g0(2)&g0(0));
  pkj_0 <= (p0(29)&p0(27)&p0(25)&p0(23)&p0(21)& p0(19)& p0(17)&p0(15)&
            p0(13)&p0(11)&p0(9)&p0(7)&p0(5)&p0(3)&p0(1)&p_1_0);
  gkj_0 <= (g0(29)&g0(27)&g0(25)&g0(23)&g0(21)&g0(19)&g0(17)&g0(15)&
            g0(13)&g0(11)&g0(9)&g0(7)&g0(5)& g0(3)&g0(1)&g_1_0);

  row1: blackbox port map(clk, pik_0, pkj_0, gik_0, gkj_0, pij_0, gij_0);

  p1(30) <= pij_0(15); p1(28) <= pij_0(14); p1(26) <= pij_0(13);
  p1(24) <= pij_0(12); p1(22) <= pij_0(11); p1(20) <= pij_0(10);
  p1(18) <= pij_0(9); p1(16) <= pij_0(8); p1(14) <= pij_0(7);
  p1(12) <= pij_0(6); p1(10) <= pij_0(5); p1(8) <= pij_0(4);
 p1(6) <= pij_0(3); p1(4) <= pij_0(2); p1(2) <= pij_0(1); p1(0) <= pij_0(0);

  g1(30) <= gij_0(15); g1(28) <= gij_0(14); g1(26) <= gij_0(13);
  g1(24) <= gij_0(12); g1(22) <= gij_0(11); g1(20) <= gij_0(10);
  g1(18) <= gij_0(9); g1(16) <= gij_0(8); g1(14) <= gij_0(7);
  g1(12) <= gij_0(6); g1(10) <= gij_0(5); g1(8) <= gij_0(4);
 g1(6) <= gij_0(3); g1(4) <= gij_0(2); g1(2) <= gij_0(1); g1(0) <= gij_0(0);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity row2 is
  port(clk:    in STD_LOGIC;
       p1, g1: in  STD_LOGIC_VECTOR(30 downto 0);
       p2, g2: out STD_LOGIC_VECTOR(30 downto 0));
end;

architecture synth of row2 is
  component blackbox is
    port (clk:      in  STD_LOGIC;
          pik, pkj: in STD_LOGIC_VECTOR(15 downto 0);
          gik, gkj: in STD_LOGIC_VECTOR(15 downto 0);
          pij:      out STD_LOGIC_VECTOR(15 downto 0);
          gij:      out STD_LOGIC_VECTOR(15 downto 0));
  end component;
  component flop is generic(width: integer);
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:   out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;

  -- internal signals for calculating p, g
  signal pik_1, gik_1, pkj_1, gkj_1,
         pij_1, gij_1: STD_LOGIC_VECTOR(15 downto 0);

  -- internal signals for pipeline registers
  signal pg1_in, pg2_out: STD_LOGIC_VECTOR(29 downto 0);

begin
  pg1_in <= (p1(28 downto 27)&p1(24 downto 23)&p1(20 downto 19)&
             p1(16 downto 15)&
             p1(12 downto 11)&p1(8 downto 7)&p1(4 downto 3)&p1(0)&
             g1(28 downto 27)&g1(24 downto 23)&g1(20 downto 19)&
             g1(16 downto 15)&
             g1(12 downto 11)&g1(8 downto 7)&g1(4 downto 3)&g1(0));
  flop2_pg: flop generic map(30) port map (clk, pg1_in, pg2_out);
```

```
  p2(28 downto 27) <= pg2_out(29 downto 28);
  p2(24 downto 23) <= pg2_out(27 downto 26);
  p2(20 downto 19) <= pg2_out( 25 downto 24);
  p2(16 downto 15) <= pg2_out(23 downto 22);
  p2(12 downto 11) <= pg2_out(21 downto 20);
  p2(8 downto 7) <= pg2_out(19 downto 18);
  p2(4 downto 3) <= pg2_out(17 downto 16);
  p2(0) <= pg2_out(15);
  g2(28 downto 27) <= pg2_out(14 downto 13);
  g2(24 downto 23) <= pg2_out(12 downto 11);
  g2(20 downto 19) <= pg2_out(10 downto 9);
  g2(16 downto 15) <= pg2_out(8 downto 7);
  g2(12 downto 11) <= pg2_out(6 downto 5);
  g2(8 downto 7) <= pg2_out(4 downto 3);
  g2(4 downto 3) <= pg2_out(2 downto 1); g2(0) <= pg2_out(0);

  -- pg calculations
  pik_1 <= (p1(30 downto 29)&p1(26 downto 25)&p1(22 downto 21)&
            p1(18 downto 17)&p1(14 downto 13)&p1(10 downto 9)&
            p1(6 downto 5)&p1(2 downto 1));
  gik_1 <= (g1(30 downto 29)&g1(26 downto 25)&g1(22 downto 21)&
            g1(18 downto 17)&g1(14 downto 13)&g1(10 downto 9)&
            g1(6 downto 5)&g1(2 downto 1));
  pkj_1 <= (p1(28)&p1(28)&p1(24)&p1(24)&p1(20)&p1(20)&p1(16)&p1(16)&
            p1(12)&p1(12)&p1(8)&p1(8)&p1(4)&p1(4)&p1(0)&p1(0));
  gkj_1 <= (g1(28)&g1(28)&g1(24)&g1(24)&g1(20)&g1(20)&g1(16)&g1(16)&
            g1(12)&g1(12)&g1(8)&g1(8)&g1(4)&g1(4)&g1(0)&g1(0));

  row2: blackbox
        port map(clk, pik_1, pkj_1, gik_1, gkj_1, pij_1, gij_1);

  p2(30 downto 29) <= pij_1(15 downto 14);
  p2(26 downto 25) <= pij_1(13 downto 12);
  p2(22 downto 21) <= pij_1(11 downto 10);
  p2(18 downto 17) <= pij_1(9 downto 8);
 p2(14 downto 13) <= pij_1(7 downto 6); p2(10 downto 9) <= pij_1(5 downto 4);
  p2(6 downto 5) <= pij_1(3 downto 2); p2(2 downto 1) <= pij_1(1 downto 0);

  g2(30 downto 29) <= gij_1(15 downto 14);
  g2(26 downto 25) <= gij_1(13 downto 12);
  g2(22 downto 21) <= gij_1(11 downto 10);
  g2(18 downto 17) <= gij_1(9 downto 8);
 g2(14 downto 13) <= gij_1(7 downto 6); g2(10 downto 9) <= gij_1(5 downto 4);
  g2(6 downto 5) <= gij_1(3 downto 2); g2(2 downto 1) <= gij_1(1 downto 0);

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity row3 is
  port(clk:    in STD_LOGIC;
       p2, g2: in  STD_LOGIC_VECTOR(30 downto 0);
       p3, g3: out STD_LOGIC_VECTOR(30 downto 0));
end;

architecture synth of row3 is
  component blackbox is
    port (clk:      in  STD_LOGIC;
          pik, pkj: in STD_LOGIC_VECTOR(15 downto 0);
          gik, gkj: in STD_LOGIC_VECTOR(15 downto 0);
          pij:      out STD_LOGIC_VECTOR(15 downto 0);
          gij:      out STD_LOGIC_VECTOR(15 downto 0));
  end component;
  component flop is generic(width: integer);
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
```

```
        q:   out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;

  -- internal signals for calculating p, g
  signal pik_2, gik_2, pkj_2, gkj_2,
         pij_2, gij_2: STD_LOGIC_VECTOR(15 downto 0);

  -- internal signals for pipeline registers
  signal pg2_in, pg3_out: STD_LOGIC_VECTOR(29 downto 0);

begin
  pg2_in <= (p2(26 downto 23)&p2(18 downto 15)&p2(10 downto 7)&
             p2(2 downto 0)&
           g2(26 downto 23)&g2(18 downto 15)&g2(10 downto 7)&g2(2 downto 0));
  flop3_pg: flop generic map(30) port map (clk, pg2_in, pg3_out);
  p3(26 downto 23) <= pg3_out(29 downto 26);
  p3(18 downto 15) <= pg3_out(25 downto 22);
  p3(10 downto 7) <= pg3_out(21 downto 18);
  p3(2 downto 0) <= pg3_out(17 downto 15);
  g3(26 downto 23) <= pg3_out(14 downto 11);
  g3(18 downto 15) <= pg3_out(10 downto 7);
  g3(10 downto 7) <= pg3_out(6 downto 3);
  g3(2 downto 0) <= pg3_out(2 downto 0);

  -- pg calculations
  pik_2 <= (p2(30 downto 27)&p2(22 downto 19)&
            p2(14 downto 11)&p2(6 downto 3));
  gik_2 <= (g2(30 downto 27)&g2(22 downto 19)&
            g2(14 downto 11)&g2(6 downto 3));
  pkj_2 <= (p2(26)&p2(26)&p2(26)&p2(26)&
            p2(18)&p2(18)&p2(18)&p2(18)&
            p2(10)&p2(10)&p2(10)&p2(10)&
            p2(2)&p2(2)&p2(2)&p2(2));
  gkj_2 <= (g2(26)&g2(26)&g2(26)&g2(26)&
            g2(18)&g2(18)&g2(18)&g2(18)&
            g2(10)&g2(10)&g2(10)&g2(10)&
            g2(2)&g2(2)&g2(2)&g2(2));

  row3: blackbox
        port map(clk, pik_2, pkj_2, gik_2, gkj_2, pij_2, gij_2);

  p3(30 downto 27) <= pij_2(15 downto 12);
  p3(22 downto 19) <= pij_2(11 downto 8);
  p3(14 downto 11) <= pij_2(7 downto 4); p3(6 downto 3) <= pij_2(3 downto 0);
  g3(30 downto 27) <= gij_2(15 downto 12);
  g3(22 downto 19) <= gij_2(11 downto 8);
  g3(14 downto 11) <= gij_2(7 downto 4); g3(6 downto 3) <= gij_2(3 downto 0);

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity row4 is
  port(clk:    in STD_LOGIC;
       p3, g3: in  STD_LOGIC_VECTOR(30 downto 0);
       p4, g4: out STD_LOGIC_VECTOR(30 downto 0));
end;

architecture synth of row4 is
  component blackbox is
    port (clk:      in  STD_LOGIC;
          pik, pkj: in STD_LOGIC_VECTOR(15 downto 0);
          gik, gkj: in STD_LOGIC_VECTOR(15 downto 0);
          pij:      out STD_LOGIC_VECTOR(15 downto 0);
          gij:      out STD_LOGIC_VECTOR(15 downto 0));
  end component;
```

```
  component flop is generic(width: integer);
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:   out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;

  -- internal signals for calculating p, g
  signal pik_3, gik_3, pkj_3, gkj_3,
         pij_3, gij_3: STD_LOGIC_VECTOR(15 downto 0);

  -- internal signals for pipeline registers
  signal pg3_in, pg4_out: STD_LOGIC_VECTOR(29 downto 0);

begin
  pg3_in <= (p3(22 downto 15)&p3(6 downto 0)&g3(22 downto 15)&g3(6 downto 0));
  flop4_pg: flop generic map(30) port map (clk, pg3_in, pg4_out);
  p4(22 downto 15) <= pg4_out(29 downto 22);
  p4(6 downto 0) <= pg4_out(21 downto 15);
  g4(22 downto 15) <= pg4_out(14 downto 7);
  g4(6 downto 0) <= pg4_out(6 downto 0);

  -- pg calculations
  pik_3 <= (p3(30 downto 23)&p3(14 downto 7));
  gik_3 <= (g3(30 downto 23)&g3(14 downto 7));
  pkj_3 <= (p3(22)&p3(22)&p3(22)&p3(22)&p3(22)&p3(22)&p3(22)&p3(22)&p3(22)&
            p3(6)&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(22)&
            g3(6)&g3(6)&g3(6)&g3(6)&g3(6)&g3(6)&g3(6)&g3(6)&g3(6));

  row4: blackbox
        port map(clk, pik_3, pkj_3, gik_3, gkj_3, pij_3, gij_3);

  p4(30 downto 23) <= pij_3(15 downto 8);
  p4(14 downto 7) <= pij_3(7 downto 0);
  g4(30 downto 23) <= gij_3(15 downto 8);
  g4(14 downto 7) <= gij_3(7 downto 0);

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity row5 is
  port(clk:    in STD_LOGIC;
       p4, g4: in  STD_LOGIC_VECTOR(30 downto 0);
       p5, g5: out STD_LOGIC_VECTOR(30 downto 0));
end;
architecture synth of row5 is
  component blackbox is
    port (clk:      in  STD_LOGIC;
       pik, pkj: in STD_LOGIC_VECTOR(15 downto 0);
          gik, gkj: in STD_LOGIC_VECTOR(15 downto 0);
          pij:      out STD_LOGIC_VECTOR(15 downto 0);
          gij:      out STD_LOGIC_VECTOR(15 downto 0));
  end component;
  component flop is generic(width: integer);
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:   out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;

  -- internal signals for calculating p, g
  signal pik_4, gik_4, pkj_4, gkj_4,
         pij_4, gij_4: STD_LOGIC_VECTOR(15 downto 0);

  -- internal signals for pipeline registers
  signal pg4_in, pg5_out: STD_LOGIC_VECTOR(29 downto 0);
```

```
begin

  pg4_in <= (p4(14 downto 0)&g4(14 downto 0));
  flop4_pg: flop generic map(30) port map (clk, pg4_in, pg5_out);
  p5(14 downto 0) <= pg5_out(29 downto 15); g5(14 downto 0) <= pg5_out(14
downto 0);

  -- pg calculations
  pik_4 <= p4(30 downto 15);
  gik_4 <= g4(30 downto 15);
  pkj_4 <= p4(14)&p4(14)&p4(14)&p4(14)&
           p4(14)&p4(14)&p4(14)&p4(14)&
           p4(14)&p4(14)&p4(14)&p4(14)&
           p4(14)&p4(14)&p4(14)&p4(14);
  gkj_4 <= g4(14)&g4(14)&g4(14)&g4(14)&
           g4(14)&g4(14)&g4(14)&g4(14)&
           g4(14)&g4(14)&g4(14)&g4(14)&
           g4(14)&g4(14)&g4(14)&g4(14);

  row5: blackbox
        port map(clk, pik_4, gik_4, pkj_4, gkj_4, pij_4, gij_4);
                 p5(30 downto 15) <= pij_4; g5(30 downto 15) <= gij_4;

end;
```

Exercise 5.50

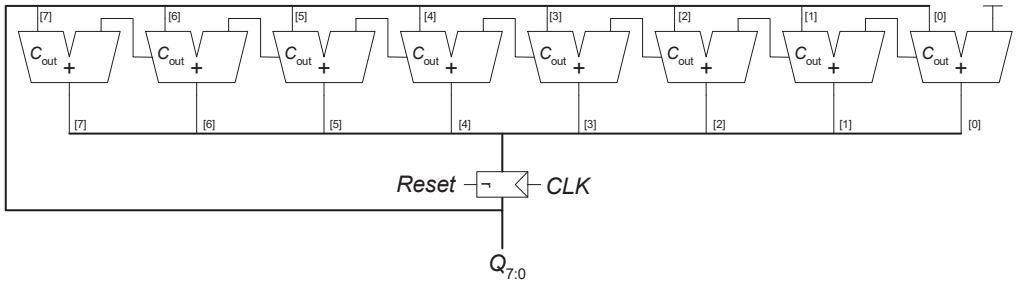


FIGURE 5.13 Incrementer built using half adders

Exercise 5.51

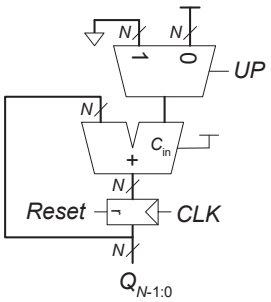


FIGURE 5.14 Up/Down counter

Exercise 5.52

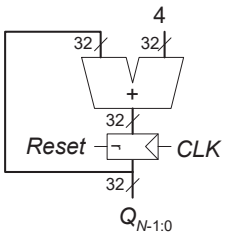


FIGURE 5.15 32-bit counter that increments by 4 on each clock edge

Exercise 5.53

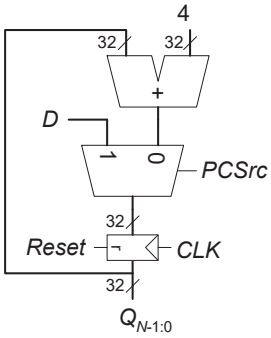


FIGURE 5.16 32-bit counter that increments by 4 or loads a new value, *D*

Exercise 5.54

(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.

5.54 (c)

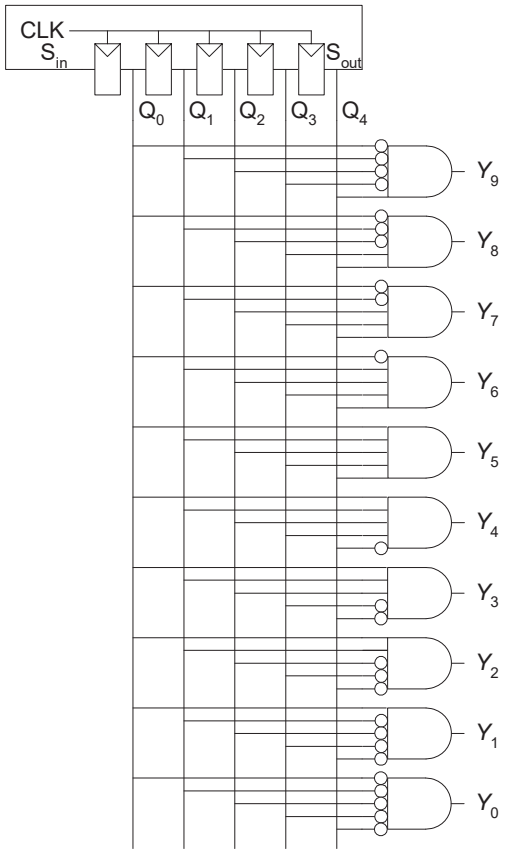


FIGURE 5.17 10-bit decimal counter using a 5-bit Johnson counter

(d) The counter uses less hardware and could be faster because it has a short critical path (a single inverter delay).

Exercise 5.55

SystemVerilog

```
module scanflop4(input  logic       clk, test, sin,
                 input  logic [3:0] d,
                 output logic [3:0] q,
                 output logic       sout);

  always_ff @(posedge clk)
    if (test)
      q <= d;
    else
      q <= {q[2:0], sin};

  assign sout = q[3];

endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity scanflop4 is
  port(clk, test, sin:  in  STD_LOGIC;
       d:  in    STD_LOGIC_VECTOR(3 downto 0);
       q:  inout STD_LOGIC_VECTOR(3 downto 0);
       sout:             out STD_LOGIC);
end;

architecture synth of scanflop4 is
begin
  process(clk, test) begin
    if rising_edge(clk) then
      if test then
        q <= d;
      else
        q <= q(2 downto 0) & sin;
      end if;
    end if;
  end process;

  sout <= q(3);

end;
```

Exercise 5.56

(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.

5.56 (b) This circuit can be built using a 16 × 2-bit memory array, with the contents given in Table 5.3.

address $a_{4:0}$	data $d_{1:0}$
00001	00
00000	00
00011	00
00101	00
01001	00
10001	00
01010	01
01011	01
01000	01
01110	01
00010	01
11010	01
10100	10
10101	10
10110	10
10000	10
11100	10
00100	10
11111	11
11110	11
11101	11
11011	11
10111	11

TABLE 5.3 Memory array values for Exercise 5.48

address $a_{4:0}$	data $d_{1:0}$
01111	11
others	XX

TABLE 5.3 Memory array values for Exercise 5.48

(c) The implementation shown in part (b) allows the encoding to change easily. Each memory address corresponds to an encoding, so simply store different data values at each memory address to change the encoding.

Exercise 5.57

http://www.intel.com/design/flash/articles/what.htm

Flash memory is a nonvolatile memory because it retains its contents after power is turned off. Flash memory allows the user to electrically program and erase information. Flash memory uses memory cells similar to an EEPROM, but with a much thinner, precisely grown oxide between a floating gate and the substrate (see Figure 5.18).

Flash programming occurs when electrons are placed on the floating gate. This is done by forcing a large voltage (usually 10 to 12 volts) on the control gate. Electrons quantum-mechanically tunnel from the source through the thin oxide onto the control gate. Because the floating gate is completely insulated by oxide, the charges are trapped on the floating gate during normal operation. If electrons are stored on the floating gate, it blocks the effect of the control gate. The electrons on the floating gate can be removed by reversing the procedure, i.e., by placing a large negative voltage on the control gate.

The default state of a flash bitcell (when there are no electrons on the floating gate) is ON, because the channel will conduct when the wordline is HIGH. After the bitcell is programmed (i.e., when there are electrons on the floating gate), the state of the bitcell is OFF, because the floating gate blocks the effect of the control gate. Flash memory is a key element in thumb drives, cell phones, digital cameras, Blackberries, and other low-power devices that must retain their memory when turned off.

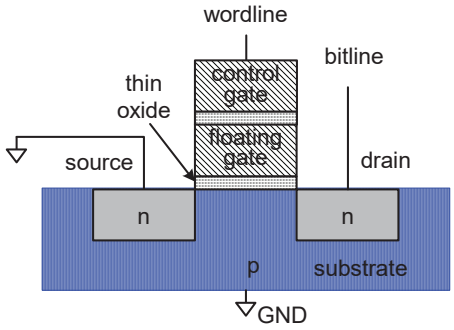


FIGURE 5.18 Flash EEPROM

Exercise 5.58

(a)

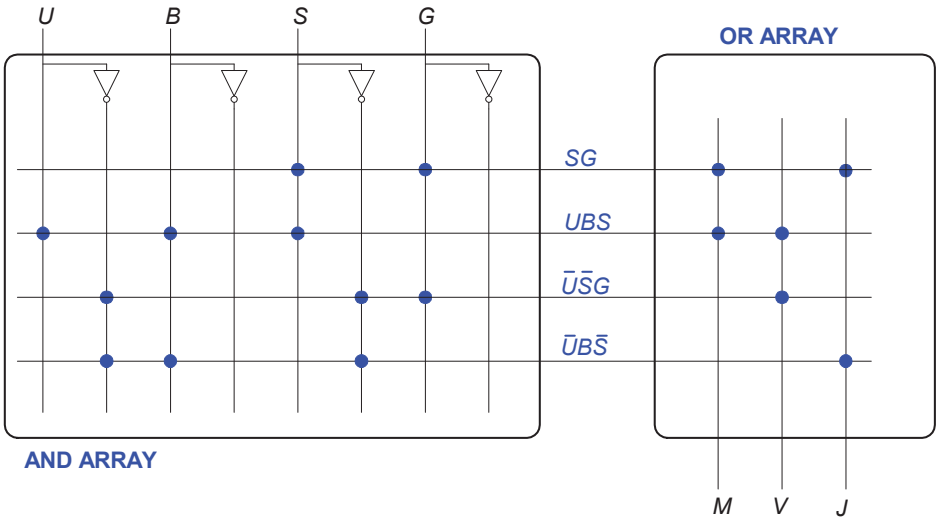


FIGURE 5.19 4 x 4 x 3 PLA implementing Exercise 5.58

5.58 (b)

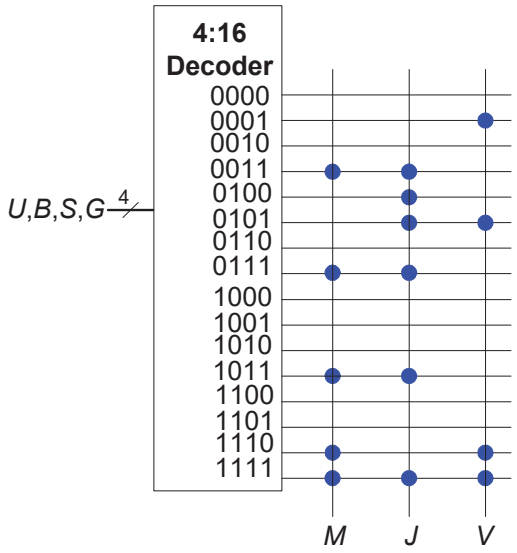


FIGURE 5.20 16 x 3 ROM implementation of Exercise 5.58

(c)

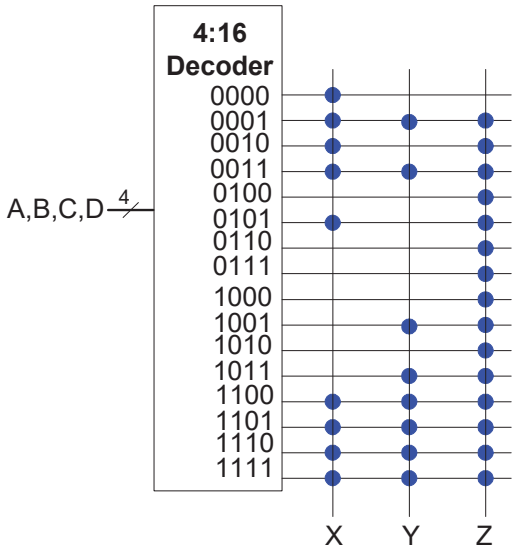
SystemVerilog

```
module ex5_44c(input  logic u, b, s, g,
               output logic m, j, v);

  assign m = s&g | u&b&s;
  assign j = ~u&b&~s | s&g;
  assign v = u&b&s | ~u&~s&g;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity ex5_44c is
  port(u, b, s, g:  in  STD_LOGIC;
       m, j, v:     out STD_LOGIC);
end;

architecture synth of ex5_44c is
begin
  m <= (s and g) or (u and b and s);
  j <= ((not u) and b and (not s)) or (s and g);
  v <= (u and b and s) or ((not u) and (not s) and g);
end;
```

Exercise 5.59



Exercise 5.60

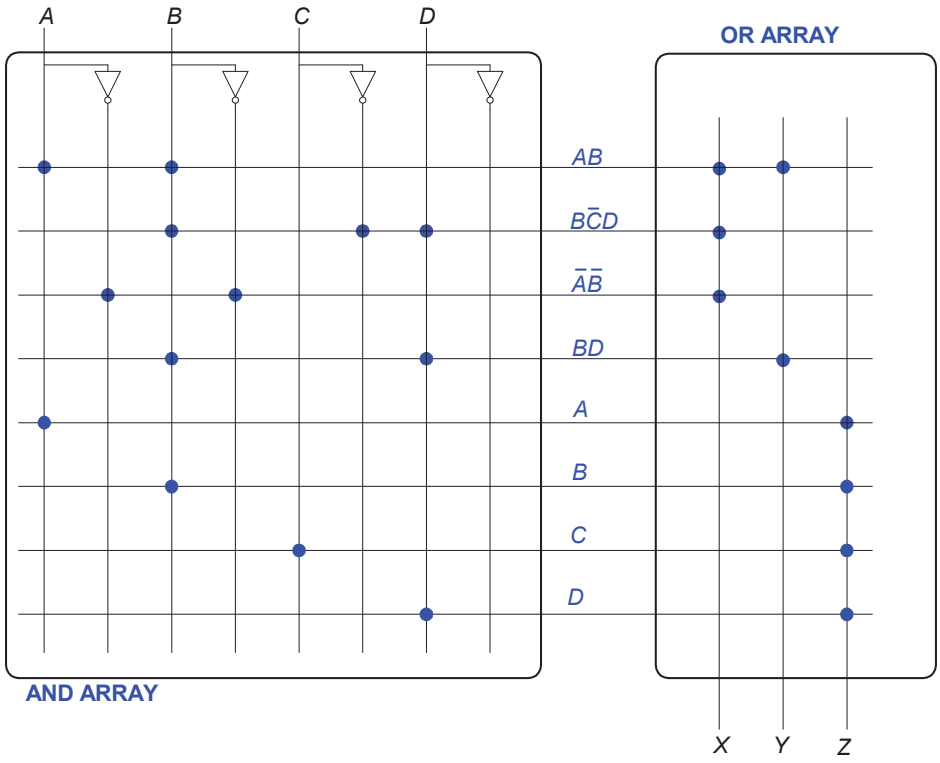


FIGURE 5.21 4 x 8 x 3 PLA for Exercise 5.52

Exercise 5.61

(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.

Exercise 5.62

(a) Yes. Both circuits can compute any function of K inputs and K outputs.

(b) No. The second circuit can only represent 2^K states. The first can represent more.

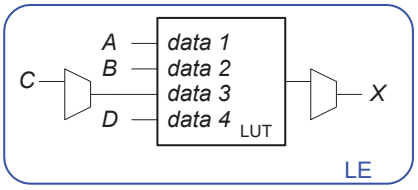
(c) Yes. Both circuits compute any function of 1 input, N outputs, and 2^K states.

(d) No. The second circuit forces the output to be the same as the state encoding, while the first one allows outputs to be independent of the state encoding.

Exercise 5.63

(a) 1 LE

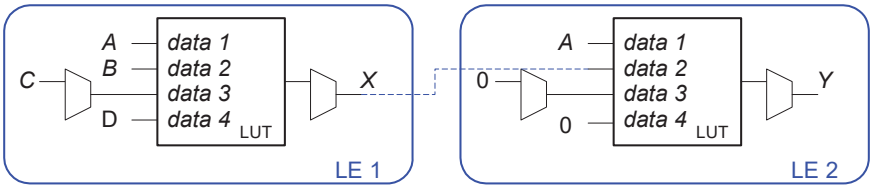
(A) data 1	(B) data 2	(C) data 3	(D) data 4	(Y) LUT output
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1



(b) 2 LEs

(B) *data 1*	(C) *data 2*	(D) *data 3*	(E) *data 4*	(X) LUT output
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

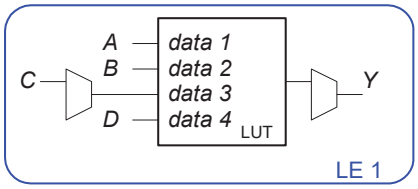
(A) *data 1*	(X) *data 2*	*data 3*	*data 4*	(Y) LUT output
0	0	X	X	0
0	1	X	X	1
1	0	X	X	1
1	1	X	X	1

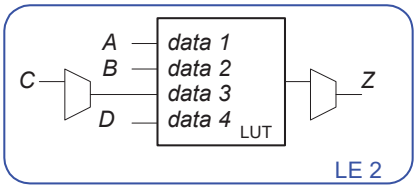


(c) 2 LEs

(A) *data 1*	(B) *data 2*	(C) *data 3*	(D) *data 4*	(Y) LUT output
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

(A) *data 1*	(B) *data 2*	(C) *data 3*	(D) *data 4*	(Z) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

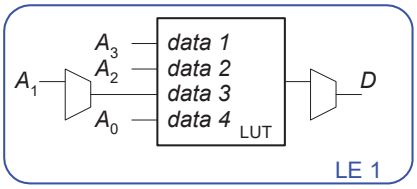


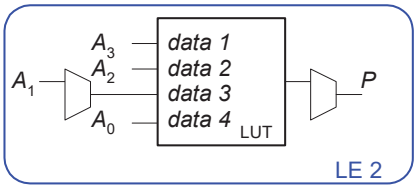


(d) 2 LEs

(A_3) *data 1*	(A_2) *data 2*	(A_1) *data 3*	(A_0) *data 4*	(D) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(A_3) *data 1*	(A_2) *data 2*	(A_1) *data 3*	(A_0) *data 4*	(P) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

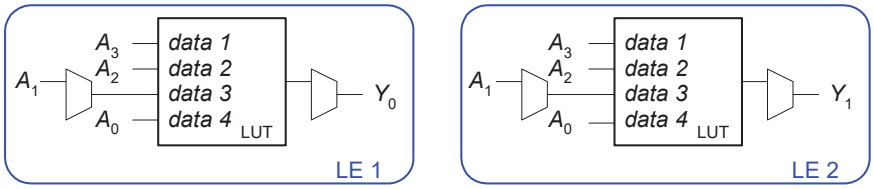




(e) 2 LEs

(A_3) *data 1*	(A_2) *data 2*	(A_1) *data 3*	(A_0) *data 4*	(Y_0) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

(A_3) *data 1*	(A_2) *data 2*	(A_1) *data 3*	(A_0) *data 4*	(Y_1) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1



Exercise 5.64

(a) 8 LEs (see next page for figure)

LE 1

	(A_2)	(A_1)	(A_0)	(Y_0)
data 1	*data 2*	*data 3*	*data 4*	LUT output
X	0	0	0	1
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 2

	(A_2)	(A_1)	(A_0)	(Y_1)
data 1	*data 2*	*data 3*	*data 4*	LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 3

	(A_2)	(A_1)	(A_0)	(Y_3)
data 1	*data 2*	*data 3*	*data 4*	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	1
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 4

	(A_2)	(A_1)	(A_0)	(Y_3)
data 1	*data 2*	*data 3*	*data 4*	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	1
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 5

	(A_2)	(A_1)	(A_0)	(Y_4)
data 1	*data 2*	*data 3*	*data 4*	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 6

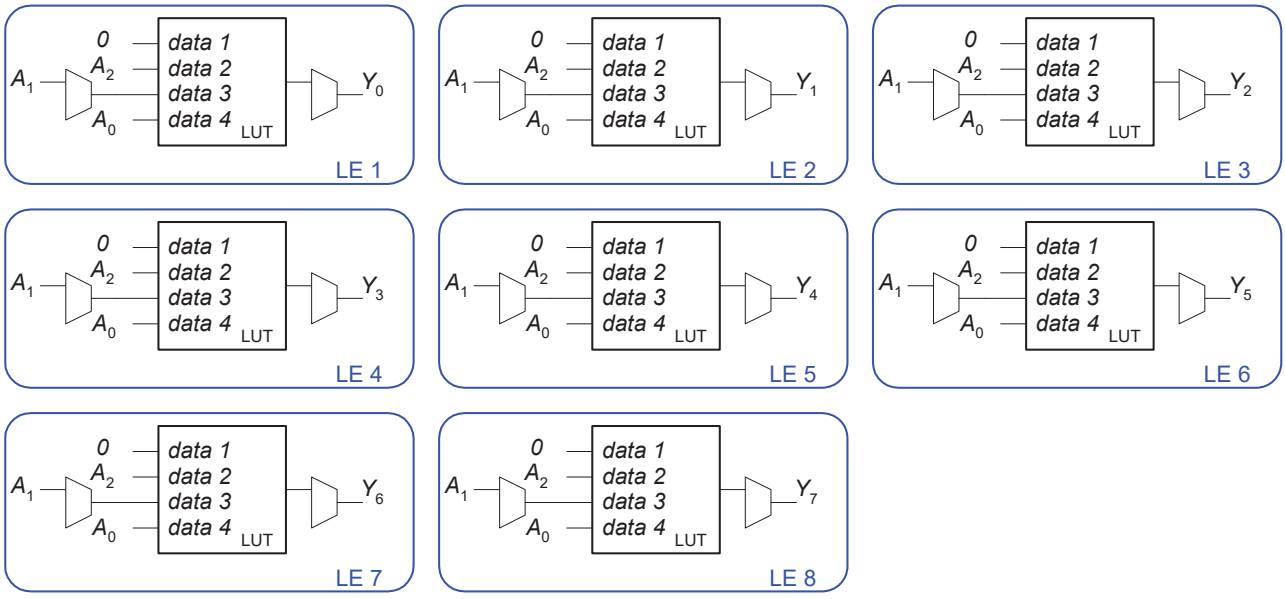
	(A_2)	(A_1)	(A_0)	(Y_5)
data 1	*data 2*	*data 3*	*data 4*	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	1
X	1	1	0	0
X	1	1	1	0

LE 7

	(A_2)	(A_1)	(A_0)	(Y_6)
data 1	*data 2*	*data 3*	*data 4*	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
X	1	1	1	0

LE 8

	(A_2)	(A_1)	(A_0)	(Y_7)
data 1	*data 2*	*data 3*	*data 4*	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	1



(b) 8 LEs (see next page for figure)

LE 7

(A_2) *data 1*	(A_1) *data 2*	(A_0) *data 3*	*data 4*	(Y_7) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	1

LE 6

(A_2) *data 1*	(A_1) *data 2*	(A_0) *data 3*	*data 4*	(Y_6) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
X	1	1	1	0

LE 5

(A_2) *data 1*	(A_1) *data 2*	(A_0) *data 3*	*data 4*	(Y_5) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	1
X	1	1	0	0
X	1	1	1	0

LE 4

(A_2) *data 1*	(A_1) *data 2*	(A_0) *data 3*	*data 4*	(Y_4) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 3

(A_2) *data 1*	(A_1) *data 2*	(A_0) *data 3*	*data 4*	(Y_3) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	1
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 2

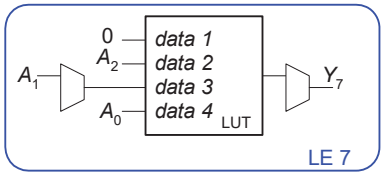
(A_2) *data 1*	(A_1) *data 2*	(A_0) *data 3*	*data 4*	(Y_2) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	1
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

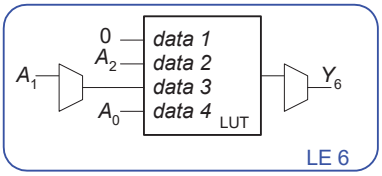
LE 1

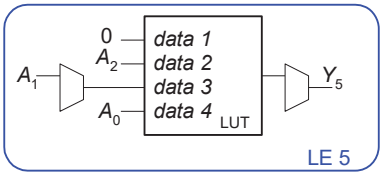
(A_2) *data 1*	(A_1) *data 2*	(A_0) *data 3*	*data 4*	(Y_1) LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

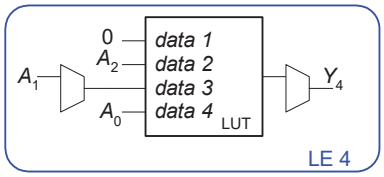
LE 0

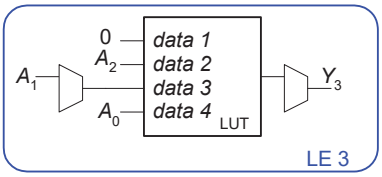
(A_2) *data 1*	(A_1) *data 2*	(A_0) *data 3*	*data 4*	(Y_0) LUT output
X	0	0	0	1
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

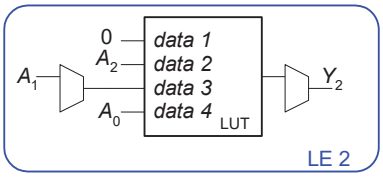


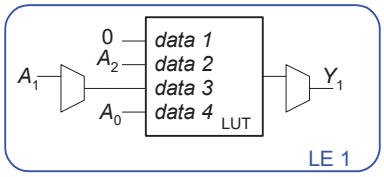


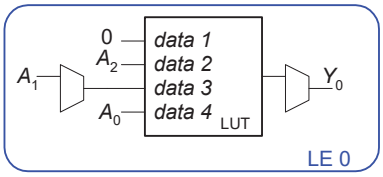












(c) 6 LEs (see next page for figure)

LE 1

data 1	data 2	(A_0) data 3	(B_0) data 4	(S_0) LUT output
X	X	0	0	0
X	X	0	1	1
X	X	1	0	1
X	X	1	1	1

LE 2

(A_0) data 1	(B_0) data 2	(A_1) data 3	(B_1) data 4	(S_1) LUT output
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

LE 3

(A_0) data 1	(B_0) data 2	(A_1) data 3	(B_1) data 4	(C_1) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

LE 4

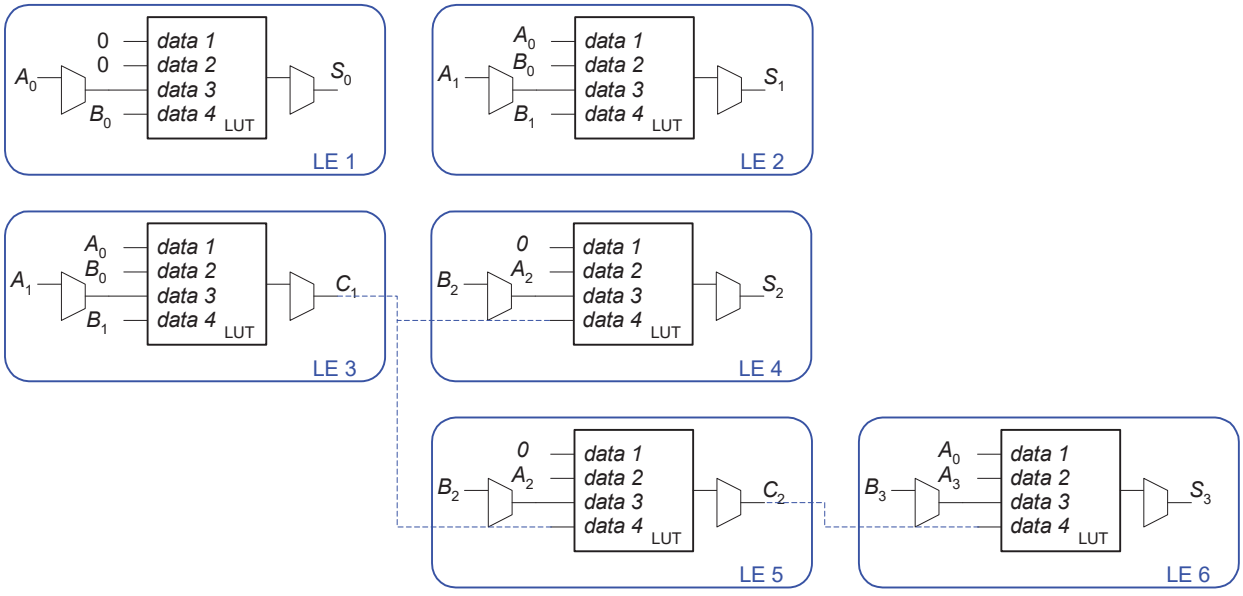
data 1	(A_2) data 2	(B_2) data 3	(C_1) data 4	(S_2) LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	1
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	1

LE 5

data 1	(A_2) data 2	(B_2) data 3	(C_1) data 4	(C_2) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	1
X	1	0	0	0
X	1	0	1	1
X	1	1	0	1
X	1	1	1	1

LE 6

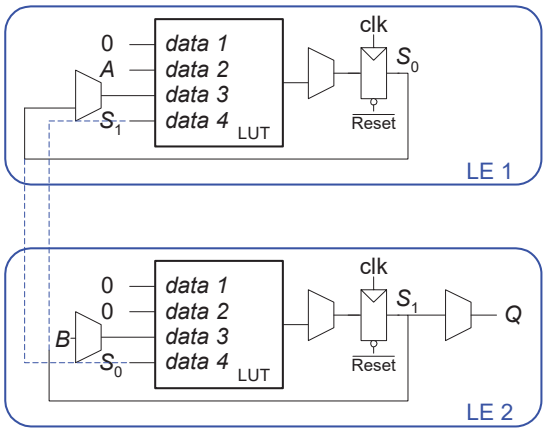
data 1	(A_3) data 2	(B_3) data 3	(C_2) data 4	(S_3) LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	1
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	1



(d) 2 LEs

	(A)	(S_0)	(S_1)	(S_0')
data 1	*data 2*	*data 3*	*data 4*	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

		(B)	(S_0)	(S_1')
data 1	*data 2*	*data 3*	*data 4*	LUT output
X	X	0	0	0
X	X	0	1	0
X	X	1	0	0
X	X	1	1	1

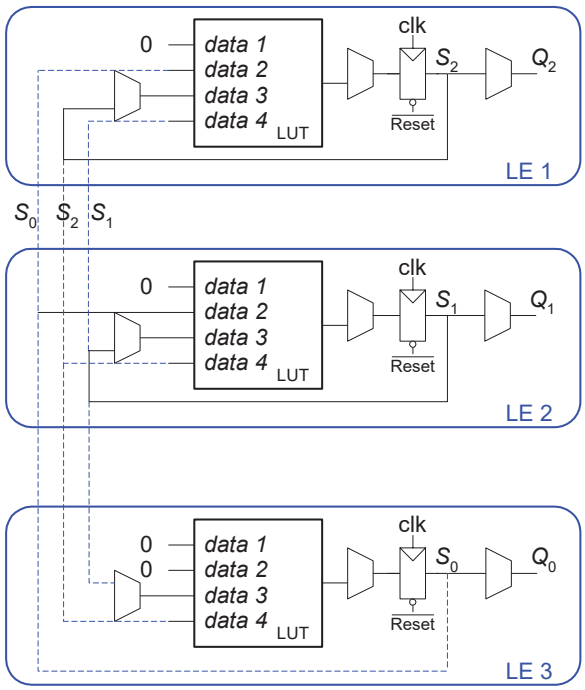


(e) 3 LEs

	(S_0)	(S_2)	(S_1)	(S_2')
data 1	*data 2*	*data 3*	*data 4*	LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	1
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
X	1	1	1	1

	(S_0)	(S_1)	(S_2)	(S_1')
data 1	*data 2*	*data 3*	*data 4*	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	1
X	0	1	1	1
X	1	0	0	1
X	1	0	1	0
X	1	1	0	1
X	1	1	1	0

		(S_1)	(S_2)	(S_0')
data 1	*data 2*	*data 3*	*data 4*	LUT output
X	X	0	0	1
X	X	0	1	0
X	X	1	0	0
X	X	1	1	1



Exercise 5.65

(a) 5 LEs (2 for next state logic and state registers, 3 for output logic)

(b)

$t_{pd} = t_{\text{pd\_LE}} + t_{\text{wire}}$
$= (381+246)$ ps
$= 627$ ps

$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}}$
$\geq [199 + 627 + 76]$ ps
$= 902$ ps

$f = 1 / 902$ ps = **1.1 GHz**

(c)

First, we check that there is no hold time violation with this amount of clock skew.

$t_{cd\_LE} = t_{pd\_LE} = 381$ ps
$t_{cd} = t_{cd\_LE} + t_{wire} = 627$ ps

$t_{\text{skew}} < (t_{ccq} + t_{cd}) - t_{\text{hold}}$
$< [(199 + 627) - 0]$ ps
< 826 ps

3 ns is less than 826 ps, so there is no hold time violation.

Now we find the fastest frequency at which it can run.

$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} + t_{\text{skew}}$
$\geq [0.902 + 3]$ ns
$= 3.902$ ns

$f = 1 / 3.902$ ns = **256 MHz**

Exercise 5.66

(a) 2 LEs (1 for next state logic and state register, 1 for output logic)

(b) Same as answer for Exercise 5.57(b)

(c) Same as answer for Exercise 5.57(c)

Exercise 5.67

First, we find the cycle time:

$T_c = 1/f = 1/100$ MHz = 10 ns

$T_c \geq t_{pcq} + Nt_{\text{LE+wire}} + t_{\text{setup}}$

10 ns $\geq [0.199 + N(0.627) + 0.076]$ ns

Thus, $N \leq 15.5$

The maximum number of LEs on the critical path is **15**.

With at most one LE on the critical path and no clock skew, the fastest the FSM will run is:

$T_c \geq [0.199 + 0.627 + 0.076]$ ns

≥ 0.902 ns

$f = 1 / 0.902$ ns = **1.1 GHz**

Question 5.1

$(2^N\text{-}1)(2^N\text{-}1) = 2^{2N} - 2^{N+1} + 1$

Question 5.2

A processor might use BCD representation so that decimal numbers, such as 1.7, can be represented exactly.

Question 5.3

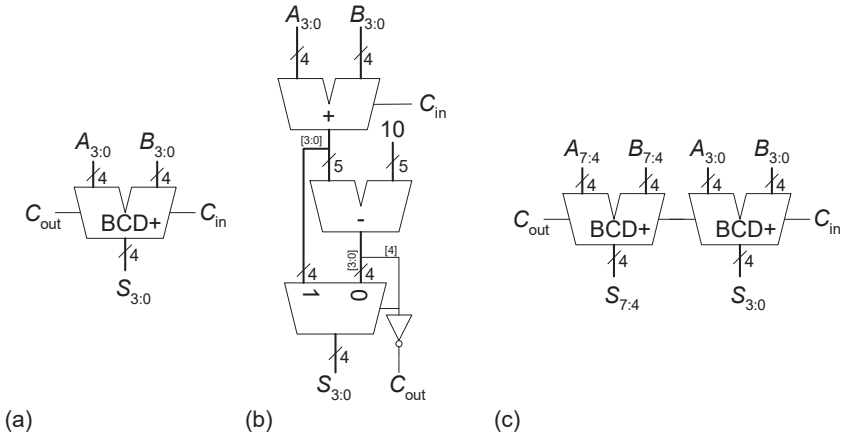


FIGURE 5.22 BCD adder: (a) 4-bit block, (b) underlying hardware, (c) 8-bit BCD adder

(continued from previous page)

SystemVerilog

```
module bcdadd_8(input  logic [7:0] a, b,
                input  logic       cin,
                output logic [7:0] s,
                output logic       cout);

  logic c0;

  bcdadd_4 bcd0(a[3:0], b[3:0], cin, s[3:0], c0);
  bcdadd_4 bcd1(a[7:4], b[7:4], c0, s[7:4], cout);

endmodule


module bcdadd_4(input  logic [3:0] a, b,
                input  logic       cin,
                output logic [3:0] s,
                output logic       cout);

  logic [4:0] result, sub10;

  assign result = a + b + cin;
  assign sub10 = result - 10;

  assign cout = ~sub10[4];
  assign s = sub10[4] ? result[3:0] : sub10[3:0];

endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity bcdadd_8 is
  port(a, b: in  STD_LOGIC_VECTOR(7 downto 0);
       cin:  in  STD_LOGIC;
       s:    out STD_LOGIC_VECTOR(7 downto 0);
       cout: out STD_LOGIC);
end;

architecture synth of bcdadd_8 is
  component bcdadd_4
  port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
       cin:  in  STD_LOGIC;
       s:    out STD_LOGIC_VECTOR(3 downto 0);
       cout: out STD_LOGIC);
  end component;
  signal c0: STD_LOGIC;
begin

  bcd0: bcdadd_4
    port map(a(3 downto 0), b(3 downto 0), cin, s(3
downto 0), c0);
  bcd1: bcdadd_4
     port map(a(7 downto 4), b(7 downto 4), c0, s(7
downto 4), cout);

end;


library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity bcdadd_4 is
  port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
       cin:  in  STD_LOGIC;
       s:    out STD_LOGIC_VECTOR(3 downto 0);
       cout: out STD_LOGIC);
end;

architecture synth of bcdadd_4 is
signal  result,  sub10,  a5,  b5:  STD_LOGIC_VECTOR(4
downto 0);
begin
  a5 <= '0' & a;
  b5 <= '0' & b;
  result <= a5 + b5 + cin;
  sub10 <= result - "01010";

  cout <= not (sub10(4));
  s <= result(3 downto 0) when sub10(4) = '1'
       else sub10(3 downto 0);

end;
```

CHAPTER 6

Exercise 6.1

(a) **Regularity supports simplicity:**

- Each instruction has a 7-bit opcode in the 7 least significant bits (lsb's) of the instruction which makes the hardware for decoding the instruction simpler.
- RISC-V has four instruction formats (R-, I-, S/B-, and U/J-type) that allows for some regularity among instructions, which then leads to simpler decoder hardware.
- Immediate bit locations are consistent across instruction formats, which minimizes wire routing and the number of multiplexers needed.
- Each instruction is 32 bits, making decoding hardware simpler.

(b) **Make the common case fast:**

- Only simple, commonly used instructions are included, which results in simpler, and thus faster, hardware.
- Registers make the access to recently used variables fast.
- Most instructions require all 32 bits of an instruction, so all instructions are 32 bits (even though some would have an advantage of a larger instruction size and others a smaller instruction size). The instruction size is chosen to make the common instructions fast.

(c) **Smaller is faster:**

- RISC-V includes a small number of commonly used instructions, which keeps the hardware small and fast.
- The instruction size is kept small to allow for fast instruction fetching and simpler decoder logic, which is, thus, faster.
- The register file only has 32 registers, which allows for fast access to them.

(d) **Good design demands good compromises:**

- RISC-V includes four instruction formats, instead of just one, which accommodates different instruction needs while still allowing for some regularity between instruction formats.
- Since RISC-V is a RISC architecture, only simple instructions are supported. However, some more complex pseudoinstructions are provided to the programmer for convenience. For example, some pseudoinstructions, such as `li`, can translate into multiple RISC-V instructions.
- Although memory access is not as fast as register access, it is a required compromise to allow for complex programs.

Exercise 6.2

Yes, it is possible to design a computer architecture without a register set. It would require using calls to memory for all instructions. It would use memory as if it were a very large register set. For example:

```
sub 0xA, 0xB, 0xC
```

The instruction set would operate the same way except on locations in memory instead of using registers. Memory address 0xA would receive the result of the value in address 0xB minus the value in address 0xC.

Advantages:

- No load or store instructions would be required as all operations would pull from and write to memory directly.
- As a result of a smaller instruction set, hardware could be further simplified.

Disadvantages:

- Either memory would have to be small or the processor would be slow.
- Instruction sizes would have to be significantly larger to accommodate listing an entire range of potential memory addresses. Or instructions might require that a source operand is also a destination operand (Ex: `sub 0xA, 0xB` would perfrom: 0xA = 0xA – 0xB).

Exercise 6.3

(a)

Character	h	e	l	l	o		t	h	e	r	e	NULL
ASCII	68	65	6C	6C	6F	20	74	68	65	72	65	00

(b)

Character	b	a	g	o	c	h	i	p	s	NULL
ASCII	62	61	67	6F	63	68	69	70	73	00

(c)

Character	T	o		t	h	e		r	e	s	c	u	e	!	NULL
ASCII	54	6F	20	74	68	65	20	72	65	73	63	75	65	21	00

* Recall that in C, the null character (0x00) specifies the end of a string.

Exercise 6.4

(a)

Character	C	o	o	l	NULL
ASCII	43	6F	6F	6C	00

(b)

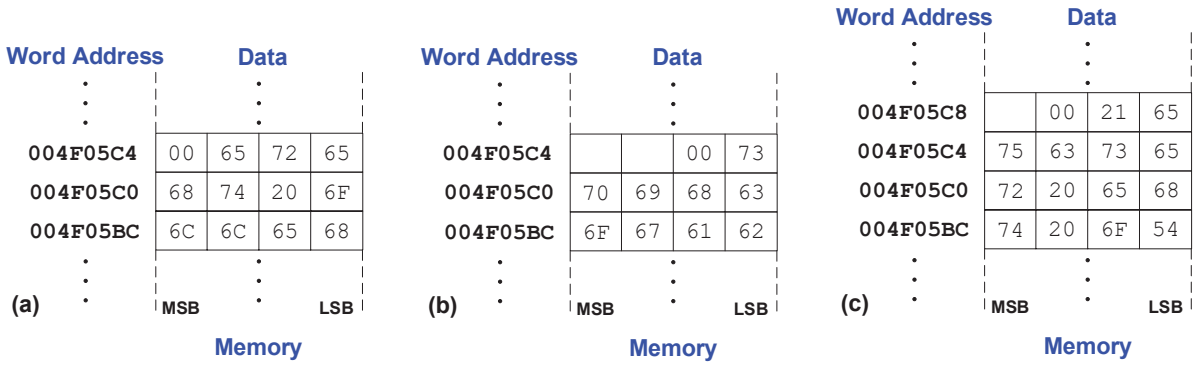
Character	R	I	S	C	-	V	NULL
ASCII	52	49	53	43	2D	56	00

(c)

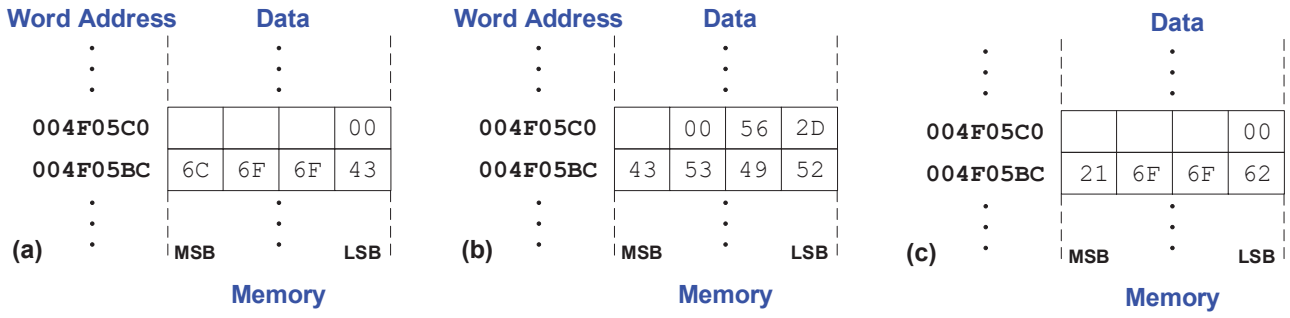
Character	b	o	o	!	NULL
ASCII	62	6F	6F	21	00

* Recall that in C, the null character (0x00) specifies the end of a string.

Exercise 6.5



Exercise 6.6



Exercise 6.7

```
or   s3, s4, s5
xori s3, s3, -1
```

Exercise 6.8

```
and  s3, s4, s5
```

```
xori s3, s3, -1
```

Exercise 6.9

a.)
```
        # a0 = g, a1 = h
        bge  a1, a0, else  # do else if (g <= h)
        addi a0, a0, 1     # g = g + 1
        j done             # jump past else block
else:   addi a1, a1, -1    # h = h - 1
done:
```

b.)
```
        # a0 = g, a1 = h
        blt  a1, a0, else  # do else if (g > h)
        addi a0, zero, 0   # g = 0
        j done             # jump past else block
else:   addi a1, zero, 0   # h = 0
done:
```

Exercise 6.10

(a)
```
        # a0 = g, a1 = h
        blt  a0, a1, else  # do else if (g < h)
        add  a0, a0, a1    # g = g + h
        j    done          # jump past else block
else:   sub  a0, a0, a1    # g = g - h
done:
```

(b)
```
        # a0 = g, a1 = h
        bge  a0, a1, else # do else if (g >= h)
        addi a1, a1, 1    # h = h + 1
        j    done         # jump past else block
else:   slli a1, a1, 1    # h = h * 2
done:
```

Exercise 6.11

```
# t1 = array1 base adr, t2 = array2 base adr
      addi t0, zero, 0      # t0(i) = 0
      addi t3, zero, 100    # t3 = 100
for:  bge  t0, t3, done     # if i >= 100, array fully copied
      slli t4, t0, 2        # i *= 4
      add  t5, t1, t4       # t5 = address of array1[i]
      add  t4, t2, t4       # t4 = address of array2[i]
      lw   t6, 0(t4)        # t6 = array2[i]
      sw   t6, 0(t5)        # array1[i] = array2[i]
      addi t0, t0, 1        # i += 1
      j    for              # loop
done:                       # end of code snippet
```

Exercise 6.12

```
# s0 = i, t3 = temp array base address
      addi s0, zero, 1      # i = 0
      addi t0, zero, 100    # t0 = 100
for:  bge  s0, t0, done     # if i >= 100 then done
      slli t2, s0, 2        # t2 = i*4
      add  t2, t2, t3       # t2 = address of temp[i]
      lw   t4, 0(t2)        # t4 = temp[i]
      slli t4, t4, 7        # t4 = temp[i] * 128
      sw   t4, 0(t2)        # temp[i] = t4
      addi s0, s0, 1        # i = i+1
      j    for              # repeat
done:                       # end of code snippet
```

Exercise 6.13

```
(a)   addi s7, zero, 29   # s7 = 29
(b)   addi s7, zero, -214 # s7 = -214
(c)   lui  s7, 0xFFFFF    # s7 = 0xFFFFF000
      addi s7, s7, 0x449  # s7 = 0xFFFFF449 = -2999
(d)   lui  s7, 0xABCDE    # s7 = 0xABCDE000
(e)   lui  s7, 0xEDCBA    # s7 = 0xEDCBA000
      addi s7, s7, 0x123  # s7 = 0xEDCBA123
(f)   lui  s7, 0xEEEEF    # s7 = 0xEEEEF000
      addi s7, s7, -85    # s7 = 0xEEEEEFAB (-85 = 0xFAB)
```

Exercise 6.14

```
(a)   addi s7, zero, 47   # s7 = 47
(b)   addi s7, zero, -349 # s7 = -349
(c)   lui  s7, 0x00001    # s7 = 0x00001000
      addi s7, s7, 0x4D0  # s7 = 0x000014D0 = 5328
(d)   lui  s7, 0xBBCCD    # s7 = 0xBBCCD000
(e)   lui  s7, 0xFEEBC    # s7 = 0xFEEBC000
      addi s7, s7, 0x789  # s7 = 0xFEEBC789
(f)   lui  s7, 0xCCAAC    # s7 = 0xCCAAC000
      addi s7, s7, -1621  # s7 = 0xCCAAB9AB (-1621 = 0x9AB)
```

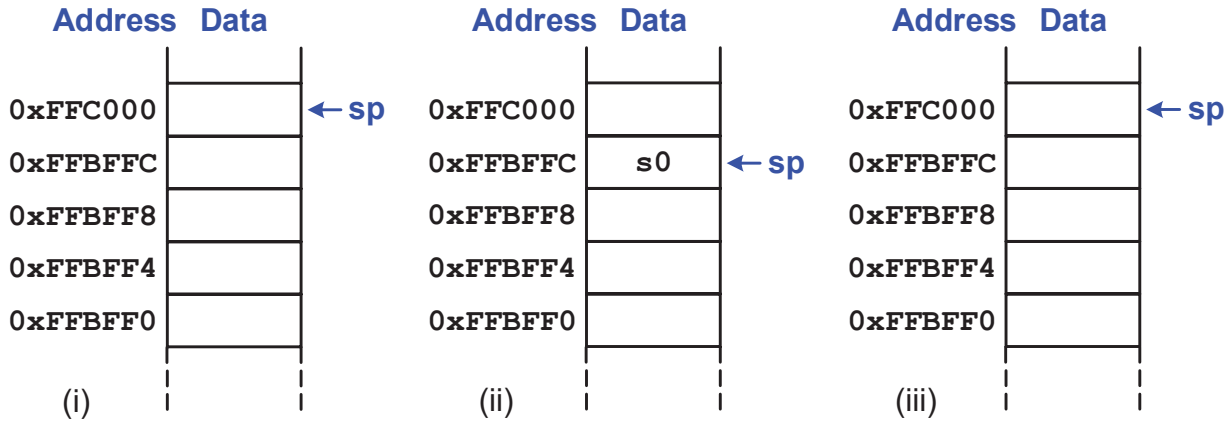
Exercise 6.15

```
int find42(int array[], int size) {
  for (int i = 0; i < size; i++) {
    if (array[i] == 42)
      return i;
    }
  return -1;
}
```

Exercise 6.16

(a)
```
# a0 = dst array base address
# a1 = src array base address
# s0 = i. Note that the Exercise prompt should say:
# The instructions should say: "Use s0 for i."
strcpy:
  addi sp, sp, -4
  sw   s0, 0(sp)       # save s0 on the stack
  addi s0, zero, 0     # i = 0
loop:
  add  t1, a1, s0      # t1 = src[i] address
  add  t2, a0, s0      # t2 = dst[i] address
  lb   t3, 0(t1)       # t3 = src[i]
  sb   t3, 0(t2)       # dst[i] = src[i]
  beq  t3, zero, done  # if src[i] = 0, we are done
  addi s0, s0, 1       # i++
  j    loop
done:
  lw   s0, 0(sp)       # restore s0 from the stack
  addi sp, sp, 4       # restore stack pointer
  jr   ra              # return
```

(b) The stack (i) before, (ii) during, and (iii) after the `strcpy` function call.



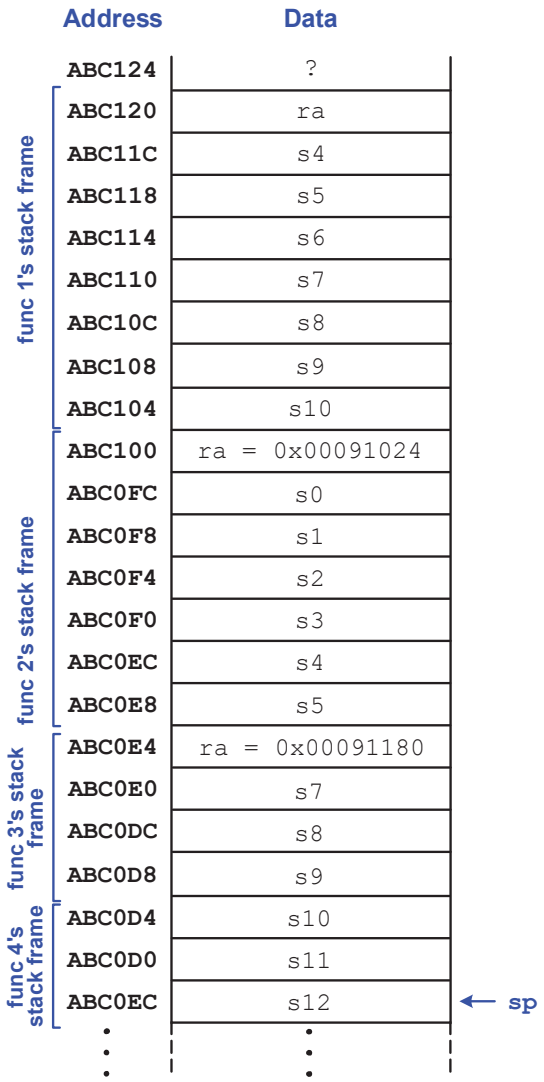
Exercise 6.17

```
find42:
  addi t0, zero, 42      # t0 = 42
  addi t1, zero, 0       # i = 0
loop:
  bge  t1, a1, notFound  # loop if i < size (if not, end reached
                         # and 42 not found)
  slli t2, t1, 2         # i * 4 (to find byte offset)
  add  t2, a0, t2        # t2 = address of array[i]
  lw   t2, 0(t2)         # t2 = array[i]
  beq  t2, t0, found     # if array[i] == 42, go to found
  addi t1, t1, 1         # i++ (i = i + 1)
  j    loop              # next iteration
found:
  add  a0, zero, t1      # a0 = i
```

```
  jr   ra                # return
notFound:
  addi a0, zero, -1      # a0 = -1
  jr   ra                # return
```

Exercise 6.18

(a)
`func1`: 8 words deep (`s4–s10, ra`)
`func2`: 7 words deep (`s0–s5, ra`)
`func3`: 4 words deep (`s7–s9, ra`)
`func4`: 3 words deep (`s10–s12`)
(b) The stack after `func4` is called:



Exercise 6.19

(a) *fib(0)* = 0, *fib(-1)* = 1

(b) **High Level Code**

```
int fib(int n) {
  int i;
  int current = 0;              // fib(i) – initialized to fib(0)
  int prev = 1;                 // fib(i-1) – initialized to fib(-1)

  for (i = 1; i <= n; i++){
    current = current + prev;   // fib(i) = fib(i-1) + fib(i-2)
    prev = current - prev;      // update prev:
                                // fib(i-1) = fib(i) - fib(i-2)
  }
  return current;               // return fib(n)
}
```

(c) RISC-V Assembly Code

```
  addi a0, zero, 9          # n = 9
  jal  fib                  # call fib(n), where n = 9
  ...                       # code after function call

fib:
  addi sp, sp, -12          # make room on stack for 3 registers
  sw   s0, 8(sp)            # save s0 on stack
  sw   s1, 4(sp)            # save s1 on stack
  sw   s2, 0(sp)            # save s2 on stack
  addi s0, zero, 0          # current = 0 (fib(i))
  addi s1, zero, 1          # prev = 1 (fib(i-1))
  addi s2, zero, 1          # i = 1
for:
  blt  a0, s2, result       # if i > n then loop ends
  add  s0, s0, s1           # fib(i) = fib(i - 1) + fib(i - 2)
  sub  s1, s0, s1           # fib(i - 1) = fib(i) - fib(i - 2)
  addi s2, s2, 1            # i = i + 1
  j    for                  # repeat loop
result:
  add  a0, zero, s0         # return fib(n) (put fib(n) in a0)
  lw   s0, 8(sp)            # restore registers from stack
  lw   s1, 4(sp)
  lw   s2, 0(sp)
  addi sp, sp, 12           # restore stack pointer
  jr   ra                   # return
```

Exercise 6.20

(a) 1* 2 * 3 * 4 * 5 = **120** (= 5! as expected)

(b) **2** – Crash by causing the stack to grow beyond the dynamic data segment because the program will jump (`jr ra`) repeatedly to 0x8528, because `ra` wasn't saved/restored. Thus, program will repeatedly increment the stack pointer by 8 until it shrinks beyond the dynamic data segment.

(c) (1) **3** – Because `n` (i.e., `a0`) isn't restored after the function call (into `t1`), the program produces the wrong result. When called with `n` = 5, it will produce: `t1*t1*t1*t1*t1*1`, which depends on the most recent value placed in `t1`.
(2) **3** – Removing instruction 0x8518 prevents the last stack frame from being deallocated. So, each stack frame will be off by 1. The incorrect return value is: `a0` = 1 * 1 * 2 * 3 * 4 = **24**. Additionally, the caller (i.e., function that called `factorial(5)`) will not be able to access its stack frame correctly because `sp` was not restored.
(3) **3** – Removing instruction 0x8530 prevents the stack frames from being deallocated. So, each recursive function (after `factorial(1)`) will read the stack frame of `factorial(2)`. The incorrect return value is: `a0` = 1 * 2 * 2 * 2 * 2 = **16**. Additionally, the caller (i.e., function that called `factorial(5)`) will not be able to access its stack frame correctly because `sp` was not restored.

Exercise 6.21

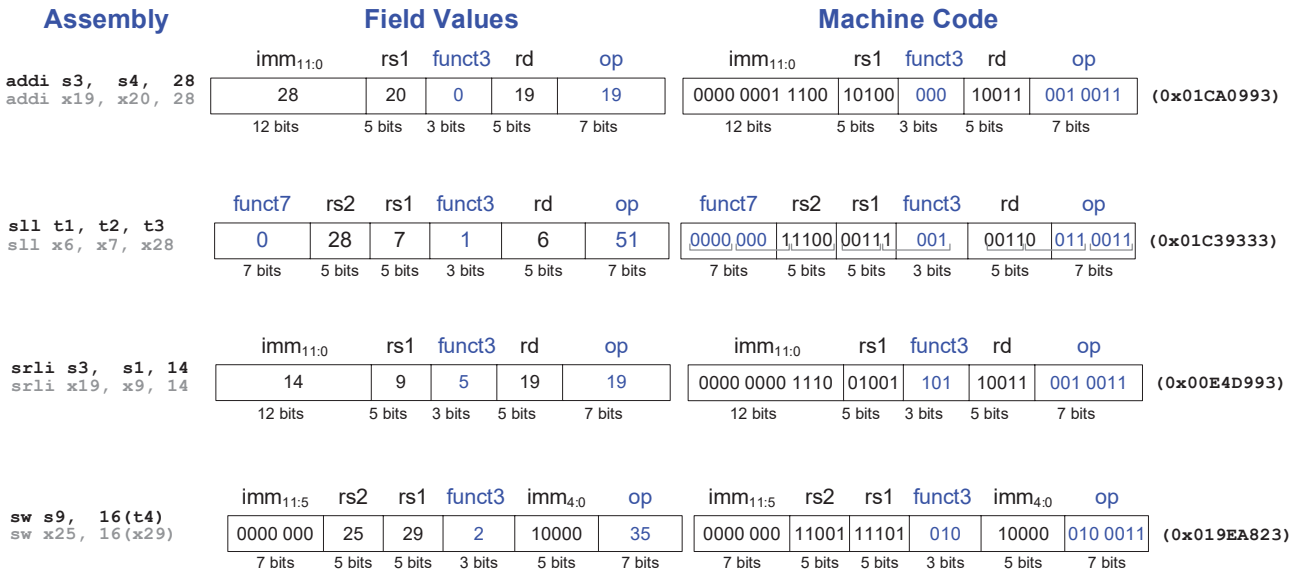
(a) By the time the program reaches the loop label, register `a0` will hold the value **19**, which is (5+5) + (3+3+3) = 2a + 3b = 19, as intended.
(b) 3 – The program will produce an incorrect value in register `a0`. The store word instruction (`sw a0, 0xC(sp)`) puts the original value of `a0` (a = 5) on the stack. When the instruction `at` 0x8030 tries to load that value, `t0` = `an unknown value (i.e., whatever was in the stack memory location before). The value in a0 can't be determined, but it will be: 5 + t0 + 9 = 14 + t0.`
(c)
(i) 3 – The program will produce an incorrect value in register `a0`. The same explanation as part (b) applies. You can't determine what is in `a0` because you would need to know the previous value of `t0`.
(ii) 2 – The program would crash due to the stack growing beyond the dynamic data segment. This is due to instruction 0x8040 being removed. `ra` is no longer restored from the stack and retains its current value, which is 0x8030. It now repeatedly executes the instructions from 0x8030 to 0x8048. Instruction 0x8044 increments the stack pointer (`sp`) so this occurs until the stack pointer increases beyond the dynamic data segment.
(iii) 4 – The program would run correctly despite the deleted lines. However, the value of register `s4` before the call to `f` would not be restored. This doesn't affect the return value of `f(5, 3)`, but if the caller (in this case, `test`) needs to use `s4` after the function call to `f`, the program retrieves an incorrect value.
(iv) 3 – The program will produce an incorrect value in register `a0`. The same explanation as in part (b) applies.
(v) 3 – The program will produce an incorrect value in register `a0`. `s4` = 5 wouldn't be saved to the stack during the first iteration of `g`. Because `s4` becomes 3 in the `g` function, the 3b part of 2a + 3b would still be carried out correctly. During the last iteration of `g`, `s4` would remain 3 so during the add instruction at 0x8038 the operation would be (a + b) instead of (a + a) as intended. The final result in register

`a0` would be **17** (i.e., 5 + **3** + 3*3 = 17).
(vi) 4 – The program would run correctly despite the deleted lines. The explanation is the same as in part (iii) above.
(vii) 2 – The program would crash due to the stack growing beyond the dynamic data segment. This is similar to the scenario for ii). The return address of 0x8030 would never be stored to register `ra`. After the first iteration of `g`, `ra` = 0x8070. With the instruction at 0x8080 (`jr ra`), instructions 0x8070 – 0x8080 would loop infinitely. Because the instruction at address 0x807C increments the stack pointer, `sp` continues to increase until it grows beyond the dynamic data segment. Then it would crash.

Exercise 6.22

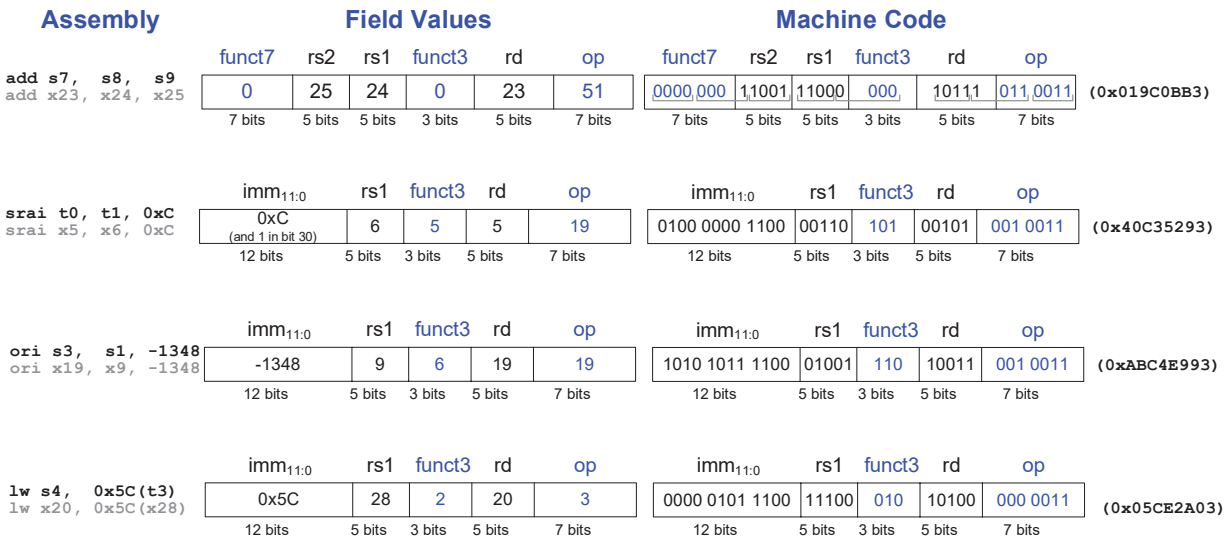
Instruction	Machine Code
`addi s3, s4, 28`	0x01CA0993
`sll t1, t2, t3`	0x01C39333
`srli s3, s1, 14`	0x00E4D993
`sw s9, 16(t4)`	0x019EA823

Supporting work:



Exercise 6.23

Instruction	Machine Code
`add s7, s8, s9`	0x019C0BB3
`srai t0, t1, 0xC`	0x40C35293
`ori s3, s1, -1348`	0xABC4E993
`lw s4, 0x5C(t3)`	0x05CE2A03



Exercise 6.24

(a) Instruction	(b) Type
`addi s3, s4, 28`	I
`srli s3, s1, 14`	I
`sw s9, 16(t4)`	S

(c)
`addi`: 28: 12-bit: 0000 0001 1100 (0x01C)
0000 0000 0000 0000 0000 0000 0001 1100 = **0x0000001C** (32-bit)

`srli`: 14: 5-bit: 0 1110 = 0x0E – not extended
`sw`: 16: 12-bit: 0000 0001 0000 = 0x010
0000 0000 0000 0000 0000 0000 0001 0000 = **0x00000010** (32-bit)

Exercise 6.25

(a) Instruction	(b) Type
`srai t0, t1, 0xC`	I
`ori s3, s1, -1348`	I
`lw s4, 0x5C(t3)`	I

(c)
`srai` – 0xC: 5-bit: 0 1100 – 0x0C (5-bit) – not extended

`ori` – 1348: 12-bit: 1010 1011 1100 = 0xABC
1111 1111 1111 1111 1111 1010 1011 1100 = **0xFFFFFABC** (32-bit)

`lw` – 0x5C: 12-bit: 0000 0101 1100 = 0x05C
0000 0000 0000 0000 0000 0000 0101 1100 = **0x0000005C** (32-bit)

Exercise 6.26

(a)
```
0x01800513         addi a0, zero, 24    # A = 24
0x00300593         addi a1, zero, 3     # B = 3
0x00000393         addi t2, zero, 0     # i = 0
0x00058E33         add  t3, a1, zero    # temp = B = 3
0x01C54863  loop:  blt  a0, t3, done    # if A < temp, done
0x00138393         addi t2, t2, 1       # i++
0x00BE0E33         add  t3, t3, a1      # temp += B
0xFF5FF06F         j    loop            # repeat loop
0x00038533  done:  add a0, t2, zero     # result = i
```

(b) An equivalent program in C would be (assuming `A` = `a0`, `B` = `a1`, `i` = `t2`, `temp` = `t3`, and result = `a0` once the assembly program reaches the `done` label:

```
int A = 24;
int B = 3;
int i;
int result;

int temp = B;
for (i = 0; A >= temp; i++)
  temp += B;

result = i;
```

(c) This program performs integer division: `result` = A/B.

Exercise 6.27

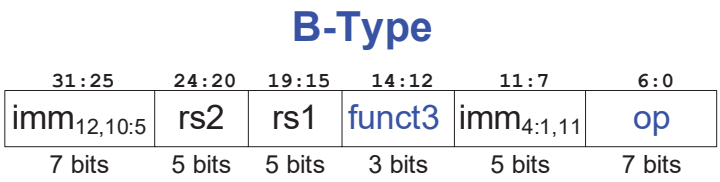
(a)
```
0x01F00393       addi t2,   zero, 31   # t2 = 31
0x00755E33  L1:  srl  t3,   a0,   t2   # t3 = a0 >> t2
0x001E7E13       andi t3,   t3,   1    # t3 = lsb of t3
0x01C580A3       sb   t3,   1(a1)      # a1[1] = t3
0x00158593       addi a1,   a1,   1    # a1++
0xFFF38393       addi t2,   t2,  -1    # t2--
0xFE03D6E3       bge  t2,   zero, L1   # if t2 >= 0, repeat
0x00008067       jr   ra               # return
```

(b)
```
# a0 = val, a1 = base address of array
# t2 = shiftAmt, t3 = tmp
void decToBin(int val, char array[]){
  int shiftAmt = 31;
  char tmp;
  int i = 1;

  do {
    tmp = (val >> shiftAmt);
    tmp = tmp & 1;
    array[i] = tmp;
```

```
        i++;
        shiftAmt--;
      } while (shiftAmt >= 0);
    }
```

(c) This program takes in a 32-bit number, `a0`, and converts it from decimal to binary. The result is stored as characters in an array that is pointed to by `a1` from index 1 to 32, i.e., from `array[1]` to `array[32]`.

Exercise 6.28



(a) **beq t4, zero, Loop:** 0xA00C - 0xA000 = 0xC: branch 0xC bytes forward:
Branch offset = $imm_{12:0}$ = 0xC: **0 0000 0000 1100**
rs1 = t4 = x29 (11101), rs2 = zero = x0 (00000)
funct3 = 000 (beq), op = 1100011 (branch)

$imm_{12,10:5}$	rs2	rs1	funct3	imm4:1,11	op
0000000	00000	11101	000	**01100**	1100011

= **0x000E8663**

(b) **bne s5, a1, L1:** 0x80174C – 0x801000 = 0x74C: branch 0x74C bytes forward.
Branch offset = $imm_{12:0}$ = 0x74C: **0 0111 0100 1100**
rs1 = s5 = x21 (10101b), rs2 = a1 = x11 (01011b)
funct3 = 001 (bne), op = 1100011 (branch)

$imm_{12,10:5}$	rs2	rs1	funct3	imm4:1,11	op
0111010	10101	01011	001	**01100**	1100011

= **0x75559663**

(c) **blt s1, s2, Back:** 0xD000 – 0xC10C = 0xEF4: branch 0xEF4 bytes back.
Flip sign of: 0xEF4 (0 1110 1111 0100): 1 0001 0000 1011 + 1 = 1 0001 0000 1100
Branch offset = $imm_{12:0}$ = 0x110C = **1 0001 0000 1100**
rs1 = s1 = x9 (01001b), rs2 = s2 = x18 (10010b)
funct3 = 100 (blt), op = 1100011 (branch)

$imm_{12,10:5}$	rs2	rs1	funct3	imm4:1,11	op
1001000	10010	01001	100	**01100**	1100011

= **0x9124C663**

(d) **bge t4, t6, L2:** 0x1031AA4 – 0x1030AAC = 0xFF8: branch 0xFF8 bytes forward:

Branch offset = $imm_{12:0}$ = **0 1111 1111 1000**
rs1 = t4 = x29 (11101b), rs2 = t6 = x31 (11111b)
funct3 = 101 (bge), op = 1100011 (branch)

$imm_{12,10:5}$	rs2	rs1	funct3	imm4:1,11	op
0111111	11111	11101	101	**1100**1	1100011

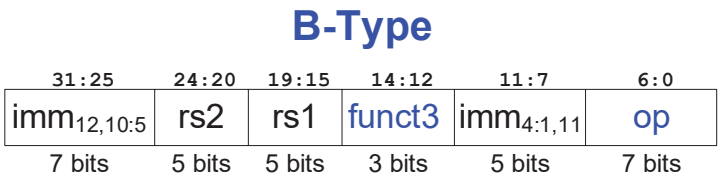
= **0x7FFEDCE3**

(e) **beq s3, s7, L3:** 0xBC09000 – 0xBC08004 = 0xFFC: branch 0xFFC bytes backward:
Flip sign of: 0xFFC (0 1111 1111 1100): 1 0000 0000 0011 + 1 = 1 0000 0000 0100
Branch offset = $imm_{12:0}$ = 0x1004: **1 0000 0000 0100**
rs1 = s3 = x19 (10011b), rs2 = s7 = x23 (10111b)
funct3 = 000 (beq), op = 1100011 (branch)

$imm_{12,10:5}$	rs2	rs1	funct3	imm4:1,11	op
1000000	10111	10011	000	**0010**0	1100011

= **0x81798263**

Exercise 6.29



a) **blt t4, s3, Loop:** 0xAA00E130 – 0xAA00E124 = 0xC: branch 0xC bytes forward.
Branch offset = $imm_{12:0}$ = **0 0000 0000 1100**
rs1 = t4 = x29 (11101b), rs2 = s3 = x19 (10011b)
funct3 = 100 (blt), op = 1100011 (branch)

$imm_{12,10:5}$	rs2	rs1	funct3	imm4:1,11	op
0000000	10011	11101	100	**0110**0	1100011

= **0x013EC663**

b) **bge t1, t2, L1:** 0xC090174C – 0xC0901000 = 0x74C: branch 0x74C bytes forward.
Branch offset = $imm_{12:0}$ = **0 0111 0100 1100**
rs1 = t1 = x6 (00110b), rs2 = t2 = x7 (00111b)
funct3 = 101 (bge), op = 1100011 (branch)

$imm_{12,10:5}$	rs2	rs1	funct3	imm4:1,11	op
0111010	00111	00110	101	**0110**0	1100011

= **0x74735663**

c) **bne s10, s11, Back:** 0x1230D908 – 0x1230D10C = 0x7FC: branch 0x7FC bytes backward.
Flip sign of: 0x7FC (0 0111 1111 1100): 1 1000 0000 0011 + 1 = 1 1000 0000 0100
Branch offset = $imm_{12:0}$ = 0x1804: **1 1000 0000 0100**
rs1 = s10 = x26 (11010b), rs2 = s11 = x27 (11011b)
funct3 = 001 (bne), op = 1100011 (branch)

$imm_{12,10:5}$	rs2	rs1	funct3	imm4:1,11	op
1000000	11011	11010	001	**00101**	1100011

= **0x81BD12E3**

d) **beq a0, s1, L2:** 0xAB0CA0FC – 0xAB0C99A8 = 0x754: branch 0x754 bytes forward.
Branch offset = $imm_{12:0}$ = 0x754: **0 0111 0101 0100**
rs1 = a0 = x10 (01010b), rs2 = s1 = x9 (01001b)
funct3 = 000 (beq), op = 1100011 (branch)

$imm_{12,10:5}$	rs2	rs1	funct3	imm4:1,11	op
0111010	01001	01010	000	**10100**	1100011

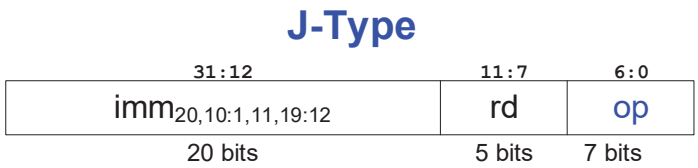
= **0x74950A63**

e) **blt s1, t3, L3:** 0xFFABD640 – 0xFFABCF04 = 0x73C: branch 0x73C bytes backward.
Flip sign of: 0x73C (0 0111 0011 1100): 1 1000 1100 0011 + 1 = 1 1000 1100 0100
Branch offset = $imm_{12:0}$ = 0x18C4 = **1 1000 1100 0100**
rs1 = s1 = x9 (01001b), rs2 = t3 = x28 (11100b)
funct3 = 100 (blt), op = 1100011 (branch)

$imm_{12,10:5}$	rs2	rs1	funct3	imm4:1,11	op
1000110	11100	01001	100	**00101**	1100011

= **0x8DC4C2E3**

Exercise 6.30



a) 0x123CABBC - 0x1234ABC0 = 0x7FFFC: jump 0x7FFFC bytes forward.
Jump offset = $imm_{20:0}$ = 0x7FFFC = **0 0111 1111 1111 1110**
rd = x0 (00000b), op = 1101111 (J-type)

$imm_{20,\ 10:1\ ,\ 11\ ,\ 19:12}$ **rd** **op**
0 111 1111 110 1 0111 1111 00000 1101111
= **0x7FD7F06F**

b) 0x123B8760 - 0x12345678 = 0x730E8: jump 0x730E8 bytes backward.
Flip sign of: 0x730E8: 0 0111 0011 0000 1110 1000:

1 1000 1100 1111 0001 0111 + 1 =
1 1000 1100 1111 0001 1000

Jump offset = $imm_{20:0}$ = 0x18CF18 = **1 1000 1100 1111 0001 100**0
rd = s0 = x8 (01000b), op = 1101111 (J-type)

$imm_{20,\ 10:1\ ,\ 11\ ,\ 19:12}$ **rd** **op**
1 111 0001 100 1 1000 1100 00000 1101111
= **0xF198C06F**

c) 0xAABDCD98 - 0xAABBCCD0 = 0x200C8: jump 0x200C8 bytes forward.
Jump offset = $imm_{20:0}$ = 0x200C8 = **0 0010 0000 0000 1100 100**0
rd = ra = x1 (00001b), op = 1101111 (J-type)

$imm_{20,\ 10:1\ ,\ 11\ ,\ 19:12}$ **rd** **op**
0 000 1100 100 0 0010 0000 00001 1101111
= **0x0C8200EF**

d) 0x1127BCDC - 0x11223344 = 0x58998: jump 0x58998 bytes forward.
Jump offset = $imm_{20:0}$ = 0x58998 = **0 0101 1000 1001 1001 100**0
rd = x0 (00000b), op = 1101111 (J-type)

$imm_{20,\ 10:1\ ,\ 11\ ,\ 19:12}$ **rd** **op**
0 001 1001 100 1 0101 1000 00000 1101111
= **0x1995806F**

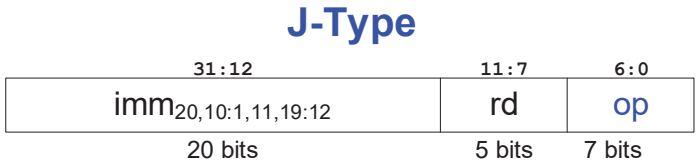
e) 0x9886543C - 0x9876543C = 0x100000: branch 0x100000 bytes backward.
Flip sign of: 0x100000: 1 0000 0000 0000 0000 0000:

0 1111 1111 1111 1111 1111 + 1 =
1 0000 0000 0000 0000 0000

Jump offset = $imm_{20:0}$ = 0x100000 = **1 0000 0000 0 000 0000 000**0
rd = ra = x1 (00001b), op = 1101111 (J-type)

$imm_{20,\ 10:1\ ,\ 11\ ,\ 19:12}$ **rd** **op**
1 000 0000 000 0 0000 0000 00000 1101111
= **0x8000006F**

Exercise 6.31



a) 0x0000EEEC - 0x0000ABC0 = 0x432C: jump 0x432C bytes forward.
Jump offset = $imm_{20:0}$ = 0x432C = **0 0000 0100 0011 0010 1100**
rd = ra = x1 (00001b), op = 1101111 (J-type)

$imm_{20,\ 10:1\ ,\ 11\ ,\ 19:12}$	rd	op
0 011 0010 110 0 0000 0100	00001	1101111

= **0x32C040EF**

b) 0x000F1230 - 0x0000C10C = 0xE5124: jump 0xE5124 bytes backward.
Flip sign of: 0xE5124: 0 1110 0101 0001 0010 0100:

1 0001 1010 1110 1101 1011 + 1 =
1 0001 1010 1110 1101 1100

Jump offset = $imm_{20:0}$ = 0x11AEDC = **1 0001 1010 1110 1101 1100**
rd = ra = x1 (00001b), op = 1101111 (J-type)

$imm_{20,\ 10:1\ ,\ 11\ ,\ 19:12}$	rd	op
1 110 1101 110 1 0001 1010	00001	1101111

= **0xEDD1A0EF**

c) 0x008FFFDC - 0x00801000 = 0xFEFDC: jump 0xFEFDC bytes forward.
Jump offset = $imm_{20:0}$ = 0xFEFDC = **0 1111 1110 1110 1101 1100**
rd = ra = x1 (00001b), op = 1101111 (J-type)

$imm_{20,\ 10:1\ ,\ 11\ ,\ 19:12}$	rd	op
0 110 1101 110 1 1111 1110	00001	1101111

= **0x6DDFE0EF**

d) 0xA131347C - 0xA1234560 = 0xDEF1C: jump 0xDEF1C bytes forward.
Jump offset = $imm_{20:0}$ = 0xDEF1C = **0 1101 1110 1111 0001 1100**
rd = x0 (00000b), op = 1101111 (J-type)

$imm_{20,\ 10:1\ ,\ 11\ ,\ 19:12}$	rd	op
0 111 0001 110 1 1101 1110	00000	1101111

= **0x71DDE06F**

e) 0xF0CBCCD4 - 0xF0BBCCD4 = 0x100000: branch 0x100000 bytes backward.
Flip sign of: 0x100000: 1 0000 0000 0000 0000 0000:

0 1111 1111 1111 1111 1111 + 1 =
1 0000 0000 0000 0000 0000

Jump offset = $imm_{20:0}$ = 0x100000 = **1 0000 0000 0 000 0000 0000**
rd = x0 (00000b), op = 1101111 (J-type)

$imm_{20,\ 10:1\ ,\ 11\ ,\ 19:12}$	rd	op
1 000 0000 000 0 0000 0000	00000	1101111

= **0x8000006F**

Exercise 6.32

Instruction	(a) Machine Code	(b) Type	Addressing mode
addi t4, a1, 0	00058E93	I	Immediate
ori a0, a0, 32	02056513	I	Immediate
sub a1, a1, a0	40A585B3	R	Register-only
jal Func2	024000EF	J	PC-relative
lw t2, 4(a0)	00452383	I	Base
sw t2, 16(a1)	0075A823	S	Base
srli t3, t2, 8	0083DE13	I	Immediate
beq t2, t3, Else	01C38463	B	PC-relative
jr ra	00008067	I	Immediate
addi a0, a0, 4	00450513	I	Immediate
j Func2	FE9FF06F	J	PC-relative

Supporting Work:

```
addi t4,  a1,  0
addi x29, x11, 0
imm              rs1      funct3   rd       op
0000 0000 0000   0101 1   000      1110 1   001 0011

ori  a0,  a0,  32
ori  x10, x10, 32
imm              rs1      funct3   rd       op
0000 0010 0000   0101 0   110      0101 0   001 0011

sub  a1,  a1,  a0
sub  x11, x11, x10
funct7     rs2       rs1      funct3   rd       op
0100 000   0 1010    0101 1   000      0101 1   011 0011

jal  Func2
jal  x1, 36   (offset = 0x58-0x34 = 0x24 = 36)
36 = 0 0000 0000 0000 0010 0100
imm20,10:1,11,19:12                  rd       op
0000 0010 0100 0000 0000             0000 1   110 1111

lw   t2, 4(a0)
lw   x7, 4(x10)
imm              rs1      funct3   rd       op
```

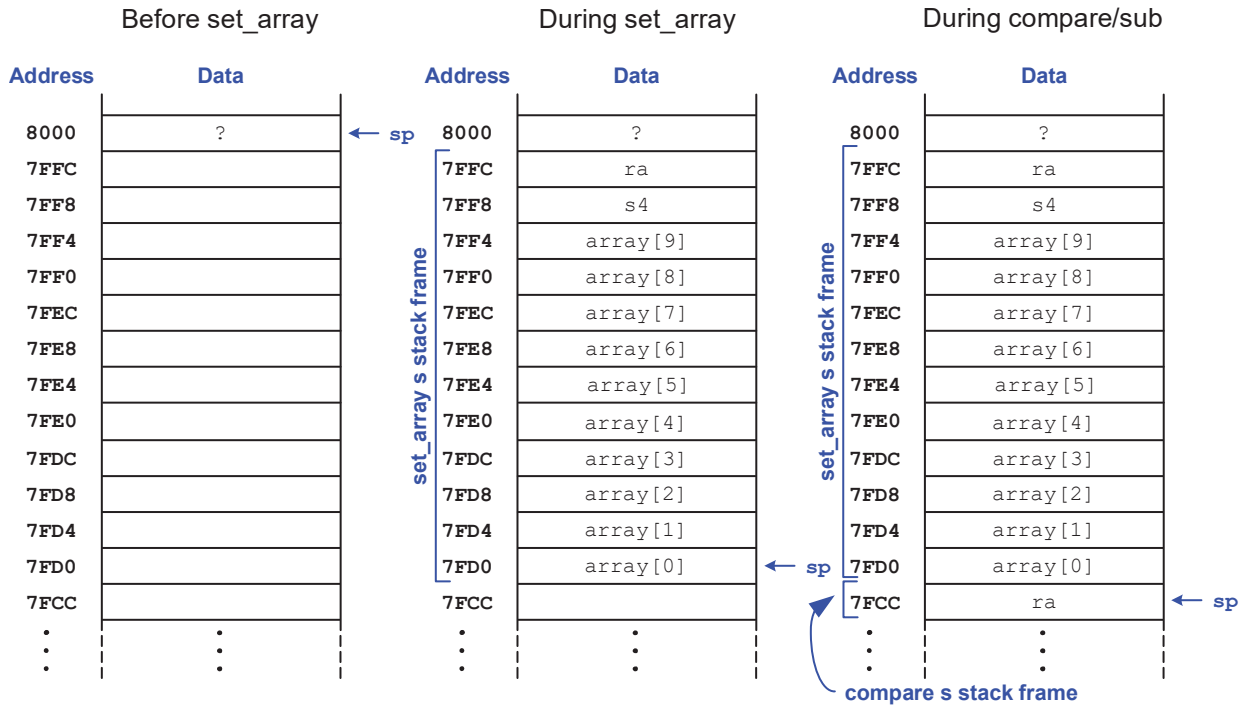
```
0000 0000 0100   0101 0   010      0011 1   000 0011

sw   t2, 16(a1)
sw   x7, 16(x11)
imm11:5     rs2      rs1      funct3   imm4:0     op
0000 000   0 0111   0101 1   010      1000 0    010 0011

srli t3,  t2, 8
srli x28, x7, 8
imm              rs1      funct3   rd       op
0000 0000 1000   0011 1   101      1110 0   001 0011

beq  t2, t3,  Else
beq  x7, x28, 8   (offset = 0x6C - 0x64 = 0x8)
8 = 0 0000 0000 1000
imm12,10:5       rs2      rs1      funct3   imm4:1,11   op
0000 000         1 1100   0011 1   000      0100 0     110 0011

jr   ra
jalr x0, x1, 0
imm              rs1      funct3   rd       op
0000 0000 0000   0000 1   000      0000 0   110 0111

addi a0,  a0,  4
addi x10, x10, 4
imm              rs1      funct3   rd       op
0000 0000 0100   0101 0   000      0101 0   001 0011

j       Func2
jal x0, -24    (offset = -(0x70-0x58) = -(0x18) = -24
-24 = 1 1111 1111 1111 1110 1000
imm20,10:1,11,19:12                rd       op
1111 1110 1001 1111 1111           0000 0   110 1111
```

Exercise 6.33

(a)

```
set_array: # a0 = num, s4 = i
  addi sp, sp, -48      # create space on the stack
  sw   ra, 44(sp)       # store ra on the stack
  sw   s4, 40(sp)       # store s4 on the stack
  addi s4, zero, 0      # i = 0
  addi t0, zero, 10     # t0 = 10 (# iterations)
loop:
  bge  s4, t0, done     # if i >= 10, exit loop
  add  a1, s4, zero     # a1 = i (second parameter)
  jal  compare          # compare(num, i)
  slli t1, s4, 2        # t1 = i*4
  add  t2, sp, t1       # t2 = address of array[i]
  sw   a0, 0(t2)        # array[i] = compare(num, i)
  addi s4, s4, 1        # i++
  j    loop             # repeat loop
done:
  lw   ra, 44(sp)       # restore ra
  lw   s4, 40(sp)       # restore s4
  addi sp, sp, 48       # restore stack pointer
  jr   ra               # return

compare: # a0 = a, a1 = b
  addi sp, sp, -4       # create space in the stack
  sw   ra, 0(sp)        # save ra on the stack
  jal  sub              # call sub(a, b)
  slt  a0, a0, zero     # a0 = 1 if sub(a,b) < 0
  xori a0, a0, -1       # invert a0
  andi a0, a0, 1        # isolate a0₀ and return
  lw   ra, 0(sp)        # restore ra
  addi sp, sp, 4        # restore sp
  jr   ra               # return

sub: # a0 = a, a1 = b
  sub a0, a0, a1        # a0 = a - b
  jr  ra                # return
```

(b)



(c) If `ra` were not saved on the stack, when the `compare` function attempts to return to `set_array` it would instead return to the instruction following the `sub` function call (`slt a0, a0, zero`) and would continue returning there every time it attempts to return. Since it will continue incrementing the stack pointer, it will eventually crash due to exceeding the stack space.

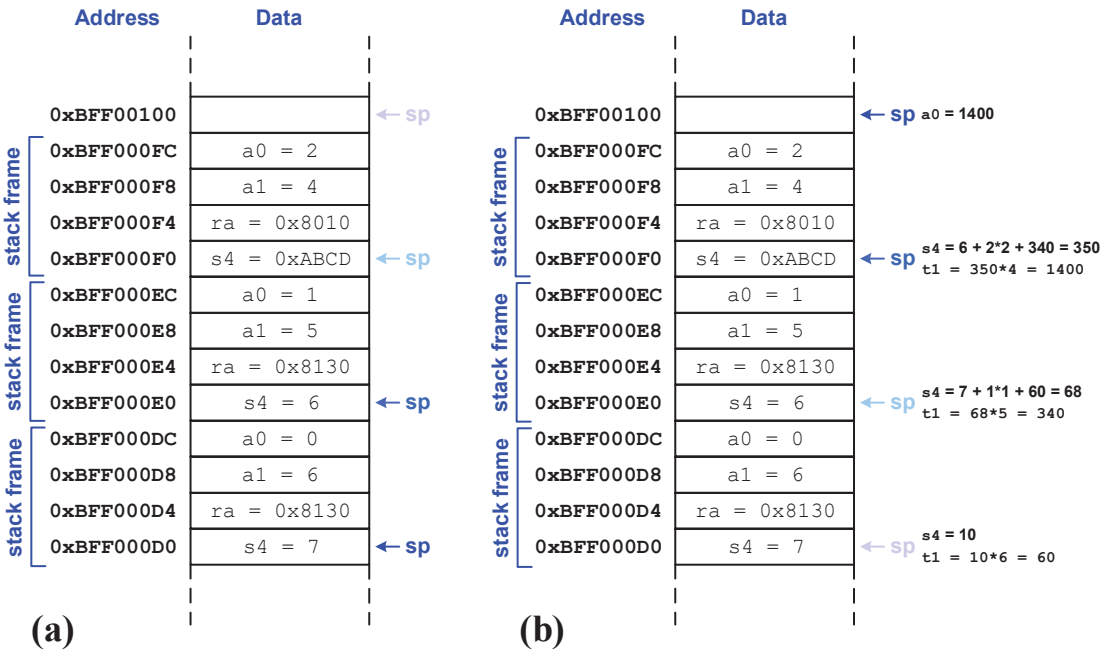
Exercise 6.34

(a) **RISC-V Assembly Code**

```
0x8100 f:      addi sp, sp, -16     # make room on stack
0x8104         sw   a0, 0xC(sp)     # save registers
0x8108         sw   a1, 0x8(sp)
0x810C         sw   ra, 0x4(sp)
0x8110         sw   s4, 0x0(sp)
0x8114         addi s4, a1, 2       # b = k + 2
0x8118         bne  a0, zero, else # if (n != 0) branch to else
0x811C         addi s4, zero, 10    # b = 10
0x8120         j    done
0x8124 else:   addi a0, a0, -1      # decrement n
0x8128         addi a1, a1, 1       # increment k
0x812C         jal  f               # call f(n - 1, k + 1)
0x8130         lw   t1, 0xC(sp)     # restore n into t1
0x8134         lw   a1, 0x8(sp)     # restore k into a1
0x8138         mul  t1, t1, t1      # t1 = n * n
0x813C         add  s4, s4, t1      # b = b + n * n
0x8140         add  s4, s4, a0      # b = b + n * n + f(n - 1, k + 1)
0x8144 done:   mul  a0, s4, a1      # return value = b * k
0x8148         lw   ra, 0x4(sp)     # restore ra
0x814C         lw   s4, 0x0(sp)     # restore s4
0x8150         addi sp, sp, 16      # restore sp
```

```
0x8154        jr    ra              # return
```

(b) Left figure: Stack after last call to `f`. Right figure: stack after return. Return value in `a0` = 1400.



Exercise 6.35

Regardless of the situation, since all branches encode the *offset* in their immediate field, the can all always jump forward 1,023 instructions. Specifically, a 13-bit signed (i.e,, two's complement) number can encode an offset of up to: $2^{12} - 1$ bytes / (4 bytes/instruction) = 2^{10} instructions – 1/4. But we can't have ¼ of an instruction, so it can encode a forward branch offset of up to 2^{10} instructions – 1 instructions = 1,023 instructions.

Exercise 6.36

The maximum distance backward a branch instruction can go is 4,096 bytes or 1,024 instructions (2^{10} instructions).
Since B-Type instructions use a 13-bit, signed immediate (where bits $imm_{1:0}$ will always be 0) the maximum possible negative number that can be encoded is:
`1 0000 0000 0000 = -4,096 bytes ( >> 2 = -1,024 instructions)`

Exercise 6.37

```
0x8000      lui t0, 0x408    # t0 = 0x40 8000
0x8004      jr  t0           # PC = 0x40 8000
```

Explanation: 2^{20} instructions = 2^{22} bytes. This is address: 0100 0000 0000 0000 0000

0000 (i.e., 0x40 0000) bytes beyond the current address (0x8000), so the code should branch to 0x40 8000.

Exercise 6.38

It is advantageous to have a large immediate field in the machine format for the jump and link instruction because the larger the immediate field, the larger the range of addresses the instruction can jump to.

Exercise 6.39

We show two options:

Option 1:

(a) **High Level Code**

```
void swapEndianness(int arr[]){
  for(int i = 0; i < 10; i++){
    arr[i] = ((arr[i] << 24) |
              ((arr[i] & 0xFF00) << 8) |
              ((arr[i] & 0xFF0000) >> 8) |
              ((arr[i] >> 24) & 0xFF));
  }
}
```

(b) **RISC-V Assembly Code**

```
# a0 = base address of arr, t0 = i
swapEndianness:
  addi  t0, zero, 0    # i = 0
  addi  t1, zero, 10   # t4 = 10 (temp value)
  lui   t4, 0xFF0      # t4 = 0xFF0000
  srli  t3, t4, 8      # t3 = 0xFF00
L1:
  bge  t0, t1, done    # if i >= 10 return
  slli t5, t0, 2       # t5 = i * 4 (byte offset)
  add  t5, t5, a0      # t5 = address of arr[i]
  lw   t6, 0(t5)       # t6 = arr[i]
  slli a1, t6, 24      # a1 = arr[i] << 24
  and  a2, t6, t3      # a2 = arr[i] & 0xFF00
  slli a2, a2, 8       # a2 =(arr[i] & 0xFF00)   << 8
  and  a3, t6, t4      # a3 = arr[i] & 0xFF0000
  srli a3, a3, 8       # a3 =(arr[i] & 0xFF0000) >> 8
  srli a4, t6, 24      # a4 = arr[i] >> 24
  or   a0, a1, a2      # a0 = combine most significant bytes
  or   a0, a0, a3      # a0 = combine 3 most significant bytes
  or   a0, a0, a4      # a0 = combine all bytes
  sw   a0, 0(t5)       # arr[i] = value with other endianness
  addi t0, t0, 1       # i++
  j    L1              # loop
done:
```

Option 2:

(a) **High Level Code**

```
void swapEndianness(int arr[]){
  char *arrBytes = (char *)arr;
  char tmp0, tmp1, tmp2, tmp3;
  int i = 10;

  do {
    tmp0 = arrBytes[0];
    tmp1 = arrBytes[1];
    tmp2 = arrBytes[2];
    tmp3 = arrBytes[3];
    arrBytes[0] = tmp3;
    arrBytes[1] = tmp2;
    arrBytes[2] = tmp1;
    arrBytes[3] = tmp0;
    arrBytes += 4;
    i--;
  } while (i != 0);
  return;
}
```

(b) **RISC-V Assembly Code**

```
# a0 = base address of array
swapEndianness:
  addi t4, zero, 10   # i = t4 = 10 (loop counter)
loop:
  lb   t0, 0(a0)      # t0 = byte 0 of arr[i]
  lb   t1, 1(a0)      # t1 = byte 1 of arr[i]
  lb   t2, 2(a0)      # t2 = byte 2 of arr[i]
  lb   t3, 3(a0)      # t3 = byte 3 of arr[i]
  sb   t3, 0(a0)      # byte 0 of arr[i] = original byte 3
  sb   t2, 1(a0)      # byte 1 of arr[i] = original byte 2
  sb   t1, 2(a0)      # byte 2 of arr[i] = original byte 1
  sb   t0, 3(a0)      # byte 3 of arr[i] = original byte 0
  addi a0, a0, 4      # increment array index
  addi t4, t4, -1     # i--
  beq  t4, zero, done # exit loop if i == 0
  j    loop
done:
  jr   ra
```

Exercise 6.40

(a) High Level Code

```
void concat(char string1[], char string2[], char stringconcat[]){
  int idx1 = 0;
  int idx2 = 0;

  while (string1[idx1] != 0){
    stringconcat[idx1] = string1[idx1];
    idx1 += 1;
  }
```

```
  while (string2[idx2] != 0){
    stringconcat[idx1] = string2[idx2];
    idx1 += 1;
    idx2 += 1;
  }
  stringconcat[idx1] = 0; //append '\0' (null) to end of string
}
```

(b) **RISC-V Assembly Code**

```
# a0 = base adr of string1[]
# a1 = base adr of string2[]
# a2 = base adr of stringconcat
concat:
  lb   t0, 0(a0)          # t0 = string1[idx1]
  beq  t0, zero, string2  # if string1[idx1] == 0, go to string2
  sb   t0, 0(a2)          # stringconcat[idx1] = string1[idx1]
  addi a0, a0, 1          # increment string1[] index
  addi a2, a2, 1          # increment stringconcat[] index
  j    concat             # loop
string2:
  lb   t0, 0(a1)          # t0 = string2[idx2]
  beq  t0, zero, done     # if string2[idx2] == 0, go to done
  sb   t0, 0(a2)          # stringconcat[idx1] = string2[idx2]
  addi a1, a1, 1          # increment string2[] index
  addi a2, a2, 1          # increment stringconcat[] index
  j    string2            # loop
done:
  sb   zero, 0(a2)        # append null to stringconcat[]
  jr   ra                 # return to caller
```

Exercise 6.41

```
# a0 = first value, a1 = second value
addFloat:
extract:
  lui  t4, 0x800
  addi t4, t4, -1       # t4 = 0x007FFFFF (mantissa mask)
  and  t0, a0, t4       # t0 = a0 mantissa
  and  t1, a1, t4       # t1 = a1 mantissa
  lui  t4, 0x800        # t4 = 0x00800000 (implicit leading 1)
  or   t0, t0, t4       # add implicit 1 to a0 mantissa
  or   t1, t1, t4       # add implicit 1 to a1 mantissa
  lui  t4, 0x7F800      # t4 = 0x7F800000 (exponent mask)
  and  t2, a0, t4       # t2 = a0 exponent
  srli t2, t2, 23       # shift a0 exponent right
  and  t3, a1, t4       # t2 = a1 exponent
  srli t3, t3, 23       # shift a1 exponent right
compare:
  beq  t2, t3, addMant # check if exponents match
```

```
  bgeu t2, t3, shift1  # check which exponent is larger
shift0:
  sub  t4, t3, t2      # calculate the difference of exponents
  sra  t0, t0, t4      # shift a0 by above difference
  add  t2, t2, t4      # update a0's exponent
  j addMant            # next we add the mantissas
shift1:
  sub  t4, t2, t3      # calculate the difference of exponents
  sra  t1, t1, t4      # shift a1 by above difference
  add  t3, t3, t4      # update a1's exponent (for regularity)
addMant:
  add  t5, t0, t1      # add the mantissas
norm:
  lui  t4, 0x1000      # t4 = 0x01000000 (overflow bit mask)
  and  t4, t5, t4      # t4 = masked bit 24
  beq  t4, zero, done  # no need to right shift if no overflow
  srli t5, t5, 1       # shift mantissa by right by one
  addi t2, t2, 1       # increment the exponent
done:
  lui  t4, 0x800
  addi t4, t4, -1      # t4 = 0x007FFFFF (mantissa mask)
  and  t4, t5, t4      # t4 = masked result mantissa
  slli t2, t2, 23      # align the exponent in proper place
  lui  t4, 0x7F800     # t4 = 0x7F800000 (exponent mask)
  and  t2, t2, t4      # t2 = result exponent
  or   a0, t5, t2      # result stored in a0
  jr   ra              # return
```

Exercise 6.42

```
# Floating point addition (w/ positive and negative numbers)
# Define the masks in the global data segment
              .data
mmask:    .word 0x007FFFFF
emask:    .word 0x7F800000
ibit:     .word 0x00800000
obit:     .word 0x01000000
sbit:     .word 0x80000000
              .text
# Assume s0 = floating point number a
#        s1 = floating point number b
flpadd:
  addi sp, sp, -4       # Make room on stack
  sw   s2, 0(sp)        # Need more temp, save s2 so we can use
  lw   t4, mmask        # load mantissa mask
  and  t0, s0, t4       # extract mantissa from fp number a
  and  t1, s1, t4       # extract mantissa from fp number b
  lw   t4, ibit         # load implicit leading 1
  or   t0, t0, t4       # add implicit leading 1 to a
  or   t1, t1, t4       # add implicit leading 1 to b
  lw   t4, emask        # load exponent mask
  and  t2, s0, t4       # extract exponent from a
```

```
  srli t2, t2, 23       # shift exponent right for comparison step
  and  t3, s1, t4       # extract exponent from b
  srli t3, t3, 23       # shift exponent right for comparison step
  lw   t4, sbit         # load mask for sign bit
  and  t6, s0, t4       # extract sign bit from a
  srli t6, t6, 31       # shift sign bit right
  and  s2, s1, t4       # extract sign bit from b
  srli s2, s2, 31       # shift sign bit right
match:
  beq  t2, t3, addequal # Is exponent of a == exponent of b?
  bgeu t2, t3, shiftb   # Determine larger exponent if not ==
  j    shifta
addequal:
  beq  t6, s2, samesignequal
  bgeu t1, t0, bgreater # If sign bits != sub smaller num from larger
agreater:
  sub  t5, t0, t1       # (Mantissa) a - b
  add  s2, zero, t6     # Update b's sign bit (not necessary)
  j    norm             # Skip to normalizing mantissa
bgreater:
  sub  t5, t1, t0       # (Mantissa) b - a
  add  t6, zero, s2     # Update a's sign bit
  j    norm             # Skip to normalizing mantissa
samesignequal:
  add  t5, t0, t1       # (Mantissa) a + b
  j    norm             # Skip to normalizing mantissa
shifta:
  sub  t4, t3, t2       # Calculate difference in exponents
  srl  t0, t0, t4       # Shift (mantissa) a by calculated difference
  add  t2, t2, t4       # Update a's exponent
  beq  t6, s2, samea    # Compare sign bits
  sub  t5, t1, t0       # (Mantissa) b - a
  j    norm             # Skip to normalizing mantissa
samea:
  add  t5, t1, t0       # (Mantissa) b + a
  j    norm             # Skip to normalizing mantissa
shiftb:
  sub  t4, t2, t3       # Calculate difference in exponents
  srl  t1, t1, t4       # Shift (mantissa) b by calculated difference
  add  t3, t3, t4       # Update b's exponent (not necessary)
  beq  t6, s2, sameb    # Compare sign bits
  sub  t5, t0, t1       # (Mantissa) a - b
  j    norm             # Skip to normalizing mantissa
sameb:
  add  t5, t0, t1       # (Mantissa) a + b
norm:
  lw   t4, obit         # load mask for overflow bit
  and  t4, t5, t4       # mask bit 24
  beq  t4, zero, done   # If overflow bit == 0, shift not needed
  srli t5, t5, 1        # Shift right by 1 bit
  addi t2, t2, 1        # Increment exponent to restore after shift
done:
  lw   t4, mmask        # Load mantissa mask
  and  t5, t5, t4       # Mask mantissa
  slli t2, t2, 23       # Shift exponent into place
  slli t6, t6, 31       # Shift sign bit into place
  lw   t4, emask        # load exponent mask
```

```
  and  t2, t2 t4        # Mask exponent
  lw   t4, sbit         # load sign bit mask
  and  t6, t6, t4       # Mask sign bit
  or   t5, t5, t2       # Place mantissa and exponent
  or   a0, t5, t6       # Place entire flp number into a0
  lw   s2, 0(sp)        # Load back original value into s2
  addi sp, sp, 4        # Restore stack
  jr   ra
```

Exercise 6.43

(a) **High Level Code**

```
// sorts a 10-element array using Bubble Sort
void sort(int scores[]){
  for (int i = 0; i < 9; i++){
    for (int j = 0; j < 9-i; j++){
      // swap places if next element is larger
      if(scores[j] > scores[j+1]){
        scores[j]   = scores[j]^scores[j+1];
        scores[j+1] = scores[j]^scores[j+1];
        scores[j]   = scores[j]^scores[j+1];
      }
    }
  }
}
```

(b) **RISC-V Assembly Code**

```
# a0 = address of scores array
# we assume 16-bit (4-byte) integer size
sort:
  addi t0, zero, 0   # i = 0
  addi t1, zero, 9   # t1 = 9
outerLoop:
  bge  t0, t1, done1 # i >= 9?
  addi t2, zero, 0   # j = 0
  sub  t3, t1, t0    # t3 = 9-i
innerLoop:
  bge  t2, t3, done2 # j >= 9-i?
  slli t4, t2, 1     # t4 = scores array offset (j*2)
  add  t4, t4, a0    # t4 = address of scores[j]
  lh   t5, 0(t4)     # t5 = scores[j]
  lh   t6, 2(t4)     # t6 = scores[j+1]
  bge  t6, t5, skip  # skip if scores[j+1] >= scores[j]
  sb   t5, 2(t4)     # scores[j] = original scores[j+1]
  sb   t6, 0(t4)     # scores[j+1] = original scores[j]
skip:
  addi t2, t2, 1     # j++
  j    innerLoop     # repeat innerLoop
done2:
  addi t0, t0, 1     # i++
  j    outerLoop     # repeat outerLoop
done1:
```

```
    jr   ra            # return
```

Exercise 6.44

(a)
```
0x8400 main: addi sp, sp, -4
0x8404       sw   ra, 0(sp)
0x8408       lw   a0, -940(gp)
0x840C       lw   a1, -936(gp)
0x8410       jal  diff
0x8414       lw   ra, 0(sp)
0x8418       addi sp, sp, 4
0x841C       jr   ra
0x8420 diff: sub  a0, a0, a1
0x8424       jr   ra
```

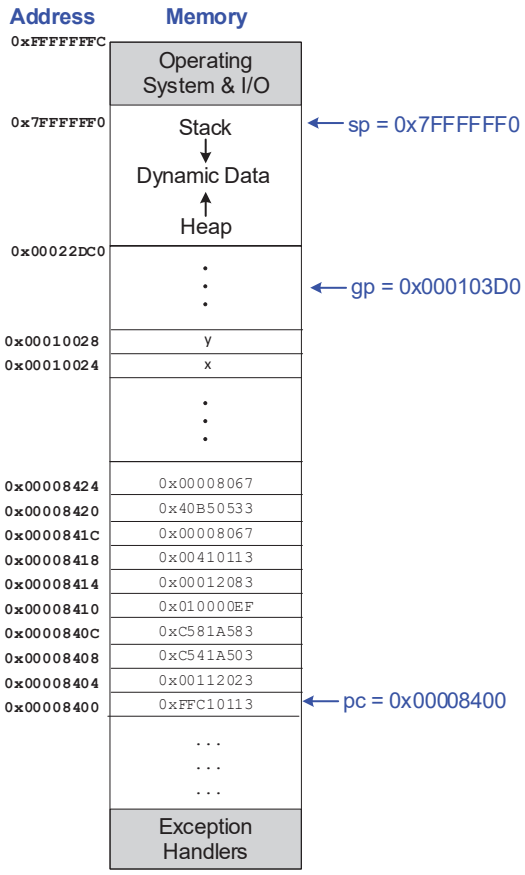
(b) Symbol Table:

Address	Size	Symbol
0x8400	00000020	main
0x8420	00000008	diff
0x10024	00000004	x
0x10028	00000004	y

(c)
```
0x8400 main: addi sp, sp, -4     # 0xFFC10113
0x8404       sw   ra, 0(sp)      # 0x00112023
0x8408       lw   a0, -940(gp)   # 0xC541A503
0x840C       lw   a1, -936(gp)   # 0xC581A583
0x8410       jal  diff           # 0x010000EF
0x8414       lw   ra, 0(sp)      # 0x00012083
0x8418       addi sp, sp, 4      # 0x00410113
0x841C       jr   ra             # 0x00008067
0x8420 diff: sub  a0, a0, a1     # 0x40B50533
0x8424       jr   ra             # 0x00008067
```

(d) Text segment = 0x20 + 0x8 = 0x28 **(40) bytes**
Data segment = 0x4 + 0x4 = 0x8 **(8) bytes**

(e)



Exercise 6.45

(a)
```
0x8534  main:      addi sp, sp, -8
0x8538             sw   ra, 4(sp)
0x853C             sw   s4, 0(sp)
0x8540             addi s4, zero, 15
0x8544             sw   s4, -300(gp)  # g = 15
0x8548             addi a1, zero, 27  # arg1 = 27
0x854C             sw   a1, -296(gp)  # h = 27
0x8550             lw   a0, -300(gp)  # arg0 = g = 15
0x8554             jal  greater
0x8558             lw   s4, 0(sp)
0x855C             lw   ra, 4(sp)
0x8560             addi sp, sp, 8
0x8564             jr   ra
0x8568  greater:   blt  a1, a0, isGreater
0x856C             addi a0, zero, 0
0x8570             jr   ra
0x8574  isGreater: addi a0, zero, 1
0x8578             jr   ra
```

(b) Symbol Table:

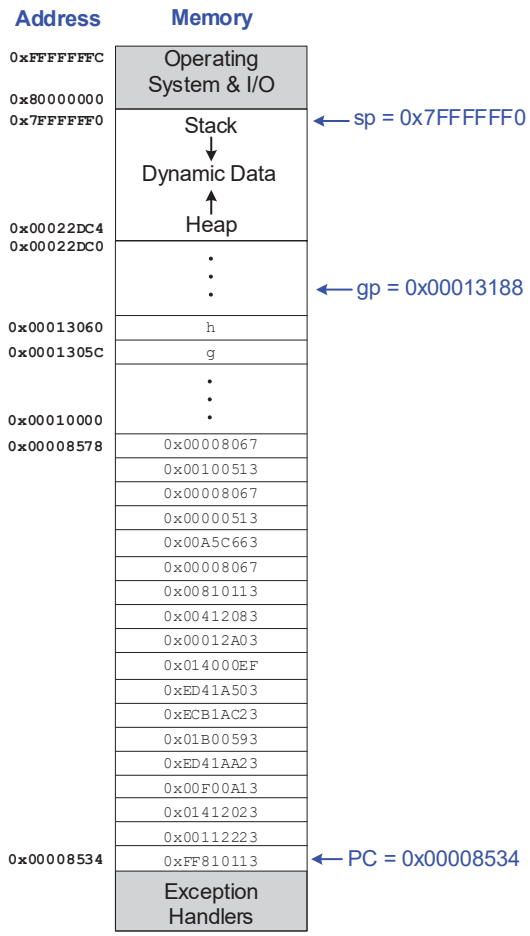
Address	Size	Symbol
0x8534	00000034	main

```
0x8568      0000000C   greater
0x8574      00000008   isGreater
0x1305C     00000004   g
0x13060     00000004   h
```

(d)
```
0x8534  main:      addi sp, sp, -8         # 0xFF810113
0x8538             sw   ra, 4(sp)          # 0x00112223
0x853C             sw   s4, 0(sp)          # 0x01412023
0x8540             addi s4, zero, 15       # 0x00F00A13
0x8544             sw   s4, -300(gp)       # 0xED41AA23
0x8548             addi a1, zero, 27       # 0x01B00593
0x854C             sw   a1, -296(gp)       # 0xECB1AC23
0x8550             lw   a0, -300(gp)       # 0xED41A503
0x8554             jal  greater            # 0x014000EF
0x8558             lw   s4, 0(sp)          # 0x00012A03
0x855C             lw   ra, 4(sp)          # 0x00412083
0x8560             addi sp, sp, 8          # 0x00810113
0x8564             jr   ra                 # 0x00008067
0x8568  greater:   blt  a1, a0, isGreater # 0x00A5C663
0x856C             addi a0, zero, 0        # 0x00000513
0x8570             jr   ra                 # 0x00008067
0x8574  isGreater: addi a0, zero, 1        # 0x00100513
0x8578             jr   ra                 # 0x00008067
```

(d) The global data segment is 8 bytes and the text segment is 72 bytes.

(e)

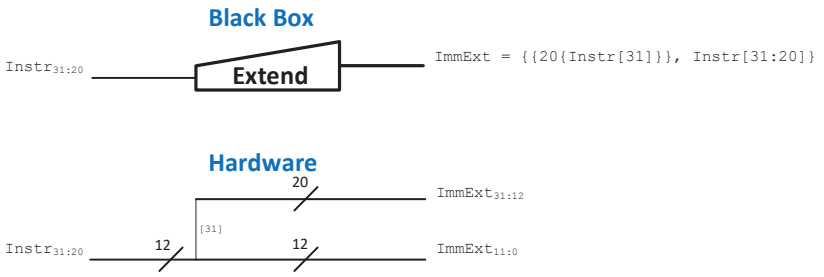


Exercise 6.46

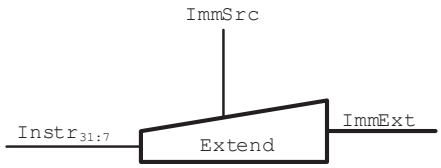
RISC-V immediates are bit-swizzled in order to optimize the hardware and make it faster overall. The disadvantage to this is that it results in confusing immediate encodings and multiple instruction formats, which can make it harder to learn the architecture. However, the bit-swizzling is abstracted away at the assembly language level (and even more so at the higher-level programming language), so the complexity of the immediate encodings gets abstracted away.

Exercise 6.47

(a)

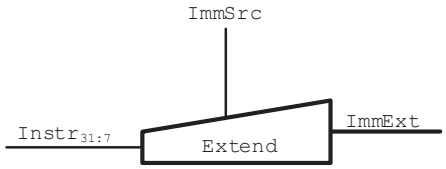


(b)



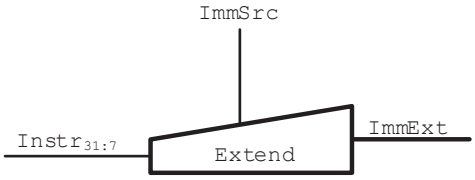
ImmSrc	ImmExt	Type
0	{{20{Instr[31]}}, Instr[31:20]}	I
1	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}	S

c)



ImmSrc	ImmExt	Type
00	{{20{Instr[31]}}, Instr[31:20]}	I
01	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}	S
11	{{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0}	B

d)



ImmSrc	ImmExt	Type
00	{{20{Instr[31]}}, Instr[31:20]}	I
01	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}	S
10	{{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0}	B
11	{{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0}	J

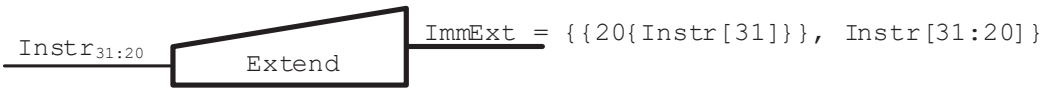
The figure below shows how the instruction bits are used to recreate the immediate for all instruction formats that encode an immediate that is extended to 32 bits. To create the 32-bit immediate, most of the immediate bits require only a 2:1 mux, thereby choosing amongst

only two possible instruction bit locations. This is in contrast to the worst case, where each immediate bit would require a 5:1 mux, to select amongst five instruction bit locations for each of the five instruction formats.

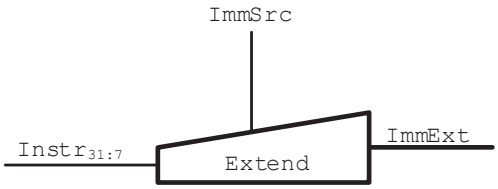
instruction bit	wire	2:1 mux	2:1 mux	4:1 mux	2:1 mux	3:1 mux	3:1 mux	
	31	31	31	31	30:25	24:21	20	I
	31	31	31	31	30:25	11:8	7	S
	31	31	31	7	30:25	11:8	0	B
	31	30:20	19:12	0	0	0	0	U
	31	31	19:12	20	30:25	24:21	0	J
immediate bit	31 30 29 28 27 26 25 24 23 22 21 20	19 18 17 16 15 14 13 12	11	10 9 8 7 6 5	4 3 2 1	0		

Exercise 6.48

(a)

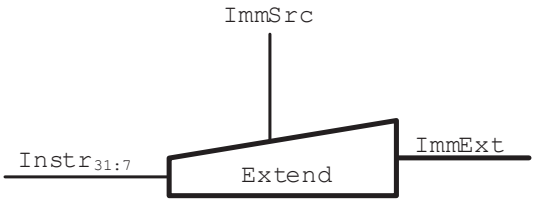


(b)



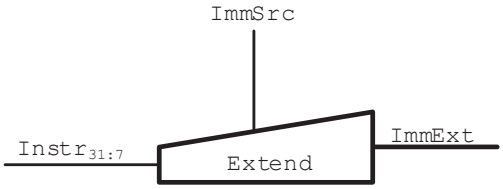
ImmSrc	ImmExt	Type
0	{{20{Instr[31]}}, Instr[31:20]}	I
1	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}	S

(c)



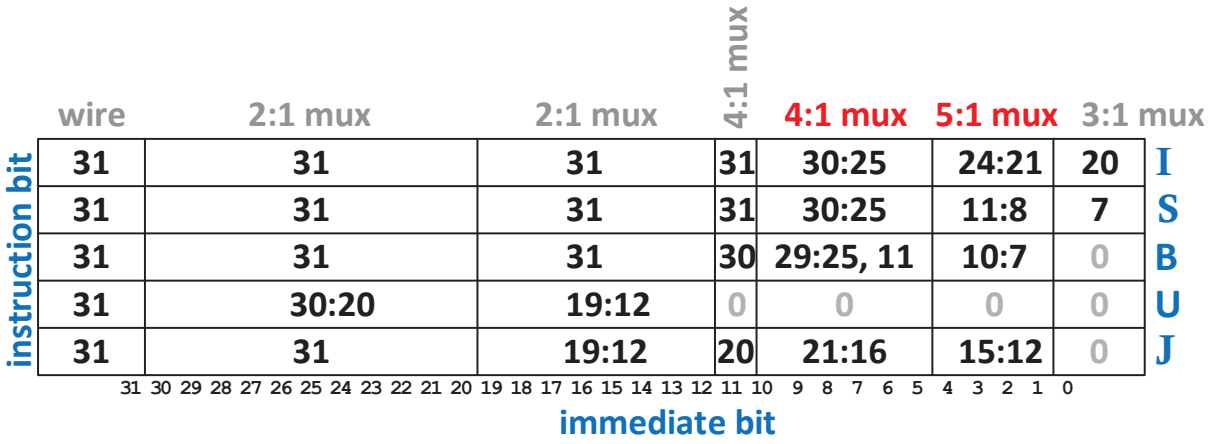
ImmSrc	ImmExt	Type
00	{{20{Instr[31]}}, Instr[31:20]}	I
01	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}	S
10	{{20{Instr[31]}}, Instr[30:25], Instr[11:7], 1'b0}	B

(d)



ImmSrc	ImmExt	Type
00	{{20{Instr[31]}}, Instr[31:20]}	I
01	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}	S
10	{{20{Instr[31]}}, Instr[30:25], Instr[11:7], 1'b0}	B
11	{{12{Instr[31]}}, Instr[30:12], 1'b0}	J

(e) Wider multiplexers are required for this extension unit as shown in the figure below. The multiplexers highlighted in red text are wider than in the bit-swizzled version of immediates – that is, the immediate encodings that actually exist in the RISC-V processor). But even without strange bit swizzling of the B- and J-type formats, some uniformity still remains amongst formats, as demonstrated by the smaller multiplexers and the wire for the most significant bit.



Exercise 6.49

(a) `jal` can jump forward $(2^{(21-1)} - 1)/4 = 2^{18} - 1 =$ **262,143** instructions forward.
(b) `jal` can jump backward $-[2^{(21-1)}]/4 = -2^{18} =$ **262,144** instructions backward.

Exercise 6.50

(a) $42 * 4 = 168 = 10101000_2 = 0xA8$
(b) 0xA8 through 0xAB
(c)

Exercise 6.51

a) $15 \times 4 = 15 \times 2^2 = 1111_2 << 2 = 111100_2 = 0x3C$
b) 0x3C through 0x3F
c)

Exercise 6.52

Because big-endian format numbers byte addresses from left to right and little-endian format numbers bytes from right to left, `lb s2,1(s7)` will store different values into `s2`. In a big-endian computer: `s2` = 0xFFFFFFCD

In a little-endian computer: `s2` = 0xFFFFFF87
Remember that `lb` sign-extends the byte to fit into the 32-bit register.

Question 6.1

```
xor a0, a0, a1 # a0 = a0 XOR a1
xor a1, a0, a1 # a1 = original a0
xor a0, a0, a1 # a0 = original a1
```

Example:

a0 = 1011 0101 (binary)
a1 = 0010 1111 (binary)

```
xor a0, a0, a1 # a0 = 1001 1010 (1's wherever different)
xor a1, a0, a1 # a1 = original a0 = 1011 0101
xor a0, a0, a1 # a0 = original a1 = 0010 1111
```

Question 6.2

```
// high-level code – this was not requested in the problem
void findMaxArray(int arr1[], int length, int arr2[]) {
  int i, j, start, end, sum = 0;
  int max = -2147483648;  // most negative number: 0x80000000

  for(i = 0; i < length; i++) {
    sum = arr1[i];
    if (sum > max) {
      max = sum;
      start = i;
      end = i;
    }
    for (j = i + 1; j < length; j++) {
      sum = sum + arr1[j];
      if (sum > max) {
        max = sum;
        start = i;
        end = j;
      }
    }
  }

  // write to arr2[]
  j = 0;
  for(i = start; i <= end; i++) {
    arr2[j] = arr1[i];
    j++;
  }
}
```

Answer:

```
# a0 = base address of arr1, a1 = length of arr1
# a2 = base address of resulting array (arr2)
# t0 = max, t1 = start, t2 = end
# t3 = i, t4 = j, t5 = sum
findMaxArray:
  addi t5, zero, 0          # sum = 0
  lui  t0, 0x80000          # max = 0x80000000 = most negative #
  addi t3, zero, 0          # i = 0
L1:
  bge  t3, a1, finish       # if i >= length, finish
  slli t6, t3, 2            # t6 = i * 4
  add  t6, t6, a0           # t6 = address of arr1[i]
  lw   t5, 0(t6)            # sum = array[i]
  bge  t0, t5, skip1        # if (max >= sum), don't update max
  addi t0, t5, 0            # max = sum
  addi t1, t3, 0            # start = i
  addi t2, t3, 0             # end = i
skip1:
```

```
        addi t4, t3, 1            # j = i + 1
L2:
  bge  t4, a1, endL2        # if j >= length, endL2
  slli t6, t4, 2            # t6 = j * 4
  add  t6, t6, a0           # t6 = address of arr1[j]
  lw   t6, 0(t6)            # t6 = arr1[j]
  add  t5, t5, t6           # sum = sum + arr1[j]
  bge  t0, t5, skip2        # if (max >= sum), don't update max
  addi t0, t5, 0            # max = sum
  addi t1, t3, 0            # start = i
  addi t2, t4, 0            # end = j
skip2:
  addi t4, t4, 1            # j++
  j    L2                   # loop (L2)
endL2:
  addi t3, t3, 1            # i++
  j    L1                   # loop (L1)

finish:
  addi t4, zero, 0          # j = 0
  addi t3, t1, 0            # i = start
L3:
  blt  t2, t3, done         # if (i > end), done
  slli t5, t3, 2            # t5 = i * 4
  add  t5, t5, a0           # t5 = address of arr1[i]
  lw   t5, 0(t5)            # t5 = arr1[i]
  slli t6, t4, 2            # t6 = j * 4
  add  t6, t6, a2           # t6 = address of arr2[j]
  sw   t5, 0(t6)            # arr2[j] = arr1[i]
  addi t3, t3, 1            # i++
  addi t4, t4, 1            # j++
       j    L3
done:
  jr   ra
```

Question 6.3

High-Level Algorithm

```
void reverseWords(char arr[]){
  // find the length of the string
  int length;
  for (length=0; arr[length] != 0; length++);

  // first reverse the entire string
  reverse(arr, 0, length-1);

  // next reverse each individual word back
```

```
  int begin = 0;
  int end = 0;

  // find start and end positions of each word
  while (end <= length){
    if ((end != length) && (arr[end] != 0x20))
      end++;
    else {
      reverse(arr, begin, end-1);
      end++;
      begin = end;
    }
  }
}

// This function reverses the characters of the passed array
// between the passed begin and end index positions
void reverse(char arr[], int begin, int end){
  while (begin < end){
    // swap characters
    arr[begin] = arr[begin]^arr[end];
    arr[end]   = arr[begin]^arr[end];
    arr[begin] = arr[begin]^arr[end];
    // move index positions in
    begin++;
    end--;
  }
}
```

RISC-V Assembly Code

```
# a0 = address of arr[], a1 = begin, a2 = end
reverseWords:
  addi t0, zero, 0    # t0 = length = 0
for:
  add  t1, a0, t0     # t1 = address of arr[length]
  lb   t1, 0(t1)      # t1 = arr[length]
  beq  t1, zero, done # stop counting when we hit null
  addi t0, t0, 1      # length++
  j    for            # repeat loop
done:
  addi a1, zero, 0    # begin = 0
  addi a2, t0, -1     # end = length - 1
  jal reverse         # reverse(arr, 0, length-1)
  addi a1, zero, 0    # a1 = begin
  addi a2, zero, 0    # a2 = end
while:
  blt  t0, a1, done2  # length < begin?
  beq  a2, t0, else   # end == length?
  addi t1, zero, 0x20 # t1 = 0x20
  add  t2, a0, a2     # t2 = address of arr[end]
  lb   t2, 0(t2)      # t2 = arr[end]
```

```
  beq  t2, t1 else    # arr[end] == 0x20?
  addi a2, a2, 1      # end++
  j    done2          # skip over else
else:
  addi a2, a2, -1     # end = end-1
  jal  reverse        # reverse(arr, begin, end-1)
  addi a2, a2, 1      # end = end+1
  addi a2, a2, 1      # end++
  add  a1, a2, zero   # begin = end
  j while             # repeat loop
done2:
  jr   ra             # return

# a0 = address of arr[], a1 = begin, a2 = end
reverse:
  add  t1, a1, a0     # t1 = address of arr[begin]
  add  t3, a2, a0     # t3 = address of arr[end]
while2:
  bge  t1, t3 done3   # begin >= end?
  lb   t2, 0(t1)      # t2 = arr[begin]
  lb   t4, 0(t3)      # t4 = arr[end]
  sb   t4, 0(t1)      # arr[begin] = original arr[end]
  sb   t2, 0(t3)      # arr[end] = original arr[begin]
  addi t1, t1, 1      # address of arr[begin]++
  addi t3, t3, -1     # address of arr[end]--
  j while2            # repeat loop
done3:
  jr   ra             # return:
```

Question 6.4

```
# Count number of 1's in a 32-bit number
# a0 = number to count, t3 = cnt
.data
checkmask: .word 0x00000001
.text

count1s:
  lw   t0, checkmask  # load mask
  add  t1, zero, a0   # t1 = number
  addi t3, zero, 0    # cnt = 0
loop:
  beq  t1, zero, done # Check if all bits have been shifted out
  and  t2, t0, t1     # t2 = num & mask
  beq  t2, zero, next # If num & mask = 1? check next bit
  addi t3, t3, 1      # If yes, increment counter
next:
  srli t1, t1, 1      # num >> 1
  j    loop           # loop
done:
  add  a0, zero, t3   # return cnt
```

```
  jr   ra             # return to caller
```

Question 6.5

```
# a0 = register on which to reverse bits
reverseBits:
  addi t1, zero, 31    # t1 = 31
  addi t2, zero, 0     # t2 = 0
  addi t3, zero, 0     # t3 = 0
L7:
  beq  t1, zero, done3 # if t1 = 0, done
  srl  t0, a0, t1      # t0 = a0 >> t1
  andi t0, t0, 1       # isolate lsb of t0
  sll  t0, t0, t2      # t0 = t0 << t2
  or   t3, t3, t0      # combine bits
  addi t1, t1, -1      # t1--
  addi t2, t2, 1       # t2++
  j    L7              # repeat the loop
done3:
  add  a0, t3, zero    # set a0 equal to its reversed self
  jr   ra              # return
```

Question 6.6

```
# Check for overflow of a3 - a2 (assuming 2's complement numbers)
# Overflow occurs when: 1. a3 and a2 have different signs, and
#                       2. The result (a3-a2) has a different
#                          sign than a3.

  sub  t0, a3, a2           # t0 = a3 - a2
  xor  t1, a2, a3           # compare operand sign bits
  srli t1, t1, 31           # shift t1 msb to lsb
  beq  t1, zero, nooverflow # if sign bits same, no overflow
  xor  t1, t0, a3           # compare result and a3 sign bits
  srli t1, t1, 31           # shift t1 msb to lsb
  beq  t1, zero, nooverflow # if result and a3 has same sign,
                            # no overflow
overflow:
# do something

nooverflow:
# do something
```

Question 6.7

High-Level Algorithm

```
bool isPalindrome(char str[]){
  int begin = 0;
  int end = 0;
```

```
  // first find the index of the last character
  for (end = 0; str[end] != 0; end++);
  end--;

  // check if each character pair matches
  // if one does not, the string is not a palindrome
  while(end > begin){
    if (str[begin] != str[end])
      return false;
    begin++;
    end--;
  }
  // if the above check passed then it is a palindrome
  return true;
}
```

RISC-V Assembly Code

```
isPalindrome:
# a0 = base address of str[]
  addi t0, zero, 0    # t0 = begin = 0
  addi t1, zero, 0    # t1 = end = 0
for:
  add  t2, a0, t1     # t2 = address of str[end]
  lb   t2, 0(t2)      # t2 = str[end]
  beq  t2, zero, done # stop counting when str[end] is null
  addi t1, t1, 1      # end++
  j    for            # repeat loop
done:
  addi t1, t1, -1     # end--
while:
  bge  t0, t1, yes    # if all chars matched, then jump to yes
  add  t2, t0, a0     # t2 = address of str[begin]
  lb   t2, 0(t2)      # t2 = str[begin]
  add  t3, t1, a0     # t3 = address of str[end]
  lb   t3, 0(t3)      # t3 = str[end]
  bne  t2, t3, isnt   # not a palindrome if not equal
  addi t0, t0, 1      # begin++
  addi t1, t1, -1     # end--
  j    while          # repeat loop

yes:
  addi a0, zero, 1    # set a0 to 1: it is a palindrome
  jr ra               # return
isnt:
  addi a0, zero, 0    # set a0 to 0: it isn't a palindrome
  jr ra               # return
```

CHAPTER 7

Exercise 7.1

(a) ***RegWrite*:** `lw`, `addi`, `jal`, and R-type instructions – *WE3* will be 0, so no data will be written to the Register File.
(b) ***ALUOp*$_1$:** R-type instructions except `add` – These instructions all require a 1 in ALUOp$_1$ for the ALU Decoder to produce the correct Fs*ALUControl* signal.
(c) ***ALUOp*$_0$:** `beq` – The ALU would incorrectly add the registers rather than subtracting them before checking the *Zero* flag.
(d) ***MemWrite*:** `sw` - *WE* will not enable the Data Memory to write.
(e) ***ImmSrc*$_1$:** `beq` and `jal` – The incorrect immediates (offsets) are selected.
(f) ***ImmSrc*$_0$:** `sw` and `jal` – The incorrect immediates are selected.
(g) ***ResultSrc*$_1$:** `jal` – *ALUResult* will be selected instead of *PCPlus4* as the result to write to the Register File.
(h) ***ResultSrc*$_0$:** `lw` – *ALUResult* will be selected instead of *ReadData* as the result to write to the Register File.
(i) ***PCSrc*:** `beq` and `jal` – *PCPlus4* will always be selected as *PCNext* instead of the new *PCTarget*.
(j) ***ALUSrc*:** `lw`, `sw`, and `addi` (and other I-type ALU instructions) – *SrcB* for the ALU will incorrectly select *RD2* instead of *ImmExt*.

Exercise 7.2

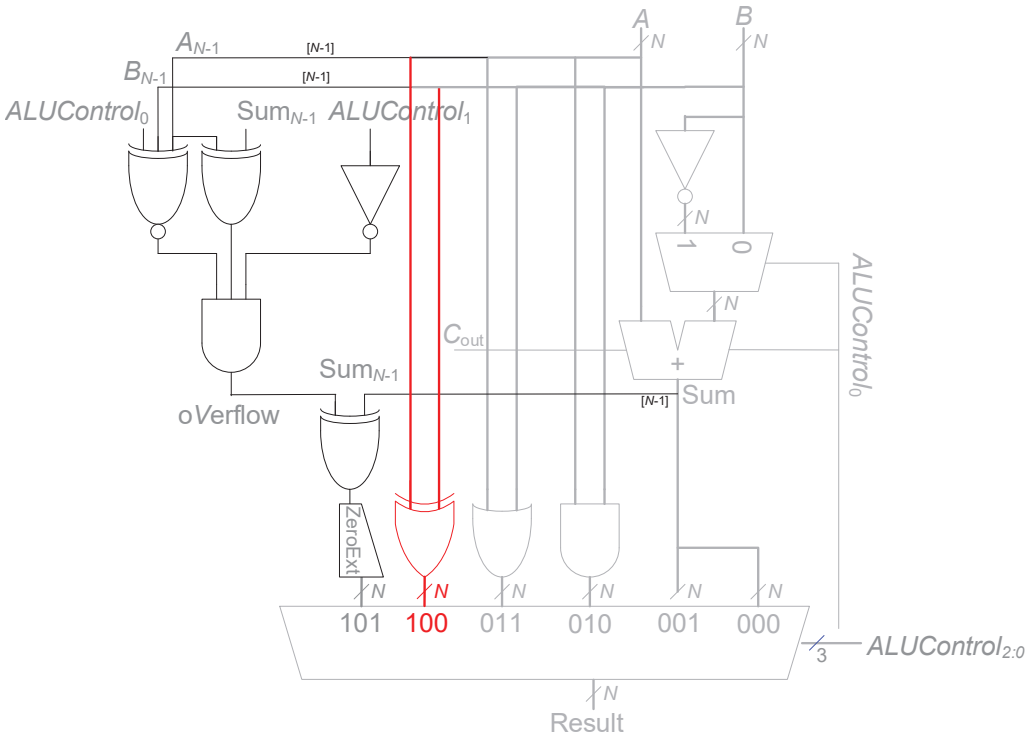
(a) ***RegWrite*:** `sw` and `beq` – These instructions do not write to a destination register.
(b) ***ALUOp*$_1$:** `lw`, `sw`, and `beq` – These instructions require *ALUOp*$_1$ = 0.
(c) ***ALUOp*$_0$:** `lw`, `sw`, `jal`, `addi` (and other I-type ALU), and R-type instructions (except `sub`) – Subtraction is the only ALU operation performed when ALUOp$_0$ = 1.
(d) ***MemWrite*:** `lw`, `beq`, `addi` (and other I-type ALU instructions), `jal`, and R-type instructions –With *MemWrite* always on, memory will be written with random values by all instructions. *MemWrite* should only be 1 for store instructions.
(e) ***ImmSrc*$_1$:** `lw`, `sw`, and `addi` (and other I-type ALU instructions) – An incorrect immediate will be selected.
(f) ***ImmSrc*$_0$:** `lw`, `beq`, and `addi` (and other I-type ALU instructions) – The reasoning is

the same as part (e). For $ImmSrc_0$ = 1, I- and B-type instructions malfunction due to selecting the incorrect immediate.
(g) ***ResultSrc*$_1$:** `lw`, `addi` (and other I-type ALU), and R-type instructions – All instructions that rely on *ALUResult* or reading data from memory will malfunction because only PCPlus4 can be output by the ResultSrc multiplexer.
(h) ***ResultSrc*$_0$:** `jal`, `addi` (and other I-type ALU), and R-type instructions – All instructions that rely on *ALUResult* or *PCPlus4* would fail because the ResultSrc multiplexer could not output those options ($ResultSrc_{1:0}$ = 00 and $ResultSrc_{1:0}$ = 10).
(i) ***PCSrc*:** `lw`, `sw`, `addi`, and R-type instructions – All instructions except branching and jump instructions would malfunction because *PCNext* comes from *PCTarget* instead of *PCPlus4*.
(j) ***ALUSrc*:** and R-type and `beq` instructions – Any instruction that uses two registers operands in the ALU will not work, because ALUSrc = 1 selects the immediate to feed into the second input of the ALU.

Exercise 7.3

(a) `xor`
The datapath does not require any changes to its interfaces. Only the ALU needs to be modified: we add another input to the multiplexer and *N* 2-bit XOR gates within the ALU. We also update the ALU Decoder truth table / logic. The Main Decoder truth table need not be updated because it already supports R-type instructions. These changes are shown below.

Modified ALU to support `xor`



Modified ALU operations to support `xor`

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
100	**xor**
101	SLT

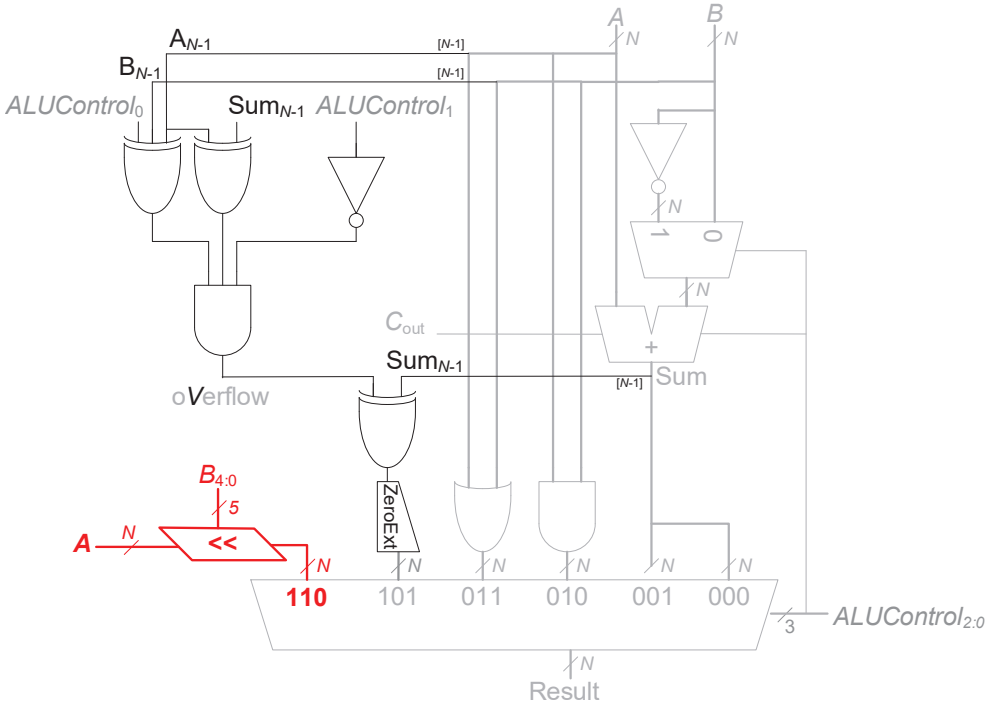
Modified ALU Decoder truth table to support `xor`

ALUOp	funct3	op_5, $funct7_5$	*ALUControl*	Instruction
00	x	x	000 (add)	`lw, sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add, addi`
	000	11	001 (subtract)	`sub`
	010	x	101 (set less than)	`slt, slti`
	100	**x**	**100 (xor)**	**`xor, xori`**
	110	x	011 (or)	`or, ori`
	111	x	010 (and)	`and, andi`

(b) `sll`

The overall datapath (interfaces and units) need not be changed. We only modify the ALU and the ALU Decoder, as shown below. We add a shifter and expand the multiplexer inside the ALU.

Modified ALU to support `sll`



Modified ALU operations to support `sll`

ALUControl$_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT
110	**sll**

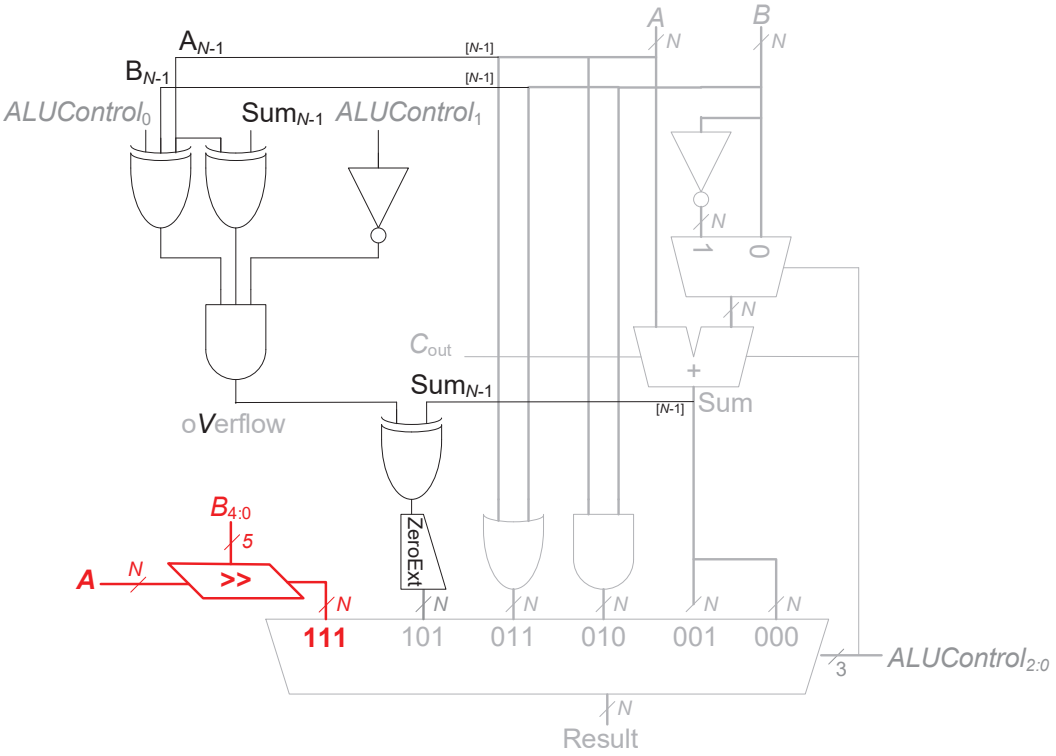
Modified ALU Decoder truth table to support `sll`

ALUOp	funct3	op$_5$, funct7$_5$	*ALUControl*	Instruction
00	x	x	000 (add)	`lw, sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add, addi`
	000	11	001 (subtract)	`sub`
	001	**x**	**110 (shift left logical)**	**`sll, slli`**
	010	x	101 (set less than)	`slt, slti`
	110	x	011 (or)	`or, ori`
	111	x	010 (and)	`and, andi`

(c) `srl`

The overall datapath (interfaces and units) need not be changed. We only modify the ALU and the ALU Decoder, as shown below. We add a shifter and expand the multiplexer inside the ALU.

Modified ALU to support `srl`



Modified ALU operations to support `srl`

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT
111	**srl**

Modified ALU Decoder truth table to support `srl`

ALUOp	funct3	op_5, $funct7_5$	*ALUControl*	Instruction
00	x	xx	000 (add)	`lw, sw`
01	x	xx	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add, addi`
	000	11	001 (subtract)	`sub`
	001	**x0**	**111 (shift right logical)**	**`srl, srli`**
	010	xx	101 (set less than)	`slt, slti`
	110	xx	011 (or)	`or, ori`
	111	xx	010 (and)	`and, andi`

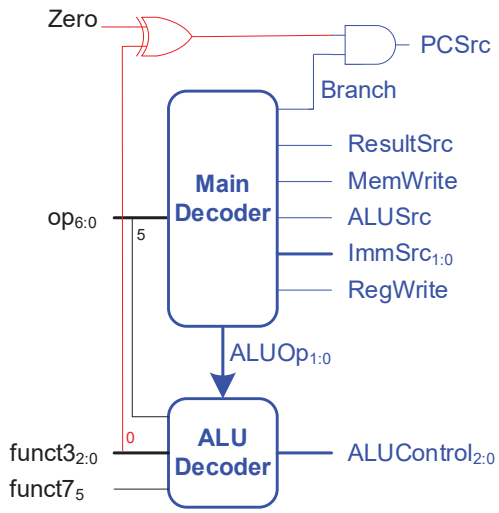
(d) `bne`

`bne` is the opposite of `beq`. `beq` and `bne` can be identified by **func3**$_0$, which is high when `bne` is the instruction. To implement, we simply need to change the control unit to branch when *Zero* is 0 and `bne` is the instruction or when *Zero* is 1 and `beq` is the instruction. This is easily achieved with *Zero* XOR **func3**$_0$.

Main Decoder truth table enhanced to support `bne`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	00	1	0	01	0	00	0
`sw`	0100011	0	01	1	1	xx	0	00	0
R-type	0110011	1	xx	0	0	00	0	10	0
`beq`**`/bne`**	1100011	0	10	0	0	xx	1	01	0
`addi`	0010011	1	00	0	0	00	0	10	0
`jal`	1101111	1	11	x	0	10	0	xx	1

Enhanced control unit for `bne`



Exercise 7.4

(a) `lui`

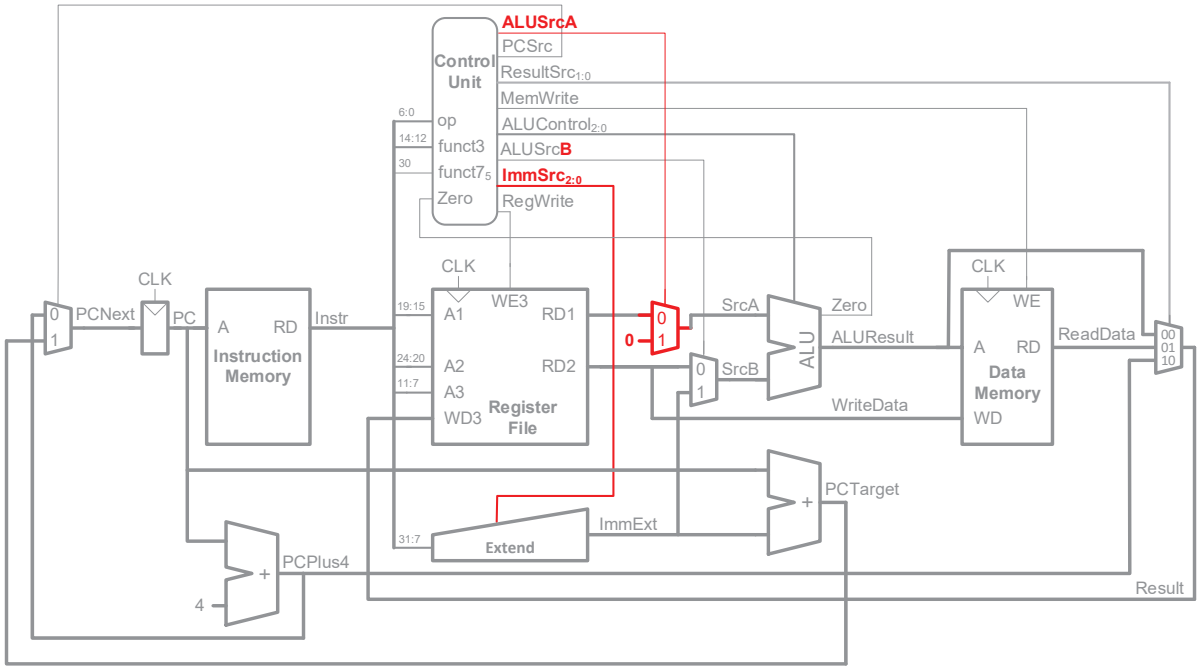
First, we update the immediate Extend unit to support `lui`.

Enhanced *ImmSrc* encoding to support `lui`

ImmSrc	*ImmExt*	Type	Description
000	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
001	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
010	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate
011	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1'b0}	J	21-bit signed immediate
100	**{*Instr*[31:12], 12'b0}**	**U**	**20-bit signed immediate**

Next, we modify the datapath by increasing the width of the *ImmSrc* control signal to 3 bits and by making 0 an option for the ALU's top input (*SrcA*).

Enhanced datapath to support `lui`



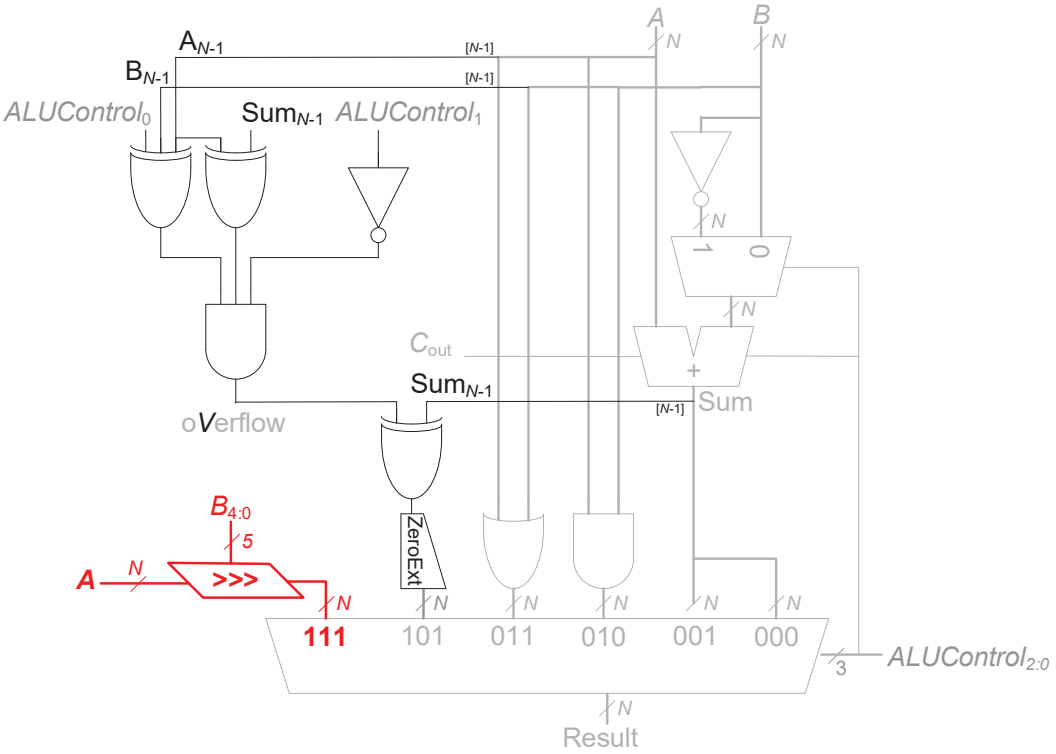
Main Decoder truth table enhanced to support `lui`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrcA	ALUSrcB	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	000	0	1	0	01	0	00	0
`sw`	0100011	0	001	0	1	1	xx	0	00	0
R-type	0110011	1	xxx	0	0	0	00	0	10	0
`beq`	1100011	0	010	0	0	0	xx	1	01	0
I-type ALU	0010011	1	000	0	0	0	00	0	10	0
`jal`	1101111	1	011	x	x	0	10	0	xx	1
`lui`	**0110111**	**1**	**100**	**1**	**1**	**0**	**00**	**0**	**xx**	**0**

(b) `sra`

The overall datapath (interfaces and units) need not be changed. We only modify the ALU and the ALU Decoder, as shown below. We add a shifter and expand the multiplexer inside the ALU.

Modified ALU to support `sra`



Modified ALU operations to support `sra`

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT
111	**sra**

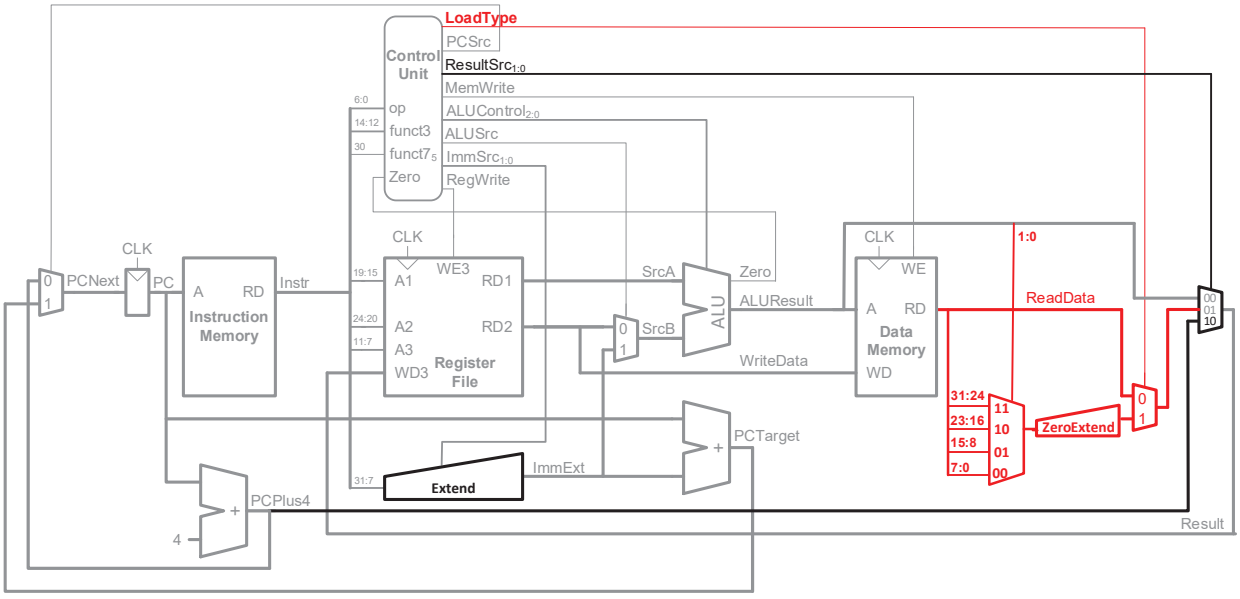
Modified ALU Decoder truth table to support `sra`

ALUOp	funct3	op_5, $funct7_5$	*ALUControl*	Instruction
00	x	xx	000 (add)	`lw, sw`
01	x	xx	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add, addi`
	000	11	001 (subtract)	`sub`
	010	xx	101 (set less than)	`slt, slti`
	101	**x1**	**111 (shift right arithmetic)**	**`sra, srai`**
	110	xx	011 (or)	`or, ori`
	111	xx	010 (and)	`and, andi`

(c) `lbu`

To implement `lbu` we create a load/store unit (LSU) within the controller that outputs a new signal *LoadType*. The LU takes in **funct3** and outputs *LoadType*. When *LoadType* is 0, it is an `lbu` instruction, and a zero-extended byte of the *ReadData* bus (selected using the two least significant bits of the address) is sent to the ResultSrc multiplexer. Otherwise, *ReadData* is sent. We add the following to the datapath: the new signal, *LoadType*, zero-extension unit, 4:1 multiplexer to select the byte within the *ReadData* word, and a LoadType multiplexer.

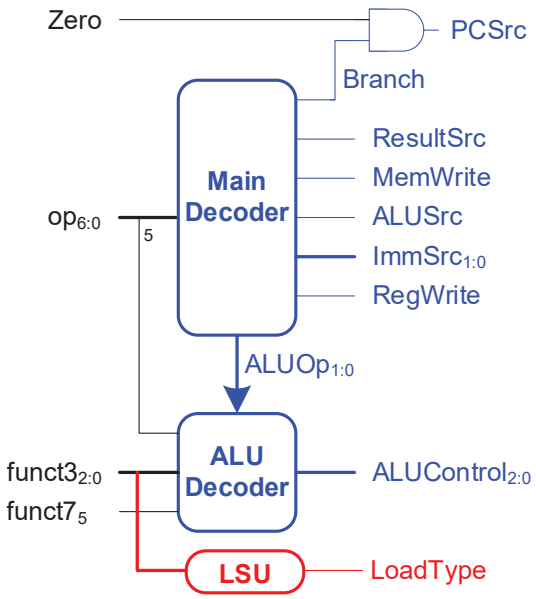
Enhanced datapath to support `lbu`



Main Decoder truth table enhanced to support `lbu`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw` / **`lbu`**	0000011	1	000	1	0	01	0	00	0
`sw`	0100011	0	001	1	1	xx	0	00	0
R-type	0110011	1	xxx	0	0	00	0	10	0
`beq`	1100011	0	010	0	0	xx	1	01	0
I-type ALU	0010011	1	000	0	0	00	0	10	0
`jal`	1101111	1	011	x	0	10	0	xx	1

Enhanced control unit for `lbu`



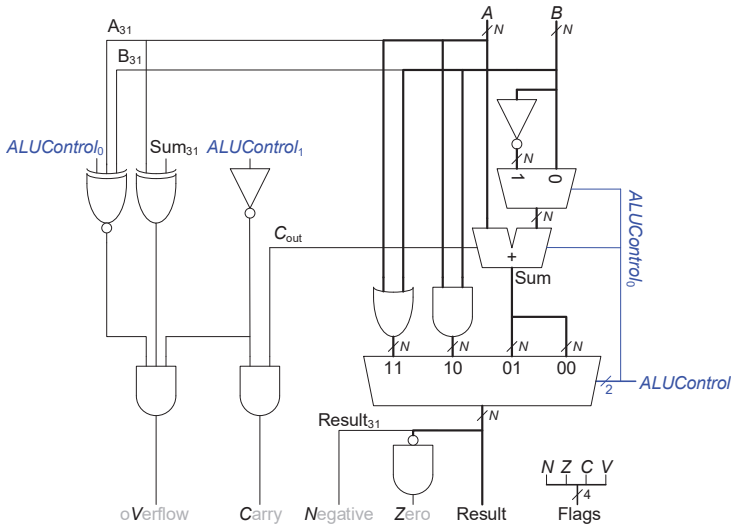
LSU truth table to support `lbu`

funct3	*LoadType*	Instruction
010	0	`lw`
100	1	`lbu`

(d) `blt`

We start by using the ALU with flags provided in the book, shown again below.

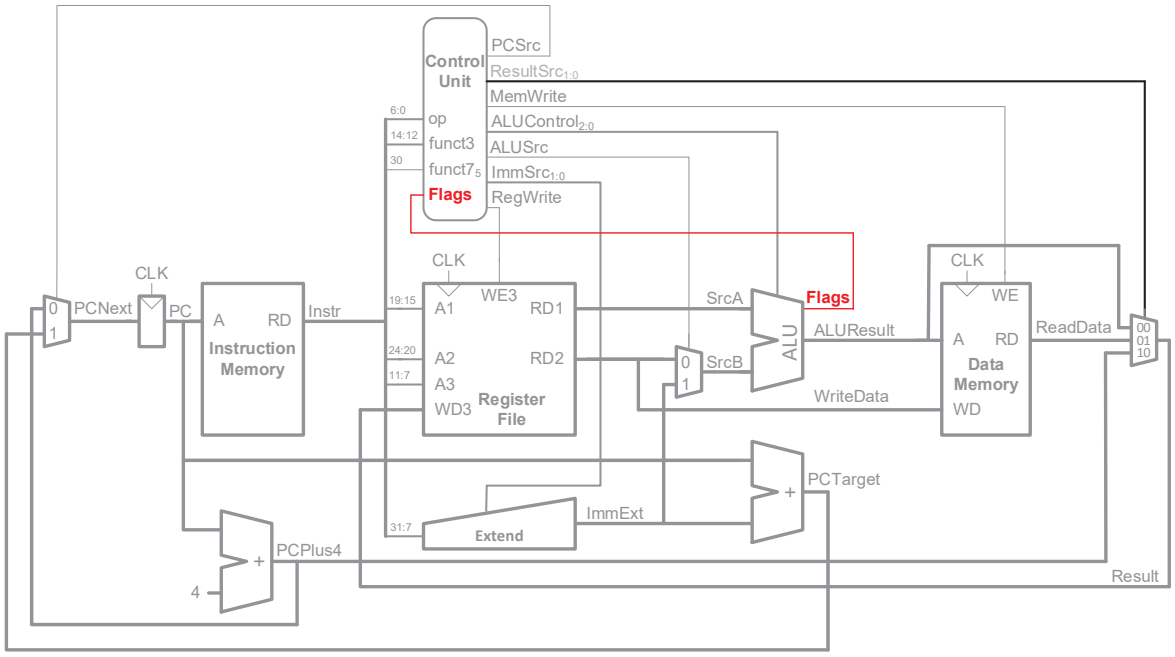
ALU with flags from the textbook



A is less than B when the Negative flag is 1 and oVerflow is 0 or when $N = 0$ and $V = 1$. (Remember that when overflow occurs, it means the result has the incorrect sign.) So, $A < B$ when N XOR $V = 1$.

We also add a *BranchType* internal signal of the Controller that is 1 when a `blt` instruction is executing and 0 when a `beq` instruction is executing.

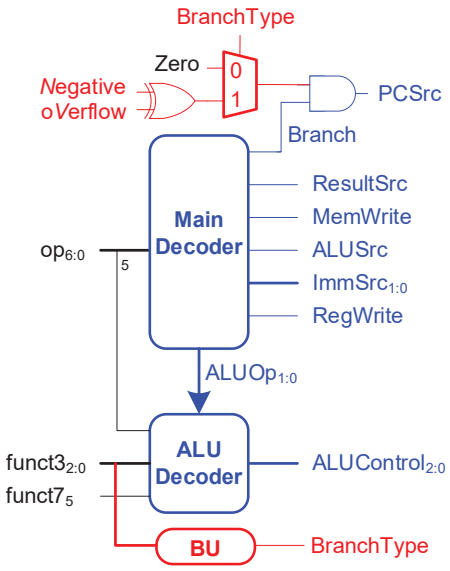
Enhanced datapath to support `blt`



Main Decoder truth table enhanced to support `blt`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	000	1	0	01	0	00	0
`sw`	0100011	0	001	1	1	xx	0	00	0
R-type	0110011	1	xxx	0	0	00	0	10	0
`beq` / **`blt`**	1100011	0	010	0	0	xx	1	01	0
I-type ALU	0010011	1	000	0	0	00	0	10	0
`jal`	1101111	1	011	x	0	10	0	xx	1

Enhanced control unit for `blt`



BU truth table to support `blt`

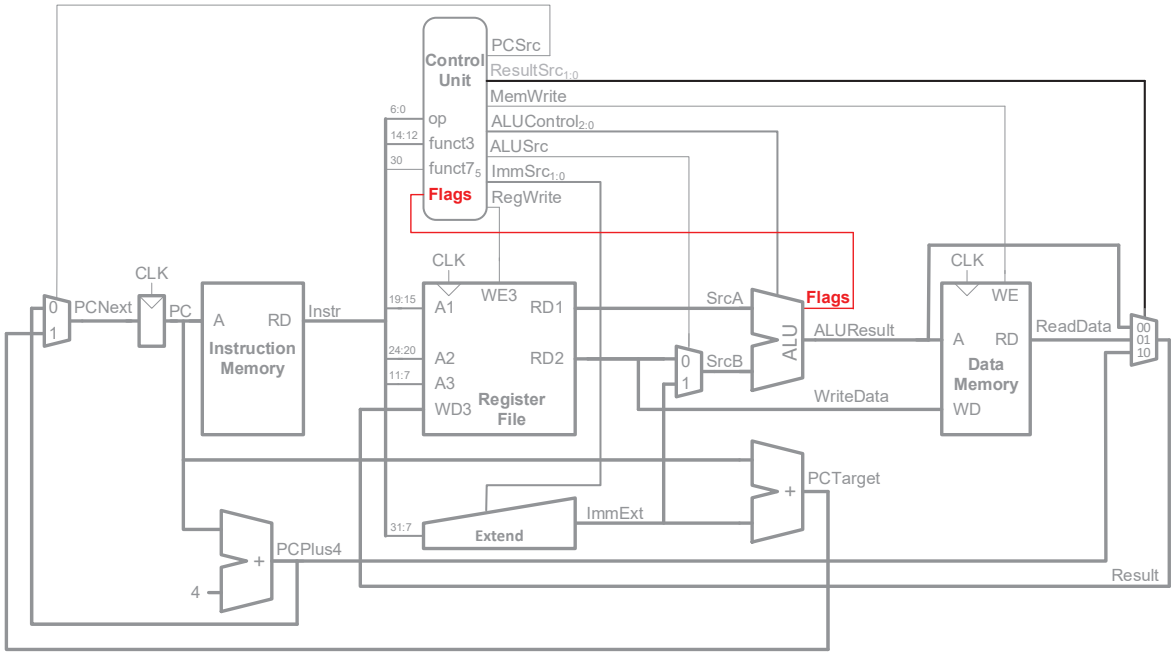
funct3	*BranchType*	Instruction
000	0	`beq`
100	1	`blt`

(e) `bltu`

We start by using the ALU with flags (see Figure 5.17 in the textbook). For unsigned numbers, *A* is less than *B* when the *C*arry flag is 0.

We also add a *BranchType* output of the Controller that is 1 when a `bltu` instruction is executing and 0 when a `beq` instruction is executing. We add a Branch Unit (BU) to the controller to produce this signal (*BranchType*) using **funct3** as its input.

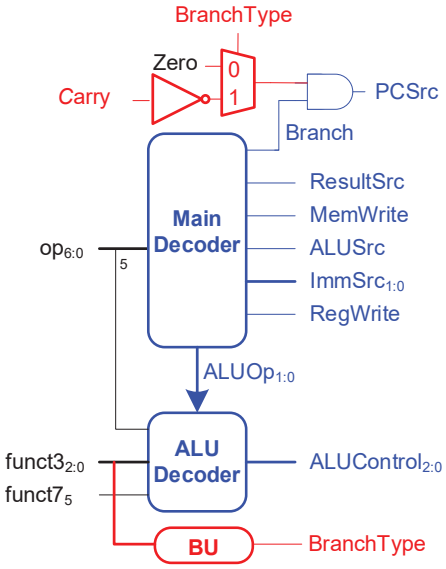
Enhanced datapath to support `bltu`



Main Decoder truth table enhanced to support `bltu`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	000	1	0	01	0	00	0
`sw`	0100011	0	001	1	1	xx	0	00	0
R-type	0110011	1	xxx	0	0	00	0	10	0
`beq` / **`bltu`**	1100011	0	010	0	0	xx	1	01	0
I-type ALU	0010011	1	000	0	0	00	0	10	0
`jal`	1101111	1	011	x	0	10	0	xx	1

Enhanced control unit for `bltu`



BU truth table to support `bltu`

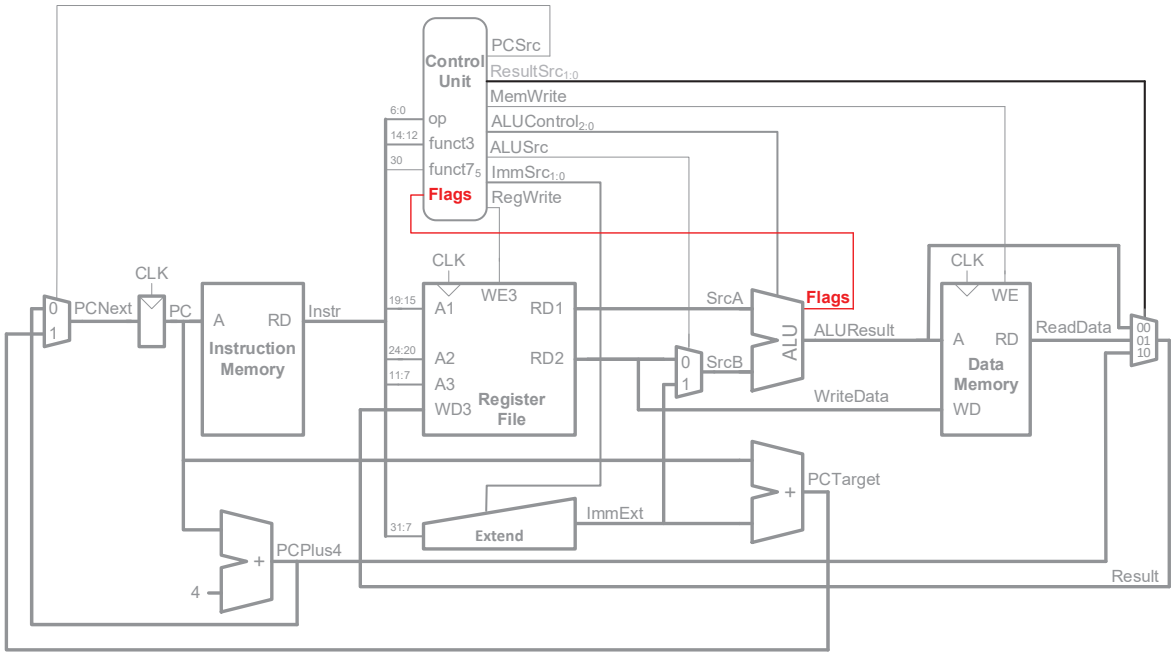
funct3	*BranchType*	Instruction
000	0	`beq`
110	1	`bltu`

(f) `bge`

We start by using the ALU with flags (see Figure 5.17 in the textbook). For signed numbers, A is greater than or equal to B when A is not less than B. So, $A >= B$, when N XNOR V.

We also add a *BranchType* internal signal of the Controller that is 1 when a `bge` instruction is executing and 0 when a `beq` instruction is executing. We add a Branch Unit (BU) to the controller to produce this signal (*BranchType*) using **funct3** as its input.

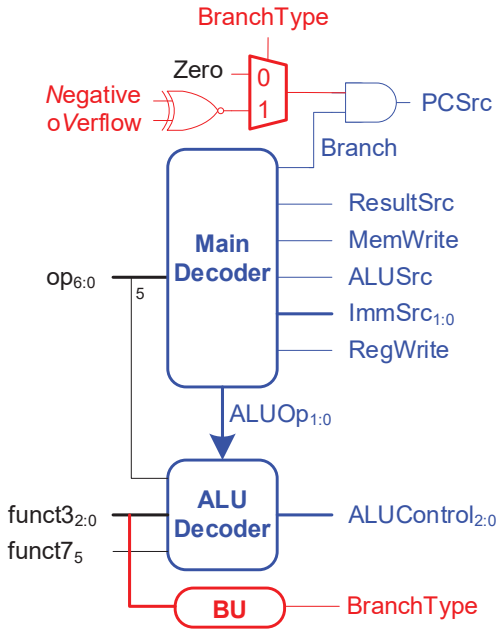
Enhanced datapath to support bge



Main Decoder truth table enhanced to support bge

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	000	1	0	01	0	00	0
`sw`	0100011	0	001	1	1	Xx	0	00	0
R-type	0110011	1	xxx	0	0	00	0	10	0
`beq` / **bge**	1100011	0	010	0	0	xx	1	01	0
I-type ALU	0010011	1	000	0	0	00	0	10	0
`jal`	1101111	1	011	x	0	10	0	xx	1

Enhanced control unit for `bge`



Branch Unit (BU) truth table to support `bge`

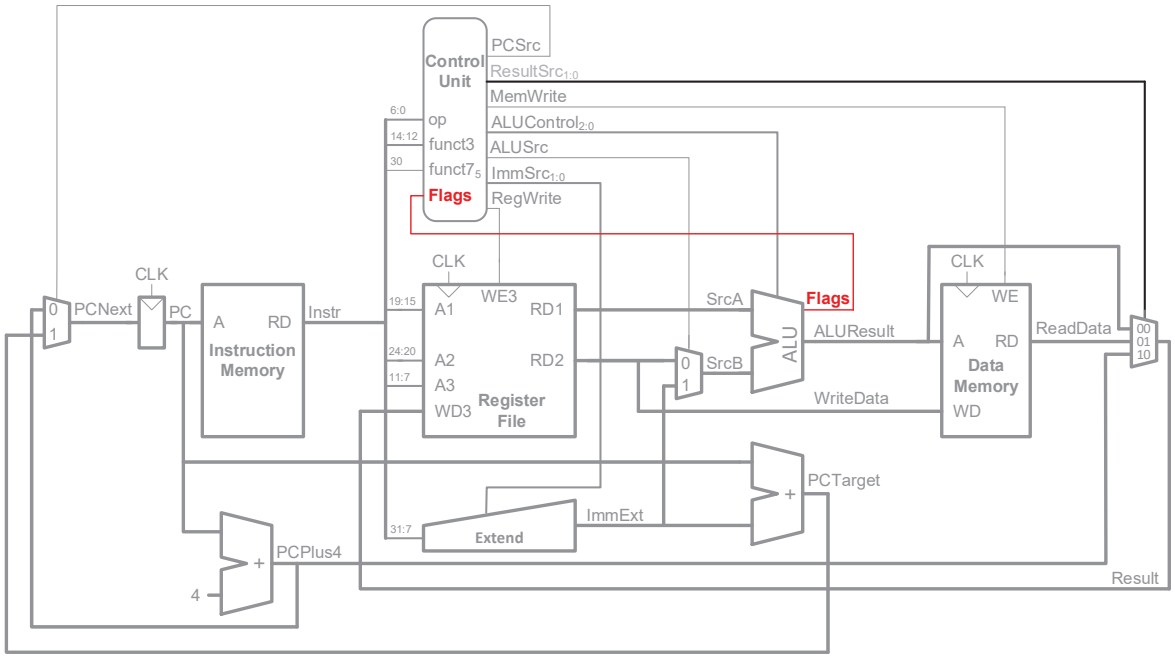
funct3	*BranchType*	Instruction
000	0	`beq`
110	1	`bge`

(f) `bgeu`

We start by using the ALU with flags (see Figure 5.17 in the textbook). For unsigned numbers, *A* is greater than or equal to *B* when the ALU performs A-B and there is a carry out.

We also add a *BranchType* internal signal of the Controller that is 1 when a `bgeu` instruction is executing and 0 when a `beq` instruction is executing. We add a Branch Unit (BU) to the controller to produce this signal (*BranchType*) using **funct3** as its input.

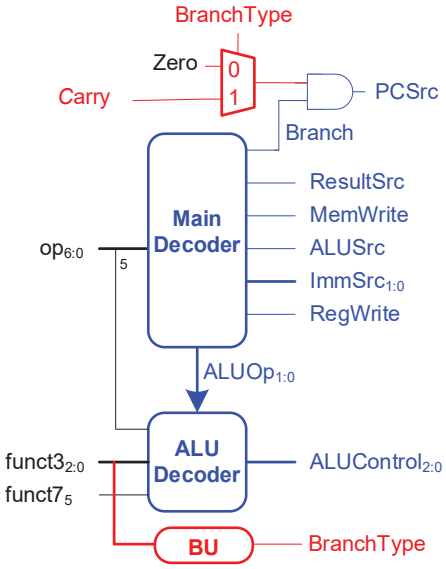
Enhanced datapath to support `bgeu`



Main Decoder truth table enhanced to support `bgeu`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	000	1	0	01	0	00	0
`sw`	0100011	0	001	1	1	Xx	0	00	0
R-type	0110011	1	xxx	0	0	00	0	10	0
`beq` / **`bgeu`**	1100011	0	010	0	0	xx	1	01	0
I-type ALU	0010011	1	000	0	0	00	0	10	0
`jal`	1101111	1	011	x	0	10	0	xx	1

Enhanced control unit for `bgeu`



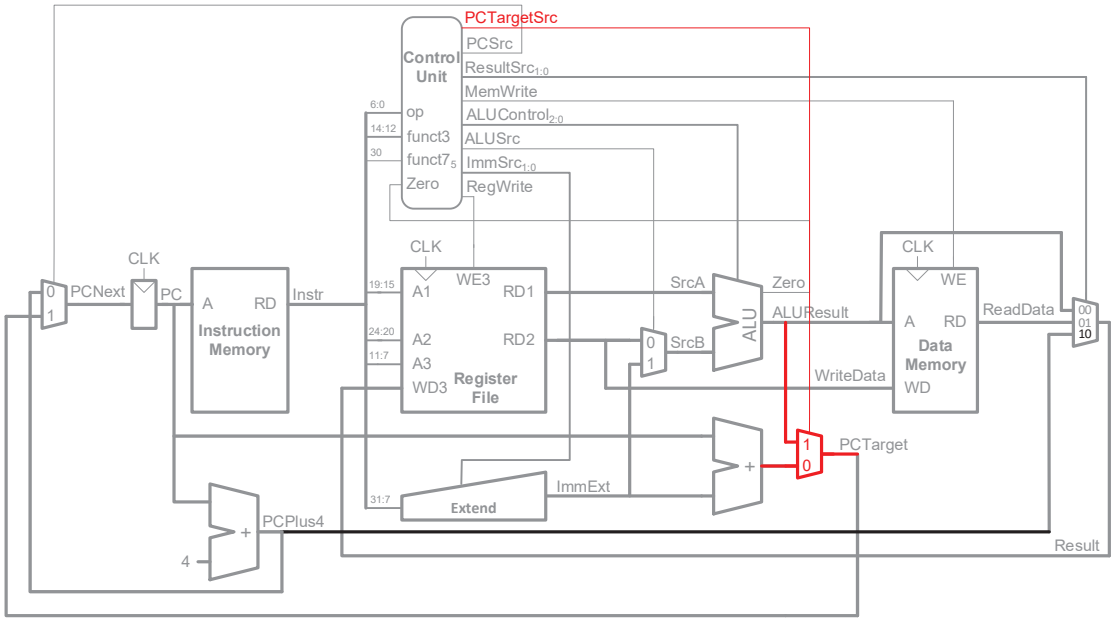
BU truth table to support `bgeu`

funct3	*BranchType*	Instruction
000	0	`beq`
111	1	`bgeu`

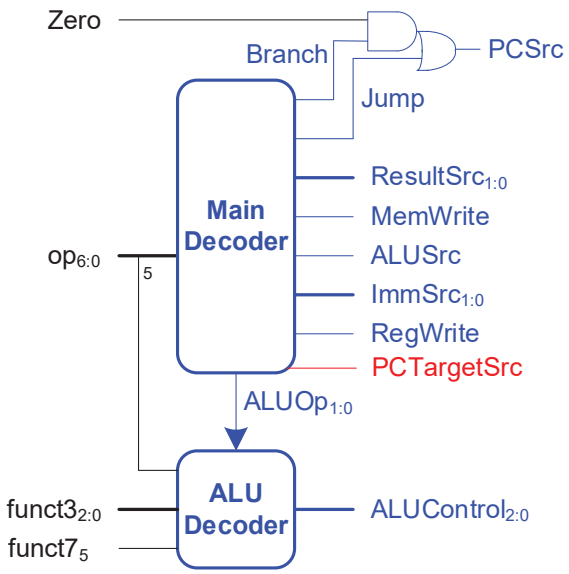
(h) `jalr`

The `jalr` instruction jumps to the target address calculated by adding **rs1** and the sign-extended 12-bit immediate. It also writes PC+4 to **rd**. The datapath already supports these calculations: the ALU can perform **rs1** + *ImmExt*; and PC+4 can be routed to the *Result* signal by the ResultSrc multiplexer. Only the ALU output needs to be routed to the PCTarget signal. So, we add a control signal (*PCTargetSrc*) and a multiplexer to select the correct PCTarget (either from the ALU or the PC-relative adder).

Enhanced datapath to support `jalr`



Main Decoder enhanced to support `jalr`



Main Decoder truth table enhanced to support `jalr`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump	PCTargetSrc
`lw`	0000011	1	000	1	0	01	0	00	0	x
`sw`	0100011	0	001	1	1	xx	0	00	0	x
R-type	0110011	1	xxx	0	0	00	0	10	0	x
`beq`	1100011	0	010	0	0	xx	1	01	0	0

I-type ALU	0010011	1	000	0	0	00	0	10	0	x
`jal`	1101111	1	011	x	0	10	0	xx	1	0
`jalr`	**1100111**	**1**	**000**	**1**	**0**	**10**	**0**	**xx**	**1**	**1**

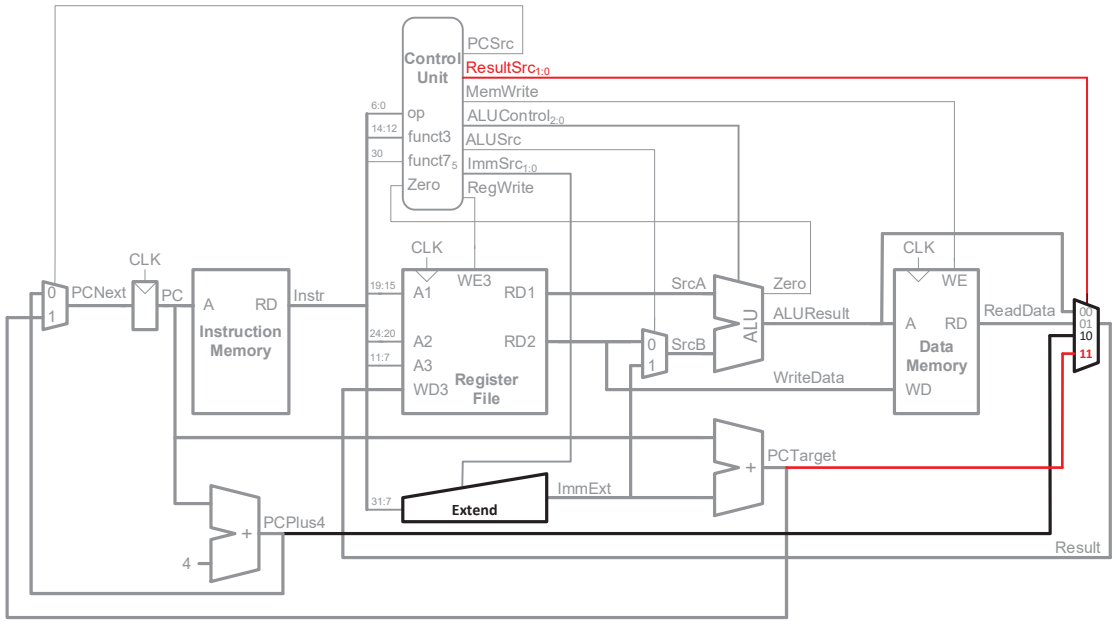
(i) `auipc`

The ImmSrc encoding table needs to be expanded to support U-type instructions (`auipc` in this case).

Enhanced ImmSrc encoding to support U-type instructions

ImmSrc	*ImmExt*	Type	Description
000	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
001	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
010	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate
011	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1'b0}	J	21-bit signed immediate
100	**{*Instr*[31:12], 12'b0}**	**U**	**20-bit upper immediate**

Enhanced datapath to support `auipc`



Main Decoder truth table enhanced to support `auipc`

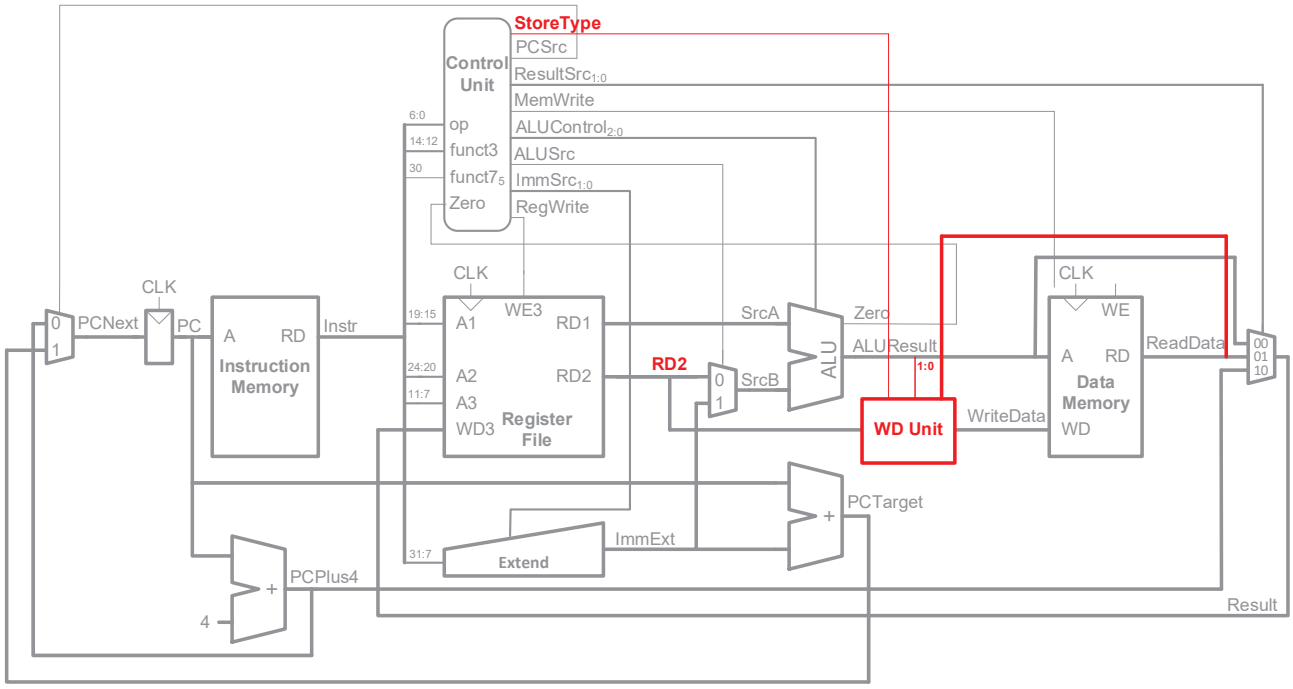
Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	000	1	0	01	0	00	0
`sw`	0100011	0	001	1	1	xx	0	00	0
R-type	0110011	1	xxx	0	0	00	0	10	0
`beq`	1100011	0	010	0	0	xx	1	01	0
I-type ALU	0010011	1	000	0	0	00	0	10	0
`jal`	1101111	1	011	x	0	10	0	xx	1
`auipc`	**0010111**	**1**	**100**	**x**	**0**	**11**	**0**	**xx**	**0**

(j) `sb`

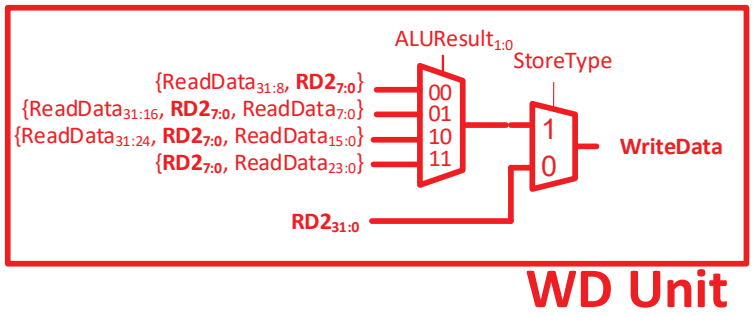
We add a WriteData Unit (WD Unit) to the datapath to write either RD2 or RD2 bit-swizzled with ReadData for sub-word writes. For `sb`, the least significant byte (lsb) of

RD2 (**rs2**'s contents), replaces a byte of data in the *ReadData* bus depending on the byte offset of the memory address, $ALUResult_{1:0}$. We add a *StoreType* output to the control unit to choose either the entire word or the bitswizzled word to write to memory. The updated figures and additional hardware is shown below.

Enhanced datapath to support `sb` (WD Unit shown in next figure)



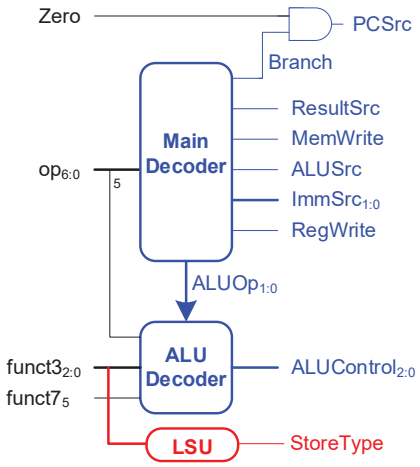
WD Unit



Main Decoder truth table enhanced to support `sb`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	000	1	0	01	0	00	0
`sw` / **`sb`**	0100011	0	001	1	1	xx	0	00	0
R-type	0110011	1	xxx	0	0	00	0	10	0
`beq`	1100011	0	010	0	0	xx	1	01	0
I-type ALU	0010011	1	000	0	0	00	0	10	0
`jal`	1101111	1	011	x	0	10	0	xx	1

Enhanced control unit for `sb` : added Load/Store Unit (LSU)



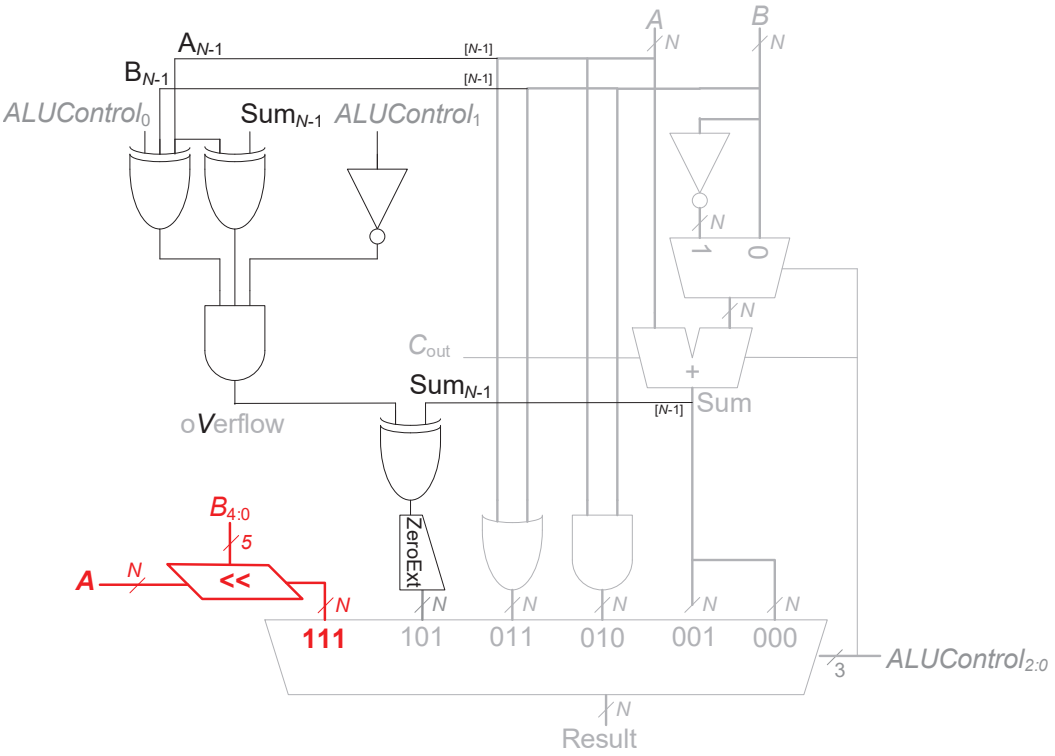
Load/Store Unit (LSU) truth table to support `sb`

funct3	StoreType	Instruction
000	1	`sb`
010	0	`sw`

(k) `slli`

The overall datapath (interfaces and units) need not be changed. We only modify the ALU and the ALU Decoder, as shown below. We add a shifter and expand the multiplexer inside the ALU.

Modified ALU to support `slli`



Modified ALU operations to support `slli`

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT
111	**sll**

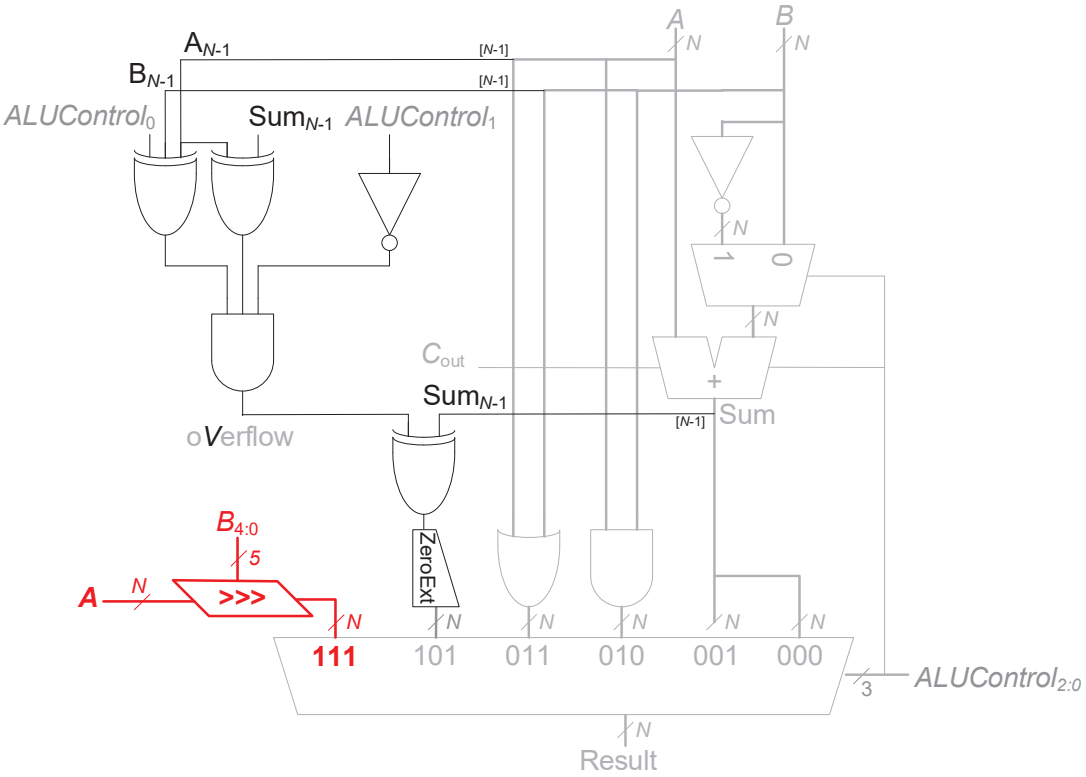
Modified ALU Decoder truth table to support `slli`

ALUOp	funct3	op_5, $funct7_5$	*ALUControl*	Instruction
00	x	x	000 (add)	`lw, sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add, addi`
	000	11	001 (subtract)	`sub`
	010	x	101 (set less than)	`slt, slti`
	001	**x**	**111 (shift left logical)**	**`sll, slli`**
	110	x	011 (or)	`or, ori`
	111	x	010 (and)	`and, andi`

(l) `srai`

The overall datapath already supports I-type ALU operations, so the datapath does not need to be changed. We only modify the ALU and the ALU Decoder, as shown below. We add a shifter and expand the multiplexer inside the ALU.

Modified ALU to support `srai`



Modified ALU operations to support `srai`

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT
111	**sra**

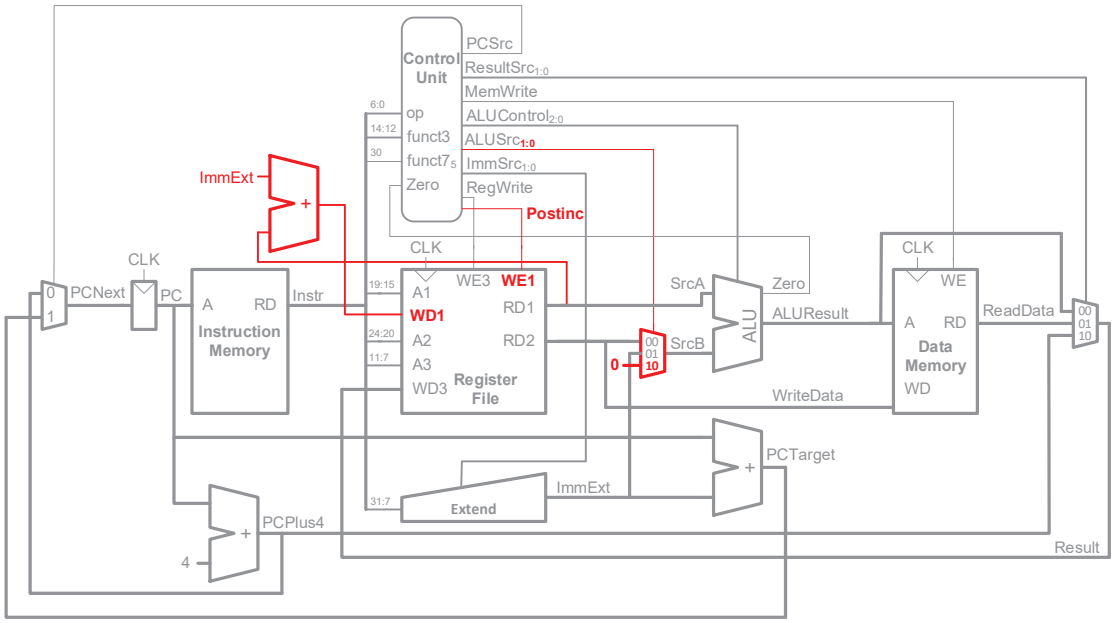
Modified ALU Decoder truth table to support `srai`

ALUOp	funct3	op$_5$, funct7$_5$	*ALUControl*	Instruction
00	x	xx	000 (add)	`lw, sw`
01	x	xx	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add, addi`
	000	11	001 (subtract)	`sub`
	010	xx	101 (set less than)	`slt, slti`
	101	**x1**	**111 (shift right arithmetic)**	**`sra, srai`**
	110	xx	011 (or)	`or, ori`
	111	xx	010 (and)	`and, andi`

Exercise 7.5

To implement `lwpostinc`, we have to modify the register file by adding another write port and another adder. We also add a new control signal, *Postinc*, and expand the ALUSrc multiplexer to include 0 as an input. That way, the address is calculated as **rs1** + 0. The extra adder produces **rs1** + *ImmExt* and writes it back to **rs1**.

Enhanced datapath to support `lwpostinc`

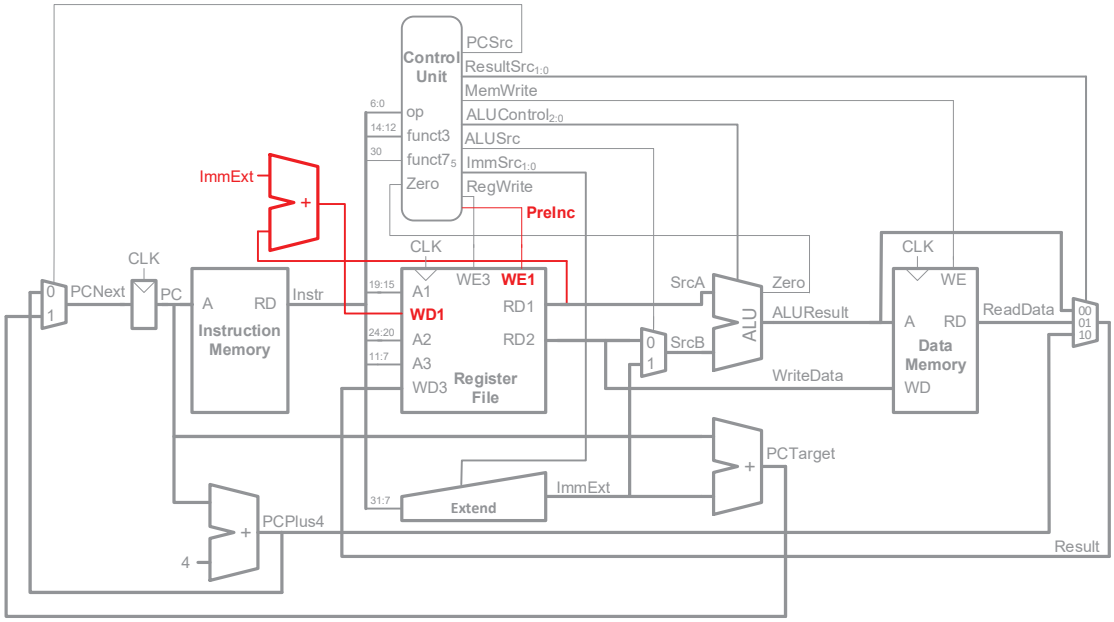


Main Decoder truth table enhanced to support `lwpostinc`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump	PostInc
`lw`	0000011	1	000	**01**	0	01	0	00	0	**0**
`sw`	0100011	0	001	**01**	1	xx	0	00	0	**0**
R-type	0110011	1	xxx	**00**	0	00	0	10	0	**0**
`beq`	1100011	0	010	**00**	0	xx	1	01	0	**0**
I-type ALU	0010011	1	000	**00**	0	00	0	10	0	**0**
`jal`	1101111	1	011	xx	0	10	0	xx	1	**0**
`lwpostinc`	**new op**	**1**	**000**	**01**	**0**	**01**	**0**	**00**	**0**	**1**

Exercise 7.6

To implement `lwpreinc`, we have to modify the register file by adding another write port and another adder. We also add a new control signal, *Preinc*. The ALU creates the address by adding rs1 + imm, as on a typical load. The extra adder produces **rs1** + *ImmExt* and writes it back to **rs1**.



Main Decoder truth table enhanced to support `lwpostinc`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump	PostInc
`lw`	0000011	1	000	1	0	01	0	00	0	**0**
`sw`	0100011	0	001	1	1	xx	0	00	0	**0**
R-type	0110011	1	xxx	0	0	00	0	10	0	**0**
`beq`	1100011	0	010	0	0	xx	1	01	0	**0**
I-type ALU	0010011	1	000	0	0	00	0	10	0	**0**
`jal`	1101111	1	011	x	0	10	0	xx	1	**0**
`lwpreinc`	**new op**	**1**	**000**	**1**	**0**	**01**	**0**	**00**	**0**	**1**

Exercise 7.7

To increase performance most (i.e., decrease cycle time), the crack circuit designer should speed up the Memory Unit. From Equation 7.3:

$$T_{c\_new} = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup}$$
$$= 40 + 2(100) + 100 + 120 + 30 + 60 = \mathbf{550}\ \boldsymbol{ps}$$

Exercise 7.8

From Equation 7.3:

$T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup}$

$= 40 + 2(200) + 100 + (\mathbf{120 - 20}) + 30 + 60 = \mathbf{730\ ps}$

It would take **73 seconds** (730 ps/instruction * 100×10^9 instructions) to execute 100 billion instructions.

Exercise 7.9

RISC-V single-cycle processor SystemVerilog:

```
// Modified to include all Exercise 7.3 instructions (xor, sll, srl, bne)

module testbench();

  logic        clk;
  logic        reset;

  logic [31:0] WriteData, DataAdr;
  logic        MemWrite;

  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check results
  always @(negedge clk)
    begin
      if(MemWrite) begin
        if(DataAdr === 216 & WriteData === 4140) begin
          $display("Simulation succeeded");
          $stop;
        end
      end
    end
endmodule

module top(input  logic        clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic        MemWrite);

  logic [31:0] PC, Instr, ReadData;
```

```
  // instantiate processor and memories
  riscvsingle rvsingle(clk, reset, PC, Instr, MemWrite, DataAdr,
                       WriteData, ReadData);
  imem imem(PC, Instr);
  dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule

module riscvsingle(input  logic        clk, reset,
                   output logic [31:0] PC,
                   input  logic [31:0] Instr,
                   output logic        MemWrite,
                   output logic [31:0] ALUResult, WriteData,
                   input  logic [31:0] ReadData);

  logic       ALUSrc, RegWrite, Jump, Zero;
  logic [1:0] ResultSrc, ImmSrc;
  logic [2:0] ALUControl;

  controller c(Instr[6:0], Instr[14:12], Instr[30], Zero,
               ResultSrc, MemWrite, PCSrc,
               ALUSrc, RegWrite, Jump,
               ImmSrc, ALUControl);
  datapath dp(clk, reset, ResultSrc, PCSrc,
              ALUSrc, RegWrite,
              ImmSrc, ALUControl,
              Zero, PC, Instr,
              ALUResult, WriteData, ReadData);
endmodule

module controller(input  logic [6:0] op,
                  input  logic [2:0] funct3,
                  input  logic       funct7b5,
                  input  logic       Zero,
                  output logic [1:0] ResultSrc,
                  output logic       MemWrite,
                  output logic       PCSrc, ALUSrc,
                  output logic       RegWrite, Jump,
                  output logic [1:0] ImmSrc,
                  output logic [2:0] ALUControl);

  logic [1:0] ALUOp;
  logic       Branch;

  maindec md(op, ResultSrc, MemWrite, Branch,
             ALUSrc, RegWrite, Jump, ImmSrc, ALUOp);
  aludec  ad(op[5], funct3, funct7b5, ALUOp, ALUControl);

  // added XOR gate for bne
  assign PCSrc = (Branch & (Zero ^ funct3[0])) | Jump;
endmodule

module maindec(input  logic [6:0] op,
               output logic [1:0] ResultSrc,
               output logic       MemWrite,
               output logic       Branch, ALUSrc,
               output logic       RegWrite, Jump,
               output logic [1:0] ImmSrc,
```

```
               output logic [1:0] ALUOp);

  logic [10:0] controls;

  assign {RegWrite, ImmSrc, ALUSrc, MemWrite,
          ResultSrc, Branch, ALUOp, Jump} = controls;

  always_comb
    case(op)
    // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump
      7'b0000011: controls = 11'b1_00_1_0_01_0_00_0; // lw
      7'b0100011: controls = 11'b0_01_1_1_00_0_00_0; // sw
      7'b0110011: controls = 11'b1_xx_0_0_00_0_10_0; // R-type
      7'b1100011: controls = 11'b0_10_0_0_00_1_01_0; // beq
      7'b0010011: controls = 11'b1_00_1_0_00_0_10_0; // I-type ALU
      7'b1101111: controls = 11'b1_11_0_0_10_0_00_1; // jal
      default:    controls = 11'bx_xx_x_x_xx_x_xx_x; // non-implemented
instruction
    endcase
endmodule

module aludec(input  logic       opb5,
              input  logic [2:0] funct3,
              input  logic       funct7b5,
              input  logic [1:0] ALUOp,
              output logic [2:0] ALUControl);

  logic  RtypeSub;
  assign RtypeSub = funct7b5 & opb5;  // TRUE for R-type subtract instruction

  always_comb
    case(ALUOp)
      2'b00:                ALUControl = 3'b000; // addition
      2'b01:                ALUControl = 3'b001; // subtraction
      default: case(funct3) // R-type or I-type ALU
                 3'b000:  if (RtypeSub)
                            ALUControl = 3'b001; // sub
                          else
                            ALUControl = 3'b000; // add, addi
                 3'b001:    ALUControl = 3'b110; // sll, slli
                 3'b010:    ALUControl = 3'b101; // slt, slti
                 3'b100:    ALUControl = 3'b100; // xor, xori
                 3'b101:    ALUControl = 3'b111; // srl, srli
                 3'b110:    ALUControl = 3'b011; // or, ori
                 3'b111:    ALUControl = 3'b010; // and, andi
                 default:   ALUControl = 3'bxxx; // ???
               endcase
    endcase
endmodule

module datapath(input  logic        clk, reset,
                input  logic [1:0]  ResultSrc,
                input  logic        PCSrc, ALUSrc,
                input  logic        RegWrite,
                input  logic [1:0]  ImmSrc,
                input  logic [2:0]  ALUControl,
                output logic        Zero,
```

```
                output logic [31:0] PC,
                input  logic [31:0] Instr,
                output logic [31:0] ALUResult, WriteData,
                input  logic [31:0] ReadData);

  logic [31:0] PCNext, PCPlus4, PCTarget;
  logic [31:0] ImmExt;
  logic [31:0] SrcA, SrcB;
  logic [31:0] Result;

  // next PC logic
  flopr #(32) pcreg(clk, reset, PCNext, PC);
  adder       pcadd4(PC, 32'd4, PCPlus4);
  adder       pcaddbranch(PC, ImmExt, PCTarget);
  mux2 #(32)  pcmux(PCPlus4, PCTarget, PCSrc, PCNext);

  // register file logic
  regfile     rf(clk, RegWrite, Instr[19:15], Instr[24:20],
                 Instr[11:7], Result, SrcA, WriteData);
  extend      ext(Instr[31:7], ImmSrc, ImmExt);

  // ALU logic
  mux2 #(32)  srcbmux(WriteData, ImmExt, ALUSrc, SrcB);
  alu         alu(SrcA, SrcB, ALUControl, ALUResult, Zero);
  mux3 #(32)  resultmux(ALUResult, ReadData, PCPlus4, ResultSrc, Result);
endmodule

module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [ 4:0] a1, a2, a3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally (A1/RD1, A2/RD2)
  // write third port on rising edge of clock (A3/WD3/WE3)
  // register 0 hardwired to 0

  always_ff @(posedge clk)
    if (we3) rf[a3] <= wd3;

  assign rd1 = (a1 != 0) ? rf[a1] : 0;
  assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule

module adder(input  [31:0] a, b,
             output [31:0] y);

  assign y = a + b;
endmodule

module extend(input  logic [31:7] instr,
              input  logic [1:0]  immsrc,
              output logic [31:0] immext);
```

```
  always_comb
    case(immsrc)
                // I-type
      2'b00:    immext = {{20{instr[31]}}, instr[31:20]};
                // S-type (stores)
      2'b01:    immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
                // B-type (branches)
      2'b10:    immext = {{20{instr[31]}}, instr[7], instr[30:25],
instr[11:8], 1'b0};
                // J-type (jal)
      2'b11:    immext = {{12{instr[31]}}, instr[19:12], instr[20],
instr[30:21], 1'b0};
      default: immext = 32'bx; // undefined
    endcase
endmodule

module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);

  assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module imem(input  logic [31:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
      $readmemh("example.txt",RAM);

  assign rd = RAM[a[31:2]]; // word aligned
endmodule

module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);
```

```
  logic [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule

module alu(input  logic [31:0] a, b,
           input  logic [2:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum;
  logic        v;             // overflow
  logic        isAddSub;      // true when is add or subtract operation

  assign condinvb = alucontrol[0] ? ~b : b;
  assign sum = a + condinvb + alucontrol[0];
  assign isAddSub = ~alucontrol[2] & ~alucontrol[1] |
                    ~alucontrol[1] &  alucontrol[0];

  always_comb
    case (alucontrol)
      3'b000:  result = sum;         // add
      3'b001:  result = sum;         // subtract
      3'b010:  result = a & b;       // and
      3'b011:  result = a | b;       // or
      3'b100:  result = a ^ b;       // xor
      3'b101:  result = sum[31] ^ v; // slt
      3'b110:  result = a << b[4:0]; // sll
      3'b111:  result = a >> b[4:0]; // srl
      default: result = 32'bx;
    endcase

  assign zero = (result == 32'b0);
  assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;

endmodule
```

Modified test program:

```
# If successful, it should write the value 4140 (0x102C) to address 216 (0xd8).
#       RISC-V Assembly         Description                 Address   Machine Code
main:   addi x2, x0, 5          # x2 = 5                    0         00500113
        addi x3, x0, 12         # x3 = 12                   4         00C00193
        addi x7, x3, -9         # x7 = (12 - 9) = 3         8         FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7         C         0023E233
        and  x5, x3, x4         # x5 = (12 AND 7) = 4       10        0041F2B3
        add  x5, x5, x4         # x5 = (4 + 7) = 11         14        004282B3
        sll  x5, x5, x3         # x5 = 11 << 12 = 45,056    18        003292b3
        srl  x5, x5, x2         # x5 = 45,056 >> 5 = 1408   1C        0022d2b3
        bne  x5, x3, skip       # 1408 != 12: branch taken  20        00329463
        sll  x5, x5, x3         # shouldn't execute         24        003292b3
skip:   beq  x5, x7, end        # shouldn't be taken        28        02728863
        slt  x4, x3, x4         # x4 = (12 < 7) = 0         2C        0041A233
        beq  x4, x0, around     # should be taken           30        00020463
        addi x5, x0, 0          # shouldn't execute         34        00000293
around: slt  x4, x7, x2         # x4 = (3 < 5)  = 1         38        0023A233
        add  x7, x4, x5         # x7 = (1 + 1408) = 1409    3C        005203B3
        sub  x7, x7, x2         # x7 = (1409 - 5) = 1404    40        402383B3
```

```
        sw   x7, 200(x3)        # [212] = 1404              44        0c71a423
        lw   x2, 212(x0)        # x2 = [212] = 1404         48        0d402103
        add  x9, x2, x5         # x9 = (1404 + 1408) = 2812 4C        005104B3
        jal  x3, end            # jump to end, x3 = 0x54    50        008001EF
        addi x2, x0, 1          # shouldn't execute         54        00100113
end:    add  x2, x2, x9         # x2 = (1404 + 2812) = 4216 58        00910133
        addi x4, x0, -1         # x4 = 0xFFFFFFFF           5C        fff00213
        addi x5, x0, 1          # x5 = 1                    60        00100293
        addi x6, x0, 31         # x6 = 31                   64        01f00313
        sll  x6, x5, x6         # x6 = 0x80000000           68        00629333
        xor  x5, x4, x6         # x5 = 0x7FFFFFFF           6C        006242b3
        slt  x6, x5, x4         # x6 = 0                    70        0042a333
wrong:  bne  x6, x0, wrong      # shouldn't be taken        74        00031063
        xor  x2, x2, x3         # x2 = 4216 ^ 0x54 = 4140   78        00314133
        sw   x2, 0x84(x3)       # mem[216] = 0x102C = 4140  7C        0821a223
done:   beq  x2, x2, done       # infinite loop             80        00210063
```

Exercise 7.10

RISC-V single-cycle processor SystemVerilog:

```
// Modified to include all Exercise 7.4 instructions:
//   lui, sra, lbu, blt, bltu, bge, bgeu, jalr, auipc, sb, slli, srai
// It also supports the added instructions from Exercise 7.9:
//   xor, sll, srl, bne
// Search for the instruction mnemonic to view changes associated with
// that instruction.

module testbench();

  logic        clk;
  logic        reset;

  logic [31:0] WriteData, DataAdr;
  logic        MemWrite;

  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check results
  always @(negedge clk)
    begin
      // DataAdr = 0xBC & WriteData = 0x797780BC
      if((MemWrite==1'b1) & (DataAdr === 188) & (WriteData === 2037874876))
begin
          $display("Simulation succeeded");
          $stop;
      end
```

```
    end
endmodule

module top(input  logic        clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic        MemWrite);

  logic [31:0] PC, Instr, ReadData;

  // instantiate processor and memories
  riscvsingle rvsingle(clk, reset, PC, Instr, MemWrite, DataAdr,
                       WriteData, ReadData);
  imem imem(PC, Instr);
  dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule

module riscvsingle(input  logic        clk, reset,
                   output logic [31:0] PC,
                   input  logic [31:0] Instr,
                   output logic        MemWrite,
                   output logic [31:0] ALUResult, WriteData,
                   input  logic [31:0] ReadData);

  logic       ALUSrcB, RegWrite, Jump;  // renamed ALUSrc to ALUSrcB for lui
  logic [3:0] Flags;       // added for blt and other branches
  logic       ALUSrcA;     // added for lui
  logic [1:0] ResultSrc;
  logic [2:0] ImmSrc;      // expand to 3 bits for lui
  logic [3:0] ALUControl;  // expand to 4 bits for sra
  logic [1:0] LoadType;    // added for lb and lbu
  logic       StoreType;   // added for sb
  logic       PCTargetSrc; // added for jalr

  controller c(Instr[6:0], Instr[14:12], Instr[30], Flags,
               ResultSrc, MemWrite, PCSrc,
               ALUSrcA, ALUSrcB, RegWrite, Jump,
               ImmSrc, ALUControl, LoadType, StoreType, PCTargetSrc);
  datapath dp(clk, reset, ResultSrc, PCSrc,
              ALUSrcA, ALUSrcB, RegWrite,
              ImmSrc, ALUControl, LoadType, StoreType,
              PCTargetSrc,
              Flags, PC, Instr,
              ALUResult, WriteData, ReadData);
endmodule

module controller(input  logic [6:0] op,
                  input  logic [2:0] funct3,
                  input  logic       funct7b5,
                  input  logic [3:0] Flags,  // changed from Zero to Flags
for blt and other branches
                  output logic [1:0] ResultSrc,
                  output logic       MemWrite,
                  output logic       PCSrc, ALUSrcA, // added for lui
                  output logic       ALUSrcB,        // changed name to
ALUSrcA for lui
                  output logic       RegWrite, Jump,
                  output logic [2:0] ImmSrc,         // expand to 3 bits for
```

```
lui
                  output logic [3:0] ALUControl,     // expand to 4 bits for
sra
                  output logic [1:0] LoadType,       // added for lbu
                  output logic       StoreType,       // added for sb
                  output logic       PCTargetSrc);   // added for jalr
  logic [1:0] ALUOp;
  logic       Branch;
  logic       branchtaken;   // added for bne, blt, and other branches

  maindec md(op, ResultSrc, MemWrite, Branch,
             ALUSrcA, ALUSrcB, RegWrite, Jump, ImmSrc, ALUOp, PCTargetSrc);
  aludec  ad(op[5], funct3, funct7b5, ALUOp, ALUControl);
  lsu     lsu(funct3, LoadType, StoreType);
  bu      branchunit(Branch, Flags, funct3, branchtaken); // added for bne,
blt, etc.

  assign PCSrc = branchtaken | Jump;
endmodule

module maindec(input  logic [6:0] op,
               output logic [1:0] ResultSrc,
               output logic       MemWrite,
               output logic       Branch, ALUSrcA,
               output logic       ALUSrcB,  // added for lui - renamed ALUSrc
to ALUSrcA
               output logic       RegWrite, Jump,
               output logic [2:0] ImmSrc,   // expand to 3 bits for lui
               output logic [1:0] ALUOp,
               output logic       PCTargetSrc);  //  added for jalr

  logic [13:0] controls; // expanded to 14 bits to include ALUSrcB (for lui)
and PCTargetSrc (for jalr)

  assign {RegWrite, ImmSrc, ALUSrcA, ALUSrcB, MemWrite,
          ResultSrc, Branch, ALUOp, Jump, PCTargetSrc} = controls;

  always_comb
    case(op)
    //
RegWrite_ImmSrc_ALUSrcA_ALUSrcB_MemWrite_ResultSrc_Branch_ALUOp_Jump_PCTarget
Src
      7'b0000011: controls = 14'b1_000_0_1_0_01_0_00_0_x; // lw
      7'b0100011: controls = 14'b0_001_0_1_1_00_0_00_0_x; // sw
      7'b0110011: controls = 14'b1_xxx_0_0_0_00_0_10_0_x; // R-type
      7'b1100011: controls = 14'b0_010_0_0_0_00_1_01_0_0; // beq
      7'b0010011: controls = 14'b1_000_0_1_0_00_0_10_0_x; // I-type ALU
      7'b1101111: controls = 14'b1_011_x_0_0_10_0_00_1_0; // jal
      7'b0110111: controls = 14'b1_100_1_1_0_00_0_00_0_x; // lui
      7'b1100111: controls = 14'b1_000_0_1_0_10_0_00_1_1; // jalr
      7'b0010111: controls = 14'b1_100_x_x_0_11_0_xx_0_0; // auipc
      default:    controls = 14'bx_xxx_x_x_x_xx_x_xx_x_x; // non-implemented
instruction
    endcase
endmodule

module aludec(input  logic       opb5,
```

```
              input  logic [2:0] funct3,
              input  logic       funct7b5,
              input  logic [1:0] ALUOp,
              output logic [3:0] ALUControl);   // expand to 4 bits for sra

  logic  RtypeSub;
  assign RtypeSub = funct7b5 & opb5;  // TRUE for R-type subtract instruction

  always_comb
    case(ALUOp)
      2'b00:                ALUControl = 4'b000; // addition
      2'b01:                ALUControl = 4'b001; // subtraction
      default: case(funct3) // R-type or I-type ALU
                 3'b000:  if (RtypeSub)
                            ALUControl = 4'b0001; // sub
                          else
                            ALUControl = 4'b0000; // add, addi
                 3'b001:    ALUControl = 4'b0110; // sll, slli
                 3'b010:    ALUControl = 4'b0101; // slt, slti
                 3'b100:    ALUControl = 4'b0100; // xor, xori
                 3'b101:  if (funct7b5)
                            ALUControl = 4'b1000; // sra, srai
                          else
                            ALUControl = 4'b0111; // srl, srli
                 3'b110:    ALUControl = 4'b0011; // or, ori
                 3'b111:    ALUControl = 4'b0010; // and, andi
                 default:   ALUControl = 4'bxxx; // ???
               endcase
    endcase
endmodule

// Load/store Unit (lsu) added for lbu
module lsu(input  logic [2:0] funct3,
           output logic [1:0] LoadType,
           output logic       StoreType);
  always_comb
    case(funct3)
      3'b000:     {LoadType, StoreType} = {2'b10, 1'b1};
      3'b010:     {LoadType, StoreType} = {2'b00, 1'b0};
      3'b100:     {LoadType, StoreType} = {2'b01, 1'bx};
      default:    {LoadType, StoreType} =  3'bxxx;
    endcase
endmodule

// Branch Unit (bu) added for bne, blt, bltu, bge, bgeu
module bu (input  logic       Branch,
           input  logic [3:0] Flags,
           input  logic [2:0] funct3,
           output logic       taken);
  logic  v, c, n, z;    // Flags: overflow, carry out, negative, zero
  logic  cond;          // cond is 1 when condition for branch met
  assign {v, c, n, z} = Flags;
  assign taken = cond & Branch;

  always_comb
    case (funct3)
      3'b000:  cond = z;        // beq
```

```
      3'b001:  cond = ~z;       // bne
      3'b100:  cond = (n ^ v);  // blt
      3'b101:  cond = ~(n ^ v); // bge
      3'b110:  cond = ~c;       // bltu
      3'b111:  cond = c;        // bgeu
      default: cond = 1'b0;
    endcase
endmodule

module datapath(input  logic        clk, reset,
                input  logic [1:0]  ResultSrc,
                input  logic        PCSrc,
                input  logic        ALUSrcA,     // added for lui
                input  logic        ALUSrcB,     // renamed for lui
                input  logic        RegWrite,
                input  logic [2:0]  ImmSrc,      // expanded to 3 bits for
lui
                input  logic [3:0]  ALUControl,  // expanded to 4 bits for
sra
                input  logic [1:0]  LoadType,    // added for lbu
                input  logic        StoreType,   // added for sb
                input  logic        PCTargetSrc, // added for jalr
                output logic [3:0]  Flags,       // changed from Zero to
Flags for blt
                output logic [31:0] PC,
                input  logic [31:0] Instr,
                output logic [31:0] ALUResult, WriteData,
                input  logic [31:0] ReadData);

  logic [31:0] PCNext, PCPlus4, PCTarget;
  logic [31:0] PCrelativeTarget;                    // added for jalr
  logic [31:0] ImmExt;
  logic [31:0] SrcA, SrcB;
  logic [31:0] RD1;                                  // added for lui
  logic [31:0] RD2;                                  // added for sb
  logic [31:0] Result;
  logic [31:0] ZeroExtendByte, ReadDataOut;          // added for lbu
  logic [31:0] SignExtendByte;                       // added for lb
  logic [7:0]  byteout;                              // added for lbu

  // next PC logic
  flopr #(32) pcreg(clk, reset, PCNext, PC);
  adder       pcadd4(PC, 32'd4, PCPlus4);
  adder       pcaddbranch(PC, ImmExt, PCrelativeTarget);  // modified for
jalr
  mux2 #(32)  pctargetmux(PCrelativeTarget, ALUResult, PCTargetSrc,
PCTarget); // jalr
  mux2 #(32)  pcmux(PCPlus4, PCTarget, PCSrc, PCNext);

  // register file logic
  regfile     rf(clk, RegWrite, Instr[19:15], Instr[24:20],
                 Instr[11:7], Result, RD1, RD2);   // RD1 output instead of
SrcA for lui
  extend      ext(Instr[31:7], ImmSrc, ImmExt);

  // ALU logic
  mux2 #(32)  srcamux(RD1, 32'b0, ALUSrcA, SrcA);        // added for lui
```

```
  mux2 #(32)  srcbmux(RD2, ImmExt, ALUSrcB, SrcB); // ALUSrcB renamed for lui
  alu         alu(SrcA, SrcB, ALUControl, ALUResult, Flags); // added Flags
for blt
  mux4 #(32)  resultmux(ALUResult, ReadDataOut, PCPlus4, PCTarget, ResultSrc,
Result);  // updated for auipc

  // ReadData logic - added for lbu
  mux4 #(8)   bytesel(ReadData[7:0], ReadData[15:8], ReadData[23:16],
ReadData[31:24], ALUResult[1:0], byteout);
  zeroextend  ze(byteout, ZeroExtendByte);
  signextend  se(byteout, SignExtendByte);
  mux3 #(32)  readdatamux(ReadData, ZeroExtendByte, SignExtendByte, LoadType,
ReadDataOut);

  // WriteData logic - added for sb
  wdunit      wdu(RD2, ReadData, StoreType, ALUResult[1:0], WriteData);
endmodule

module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [ 4:0] a1, a2, a3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally (A1/RD1, A2/RD2)
  // write third port on rising edge of clock (A3/WD3/WE3)
  // register 0 hardwired to 0

  always_ff @(posedge clk)
    if (we3) rf[a3] <= wd3;

  assign rd1 = (a1 != 0) ? rf[a1] : 0;
  assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule

module adder(input  [31:0] a, b,
             output [31:0] y);

  assign y = a + b;
endmodule

module extend(input  logic [31:7] instr,
              input  logic [2:0]  immsrc,  // extended to 3 bits for lui
              output logic [31:0] immext);

  always_comb
    case(immsrc)
               // I-type
      3'b000:   immext = {{20{instr[31]}}, instr[31:20]};
               // S-type (stores)
      3'b001:   immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
               // B-type (branches)
      3'b010:   immext = {{20{instr[31]}}, instr[7], instr[30:25],
instr[11:8], 1'b0};
```

```
                // J-type (jal)
      3'b011:   immext = {{12{instr[31]}}, instr[19:12], instr[20],
instr[30:21], 1'b0};
                // U-type (lui, auipc)
      3'b100:   immext = {instr[31:12], 12'b0};
      default:  immext = 32'bx; // undefined
    endcase
endmodule

module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);

  assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

// mux4 added for lbu
module mux4 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2, d3,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);
  always_comb
    case (s)
      2'b00: y = d0;
      2'b01: y = d1;
      2'b10: y = d2;
      2'b11: y = d3;
      default: y = 'x;
  endcase
endmodule

module imem(input  logic [31:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[127:0];

  initial
      $readmemh("example.txt", RAM);
```

```
  assign rd = RAM[a[31:2]]; // word aligned
endmodule

module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

  logic [31:0] RAM[127:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule

module alu(input  logic [31:0] a,
           input  logic [31:0] b,
           input  logic [3:0]  alucontrol,  // expanded to 4 bits for sra
           output logic [31:0] result,
           output logic [3:0]  flags);      // added for blt and other
branches

  logic [31:0] condinvb, sum;
  logic        v, c, n, z;     // flags: overflow, carry out, negative, zero
  logic        cout;           // carry out of adder
  logic        isAddSub;       // true if is an add, subtract, or slt

  assign flags = {v, c, n, z};
  assign condinvb = alucontrol[0] ? ~b : b;
  assign {cout, sum} = a + condinvb + alucontrol[0];
  assign isAddSub = (~alucontrol[3] & ~alucontrol[2] & ~alucontrol[1]) |
                    (~alucontrol[3] & ~alucontrol[1] & alucontrol[0]);

  always_comb
    case (alucontrol)
      4'b0000:  result = sum;                    // add
      4'b0001:  result = sum;                    // subtract
      4'b0010:  result = a & b;                  // and
      4'b0011:  result = a | b;                  // or
      4'b0100:  result = a ^ b;                  // xor
      4'b0101:  result = sum[31] ^ v;            // slt
      4'b0110:  result = a << b[4:0];            // sll
      4'b0111:  result = a >> b[4:0];            // srl
      4'b1000:  result = $signed(a) >>> b[4:0];  // sra
      default:  result = 32'bx;
    endcase

  // added for blt and other branches
  assign z = (result == 32'b0);
  assign n = result[31];
  assign c = cout & isAddSub;
  assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;
endmodule

// zeroextend module added for lbu
module zeroextend(input logic  [7:0]  a,
```

```
                  output logic [31:0] zeroimmext);

  assign zeroimmext = {24'b0, a};
endmodule


// signextend module added for lb
module signextend(input logic  [7:0]  a,
                  output logic [31:0] signext);

  assign signext = {{24{a[7]}}, a};
endmodule

module wdunit(input  logic [31:0] rd2,
              input  logic [31:0] readdata,
              input  logic        StoreType,
              input  logic [1:0]  byteoffset,
              output logic [31:0] WriteData);
  logic [31:0] storeb0, storeb1, storeb2, storeb3, sbword;

  assign storeb0 = {readdata[31:8],  rd2[7:0]};
  assign storeb1 = {readdata[31:16], rd2[7:0], readdata[7:0]};
  assign storeb2 = {readdata[31:24], rd2[7:0], readdata[15:0]};
  assign storeb3 = {rd2[7:0], readdata[23:0]};

  mux4 #(32) sbmux(storeb0, storeb1, storeb2, storeb3, byteoffset, sbword);
  mux2 #(32) wdmux(rd2, sbword, StoreType, WriteData);
endmodule
```

Modified test program:

```
# If successful, it should write the value 0x797780BC (2037874876) to address 0xBC (188).
#       RISC-V Assembly         Description                  Address      Machine Code
main:   addi  x1, x0, 0x1C       # x1 = 0x1C                    0     01c00093
        lui   x2, 0xAABBC        # x2 = 0xAABBC000              4     aabbc137
        sw    x2, 200(x0)        # mem[200] = 0xAABBC000        8     0c202423
        srli  x2, x2, 12         # x2 = 0x000AABBC              C     00c15113
        sb    x2, 203(x0)        # mem[203] = 0xBCBBC000        10    0c2005a3
        lw    x3, 200(x0)        # x3 = 0xBCBBC000              14    0c802183
        lb    x4, 201(x0)        # x4 = 0xFFFFFFC0              18    0c900203
        auipc x5, 0xFFAAB        # x5 = 0xFFAAB01C              1C    ffaab297
        jalr  x2, x1, 0xC        # x2 = 0x24, PC = 0x28         20    00c08167
        addi  x3, x0, 4          # skipped                      24    00400193
        sra   x7, x3, x1         # x7 = 0xFFFFFFFB              28    4011d3b3
        addi  x4, x0, 100        # x4 = 100                     2C    06400213
        sw    x3, 221(x7)        # mem[216] = 0xBCBBC000        30    0c33aea3
        lbu   x5, 191(x1)        # x5 = mem[219] = 0xBC         34    0bf0c283
        blt   x5, x7, around     # should not be taken          38    0272c663
        bge   x3, x5, skip       # should hot be taken          3C    0051da63
        bge   x5, x3, skip       # should be taken              40    0032d863
L2:     bltu  x3, x2, around     # should not be taken          44    0221e063
        bltu  x2, x3, around     # should be taken              48    00316e63
        sll   x5, x5, x3         # skip: shouldn't execute      4C    003292b3
skip:   bgeu  x5, x3, L2         # should not be taken          50    fe32fae3
        bgeu  x3, x5, L2         # should be taken              54    fe51f8e3
        slt   x4, x3, x4         # x4 = (12 < 7) = 0            58    0041A233
        beq   x4, x0, around     # should be taken              5C    00020463
        addi  x5, x0, 0          # shouldn't execute            60    00000293
around: slt   x4, x7, x2         # x4 = (3 < 5)  = 1            64    0023A233
        sll   x7, x5, x4         # x7 = 0xBC << 1 = 0x178       68    004293b3
        sub   x7, x7, x2         # x7 = 0x178 - 0x24 = 0x154    6C    402383b3
        sw    x3, -84(x7)        # [0x100] = 0xBCBBC000         70    fa33a623
        lw    x2, 0x100(x0)      # x2 = [0x100] = 0xBCBBC000    74    10002103
```

```
        add   x9, x2, x5         # x9 = 0xBCBBC0BC               78    005104B3
        jal   x3, end            # jump to end, x3 = 0x80        7C    008001EF
        addi  x2, x0, 1          # shouldn't execute             80    00100113
end:    add   x2, x2, x9         # x2 = 0x797780BC               84    00910133
        lui   x4, 0x7FFFE        # x4 = 0x7FFFF000               88    80000237
        addi  x4, x4, -1         # x4 = 0x7FFFFFFF               8C    fff20213
        addi  x5, zero, -1       # x5 = -1                       90    fff00293
        slt   x6, x4, x5         # x6 = 0                        94    00522333
wrong:  bne   x6, x0, wrong      # shouldn't be taken            98    00031063
        sw    x2, 0x3C(x3)       # mem[0xBC] = 0x797780BC        9C    0221AE23
done:   beq   x2, x2, done       # infinite loop                 A0    00210063
```

Exercise 7.11

(a) ***ResultSrc*$_1$:** All instructions will fail because the program counter will not update to PC+4. Instead, it will be incorrectly updated to the previous ALU result because *ResultSrc* would be 00 instead of 10.
(b) ***ResultSrc*$_0$:** `lw` – Instead of loading the data read from the memory into the register, the previous ALU result will be stored as the result because *ResultSource* would be 00 instead of 01.
(c) ***ALUSrcB*$_1$:** All instructions will fail because during the Fetch state, the PC will increment by whatever happens to be in *WriteData* at the time instead of by 4 because to the ALU receiving the wrong source.
(d) ***ALUSrcB*$_0$:** All instructions that use an immediate (`addi`, `beq`, `lw`, `sw`, etc) will fail because the immediate can never be selected and used due to the stuck-at-0 *ALUSrcB*$_0$.
(e) ***ALUSrcA*$_1$:** `lw`, `sw`, `beq`, R-type, and I-type ALU instructions – RD1 will never be able to be selected and used due to the stuck-at-0 *ALUSrcA*$_1$.
(f) ***ALUSrcA*$_0$:** `jal` and `beq` – The branch/jump target address will not be able to be calculated because *OldPC* can never be selected and used due to the stuck-at-0 *ALUSrcA*$_0$.
(g) ***ImmSrc*$_1$:** `jal` and `beq` – The branch/jump offsets (immediates) could not be selected.
(h) ***ImmSrc*$_0$:** `sw` and `jal` – The address and jump offsets (immediates), respectively, could not be selected.
(i) ***RegWrite*:** `lw`, `jal`, R-type, and I-type ALU instructions – The register file will never be written to.
(j) ***PCUpdate*:** All instructions will fail because the program counter will never be updated and, thus, we will execute the same instruction repeatedly.
(k) ***Branch*:** `beq` will malfunction whenever the branch is taken. Because *Branch* is stuck at zero, the PC will never update to the target PC whenever a branch should be taken.
(l) ***AdrSrc*:** `lw` and `sw` – Instead of the proper data address being used for loads and stores, the PC will be used which in the case of a load would fetch garbage data and, in the case of a store, would corrupt the program.
(m) ***MemWrite*:** `sw` – Memory cannot be written.
(n) ***IRWrite*:** All instructions will fail because the instruction would never be written to the instruction register and thus no instructions could execute.

Exercise 7.12

(a) ***ResultSrc*$_1$:** All instructions would malfunction – The Main FSM from Figure 45 shows that all instructions use either ResultSrc = 00 or 01 before they complete. With ResultSrc1 locked into

"1" none of these instructions would be able to retrieve the value needed for proper operation.

(b) ***ResultSrc*$_0$:** All instructions would malfunction – Because all instructions pass through the Fetch state, PC+4 could not be written to the PC register if ResultSrc0 is stuck at 1. Even for the instructions that might "jump" or "branch" to a different instruction, *ResultSrc* = 00 is required for correct operation.

(c) ***ALUSrcB*$_1$:** All instructions would malfunction – With ALUSrcB$_1$ locked into "1", the only possible *SrcB* would be a constant of "4". No instruction could use an immediate or **rs2** during any state so all instructions would fail at some point.

(d) ***ALUSrcB*$_0$:** All instructions would malfunction – With *ALUSrcB$_0$* stuck at "1", the only possible *SrcB* would be *ImmExt*. This means no instruction could produce PC+4 during the Fetch state. Even though `jal` always skips to the *JTA*, the instruction could not properly write PC+4 to **rd**.

(e) ***ALUSrcA*$_1$:** All instructions would malfunction – No instruction could calculate PC+4 during the Fetch state because *SrcA* is locked into being *RD1* from the register file. `beq` and `jal` would not be able to calculate their target addresses because the SrcA multiplexer can't select *OldPC*.

(f) ***ALUSrcA*$_0$:** `lw`, `sw`, R-type, I-type ALU, and `beq` – No instruction could access *RD1* as *SrcA* so all instructions that use **rs1** would malfunction. `jal` would function correctly. `jal` would not calculate PC+4 as intended during the Fetch state, but this gets overwritten by *JTA* which is calculated in the *Decode* state and then stored as *PCNext* in the JAL state. In `jal`, PC+4 must also be written to **rd**, which requires using *OldPC* and a constant of "4" which would still function under these conditions.

(g) ***ImmSrc*$_1$:** I-type ALU and `sw` would malfunction – The incorrect immediate would be selected.

(h) ***ImmSrc*$_0$**`[SH1]`**:** I-type ALU and `beq` would malfunction – The incorrect immediate would be selected.

(i) ***RegWrite*:** All instructions would malfunction because the register file would be written on every clock cycle.

(j) ***PCUpdate*:** All instructions would malfunction – Because all instructions use the Result multiplexer, the Program Counter will always be overwritten with an unintended value. For `jal`, the *JTA* will be overwritten with PC+4. `beq` will always branch, even if it shouldn't. For all other instructions, *ALUOut/ReadData* will overwrite PC+4. Then, when the next Fetch state occurs, it will output PC as those unintended values.

(k) ***Branch*:** All instructions would malfunction – Because the processor could randomly branch to a random target address on every clock cycle (when *Zero* = 1).

(l) ***AdrSrc*:** All instructions would malfunction – During the Fetch state, PC is not selected as the address to send to the combined Instruction / Data memory.

(m) ***MemWrite*:** All instructions would malfunction – Because the processor would write a random value to a random address in memory on each clock cycle.

(n) ***IRWrite*:** All instructions would malfunction. The instruction register would be written with the correct instruction in the fetch stage but then with the instruction at PC+4 in the Decode stage. Thus, after the Decode stage, the control signals would be from the next instruction, instead of the current instruction.

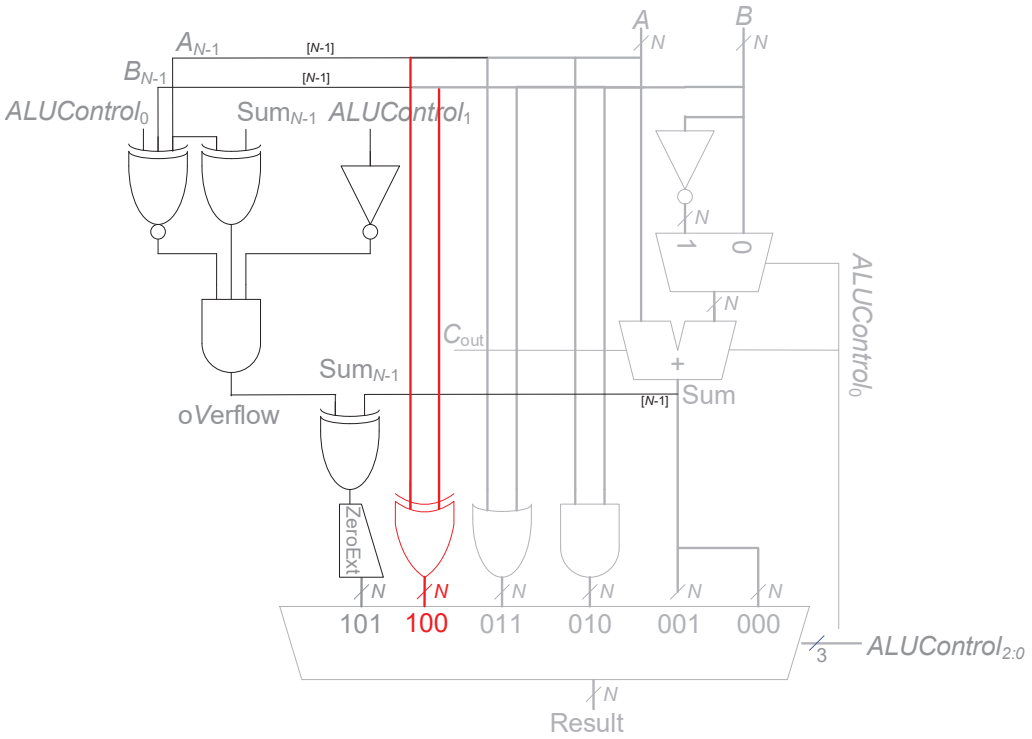
Exercise 7.13

(a) `xor`

Neither the datapath or the Main FSM need to be modified, because they already support R-type instructions.

Only the ALU needs to be modified: we add another input to the multiplexer and N 2-bit XOR gates within the ALU. We also update the ALU Decoder truth table / logic. The changes are shown below.

Modified ALU to support `xor`



Modified ALU operations to support `xor`

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
100	**xor**
101	SLT

Modified ALU Decoder truth table to support `xor`

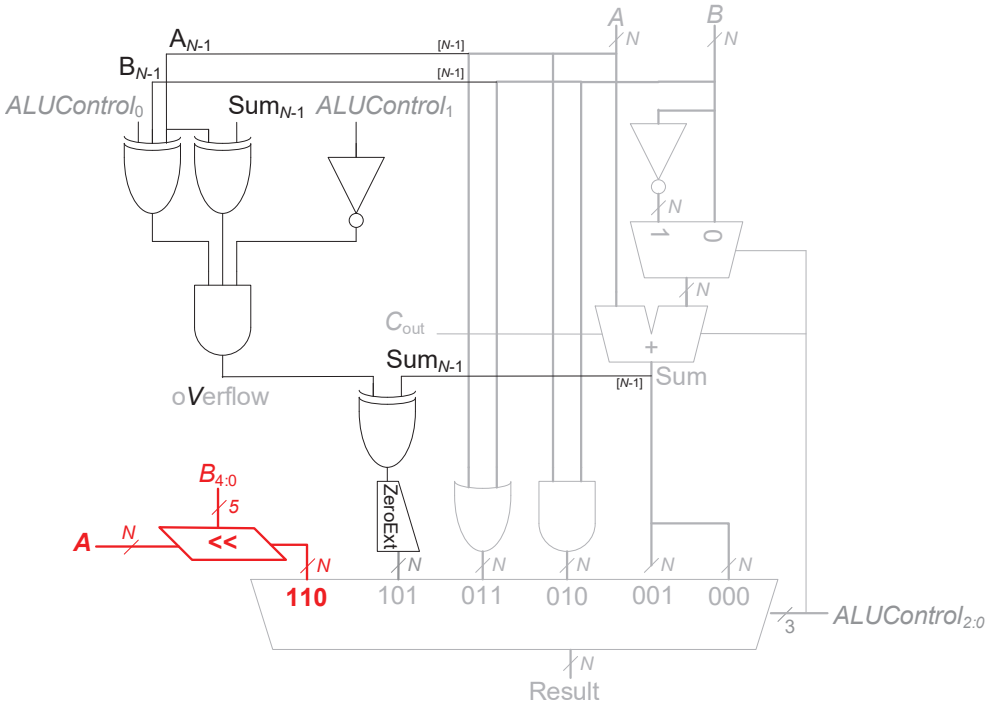
ALUOp	funct3	op_5, $funct7_5$	*ALUControl*	Instruction
00	x	x	000 (add)	`lw, sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add, addi`
	000	11	001 (subtract)	`sub`
	010	x	101 (set less than)	`slt, slti`
	100	**x**	**100 (xor)**	**`xor, xori`**
	110	x	011 (or)	`or, ori`
	111	x	010 (and)	`and, andi`

(b) `sll`

Neither the datapath or the Main FSM need to be modified, because they already support R-type instructions.

We only modify the ALU and the ALU Decoder, as shown below. We add a shifter and expand the multiplexer inside the ALU.

Modified ALU to support `sll`



Modified ALU operations to support `sll`

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT
110	**sll**

Modified ALU Decoder truth table to support `sll`

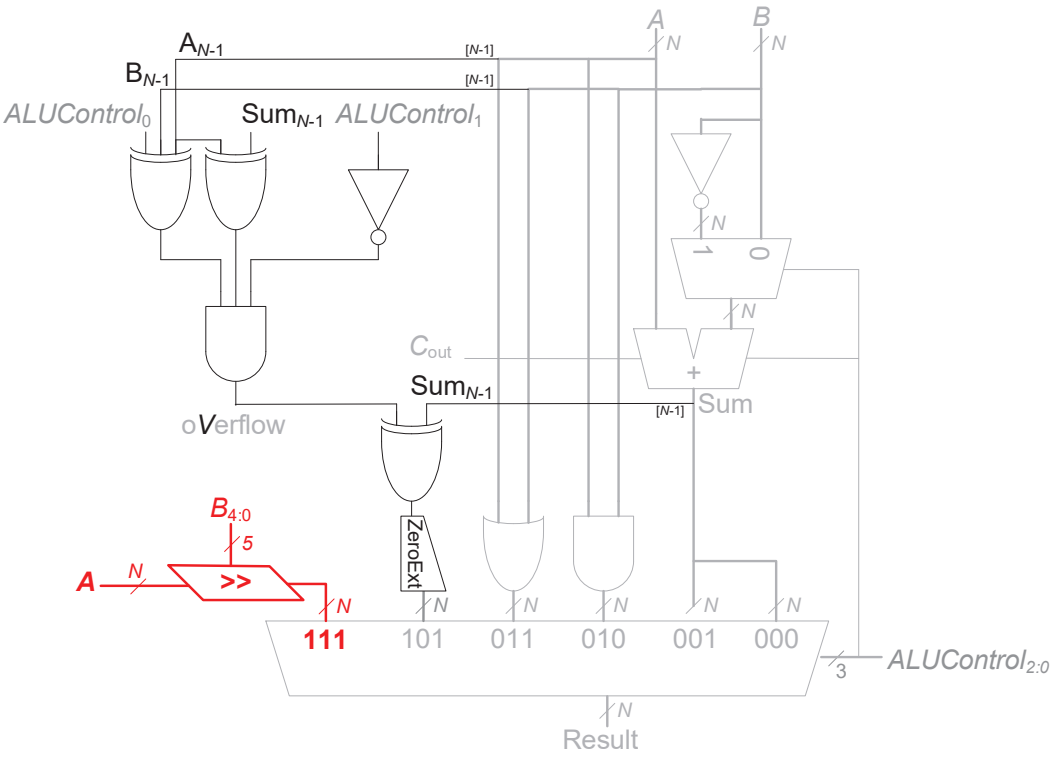
ALUOp	funct3	op_5, $funct7_5$	*ALUControl*	Instruction
00	x	x	000 (add)	`lw, sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add, addi`
	000	11	001 (subtract)	`sub`
	001	**x**	**110 (shift left logical)**	**`sll, slli`**
	010	x	101 (set less than)	`slt, slti`
	110	x	011 (or)	`or, ori`
	111	x	010 (and)	`and, andi`

(c) `srl`

Neither the datapath or the Main FSM need to be modified, because they already support R-type instructions.

We only modify the ALU and the ALU Decoder, as shown below. We add a shifter and expand the multiplexer inside the ALU.

Modified ALU to support `srl`



Modified ALU operations to support `srl`

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT
111	**srl**

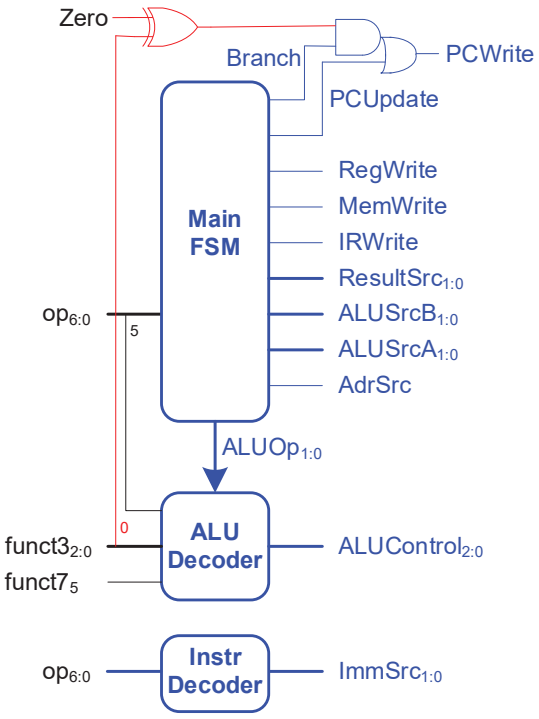
Modified ALU Decoder truth table to support `srl`

ALUOp	funct3	op_5, $funct7_5$	*ALUControl*	Instruction
00	x	xx	000 (add)	`lw, sw`
01	x	xx	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add, addi`
	000	11	001 (subtract)	`sub`
	001	**x0**	**111 (shift right logical)**	**`srl, srli`**
	010	xx	101 (set less than)	`slt, slti`
	110	xx	011 (or)	`or, ori`
	111	xx	010 (and)	`and, andi`

(d) `bne`

`bne` is the opposite of `beq`. `beq` and `bne` can be identified by **func3**$_0$, which is high when `bne` is the instruction and low for `beq`. To implement, we simply need to change the control unit to branch when *Zero* is 0 and `bne` is the instruction or when *Zero* is 1 and `beq` is the instruction. This is easily achieved with *Zero* XOR **func3**$_0$.

Enhanced control unit for `bne`



Exercise 7.14

(a) `lui`

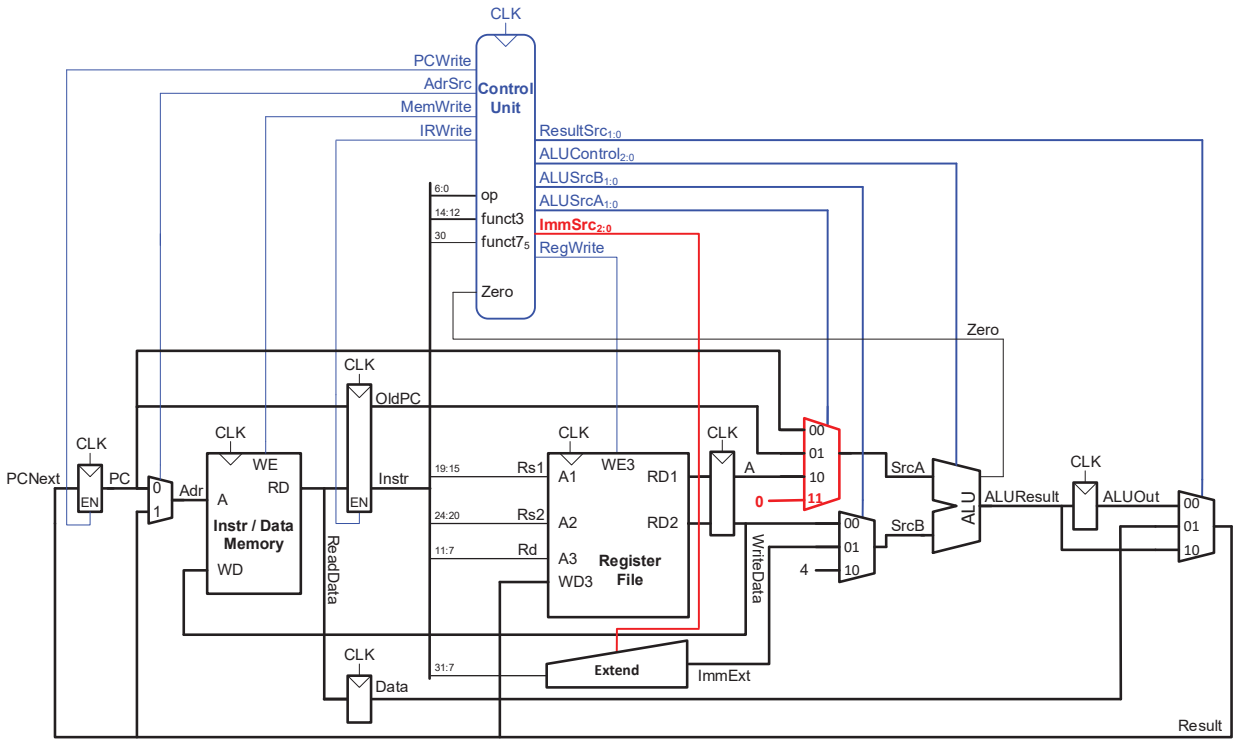
First, we update the immediate Extend unit to support `lui`.

Enhanced *ImmSrc* encoding to support `lui`

ImmSrc	*ImmExt*	Type	Description
000	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
001	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
010	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate
011	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1'b0}	J	21-bit signed immediate
100	**{*Instr*[31:12], 12'b0}**	**U**	**20-bit signed immediate**

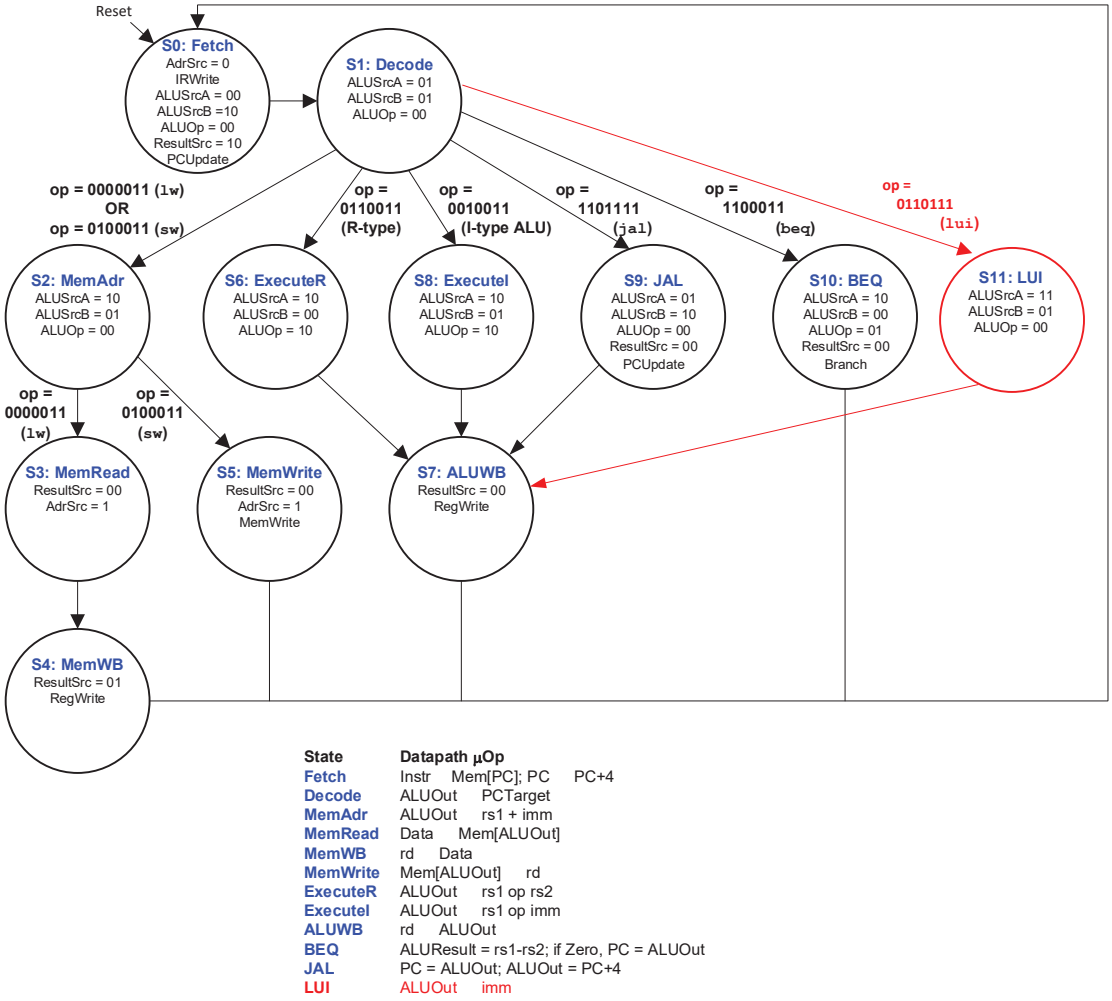
Next, we modify the datapath by increasing the width of the *ImmSrc* control signal to 3 bits and by making 0 an option for the ALU's top input (SrcA).

Enhanced datapath to support `lui`



Next, we modify the Main FSM by creating a new state, LUI, adds 0 + imm (i.e., the upper immediate) and places the result in the ALUOut register at the next clock edge. `lui` then proceeds to the ALUWB (ALU Writeback) state where the results in the ALUOut register are written to **rd**.

Enhanced Main FSM to support `lui`

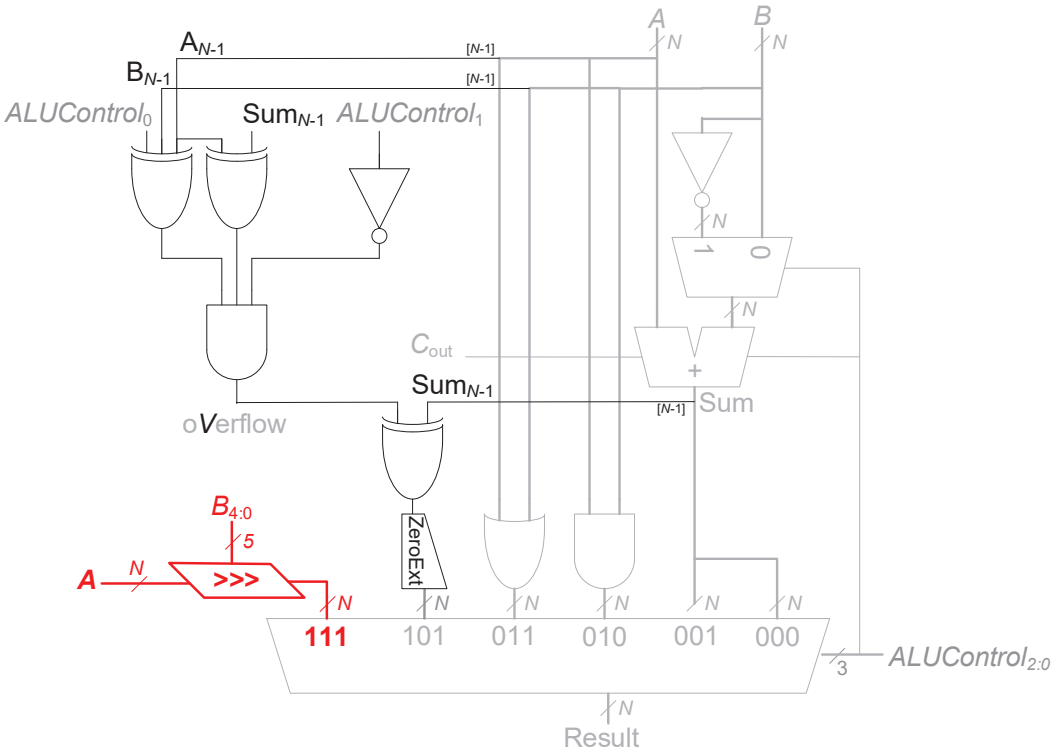


State	Datapath μOp
Fetch	Instr Mem[PC]; PC PC+4
Decode	ALUOut PCTarget
MemAdr	ALUOut rs1 + imm
MemRead	Data Mem[ALUOut]
MemWB	rd Data
MemWrite	Mem[ALUOut] rd
ExecuteR	ALUOut rs1 op rs2
ExecuteI	ALUOut rs1 op imm
ALUWB	rd ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4
LUI	ALUOut imm

(b) `sra`

The overall datapath (interfaces and units) need not be changed. We only modify the ALU and the ALU Decoder, as shown below. We add a shifter and expand the multiplexer inside the ALU.

Modified ALU to support `sra`



Modified ALU operations to support `sra`

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT
111	**sra**

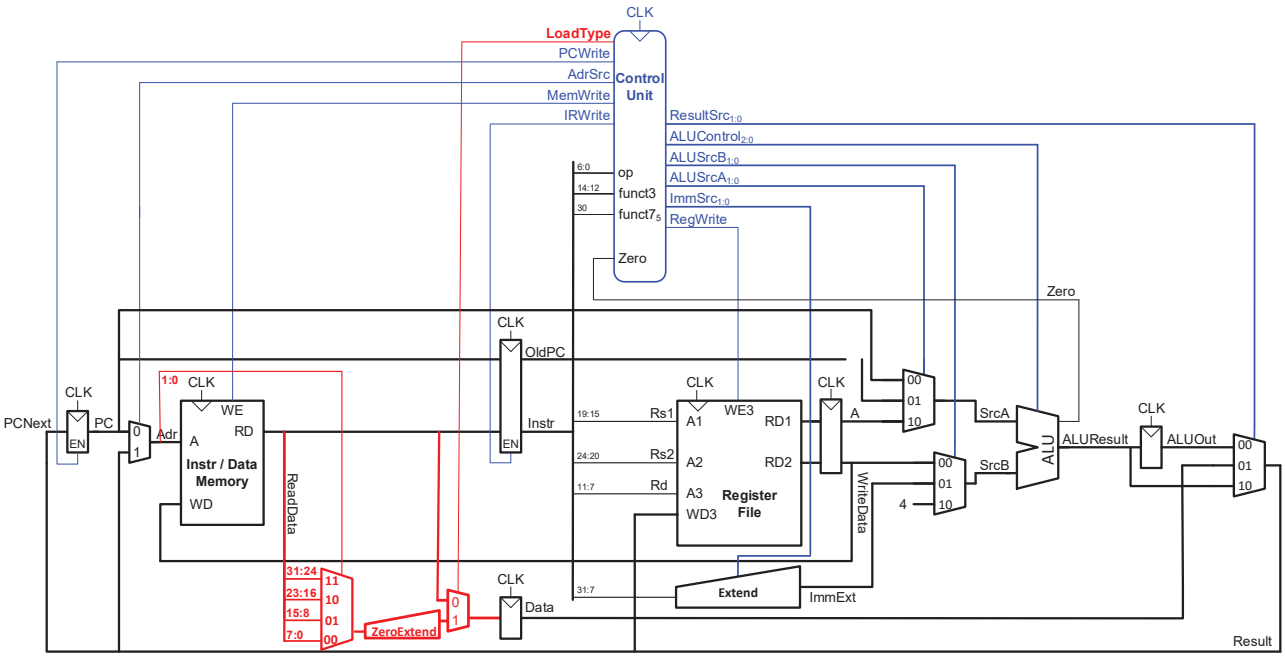
Modified ALU Decoder truth table to support `sra`

ALUOp	funct3	op_5, $funct7_5$	*ALUControl*	Instruction
00	x	xx	000 (add)	`lw, sw`
01	x	xx	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add, addi`
	000	11	001 (subtract)	`sub`
	010	xx	101 (set less than)	`slt, slti`
	101	**x1**	**111 (shift right arithmetic)**	**`sra, srai`**
	110	xx	011 (or)	`or, ori`
	111	xx	010 (and)	`and, andi`

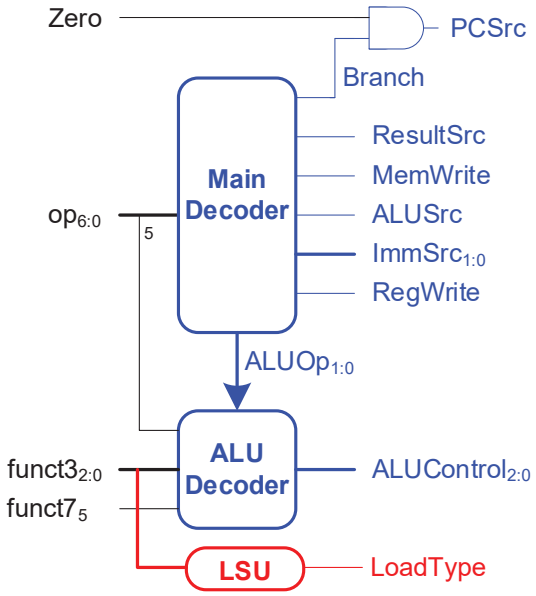
(c) `lbu`

To implement `lbu` we create a load/store unit (LSU) within the controller that outputs a new signal *LoadType*. The LSU takes in **funct3** and outputs *LoadType*. When *LoadType* is 0, it is an `lbu` instruction, and a zero-extended byte (selected by the two least significant address bits, $Adr_{1:0}$) of the *ReadData* bus is sent to the ResultSrc multiplexer. Otherwise, *ReadData* is sent. We add the new signal, *LoadType*, zero-extension unit, and LoadType multiplexer to the datapath.

Enhanced datapath to support `lbu`



Enhanced control unit for `lbu`

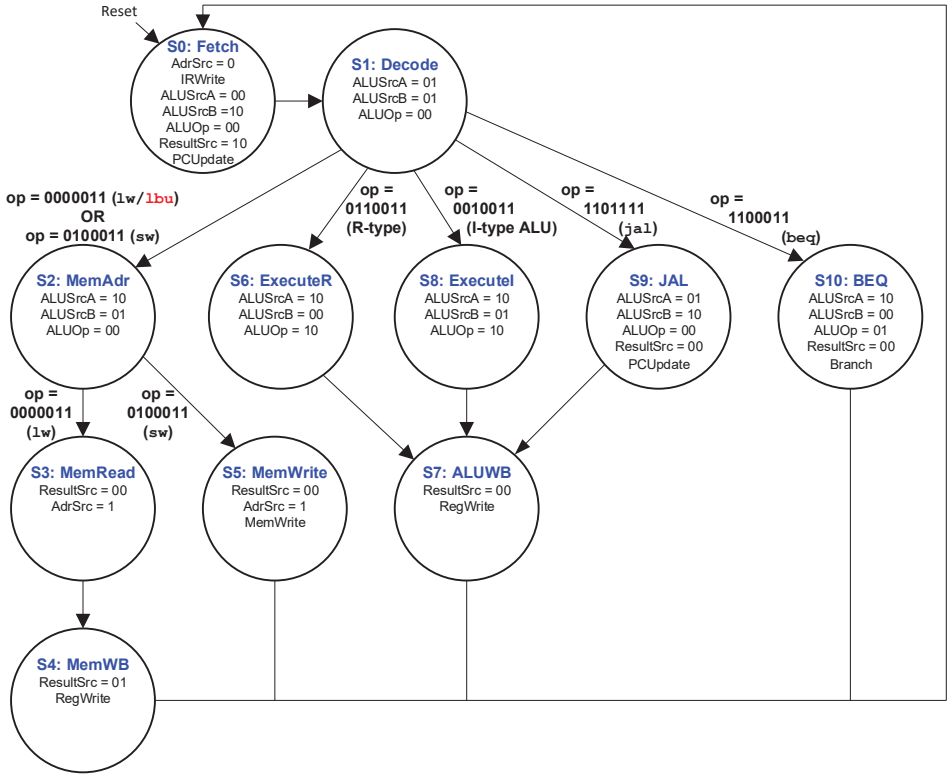


LSU truth table to support `lbu`

funct3	*LoadType*	Instruction
010	0	`lw`
100	1	`lbu`

The Main FSM does not need to be modified, but we add the `lbu` label into state S2 for completeness. Remember that `lw` and `lbu` have the same opcode.

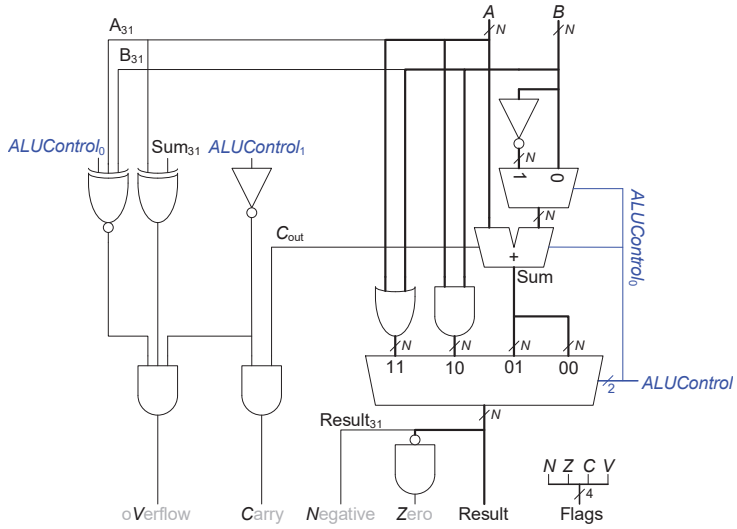
Enhanced Main FSM with `lbu` label



(d) `blt`

We start by using the ALU with flags provided in the book, shown again below.

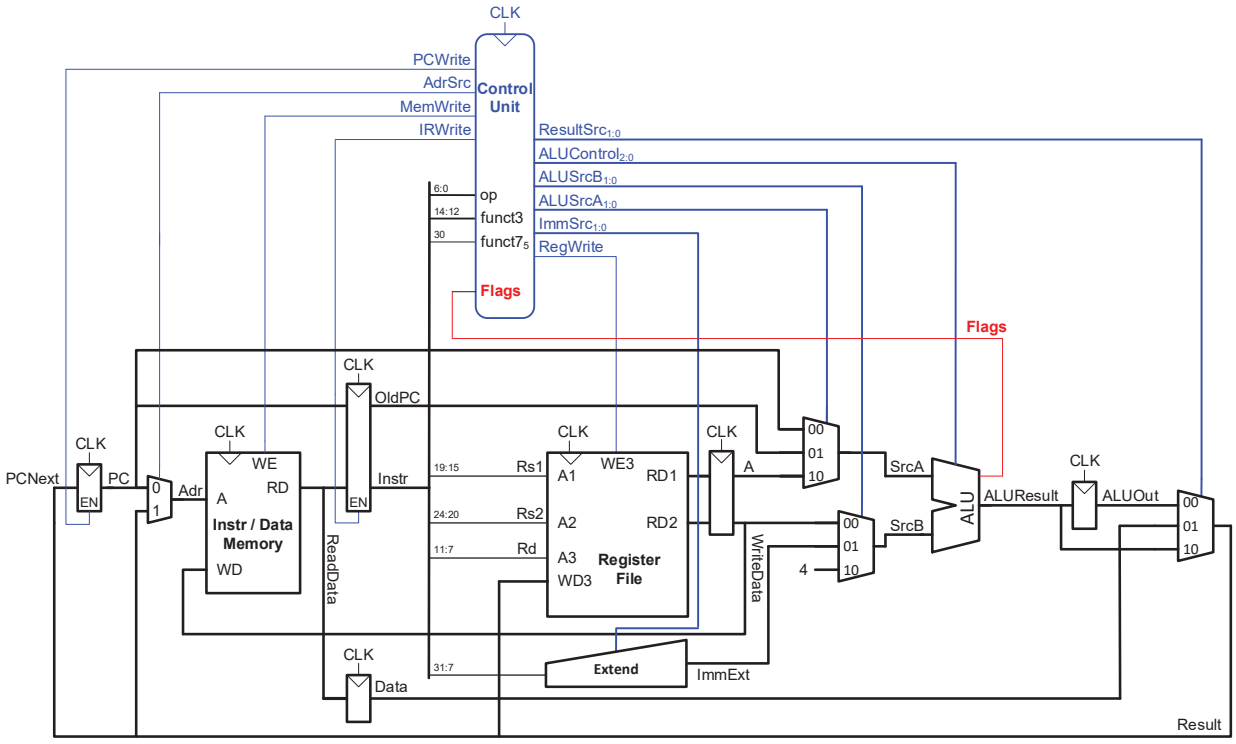
ALU with flags from the textbook



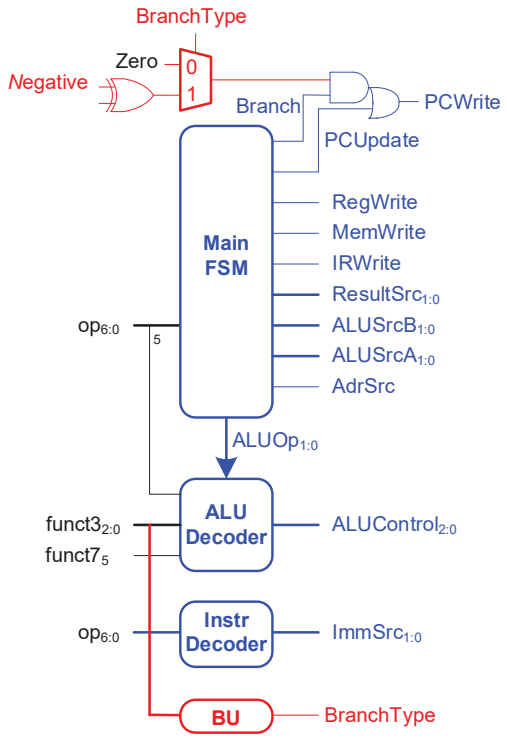
A is less than B when the *N*egative flag is 1 and o*V*erflow is 0 or when $N = 0$ and $V = 1$. (Remember that when overflow occurs, it means the result has the incorrect sign.) So, $A < B$ when N XOR $V = 1$.

We also add a *BranchType* internal signal of the Controller that is 1 when a `blt` instruction is executing and 0 when a `beq` instruction is executing.

Enhanced datapath to support `blt`



Enhanced control unit for `blt`

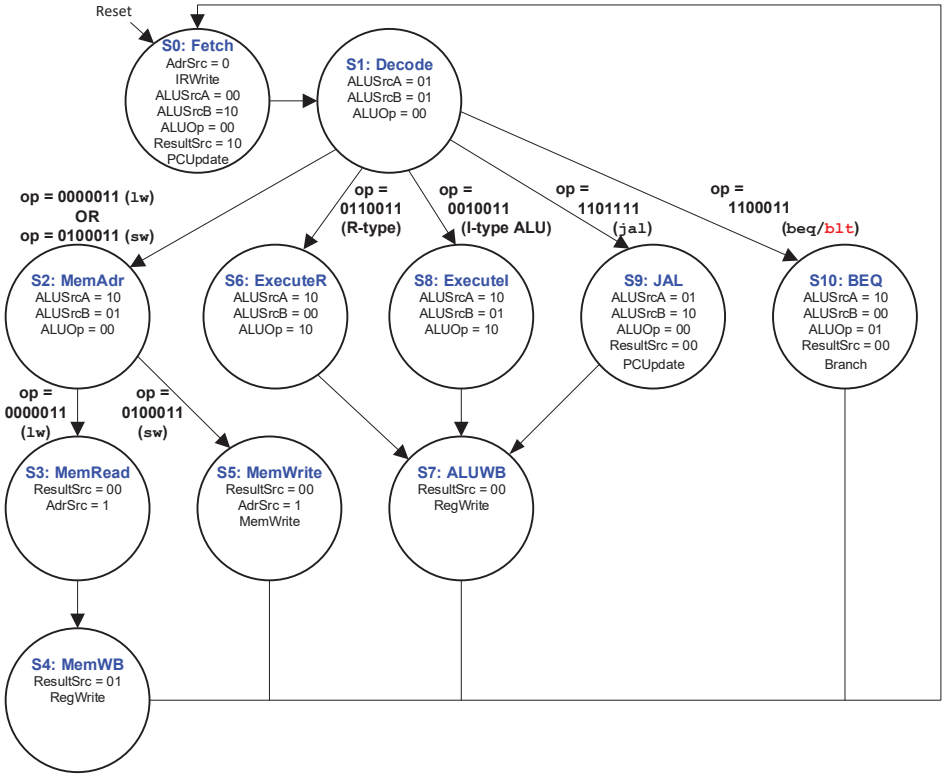


Branch Unit (BU) truth table to support `blt`

funct3	*BranchType*	Instruction
000	0	`beq`
100	1	`blt`

The Main FSM is not modified, but we add the blt label into the BEQ state for completeness.

Main FSM with `blt` label into the BEQ state

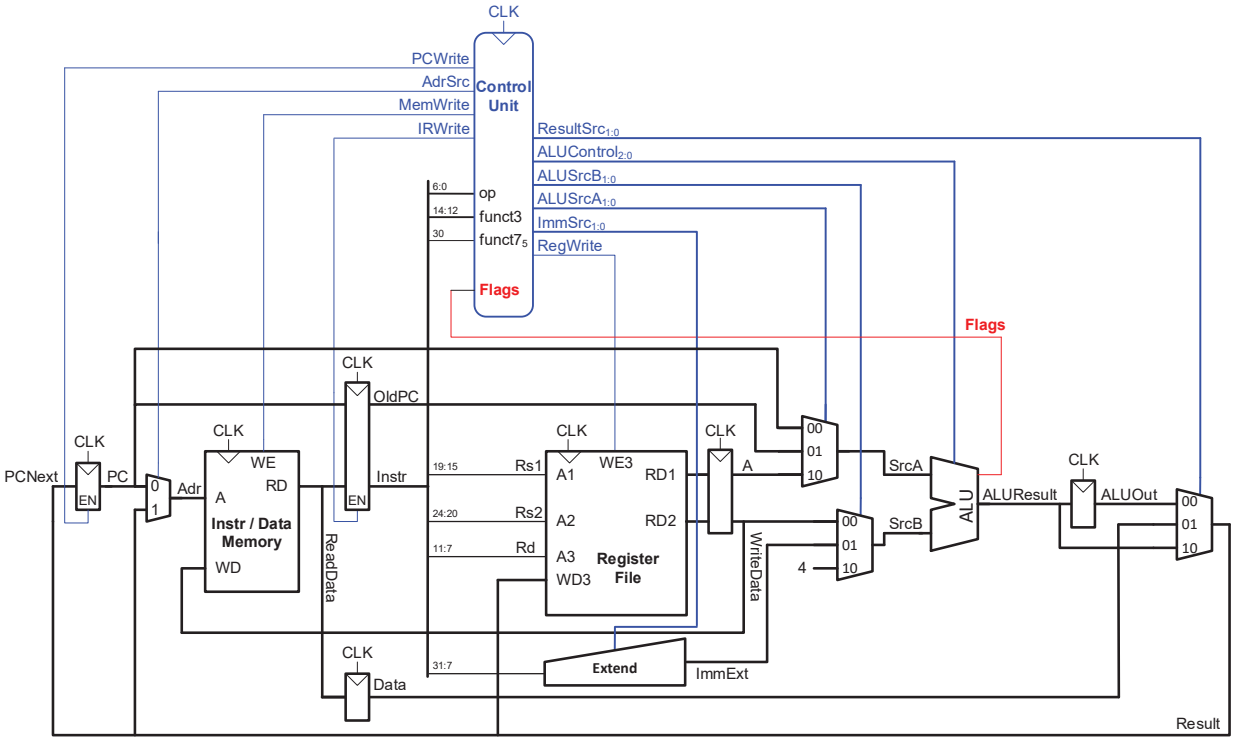


(e) `bltu`

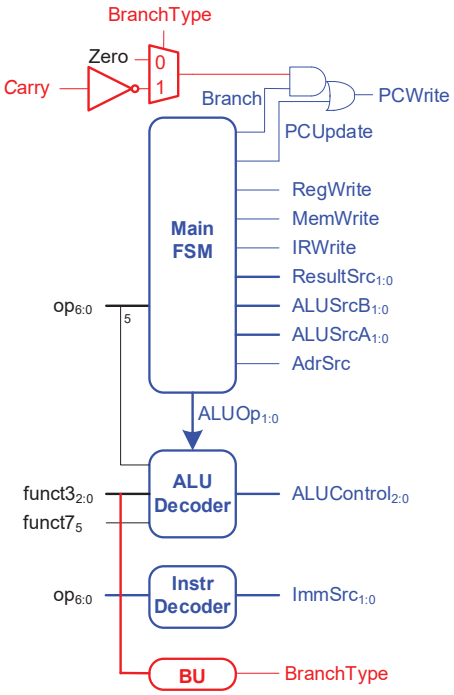
We start by using the ALU with flags (see Figure 5.17 in the textbook). For unsigned numbers, *A* is less than *B* when the *C*arry flag is 0.

We also add a *BranchType* output of the Controller that is 1 when a `bltu` instruction is executing and 0 when a `beq` instruction is executing. We add a Branch Unit (BU) to the controller to produce this signal (*BranchType*) using **funct3** as its input.

Enhanced datapath to support `bltu`



Enhanced control unit for `bltu`

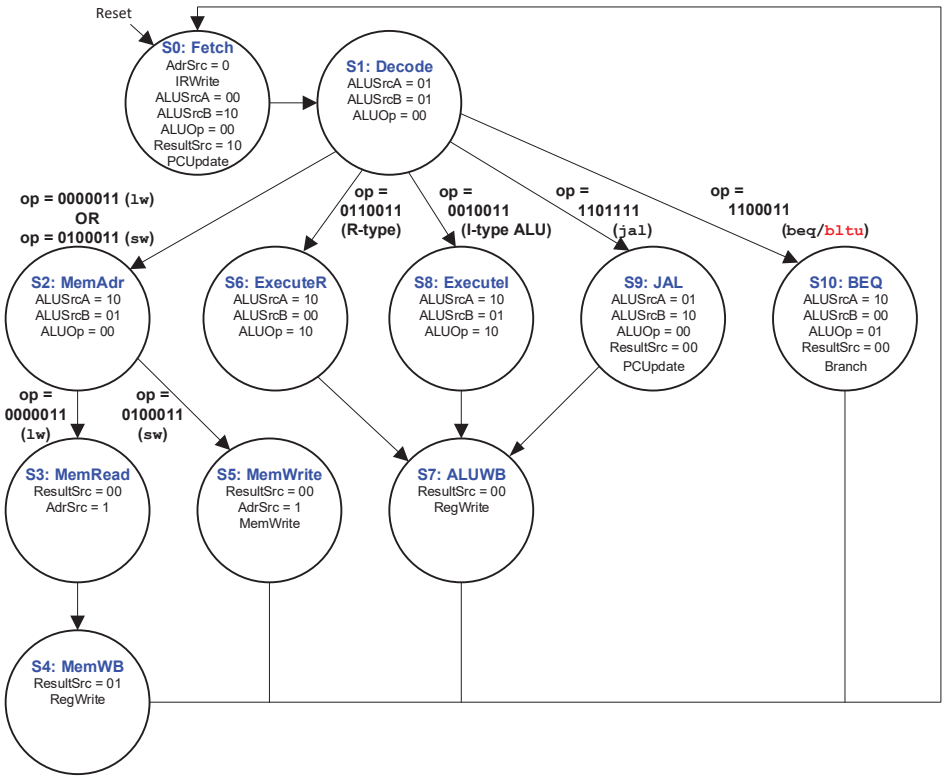


Branch Unit (BU) truth table to support `bltu`

funct3	*BranchType*	Instruction
000	0	`beq`
110	1	`bltu`

The Main FSM is not modified, but we add the `bltu` label into the BEQ state for completeness.

Main FSM with `bltu` label into the BEQ state

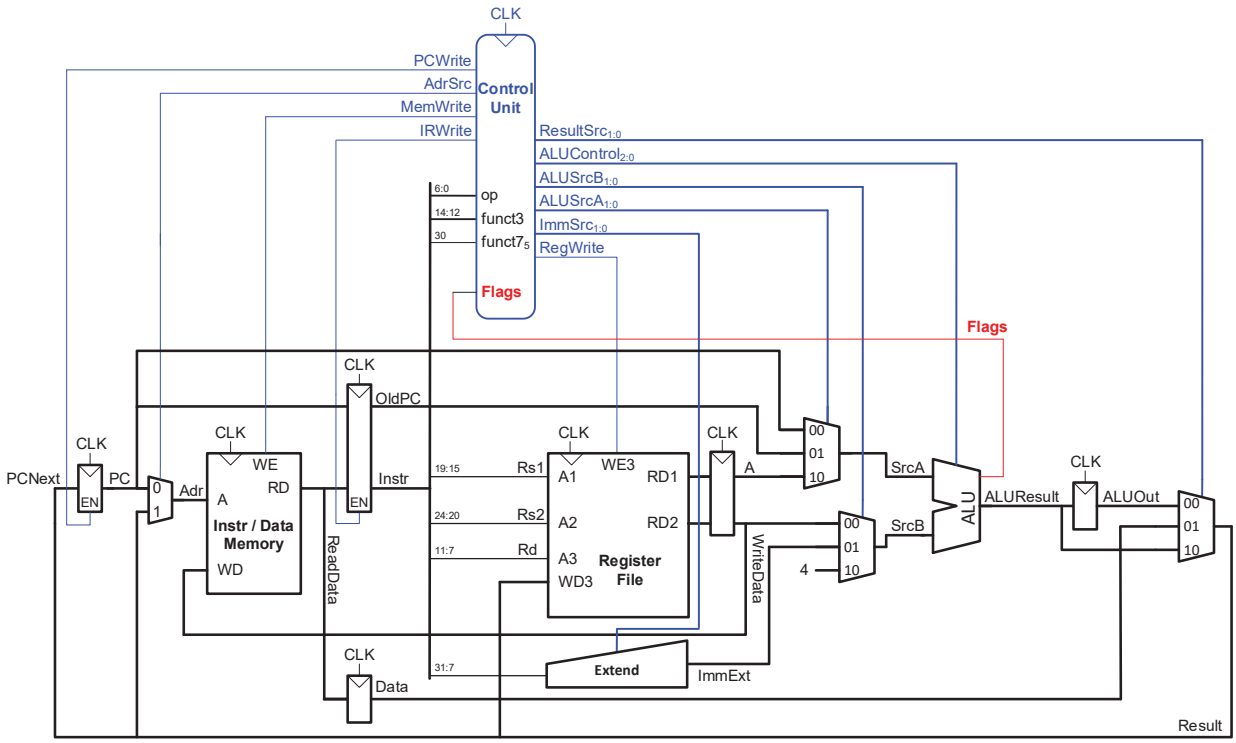


(f) `bge`

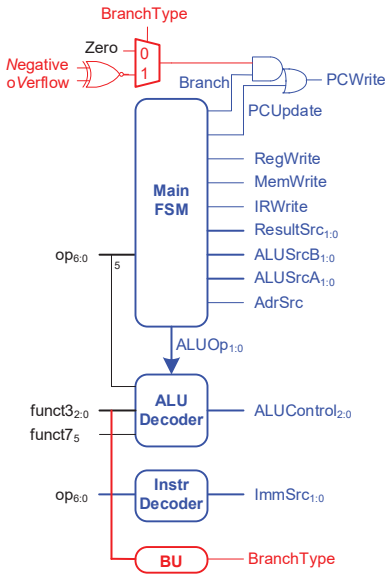
We start by using the ALU with flags (see Figure 5.17 in the textbook). For signed numbers, A is greater than or equal to B when A is not less than B. So, $A >= B$, when N XNOR V.

We also add a *BranchType* output of the Controller that is 1 when a `bge` instruction is executing and 0 when a `beq` instruction is executing. We add a Branch Unit (BU) to the controller to produce this signal (*BranchType*) using **funct3** as its input.

Enhanced datapath to support bge



Enhanced control unit for bge

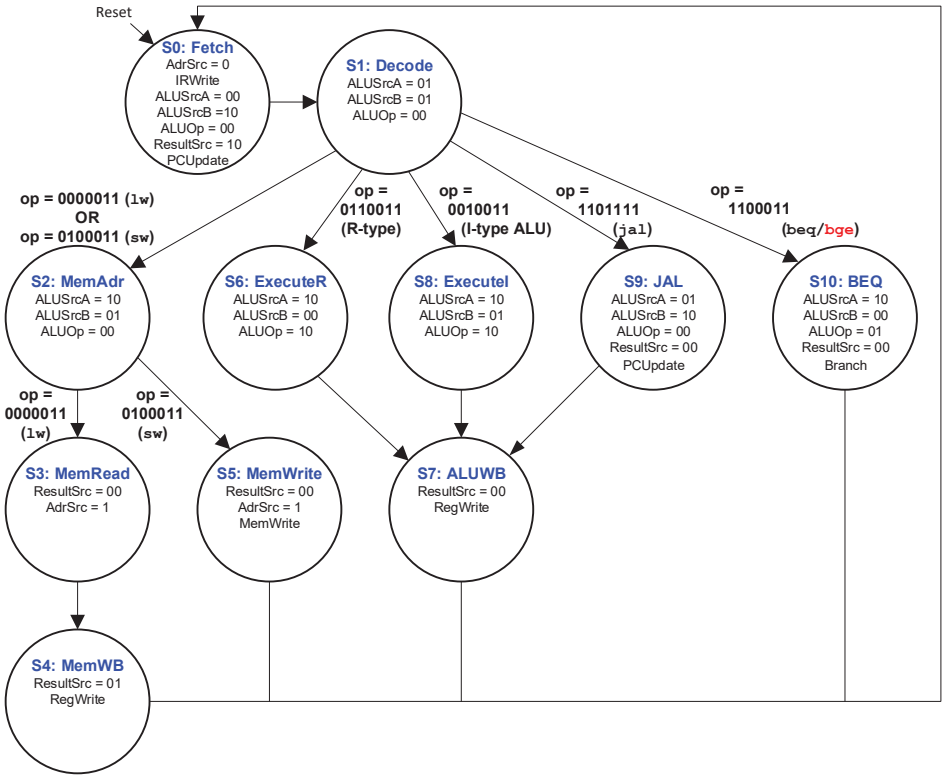


Branch Unit (BU) truth table to support bge

funct3	*BranchType*	Instruction
000	0	beq
110	1	bge

The Main FSM is not modified, but we add the `bge` label into the BEQ state for completeness.

Main FSM with `bge` label into the BEQ state

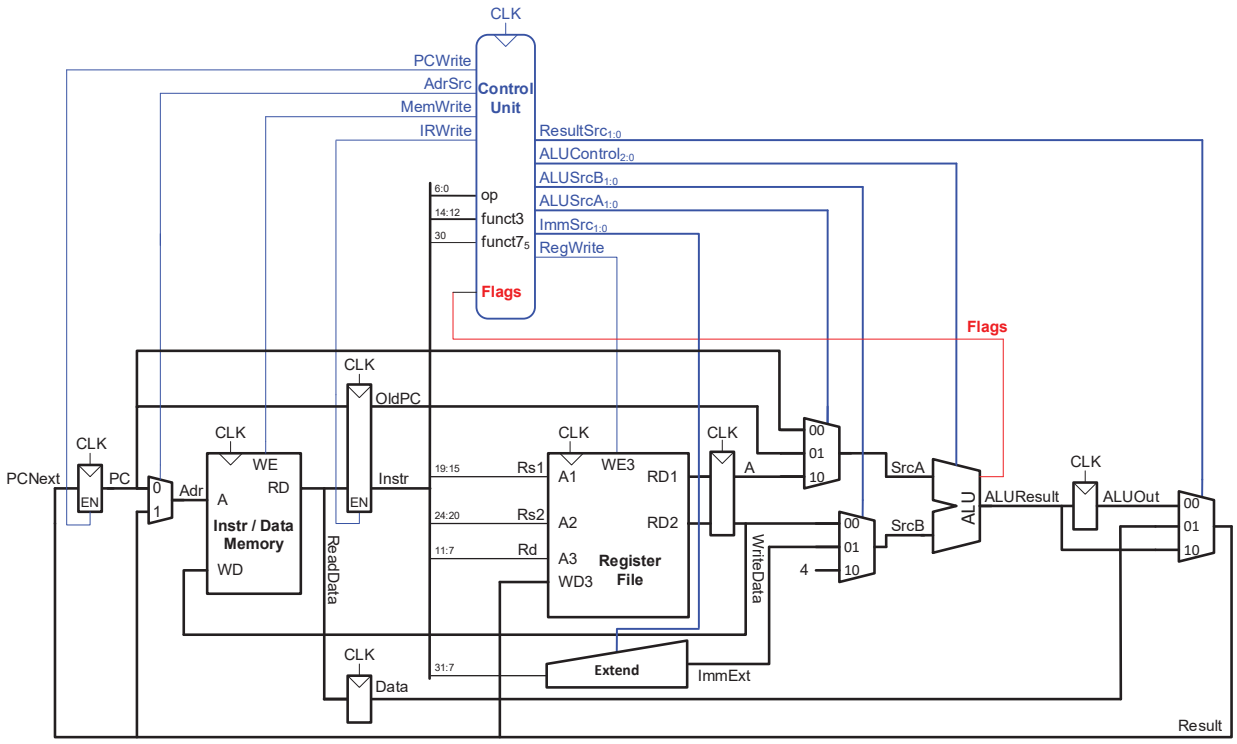


(g) `bgeu`

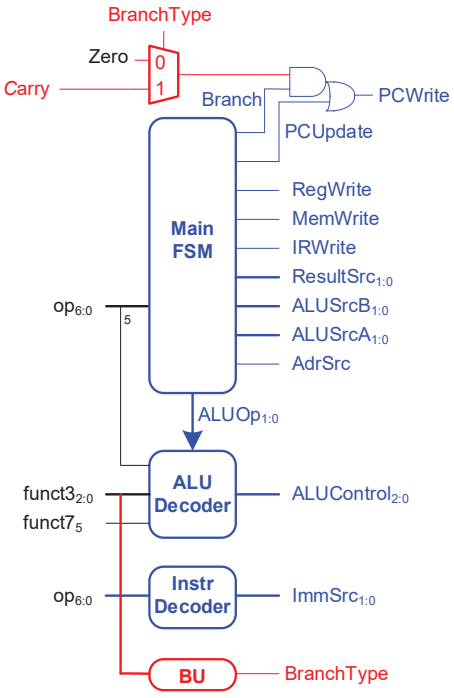
We start by using the ALU with flags (see Figure 5.17 in the textbook). For unsigned numbers, *A* is greater than or equal to *B* when the ALU performs *A* - *B* and the *C*arry flag is 1.

We also add a *BranchType* output of the Controller that is 1 when a `bgeu` instruction is executing and 0 when a `beq` instruction is executing. We add a Branch Unit (BU) to the controller to produce this signal (*BranchType*) using **funct3** as its input.

Enhanced datapath to support bgeu



Enhanced control unit for bgeu

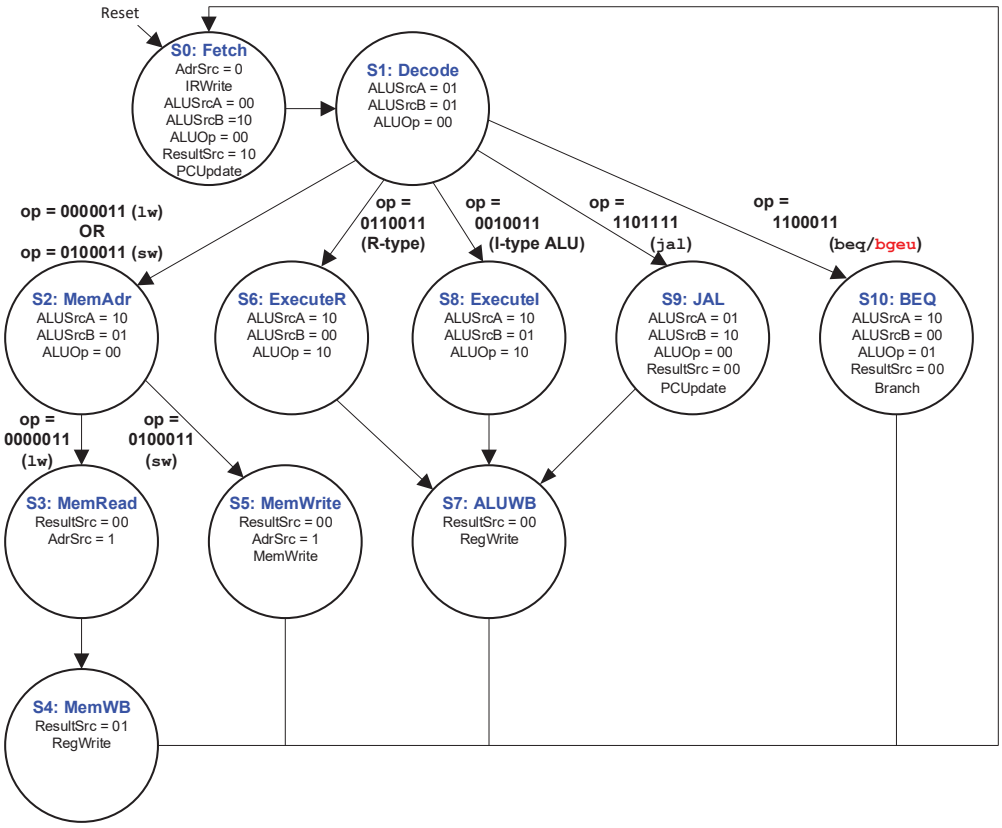


Branch Unit (BU) truth table to support `bgeu`

funct3	*BranchType*	Instruction
000	0	`beq`
111	1	`bltu`

The Main FSM is not modified, but we add the `bgeu` label into the BEQ state for completeness.

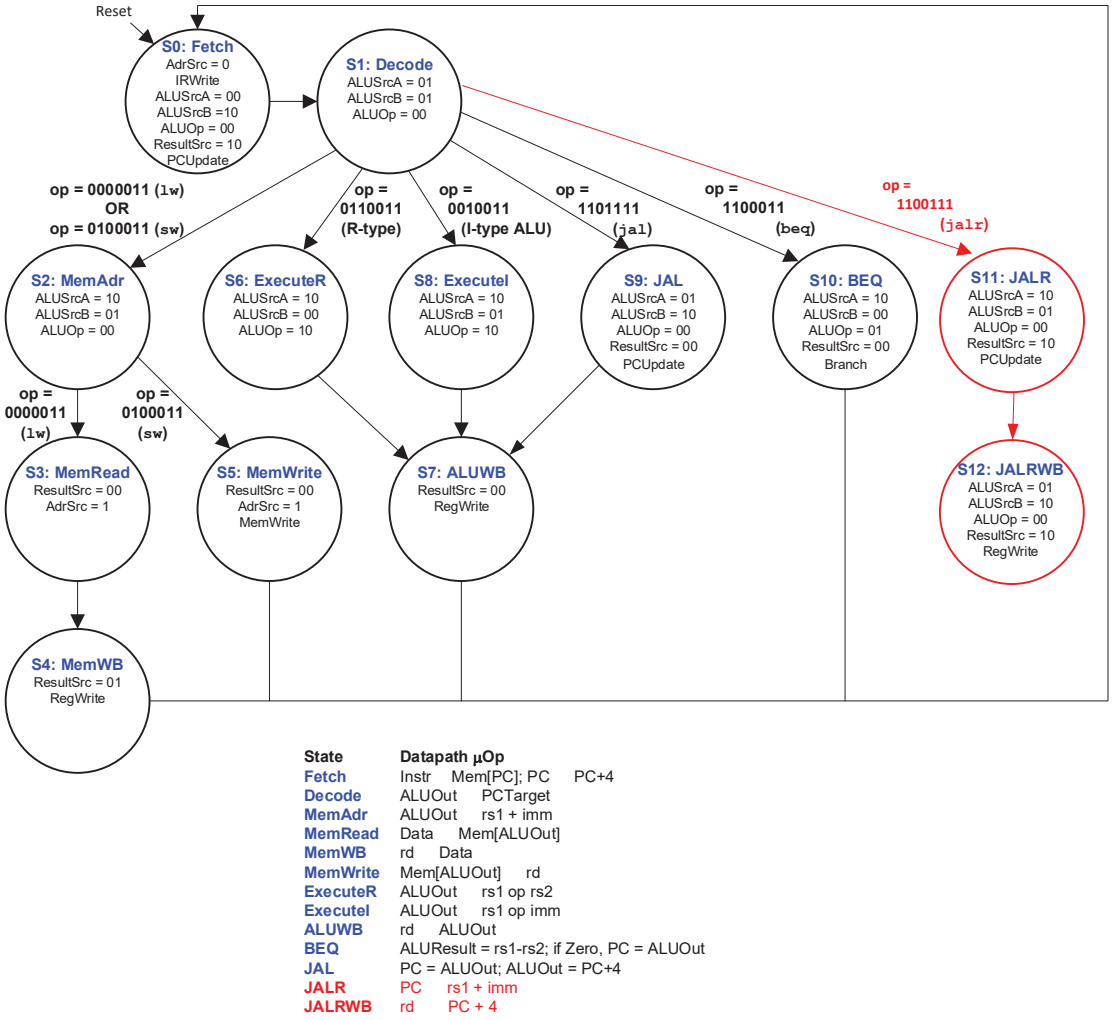
Main FSM with `bltu` label into the BEQ state



(h) `jalr`

The `jalr` instruction jumps to the target address calculated by adding **rs1** and the sign-extended 12-bit immediate. It also writes PC+4 to **rd**. The multicycle datapath already supports these calculations. We only modify the Main FSM by adding two states, JALR and JALRWB to, respectively, update the PC with the target address (**rs1** + imm) and update **rd** with PC + 4, as shown below.

Main FSM with two added states (JALR and JALRWB) to support `jalr`



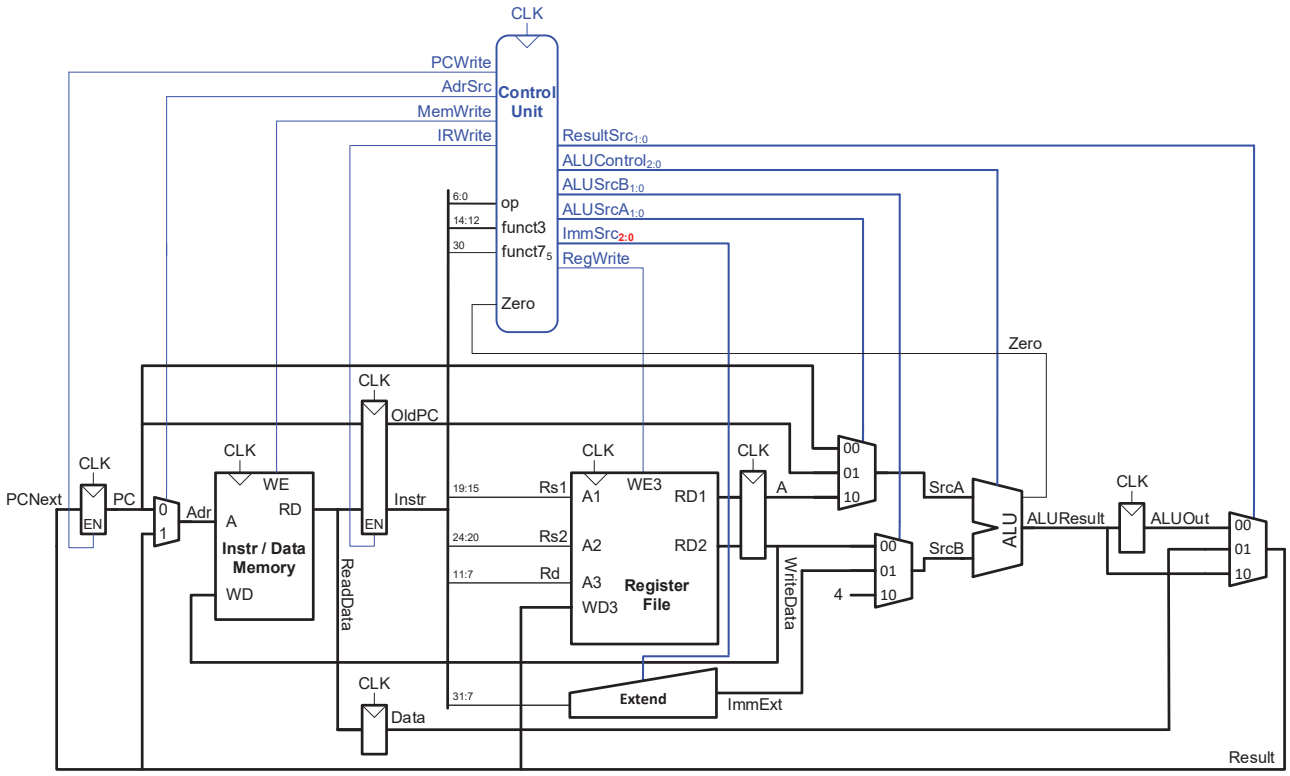
(i) `auipc`

The ImmSrc encoding table needs to be expanded to support U-type instructions (`auipc` in this case).

Enhanced ImmSrc encoding to support U-type instructions

ImmSrc	*ImmExt*	Type	Description
000	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
001	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
010	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate
011	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1'b0}	J	21-bit signed immediate
100	**{*Instr*[31:12], 12'b0}**	**U**	**20-bit upper immediate**

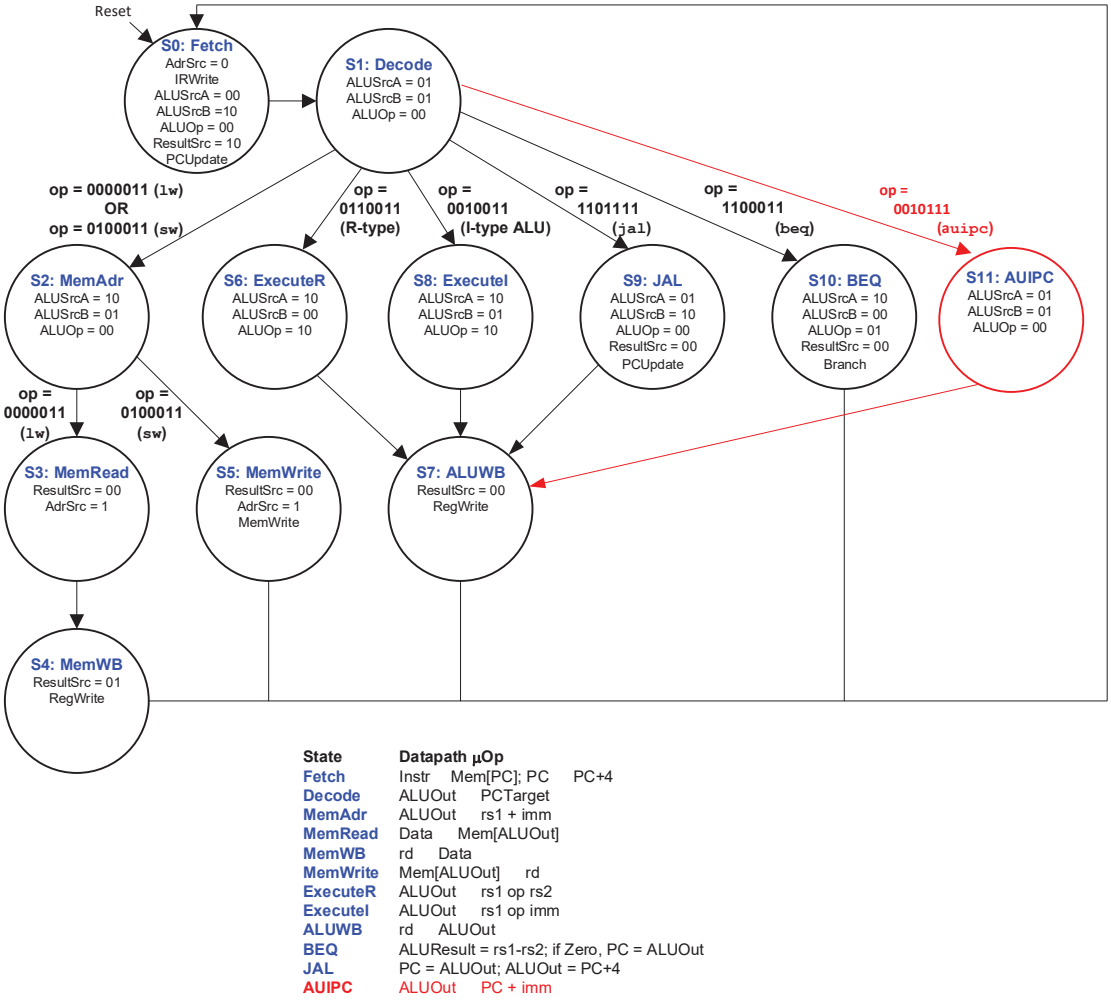
Next, we modify the datapath by increasing the width of the *ImmSrc* control signal to 3 bits.



Enhanced datapath to support `auipc`

Next, we modify the Main FSM by creating a new state, AUIPC, that adds PC + imm (i.e., the upper immediate) and places the result in the ALUOut register at the next clock edge. `auipc` then proceeds to the ALUWB (ALU Writeback) state where the results in the ALUOut register are written to **rd**.

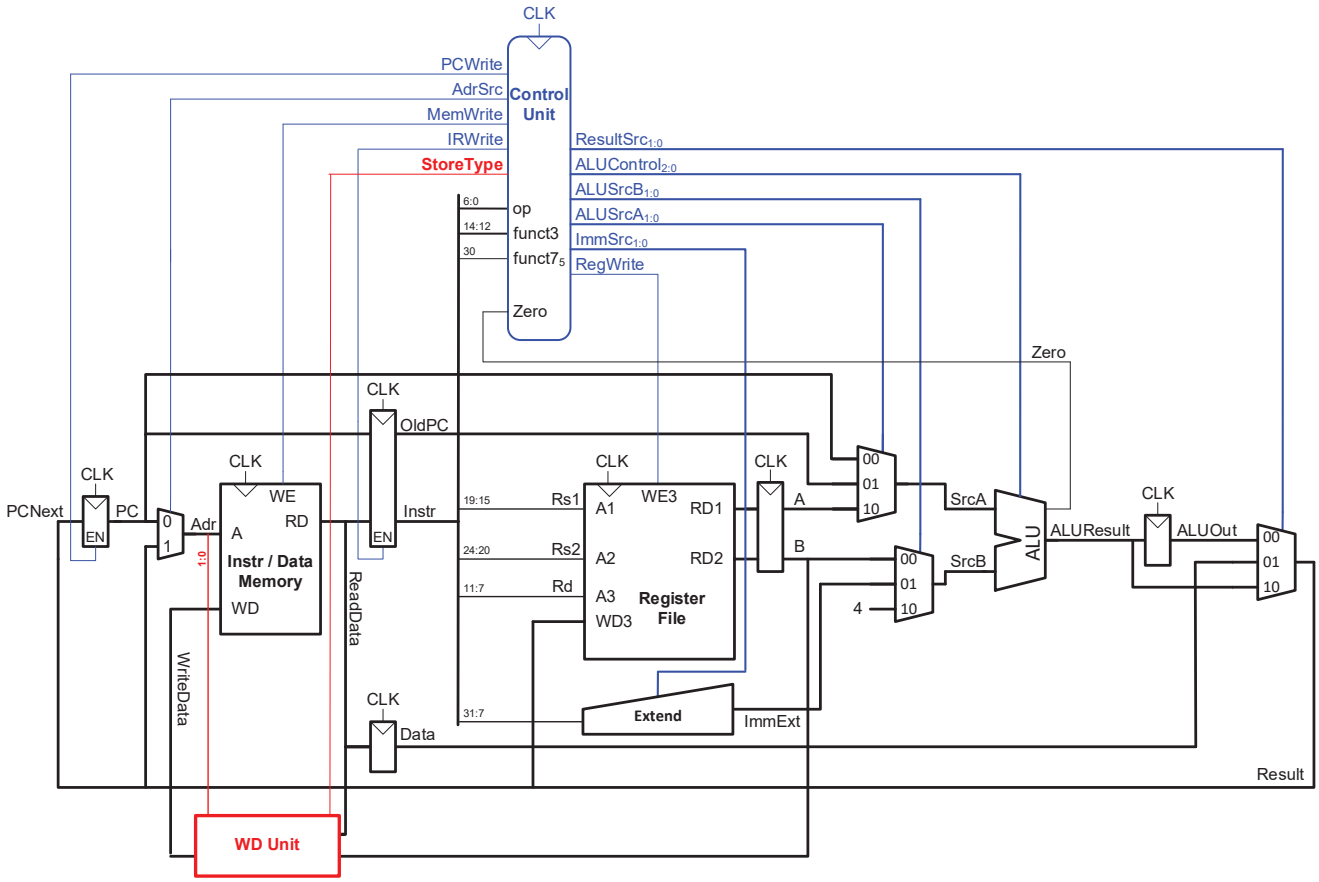
Enhanced Main FSM to support `auipc`

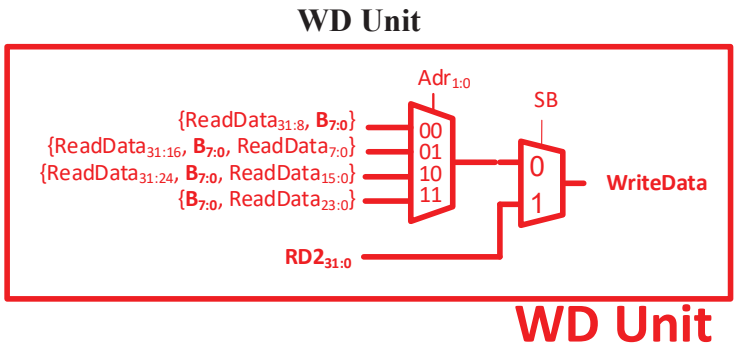


(j) `sb`

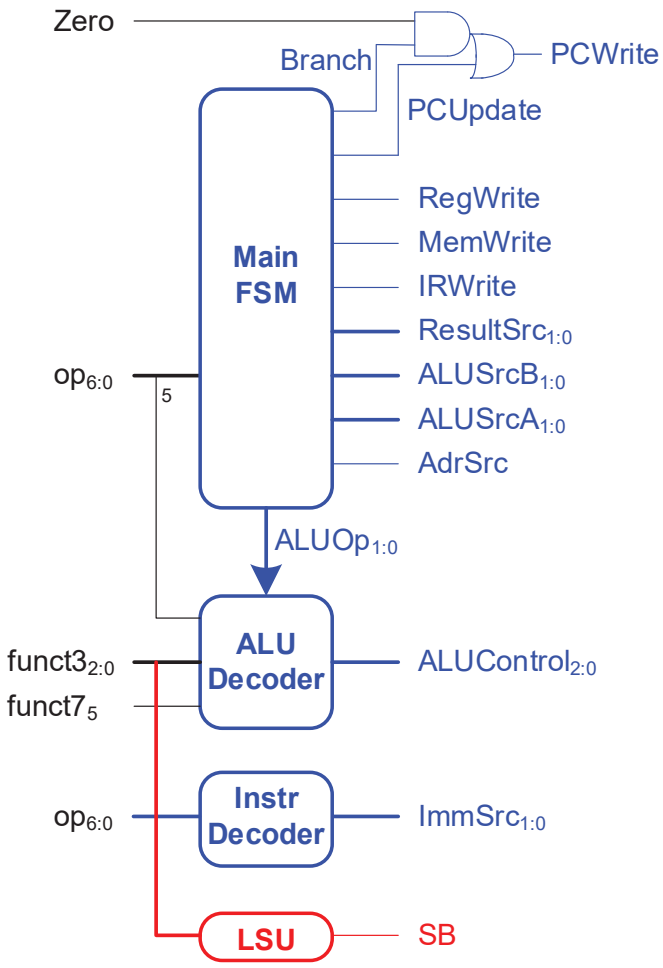
We add a WriteData Unit (WD Unit) to the datapath to write either RD2 or RD2 bit-swizzled with ReadData for sub-word writes. For `sb`, the least significant byte (lsb) of RD2 (**rs2**'s contents), replaces a byte of data in the *ReadData* bus depending on the byte offset of the memory address, $ALUResult_{1:0}$. We add an *StoreType* output of the control unit to choose either the entire word or the bitswizzled word to write to memory. The updated figures and additional hardware is shown below.

Enhanced datapath to support `sb` (WD Unit shown in next figure)


CLK
PCWrite
AdrSrc
MemWrite
IRWrite
StoreType
Control Unit
ResultSrc1:0
ALUControl2:0
ALUSrcB1:0
ALUSrcA1:0
ImmSrc1:0
RegWrite
6:0
14:12
30
op
funct3
funct75
Zero
Zero
CLK
OldPC
CLK
PCNext
PC
EN
CLK
WE
RD
A
Instr / Data Memory
WD
Adr
1:0
WriteData
ReadData
EN
Instr
CLK
19:15
24:20
11:7
Rs1
Rs2
Rd
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
CLK
A
B
00
01
10
4
SrcA
SrcB
ALU
ALUResult
CLK
ALUOut
00
01
10
CLK
Data
31:7
Extend
ImmExt
Result
WD Unit


WD Unit
Adr1:0
SB
{ReadData31:8, B7:0}
{ReadData31:16, B7:0, ReadData7:0}
{ReadData31:24, B7:0, ReadData15:0}
{B7:0, ReadData23:0}
00
01
10
11
0
1
WriteData
RD231:0
WD Unit

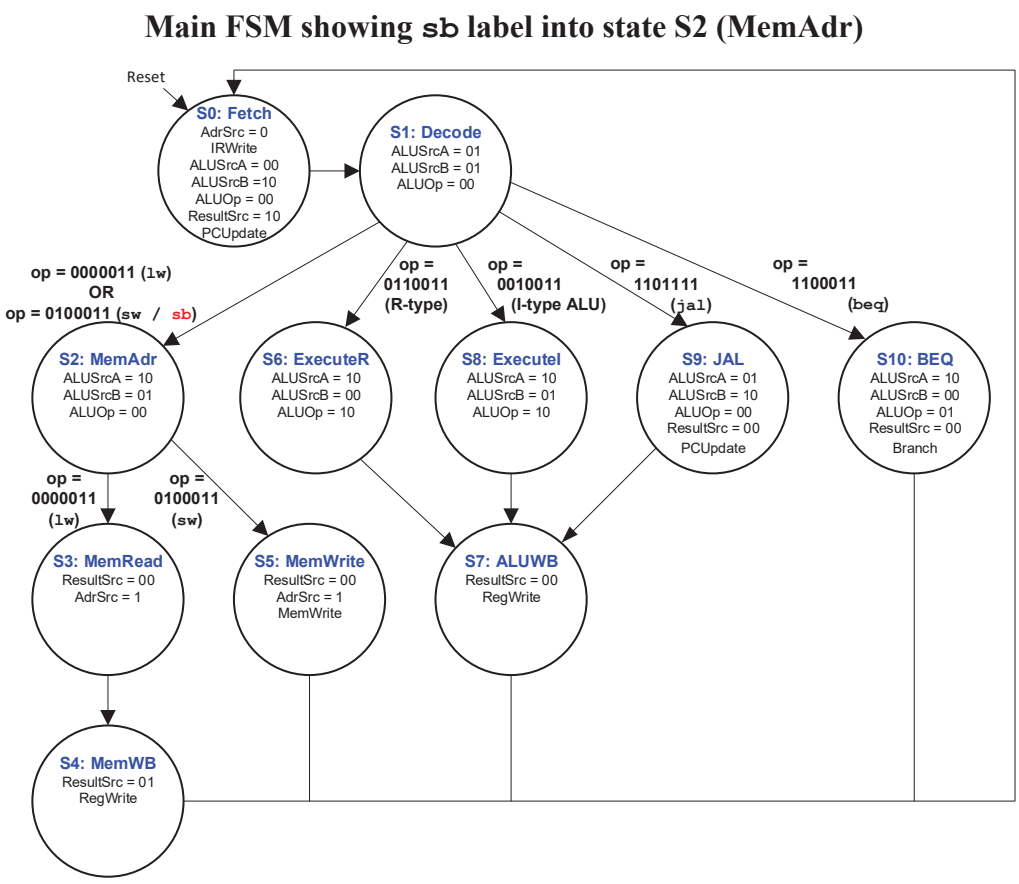
Enhanced control unit for `sb` : added Load/Store Unit (LSU)



Load/Store Unit (LSU) truth table to support `sb`

funct3	SB	Instruction
000	1	`sb`
010	0	`sw`

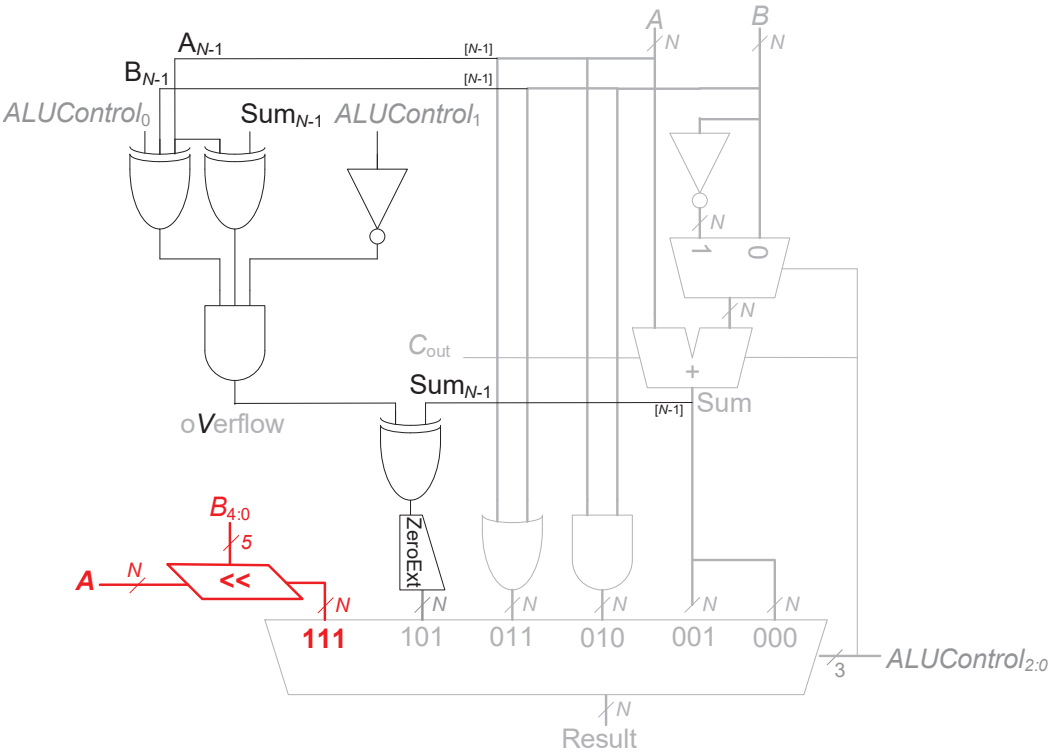
The Main FSM does not change, but we show the added label for sb. Remember that sw and sb share the same opcode.



(k) `slli`

The datapath and control already support I-type ALU operations, so we don't need to make any changes to those. We only modify the ALU and the ALU Decoder, as shown below. We add a shifter and expand the multiplexer inside the ALU.

Modified ALU to support `slli`



Modified ALU operations to support `slli`

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT
111	**sll**

Modified ALU Decoder truth table to support `slli`

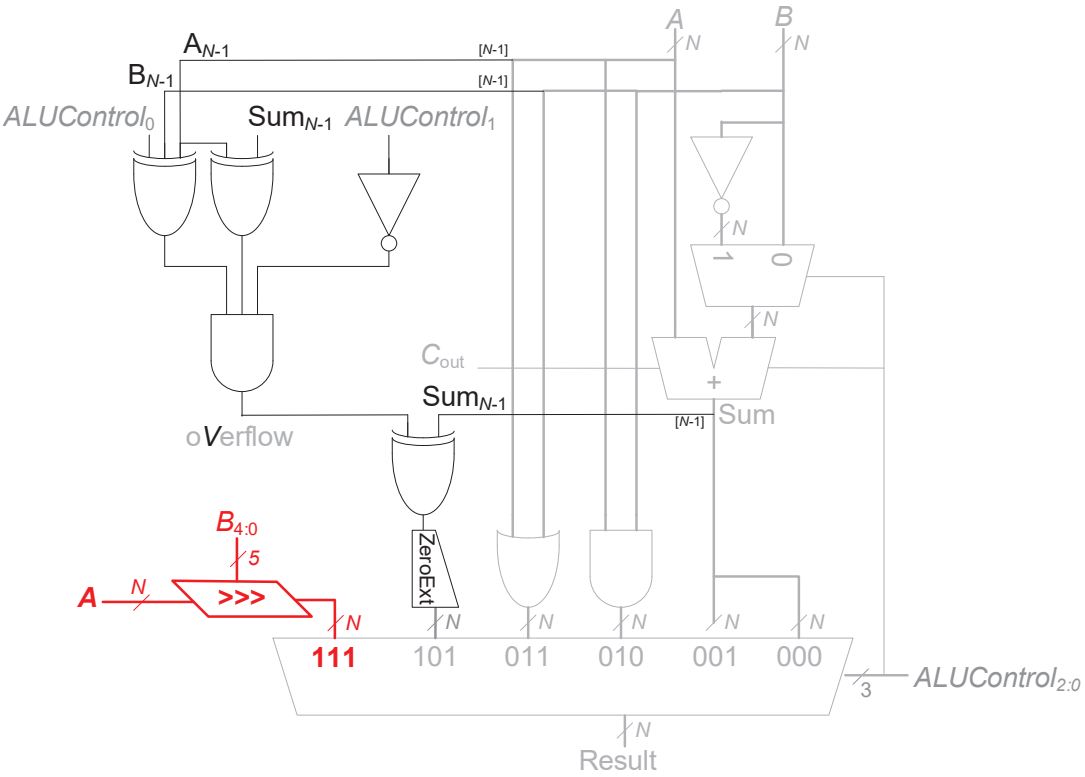
ALUOp	funct3	op_5, $funct7_5$	*ALUControl*	Instruction
00	x	x	000 (add)	`lw, sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add, addi`
	000	11	001 (subtract)	`sub`
	010	x	101 (set less than)	`slt, slti`
	001	**x**	**111 (shift left logical)**	**`sll, slli`**
	110	x	011 (or)	`or, ori`
	111	x	010 (and)	`and, andi`

(l) `srai`

The datapath and control already support I-type ALU operations, so we don't need to

make any changes to those. We only modify the ALU and the ALU Decoder, as shown below. We add a shifter and expand the multiplexer inside the ALU.

Modified ALU to support `srai`



Modified ALU operations to support `srai`

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
101	SLT
111	**sra**

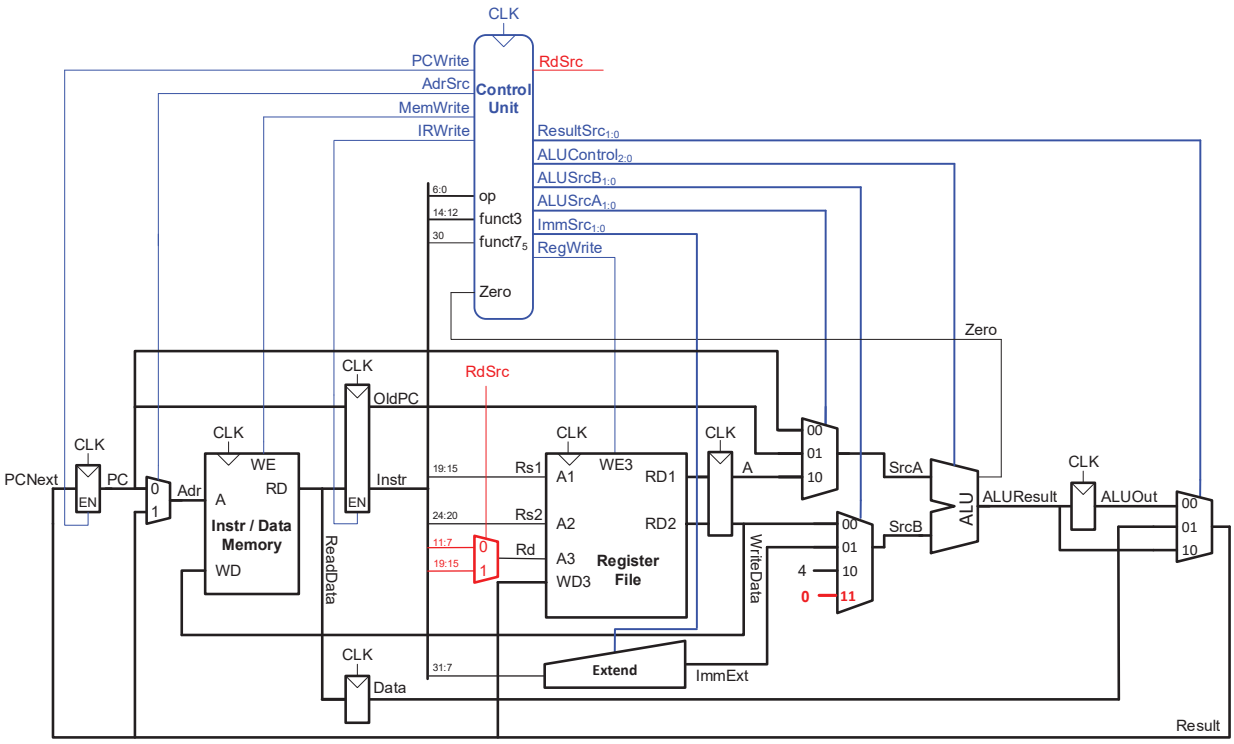
Modified ALU Decoder truth table to support `srai`

ALUOp	funct3	op_5, $funct7_5$	*ALUControl*	Instruction
00	x	xx	000 (add)	`lw, sw`
01	x	xx	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add, addi`
	000	11	001 (subtract)	`sub`
	010	xx	101 (set less than)	`slt, slti`
	101	**x1**	**111 (shift right arithmetic)**	**`sra, srai`**
	110	xx	011 (or)	`or, ori`
	111	xx	010 (and)	`and, andi`

Exercise 7.15

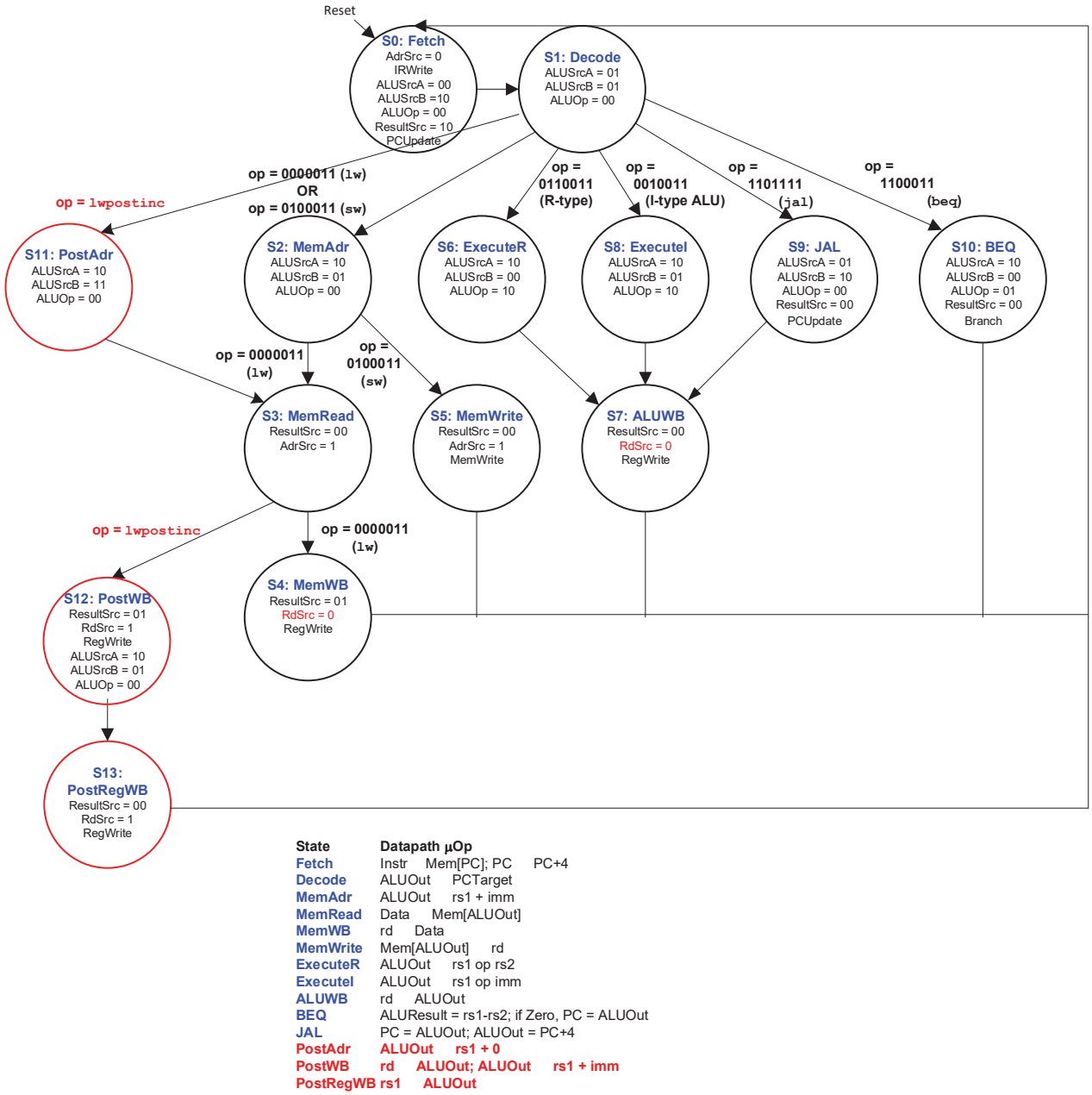
It is possible to add the `lwpostinc` instruction without modifying the register file. We modify the datapath so that the address can be calculated as: **rs1** + 0, as shown below. We added a 0 input to the ALUSrcB multiplexer, a multiplexer to choose between **rs1** and **rd** as the register to be written, and a *RdSrc* signal for the added mux.

Enhanced datapath to support `lwpostinc`



We add three states to the Main FSM: PostAdr, PostWB, PostRegWB. PostAdr calculates the address as **rs1** + 0. `lwpostinc` then proceeds to the MemRead state followed by the PostWB state, where it writes the loaded data to **rd** (and simultaneously calculates **rs1** + imm). Finally the instruction writes **rs1** + imm back to **rs1** in the PostRegWB state.

Main FSM showing added states and signals to support `lwpostinc`

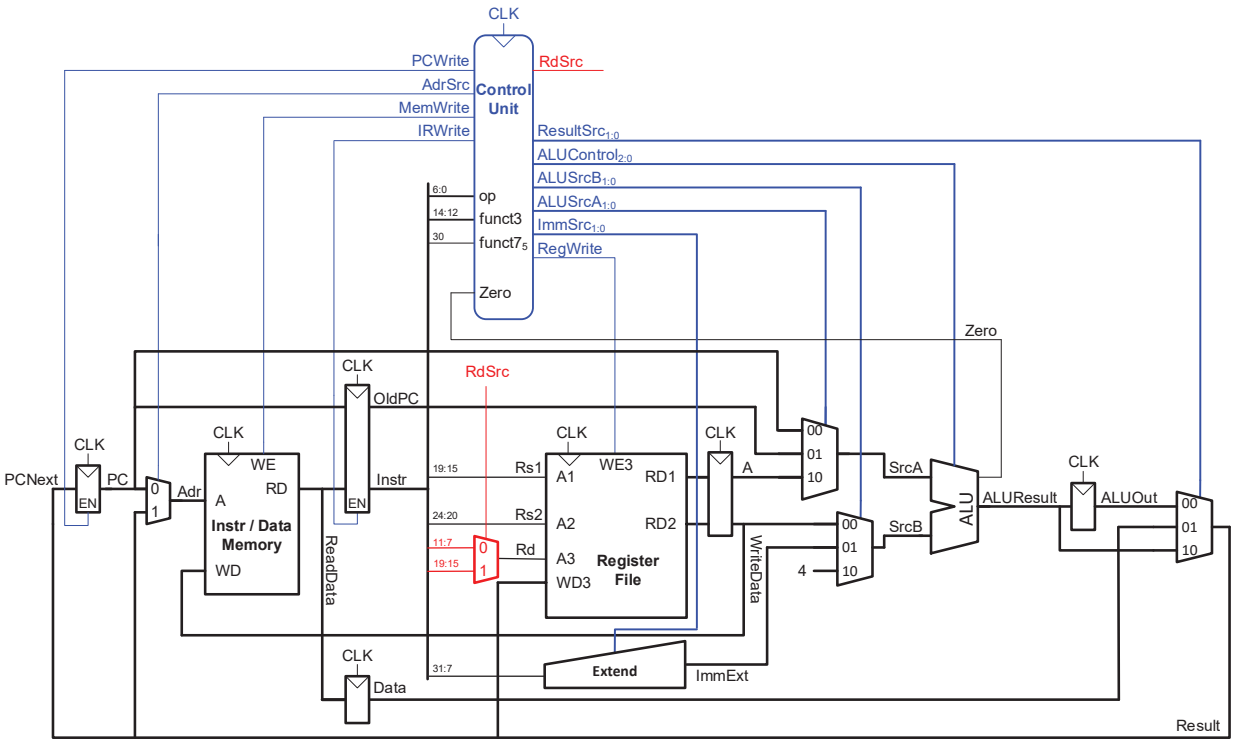


State	Datapath µOp
Fetch	Instr Mem[PC]; PC PC+4
Decode	ALUOut PCTarget
MemAdr	ALUOut rs1 + imm
MemRead	Data Mem[ALUOut]
MemWB	rd Data
MemWrite	Mem[ALUOut] rd
ExecuteR	ALUOut rs1 op rs2
ExecuteI	ALUOut rs1 op imm
ALUWB	rd ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4
PostAdr	ALUOut rs1 + 0
PostWB	rd ALUOut; ALUOut rs1 + imm
PostRegWB	rs1 ALUOut

Exercise 7.16

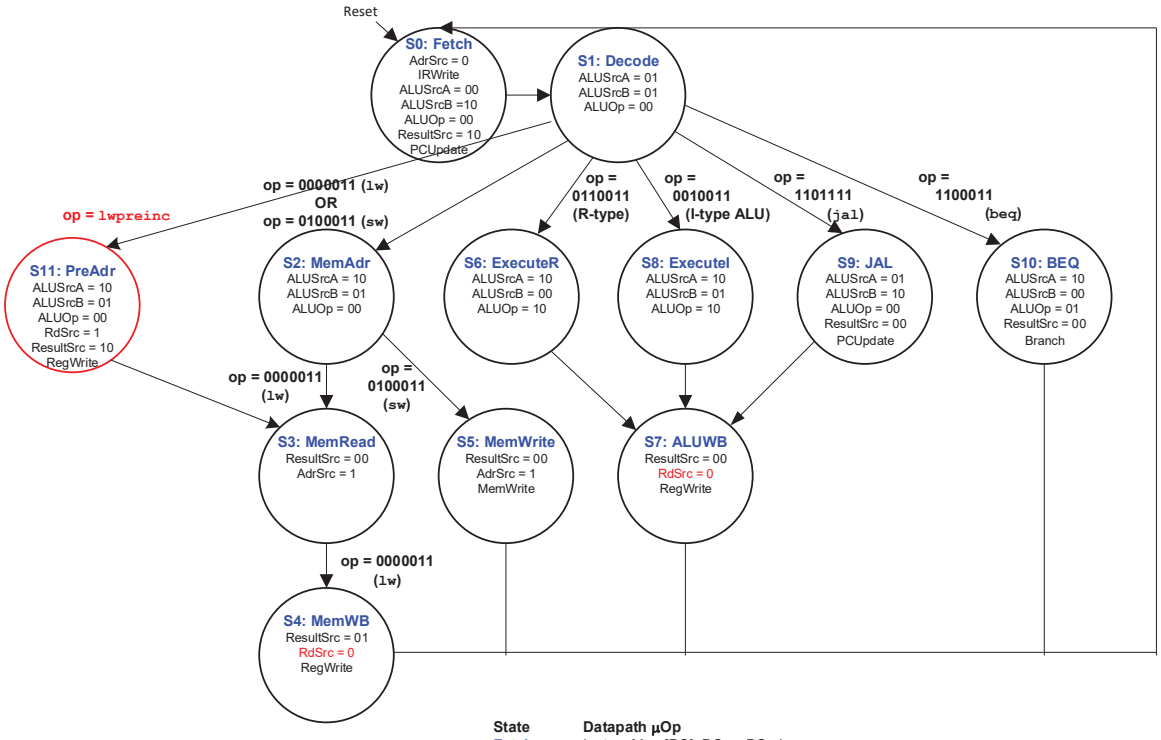
It is possible to add the `lwpreinc` instruction without modifying the register file. We added a multiplexer to the datapath to choose between **rs1** and **rd** as the register to be written, and a *RdSrc* signal for the added mux, as shown below.

Enhanced datapath to support `lwpreinc`



We add one state to the Main FSM: PreAdr. PreAdr calculates the address as **rs1** + imm and simultaneously writes this result to rs1. It then proceeds to the same states as a normal `lw` instructions.

Main FSM showing added states and signals to support `lwpreinc`



State	Datapath µOp
Fetch	Instr Mem[PC]; PC PC+4
Decode	ALUOut PCTarget
MemAdr	ALUOut rs1 + imm
MemRead	Data Mem[ALUOut]
MemWB	rd Data
MemWrite	Mem[ALUOut] rd
ExecuteR	ALUOut rs1 op rs2
ExecuteI	ALUOut rs1 op imm
ALUWB	rd ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4
PreAdr	ALUOut rs1 + imm; rs1 rs1 + imm

Exercise 7.17

The crack circuit designer should speed up the Memory Unit.

$$T_{c\_multi\_new} = t_{pcq} + t_{dec} + 2t_{mux} + \max[t_{ALU}, t_{mem}] + t_{setup}$$
$$= 40 + 25 + 2(30) + 120 + 50 = \mathbf{295\ ps}$$

Exercise 7.18

Because the ALU is not on the critical path (for max(t_{ALU}, t_{mem}), t_{mem} has the maximum delay, of 200 ps), the improved performance of the ALU will not change T_c. Thus, the cycle time will be the same as in Example 7.8, 375 ps. Given the instruction mix from Example 7.7 (resulting in a CPI of 4.12), the execution time for 100 billion instructions is still 155 seconds (see Example 7.8 for additional details).

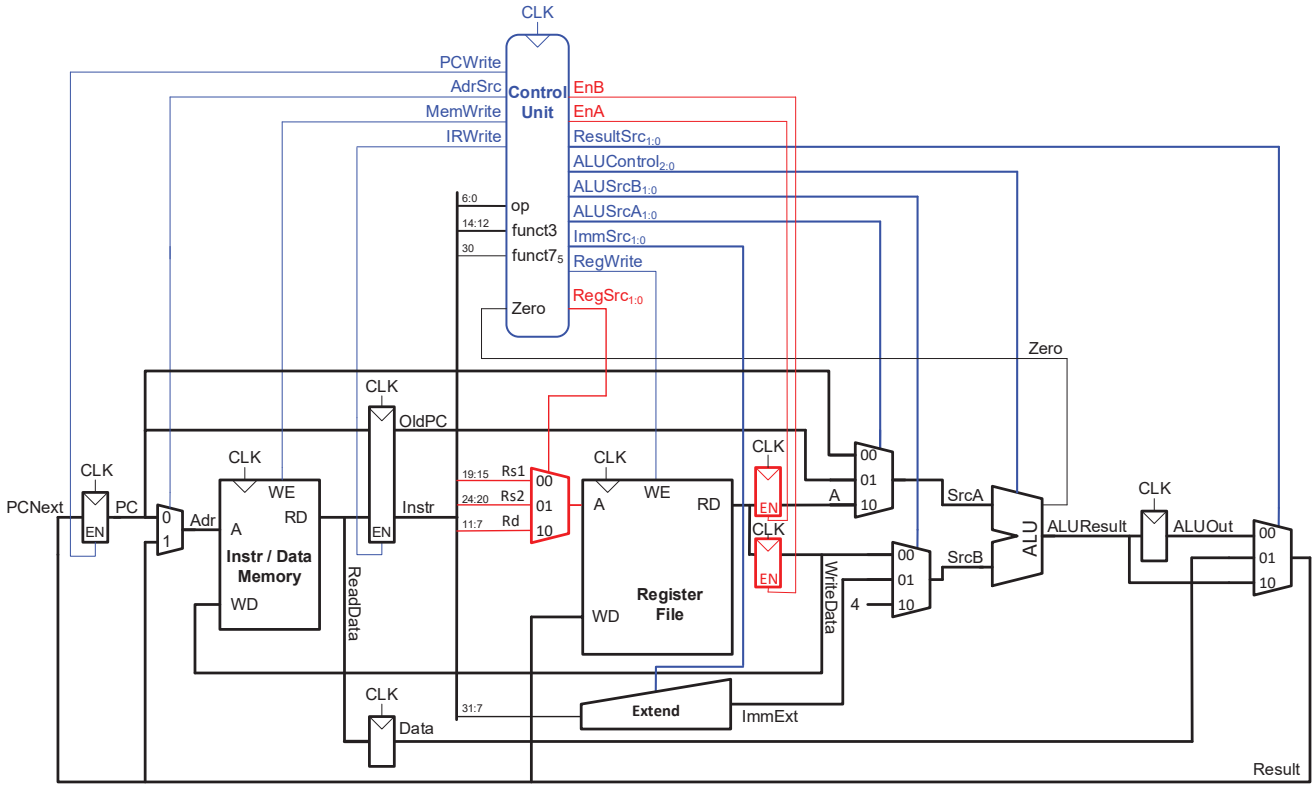
Exercise 7.19

The crack circuit designer should speed up the Memory Unit and should make it equal to the delay of the ALU (120ps). The cycle time would then be 295 ps, the same calculation as in Exercise 7.17.

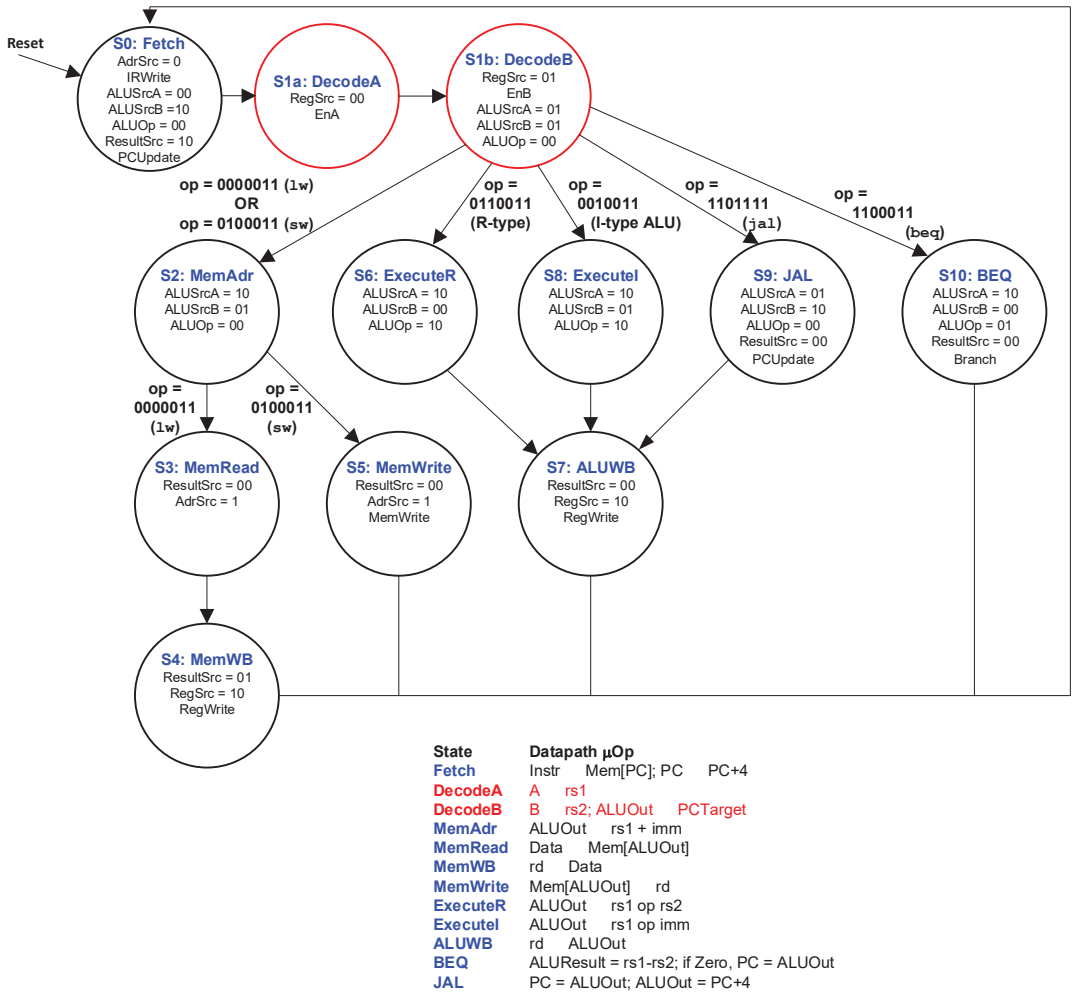
Exercise 7.20

We separate the A/WriteData register into two separate registers, each with enable signals, *EnA* and *EnB*. We separate the Decode state into two states, one to write the A register and one to write the B (i.e., WriteData) register. The updated datapath and Main FSM are below.

Revised datapath



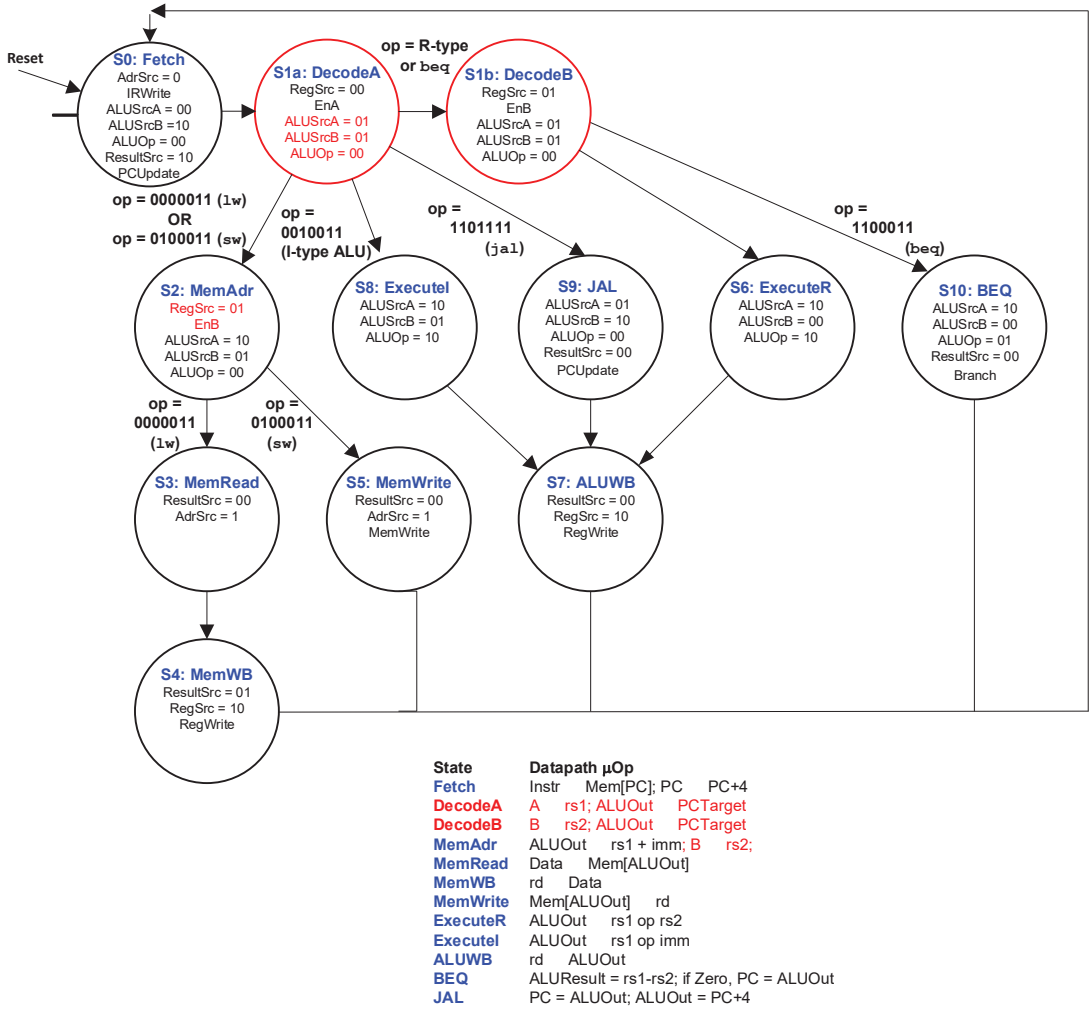
Revised Main FSM



State	Datapath µOp
Fetch	Instr Mem[PC]; PC PC+4
DecodeA	A rs1
DecodeB	B rs2; ALUOut PCTarget
MemAdr	ALUOut rs1 + imm
MemRead	Data Mem[ALUOut]
MemWB	rd Data
MemWrite	Mem[ALUOut] rd
ExecuteR	ALUOut rs1 op rs2
ExecuteI	ALUOut rs1 op imm
ALUWB	rd ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4

Note that `jal`, I-type ALU, and `lw` instructions only need to decode one source register, so those two instructions would work with just a single decode stage (DecodeA). For `jal`, the signals for calculating the target address would need to move to the DecodeA state. Similarly, `sw` instructions could save a cycle by using the first Decode state only (DecodeA) and then reading **rs2** during the MemAdr state.
Here is the optimized version of the Main FSM.

Optimized Main FSM



Exercise 7.21

Alyssa should switch to the slower but lower power register file. By doubling the delay of the register file, it still does not place it on the critical path. This means that power will be saved without affecting the cycle time.

Specifically, the path that includes the register files would require the following constraints:

$T_{c\_multi\_RF} = t_{pcq} + t_{RFread} + t_{setup}$

$T_{c\_multi\_RF} = (40 + 100 + 50)\text{ ps} = 190\text{ ps}$

With twice as much register file (RF) delay, this constraint would be:

$T_{c\_multi\_RF} = (40 + \mathbf{2 * 100} + 50)\text{ ps} = 290\text{ ps}$, which is still less than the 375 ps cycle time required by the path through memory.

Exercise 7.22

The updated amount of cycles per instruction is:
Loads = 5 cycles (same)
Stores = 4 cycles (same)
R-types = 5 cycles (1 additional cycle)
Branches = 4 cycles (1 additional cycle)
CPI = (0.25 + 0.52)(5) + (0.10 + 0.13)(4) = **4.77**

Exercise 7.23

The program will execute 6 `addi` (4 cycles each), 6 `bge` (3 cycles each), and 5 `jal` (4 cycles each) instructions for a total of (6×4) + (6×3) + (5×4) = **62 clock cycles** for 17 instructions. Thus, the CPI of this program is 62/17 = **3.65 CPI**. (Remember that `j L1` is a pseudoinstruction for `jal x0, L1`.)

Exercise 7.24

The program executes:

- 3 + 10×2 = **23 `add` or `addi`** instructions (3 in the preamble and 2 for each of 10 loop iterations). Each `add` or `addi` takes **4 cycles**.
- **11 `beq`** instructions (1 for each of 10 loop iterations, plus one for last check before exiting the loop). Each `beq` takes **3 cycles**.
- **10 `jal`** instructions (1 for each of 10 loop iterations). Each `jal` takes **4 cycles**.

So, the program takes: ((23+10) × 4) + (11 × 4) = **165 clock cycles** `to execute` 3 + (4 × 10) + 1 = **44 instructions**. Thus, the CPI equals
165 clock cycles / 44 instructions = **3.75 CPI.**

Exercise 7.25

RISC-V multicycle processor

```
module testbench();

  logic        clk;
  logic        reset;

  logic [31:0] WriteData, DataAdr;
  logic        MemWrite;

  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end
```

```
  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check results
  always @(negedge clk)
    begin
      if(MemWrite) begin
        if(DataAdr === 100 & WriteData === 25) begin
          $display("Simulation succeeded");
          $stop;
        end else if (DataAdr !== 96) begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule

module top(input  logic        clk, reset,
           output logic [31:0] WriteDataM, DataAdrM,
           output logic        MemWriteM);

  logic [31:0] PCF, InstrF, ReadDataM;

  // instantiate processor and memories
  riscv riscv(clk, reset, PCF, InstrF, MemWriteM, DataAdrM,
              WriteDataM, ReadDataM);
  imem imem(PCF, InstrF);
  dmem dmem(clk, MemWriteM, DataAdrM, WriteDataM, ReadDataM);
endmodule

module riscv(input  logic        clk, reset,
             output logic [31:0] PCF,
             input  logic [31:0] InstrF,
             output logic        MemWriteM,
             output logic [31:0] ALUResultM, WriteDataM,
             input  logic [31:0] ReadDataM);

  logic [6:0]  opD;
  logic [2:0]  funct3D;
  logic        funct7b5D;
  logic [1:0]  ImmSrcD;
  logic        ZeroE;
  logic        PCSrcE;
  logic [2:0]  ALUControlE;
  logic        ALUSrcE;
  logic        ResultSrcEb0;
  logic        RegWriteM;
  logic [1:0]  ResultSrcW;
  logic        RegWriteW;

  logic [1:0]  ForwardAE, ForwardBE;
  logic        StallF, StallD, FlushD, FlushE;
```

```
  logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW;

  controller c(clk, reset,
               opD, funct3D, funct7b5D, ImmSrcD,
               FlushE, ZeroE, PCSrcE, ALUControlE, ALUSrcE, ResultSrcEb0,
               MemWriteM, RegWriteM,
                     RegWriteW, ResultSrcW);

  datapath dp(clk, reset,
              StallF, PCF, InstrF,
                    opD, funct3D, funct7b5D, StallD, FlushD, ImmSrcD,
                    FlushE, ForwardAE, ForwardBE, PCSrcE, ALUControlE,
ALUSrcE, ZeroE,
              MemWriteM, WriteDataM, ALUResultM, ReadDataM,
              RegWriteW, ResultSrcW,
              Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW);

  hazard  hu(Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
             PCSrcE, ResultSrcEb0, RegWriteM, RegWriteW,
             ForwardAE, ForwardBE, StallF, StallD, FlushD, FlushE);

endmodule


module controller(input  logic             clk, reset,
                  // Decode stage control signals
                  input logic [6:0]  opD,
                  input logic [2:0]  funct3D,
                  input logic            funct7b5D,
                  output logic [1:0] ImmSrcD,
                  // Execute stage control signals
                  input logic            FlushE,
                  input logic            ZeroE,
                  output logic       PCSrcE,        // for datapath and
Hazard Unit
                  output logic [2:0] ALUControlE,
                  output logic           ALUSrcE,
                  output logic       ResultSrcEb0,  // for Hazard Unit
                  // Memory stage control signals
                  output logic           MemWriteM,
                  output logic       RegWriteM,     // for Hazard Unit

                  // Writeback stage control signals
                  output logic           RegWriteW,     // for datapath and
Hazard Unit
                  output logic [1:0] ResultSrcW);

  // pipelined control signals
  logic       RegWriteD, RegWriteE;
  logic [1:0] ResultSrcD, ResultSrcE, ResultSrcM;
  logic       MemWriteD, MemWriteE;
  logic             JumpD, JumpE;
  logic             BranchD, BranchE;
  logic     [1:0] ALUOpD;
  logic [2:0] ALUControlD;
  logic       ALUSrcD;
```

```
  // Decode stage logic
  maindec md(opD, ResultSrcD, MemWriteD, BranchD,
             ALUSrcD, RegWriteD, JumpD, ImmSrcD, ALUOpD);
  aludec  ad(opD[5], funct3D, funct7b5D, ALUOpD, ALUControlD);

  // Execute stage pipeline control register and logic
  floprc #(10) controlregE(clk, reset, FlushE,
                           {RegWriteD, ResultSrcD, MemWriteD, JumpD, BranchD,
ALUControlD, ALUSrcD},
                           {RegWriteE, ResultSrcE, MemWriteE, JumpE, BranchE,
ALUControlE, ALUSrcE});

  assign PCSrcE = (BranchE & ZeroE) | JumpE;
  assign ResultSrcEb0 = ResultSrcE[0];

  // Memory stage pipeline control register
  flopr #(4) controlregM(clk, reset,
                         {RegWriteE, ResultSrcE, MemWriteE},
                         {RegWriteM, ResultSrcM, MemWriteM});

  // Writeback stage pipeline control register
  flopr #(3) controlregW(clk, reset,
                         {RegWriteM, ResultSrcM},
                         {RegWriteW, ResultSrcW});
endmodule

module maindec(input  logic [6:0] op,
               output logic [1:0] ResultSrc,
               output logic       MemWrite,
               output logic       Branch, ALUSrc,
               output logic       RegWrite, Jump,
               output logic [1:0] ImmSrc,
               output logic [1:0] ALUOp);

  logic [10:0] controls;

  assign {RegWrite, ImmSrc, ALUSrc, MemWrite,
          ResultSrc, Branch, ALUOp, Jump} = controls;

  always_comb
    case(op)
    // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump
      7'b0000011: controls = 11'b1_00_1_0_01_0_00_0; // lw
      7'b0100011: controls = 11'b0_01_1_1_00_0_00_0; // sw
      7'b0110011: controls = 11'b1_xx_0_0_00_0_10_0; // R-type
      7'b1100011: controls = 11'b0_10_0_0_00_1_01_0; // beq
      7'b0010011: controls = 11'b1_00_1_0_00_0_10_0; // I-type ALU
      7'b1101111: controls = 11'b1_11_0_0_10_0_00_1; // jal
      7'b0000000: controls = 11'b0_00_0_0_00_0_00_0; // need valid values at
reset
      default:    controls = 11'bx_xx_x_x_xx_x_xx_x; // non-implemented
instruction
    endcase
endmodule

module aludec(input  logic       opb5,
              input  logic [2:0] funct3,
```

```
              input  logic       funct7b5,
              input  logic [1:0] ALUOp,
              output logic [2:0] ALUControl);

  logic  RtypeSub;
  assign RtypeSub = funct7b5 & opb5;  // TRUE for R-type subtract instruction

  always_comb
    case(ALUOp)
      2'b00:                ALUControl = 3'b000; // addition
      2'b01:                ALUControl = 3'b001; // subtraction
      default: case(funct3) // R-type or I-type ALU
                 3'b000:  if (RtypeSub)
                            ALUControl = 3'b001; // sub
                          else
                            ALUControl = 3'b000; // add, addi
                 3'b010:    ALUControl = 3'b101; // slt, slti
                 3'b110:    ALUControl = 3'b011; // or, ori
                 3'b111:    ALUControl = 3'b010; // and, andi
                 default:   ALUControl = 3'bxxx; // ???
               endcase
    endcase
endmodule

module datapath(input logic clk, reset,
                // Fetch stage signals
                input  logic        StallF,
                output logic [31:0] PCF,
                input  logic [31:0] InstrF,
                // Decode stage signals
                output logic [6:0]  opD,
                output logic [2:0]  funct3D,
                output logic        funct7b5D,
                input  logic        StallD, FlushD,
                input  logic [1:0]  ImmSrcD,
                // Execute stage signals
                input  logic        FlushE,
                input  logic [1:0]  ForwardAE, ForwardBE,
                input  logic        PCSrcE,
                input  logic [2:0]  ALUControlE,
                input  logic        ALUSrcE,
                output logic        ZeroE,
                // Memory stage signals
                input  logic        MemWriteM,
                output logic [31:0] WriteDataM, ALUResultM,
                input  logic [31:0] ReadDataM,
                // Writeback stage signals
                input  logic        RegWriteW,
                input  logic [1:0]  ResultSrcW,
                // Hazard Unit signals
                output logic [4:0]  Rs1D, Rs2D, Rs1E, Rs2E,
                output logic [4:0]  RdE, RdM, RdW);

  // Fetch stage signals
  logic [31:0] PCNextF, PCPlus4F;
  // Decode stage signals
  logic [31:0] InstrD;
```

```
logic [31:0] PCD, PCPlus4D;
logic [31:0] RD1D, RD2D;
logic [31:0] ImmExtD;
logic [4:0]  RdD;
// Execute stage signals
logic [31:0] RD1E, RD2E;
logic [31:0] PCE, ImmExtE;
logic [31:0] SrcAE, SrcBE;
logic [31:0] ALUResultE;
logic [31:0] WriteDataE;
logic [31:0] PCPlus4E;
logic [31:0] PCTargetE;
// Memory stage signals
logic [31:0] PCPlus4M;
// Writeback stage signals
logic [31:0] ALUResultW;
logic [31:0] ReadDataW;
logic [31:0] PCPlus4W;
logic [31:0] ResultW;

// Fetch stage pipeline register and logic
mux2    #(32) pcmux(PCPlus4F, PCTargetE, PCSrcE, PCNextF);
flopenr #(32) pcreg(clk, reset, ~StallF, PCNextF, PCF);
adder         pcadd(PCF, 32'h4, PCPlus4F);

// Decode stage pipeline register and logic
flopenrc #(96) regD(clk, reset, FlushD, ~StallD,
                    {InstrF, PCF, PCPlus4F},
                    {InstrD, PCD, PCPlus4D});
assign opD       = InstrD[6:0];
assign funct3D   = InstrD[14:12];
assign funct7b5D = InstrD[30];
assign Rs1D      = InstrD[19:15];
assign Rs2D      = InstrD[24:20];
assign RdD       = InstrD[11:7];

regfile        rf(clk, RegWriteW, Rs1D, Rs2D, RdW, ResultW, RD1D, RD2D);
extend         ext(InstrD[31:7], ImmSrcD, ImmExtD);

// Execute stage pipeline register and logic
floprc #(175) regE(clk, reset, FlushE,
                   {RD1D, RD2D, PCD, Rs1D, Rs2D, RdD, ImmExtD, PCPlus4D},
                   {RD1E, RD2E, PCE, Rs1E, Rs2E, RdE, ImmExtE, PCPlus4E});

mux3   #(32)  faemux(RD1E, ResultW, ALUResultM, ForwardAE, SrcAE);
mux3   #(32)  fbemux(RD2E, ResultW, ALUResultM, ForwardBE, WriteDataE);
mux2   #(32)  srcbmux(WriteDataE, ImmExtE, ALUSrcE, SrcBE);
alu           alu(SrcAE, SrcBE, ALUControlE, ALUResultE, ZeroE);
adder         branchadd(ImmExtE, PCE, PCTargetE);

// Memory stage pipeline register
flopr  #(101) regM(clk, reset,
                   {ALUResultE, WriteDataE, RdE, PCPlus4E},
                   {ALUResultM, WriteDataM, RdM, PCPlus4M});

// Writeback stage pipeline register and logic
flopr  #(101) regW(clk, reset,
```

```
                    {ALUResultM, ReadDataM, RdM, PCPlus4M},
                    {ALUResultW, ReadDataW, RdW, PCPlus4W});
  mux3   #(32)  resultmux(ALUResultW, ReadDataW, PCPlus4W, ResultSrcW,
ResultW);
endmodule

// Hazard Unit: forward, stall, and flush
module hazard(input  logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
              input  logic       PCSrcE, ResultSrcEb0,
              input  logic       RegWriteM, RegWriteW,
              output logic [1:0] ForwardAE, ForwardBE,
              output logic       StallF, StallD, FlushD, FlushE);

  logic lwStallD;

  // forwarding logic
  always_comb begin
    ForwardAE = 2'b00;
    ForwardBE = 2'b00;
    if (Rs1E != 5'b0)
      if      ((Rs1E == RdM) & RegWriteM) ForwardAE = 2'b10;
      else if ((Rs1E == RdW) & RegWriteW) ForwardAE = 2'b01;

    if (Rs2E != 5'b0)
      if      ((Rs2E == RdM) & RegWriteM) ForwardBE = 2'b10;
      else if ((Rs2E == RdW) & RegWriteW) ForwardBE = 2'b01;
  end

  // stalls and flushes
  assign lwStallD = ResultSrcEb0 & ((Rs1D == RdE) | (Rs2D == RdE));
  assign StallD = lwStallD;
  assign StallF = lwStallD;
  assign FlushD = PCSrcE;
  assign FlushE = lwStallD | PCSrcE;
endmodule

module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [ 4:0] a1, a2, a3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally (A1/RD1, A2/RD2)
  // write third port on rising edge of clock (A3/WD3/WE3)
  // write occurs on falling edge of clock
  // register 0 hardwired to 0

  always_ff @(negedge clk)
    if (we3) rf[a3] <= wd3;

  assign rd1 = (a1 != 0) ? rf[a1] : 0;
  assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule
```

```
module adder(input  [31:0] a, b,
             output [31:0] y);

  assign y = a + b;
endmodule

module extend(input  logic [31:7] instr,
              input  logic [1:0]  immsrc,
              output logic [31:0] immext);

  always_comb
    case(immsrc)
               // I-type
      2'b00:   immext = {{20{instr[31]}}, instr[31:20]};
               // S-type (stores)
      2'b01:   immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
               // B-type (branches)
      2'b10:   immext = {{20{instr[31]}}, instr[7], instr[30:25],
instr[11:8], 1'b0};
               // J-type (jal)
      2'b11:   immext = {{12{instr[31]}}, instr[19:12], instr[20],
instr[30:21], 1'b0};
      default: immext = 32'bx; // undefined
    endcase
endmodule

module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule

module flopenr #(parameter WIDTH = 8)
                (input  logic             clk, reset, en,
                 input  logic [WIDTH-1:0] d,
                 output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en) q <= d;
endmodule

module flopenrc #(parameter WIDTH = 8)
                 (input  logic             clk, reset, clear, en,
                  input  logic [WIDTH-1:0] d,
                  output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en)
      if (clear) q <= 0;
      else       q <= d;
endmodule
```

```
module floprc #(parameter WIDTH = 8)
               (input  logic clk,
                input  logic reset,
                input  logic clear,
                input  logic [WIDTH-1:0] d,
                output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else
      if (clear) q <= 0;
      else       q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);

  assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module imem(input  logic [31:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
      $readmemh("riscvtest.txt",RAM);

  assign rd = RAM[a[31:2]]; // word aligned
endmodule

module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule

module alu(input  logic [31:0] a, b,
           input  logic [2:0]  alucontrol,
           output logic [31:0] result,
```

```
          output logic        zero);

  logic [31:0] condinvb, sum;
  logic        v;             // overflow
  logic        isAddSub;      // true when is add or subtract operation

  assign condinvb = alucontrol[0] ? ~b : b;
  assign sum = a + condinvb + alucontrol[0];
  assign isAddSub = ~alucontrol[2] & ~alucontrol[1] |
                    ~alucontrol[1] &  alucontrol[0];

  always_comb
    case (alucontrol)
      3'b000:  result = sum;         // add
      3'b001:  result = sum;         // subtract
      3'b010:  result = a & b;       // and
      3'b011:  result = a | b;       // or
      3'b100:  result = a ^ b;       // xor
      3'b101:  result = sum[31] ^ v; // slt
      3'b110:  result = a << b[4:0]; // sll
      3'b111:  result = a >> b[4:0]; // srl
      default: result = 32'bx;
    endcase

  assign zero = (result == 32'b0);
  assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;
endmodule
```

Exercise 7.26

RISC-V multicycle processor

Enhanced to support all instructions from **Exercise 7.14**:

```
lui, sra, lbu, blt, bltu, bge, bgeu, jalr, auipc, sb, slli, srai
```

Also includes added instructions from Exercise 7.13:

```
xor, sll, srl, bne
```

```
module testbench();

  logic        clk;
  logic        reset;

  logic [31:0] WriteData, DataAdr;
  logic        MemWrite;

  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
```

```
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check results
  always @(negedge clk)
    begin
      // DataAdr = 0xBC & WriteData = 0x797780BC
      if((MemWrite==1'b1) & (DataAdr === 188) & (WriteData === 2037874876))
begin
          $display("Simulation succeeded");
          $stop;
      end
    end
endmodule

module top(input  logic        clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic        MemWrite);

  logic [31:0] ReadData;

  // instantiate processor and memories
  riscvmulti rvmulti(clk, reset, MemWrite, DataAdr,
                     WriteData, ReadData);
  mem mem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule

module riscvmulti(input  logic        clk, reset,
                  output logic        MemWrite,
                  output logic [31:0] Adr, WriteData,
                  input  logic [31:0] ReadData);

  logic        RegWrite, jump;
  logic [1:0]  ResultSrc;
  logic [2:0]  ImmSrc;      // expand to 3-bits for lui and auipc
  logic [3:0]  ALUControl;
  logic           PCWrite;
  logic           IRWrite;
  logic [1:0]  ALUSrcA;
  logic [1:0]  ALUSrcB;
  logic                AdrSrc;
  logic [3:0]  Flags;  // added for other branches
  logic [6:0]  op;
  logic [2:0]  funct3;
  logic        funct7b5;
  logic [1:0]  LoadType;    // added for lbu
  logic        StoreType;   // added for sb
  logic        PCTargetSrc; // added for jalr

  controller c(clk, reset, op, funct3, funct7b5, Flags,
               ImmSrc, ALUSrcA, ALUSrcB,
               ResultSrc, AdrSrc, ALUControl,
               IRWrite, PCWrite, RegWrite, MemWrite,
               LoadType, StoreType,                   // lbu, sb
               PCTargetSrc);                          // jalr
```

```
  datapath   dp(clk, reset,
                ImmSrc, ALUSrcA, ALUSrcB,
                ResultSrc, AdrSrc, IRWrite, PCWrite,
                RegWrite, MemWrite, ALUControl,
                LoadType, StoreType, PCTargetSrc,
                op, funct3,
                funct7b5, Flags, Adr, ReadData, WriteData);
endmodule

module controller(input  logic       clk,
                  input  logic       reset,
                  input  logic [6:0] op,
                  input  logic [2:0] funct3,
                  input  logic       funct7b5,
                  input  logic [3:0] Flags,
                  output logic [2:0] ImmSrc,
                  output logic [1:0] ALUSrcA, ALUSrcB,
                  output logic [1:0] ResultSrc,
                  output logic       AdrSrc,
                  output logic [3:0] ALUControl,
                  output logic       IRWrite, PCWrite,
                  output logic       RegWrite, MemWrite,
                  output logic [1:0] LoadType,     // lbu
                  output logic       StoreType,    // sb
                  output logic       PCTargetSrc); // jalr

  logic [1:0] ALUOp;
  logic       Branch, PCUpdate;
  logic       branchtaken; // added for other branches

  // Main FSM
  mainfsm fsm(clk, reset, op,
              ALUSrcA, ALUSrcB, ResultSrc, AdrSrc,
              IRWrite, PCUpdate, RegWrite, MemWrite,
              ALUOp, Branch);

  // ALU Decoder
  aludec  ad(op[5], funct3, funct7b5, ALUOp, ALUControl);

  // Instruction Decoder
  instrdec id(op, ImmSrc);

  // Branch logic
  lsu     lsu(funct3, LoadType, StoreType);
  bu      branchunit(Branch, Flags, funct3, branchtaken); // added for bne,
blt, etc.

  assign PCWrite = branchtaken | PCUpdate;

endmodule

module mainfsm(input  logic         clk,
               input  logic         reset,
               input  logic [6:0]   op,
               output logic [1:0]   ALUSrcA, ALUSrcB,
               output logic [1:0]   ResultSrc,
```

```
              output logic         AdrSrc,
              output logic         IRWrite, PCUpdate,
              output logic         RegWrite, MemWrite,
              output logic [1:0]   ALUOp,
              output logic         Branch);

  typedef enum logic [3:0] {FETCH, DECODE, MEMADR, MEMREAD, MEMWB, MEMWRITE,
                            EXECUTER, EXECUTEI, ALUWB,
                            BEQ, JAL, LUI, JALR, JALRWB, AUIPC, UNKNOWN}
statetype;

  statetype state, nextstate;
  logic [14:0] controls;

  // state register
  always @(posedge clk or posedge reset)
    if (reset) state <= FETCH;
    else state <= nextstate;

  // next state logic
  always_comb
    case(state)
      FETCH:                    nextstate = DECODE;
      DECODE: casez(op)
                7'b0?00011:     nextstate = MEMADR;    // lw or sw
                7'b0110011:     nextstate = EXECUTER;  // R-type
                7'b0010011:     nextstate = EXECUTEI;  // addi
                7'b1100011:     nextstate = BEQ;       // beq
                7'b1101111:     nextstate = JAL;       // jal
                7'b0110111:     nextstate = LUI;       // lui
                7'b1100111:     nextstate = JALR;      // jalr
                7'b0010111:     nextstate = AUIPC;     // auipc
                default:        nextstate = UNKNOWN;
              endcase
      MEMADR:
        if (op[5])              nextstate = MEMWRITE;  // sw
        else                    nextstate = MEMREAD;   // lw
      MEMREAD:                  nextstate = MEMWB;
      EXECUTER:                 nextstate = ALUWB;
      EXECUTEI:                 nextstate = ALUWB;
      JAL:                      nextstate = ALUWB;
      LUI:                      nextstate = ALUWB;
      JALR:                     nextstate = JALRWB;
      JALRWB:                   nextstate = FETCH;
      AUIPC:                    nextstate = ALUWB;
      default:                  nextstate = FETCH;
    endcase

  // state-dependent output logic
  always_comb
    case(state)
      FETCH:        controls = 15'b00_10_10_0_1100_00_0;
      DECODE:     controls = 15'b01_01_00_0_0000_00_0;
      MEMADR:     controls = 15'b10_01_00_0_0000_00_0;
      MEMREAD:  controls = 15'b00_00_00_1_0000_00_0;
      MEMWRITE: controls = 15'b00_00_00_1_0001_00_0;
      MEMWB:      controls = 15'b00_00_01_0_0010_00_0;
```

```
      EXECUTER:   controls = 15'b10_00_00_0_0000_10_0;
      EXECUTEI: controls = 15'b10_01_00_0_0000_10_0;
      ALUWB:    controls = 15'b00_00_00_0_0010_00_0;
      BEQ:           controls = 15'b10_00_00_0_0000_01_1;
      JAL:           controls = 15'b01_10_00_0_0100_00_0;
      LUI:      controls = 15'b11_01_00_0_0000_00_0;
      JALR:         controls = 15'b10_01_10_0_0100_00_0;
      JALRWB:   controls = 15'b01_10_10_0_0010_00_0;
      AUIPC:    controls = 15'b01_01_00_0_0000_00_0;
      default:    controls = 15'bxx_xx_xx_x_xxxx_xx_x;
    endcase

  assign {ALUSrcA, ALUSrcB, ResultSrc, AdrSrc, IRWrite, PCUpdate,
  RegWrite, MemWrite, ALUOp, Branch} = controls;

endmodule

module aludec(input  logic       opb5,
              input  logic [2:0] funct3,
              input  logic       funct7b5,
              input  logic [1:0] ALUOp,
              output logic [3:0] ALUControl);   // expand to 4 bits for sra

  logic  RtypeSub;
  assign RtypeSub = funct7b5 & opb5;  // TRUE for R-type subtract instruction

  always_comb
    case(ALUOp)
      2'b00:                ALUControl = 4'b000; // addition
      2'b01:                ALUControl = 4'b001; // subtraction
      default: case(funct3) // R-type or I-type ALU
                 3'b000:  if (RtypeSub)
                            ALUControl = 4'b0001; // sub
                          else
                            ALUControl = 4'b0000; // add, addi
                 3'b001:    ALUControl = 4'b0110; // sll, slli
                 3'b010:    ALUControl = 4'b0101; // slt, slti
                 3'b100:    ALUControl = 4'b0100; // xor, xori
                 3'b101:  if (funct7b5)
                            ALUControl = 4'b1000; // sra, srai
                          else
                            ALUControl = 4'b0111; // srl, srli
                 3'b110:    ALUControl = 4'b0011; // or, ori
                 3'b111:    ALUControl = 4'b0010; // and, andi
                 default:   ALUControl = 4'bxxx; // ???
               endcase
    endcase
endmodule

module instrdec (input  logic [6:0] op,
                 output logic [2:0] ImmSrc);
  always_comb
    case(op)
      7'b0110011: ImmSrc = 3'bxxx; // R-type
      7'b0010011: ImmSrc = 3'b000; // I-type ALU
      7'b0000011: ImmSrc = 3'b000; // lw / lbu
      7'b0100011: ImmSrc = 3'b001; // sw / sb
```

```
      7'b1100011: ImmSrc = 3'b010; // branches
      7'b1101111: ImmSrc = 3'b011; // jal
      7'b0110111: ImmSrc = 3'b100; // lui
      7'b1100111: ImmSrc = 3'b000; // jalr
      7'b0010111: ImmSrc = 3'b100; // auipc
      default:    ImmSrc = 3'bxxx; // ???
    endcase
endmodule

module datapath(input  logic        clk, reset,
                input  logic [2:0]  ImmSrc,
                input  logic [1:0]  ALUSrcA, ALUSrcB,
                input  logic [1:0]  ResultSrc,
                                 input  logic        AdrSrc,
                input  logic        IRWrite, PCWrite,
                input  logic        RegWrite, MemWrite,
                input  logic [3:0]  alucontrol,
                input  logic [1:0]  LoadType,  // lb, lbu
                input  logic        StoreType, // sb
                input  logic        PCTargetSrc,
                output logic [6:0]  op,
                output logic [2:0]  funct3,
                output logic        funct7b5,
                output logic [3:0]  Flags,
                output logic [31:0] Adr,
                input  logic [31:0] ReadData,
                output logic [31:0] WriteData);

  logic [31:0] PC, OldPC, Instr, immext, ALUResult;
  logic [31:0] SrcA, SrcB, RD1, RD2, A, B;
  logic [31:0] Result, Data, ALUOut;
  logic [31:0] ZeroExtendByte, ReadDataOut;          // added for lbu
  logic [31:0] SignExtendByte;                       // added for lbu
  logic [7:0]  byteout;                              // added for lbu


  // next PC logic
  flopenr #(32) pcreg(clk, reset, PCWrite, Result, PC);
  flopenr #(32) oldpcreg(clk, reset, IRWrite, PC, OldPC);

  // memory logic
  mux2    #(32) adrmux(PC, Result, AdrSrc, Adr);
  flopenr #(32) ir(clk, reset, IRWrite, ReadData, Instr);
  flopr   #(32) datareg(clk, reset, ReadDataOut, Data);

  // register file logic
  regfile       rf(clk, RegWrite, Instr[19:15], Instr[24:20],
                   Instr[11:7], Result, RD1, RD2);
  extend        ext(Instr[31:7], ImmSrc, immext);
  flopr  #(32)  srcareg(clk, reset, RD1, A);
  flopr  #(32)  wdreg(clk, reset, RD2, B);

  // ALU logic
  mux4   #(32)  srcamux(PC, OldPC, A, 32'b0, ALUSrcA, SrcA);  // lui
  mux3   #(32)  srcbmux(B, immext, 32'd4, ALUSrcB, SrcB);
  alu           alu(SrcA, SrcB, alucontrol, ALUResult, Flags);
  flopr  #(32)  aluoutreg(clk, reset, ALUResult, ALUOut);
```

```
  mux3   #(32)  resmux(ALUOut, Data, ALUResult, ResultSrc, Result);

  // ReadData logic - added for lbu
  mux4 #(8)   bytesel(ReadData[7:0], ReadData[15:8], ReadData[23:16],
ReadData[31:24], Adr[1:0], byteout);
  zeroextend  ze(byteout, ZeroExtendByte);
  signextend  se(byteout, SignExtendByte);
  mux3 #(32)  readdatamux(ReadData, ZeroExtendByte, SignExtendByte, LoadType,
ReadDataOut);

  // WriteData logic - added for sb
  wdunit      wdu(B, ReadData, StoreType, Adr[1:0], WriteData);

  // outputs to control unit
  assign op       = Instr[6:0];
  assign funct3   = Instr[14:12];
  assign funct7b5 = Instr[30];

endmodule


module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [ 4:0] a1, a2, a3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally (A1/RD1, A2/RD2)
  // write third port on rising edge of clock (A3/WD3/WE3)
  // register 0 hardwired to 0

  always_ff @(posedge clk)
    if (we3) rf[a3] <= wd3;

  assign rd1 = (a1 != 0) ? rf[a1] : 0;
  assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule

module adder(input  [31:0] a, b,
             output [31:0] y);

  assign y = a + b;
endmodule

module extend(input  logic [31:7] instr,
              input  logic [2:0]  immsrc,  // extended to 3 bits for lui
              output logic [31:0] immext);

  always_comb
    case(immsrc)
               // I-type
      3'b000:   immext = {{20{instr[31]}}, instr[31:20]};
               // S-type (stores)
      3'b001:   immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
```

```
               // B-type (branches)
      3'b010:   immext = {{20{instr[31]}}, instr[7], instr[30:25],
instr[11:8], 1'b0};
               // J-type (jal)
      3'b011:   immext = {{12{instr[31]}}, instr[19:12], instr[20],
instr[30:21], 1'b0};
               // U-type (lui, auipc)
      3'b100:   immext = {instr[31:12], 12'b0};
      default:  immext = 32'bx; // undefined
    endcase
endmodule

// zeroextend module added for lbu
module zeroextend(input logic  [7:0]  a,
                  output logic [31:0] zeroext);

  assign zeroext = {24'b0, a};
endmodule

// signextend module added for lb
module signextend(input logic  [7:0]  a,
                  output logic [31:0] signext);

  assign signext = {{24{a[7]}}, a};
endmodule

// WriteData Unit (wdunit) - added for sb
module wdunit(input  logic [31:0] rd2,
              input  logic [31:0] readdata,
              input  logic        StoreType,
              input  logic [1:0]  byteoffset,
              output logic [31:0] WriteData);
  logic [31:0] storeb0, storeb1, storeb2, storeb3, sbword;

  assign storeb0 = {readdata[31:8],  rd2[7:0]};
  assign storeb1 = {readdata[31:16], rd2[7:0], readdata[7:0]};
  assign storeb2 = {readdata[31:24], rd2[7:0], readdata[15:0]};
  assign storeb3 = {rd2[7:0], readdata[23:0]};

  mux4 #(32) sbmux(storeb0, storeb1, storeb2, storeb3, byteoffset, sbword);
  mux2 #(32) wdmux(rd2, sbword, StoreType, WriteData);
endmodule


module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule

module flopenr #(parameter WIDTH = 8)
                (input  logic             clk, reset, en,
                 input  logic [WIDTH-1:0] d,
```

```
                output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en) q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);

  assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module mux4 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2, d3,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);

  assign y = s[1] ? (s[0] ? d3: d2) : (s[0] ? d1 : d0);
endmodule

module mem(input  logic        clk, we,
           input  logic [31:0] a, wd,
           output logic [31:0] rd);

  logic [31:0] RAM[127:0];          // increased size of memory

  initial
      $readmemh("example.txt",RAM);

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule

module alu(input  logic [31:0] a,
           input  logic [31:0] b,
           input  logic [3:0]  alucontrol,  // expanded to 4 bits for sra
           output logic [31:0] result,
           output logic [3:0]  flags);      // added for blt and other
branches

  logic [31:0] condinvb, sum;
  logic        v, c, n, z;     // flags: overflow, carry out, negative, zero
  logic        cout;           // carry out of adder
  logic        isAddSub;       // true if is an add or subtract
```

```
  assign flags = {v, c, n, z};
  assign condinvb = alucontrol[0] ? ~b : b;
  assign {cout, sum} = a + condinvb + alucontrol[0];
  assign isAddSub = (~alucontrol[3] & ~alucontrol[2] & ~alucontrol[1]) |
                    (~alucontrol[3] & ~alucontrol[1] &  alucontrol[0]);

  always_comb
    case (alucontrol)
      4'b0000:  result = sum;                   // add
      4'b0001:  result = sum;                   // subtract
      4'b0010:  result = a & b;                 // and
      4'b0011:  result = a | b;                 // or
      4'b0100:  result = a ^ b;                 // xor
      4'b0101:  result = sum[31] ^ v;           // slt
      4'b0110:  result = a << b[4:0];           // sll
      4'b0111:  result = a >> b[4:0];           // srl
      4'b1000:  result = $signed(a) >>> b[4:0]; // sra
      default:  result = 32'bx;
    endcase

  // added for blt and other branches
  assign z = (result == 32'b0);
  assign n = result[31];
  assign c = cout & isAddSub;
  assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;
endmodule

// Load/store Unit (lsu) added for lbu
module lsu(input  logic [2:0] funct3,
           output logic [1:0] LoadType,
           output logic       StoreType);
  always_comb
    case(funct3)
      3'b000:     {LoadType, StoreType} = {2'b10, 1'b1};
      3'b010:     {LoadType, StoreType} = {2'b00, 1'b0};
      3'b100:     {LoadType, StoreType} = {2'b01, 1'bx};
      default:    {LoadType, StoreType} =  3'bxxx;
    endcase
endmodule

// Branch Unit (bu) added for bne, blt, bltu, bge, bgeu
module bu(input  logic       Branch,
          input  logic [3:0] Flags,
          input  logic [2:0] funct3,
          output logic       taken);
  logic  v, c, n, z;    // Flags: overflow, carry out, negative, zero
  logic  cond;          // cond is 1 when condition for branch met
  assign {v, c, n, z} = Flags;
  assign taken = cond & Branch;

  always_comb
    case (funct3)
      3'b000:  cond = z;        // beq
      3'b001:  cond = ~z;       // bne
      3'b100:  cond = (n ^ v);  // blt
      3'b101:  cond = ~(n ^ v); // bge
```

```
      3'b110:  cond = ~c;       // bltu
      3'b111:  cond = c;        // bgeu
      default: cond = 1'b0;
    endcase
endmodule
```

Test Program:
Use the same test program as shown in the solutions for Exercise 7.10.

Exercise 7.27

RISC-V multicycle processor

Enhanced to support all instructions from **Exercise 7.13**:

```
      xor, sll, srl, bne

module testbench();

  logic        clk;
  logic        reset;

  logic [31:0] WriteData, DataAdr;
  logic        MemWrite;

  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check results
  always @(negedge clk)
    begin
      if(MemWrite) begin
        if(DataAdr === 216 & WriteData === 4140) begin
          $display("Simulation succeeded");
          $stop;
        end
      end
    end
endmodule

module top(input  logic        clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic        MemWrite);

  logic [31:0] ReadData;

  // instantiate processor and memories
  riscvmulti rvmulti(clk, reset, MemWrite, DataAdr,
                     WriteData, ReadData);
  mem mem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule

module riscvmulti(input  logic        clk, reset,
                  output logic        MemWrite,
                  output logic [31:0] Adr, WriteData,
```

```
                  input  logic [31:0] ReadData);

  logic        RegWrite, jump;
  logic [1:0]  ResultSrc;
  logic [2:0]  ImmSrc;      // expand to 3-bits for lui and auipc
  logic [3:0]  ALUControl;
  logic          PCWrite;
  logic          IRWrite;
  logic [1:0]  ALUSrcA;
  logic [1:0]  ALUSrcB;
  logic               AdrSrc;
  logic [3:0]  Flags;  // added for other branches
  logic [6:0]  op;
  logic [2:0]  funct3;
  logic        funct7b5;
  logic        LoadType;    // added for lbu
  logic        StoreType;   // added for sb
  logic        PCTargetSrc; // added for jalr

  controller c(clk, reset, op, funct3, funct7b5, Flags,
               ImmSrc, ALUSrcA, ALUSrcB,
               ResultSrc, AdrSrc, ALUControl,
               IRWrite, PCWrite, RegWrite, MemWrite,
               LoadType, StoreType,                  // lbu, sb
               PCTargetSrc);                         // jalr

  datapath   dp(clk, reset,
                ImmSrc, ALUSrcA, ALUSrcB,
                ResultSrc, AdrSrc, IRWrite, PCWrite,
                RegWrite, MemWrite, ALUControl,
                LoadType, StoreType, PCTargetSrc,
                op, funct3,
                funct7b5, Flags, Adr, ReadData, WriteData);
endmodule

module controller(input  logic       clk,
                  input  logic       reset,
                  input  logic [6:0] op,
                  input  logic [2:0] funct3,
                  input  logic       funct7b5,
                  input  logic [3:0] Flags,
                  output logic [2:0] ImmSrc,
                  output logic [1:0] ALUSrcA, ALUSrcB,
                  output logic [1:0] ResultSrc,
                  output logic       AdrSrc,
                  output logic [3:0] ALUControl,
                  output logic       IRWrite, PCWrite,
                  output logic       RegWrite, MemWrite,
                  output logic       LoadType,     // lbu
                  output logic       StoreType,    // sb
                  output logic       PCTargetSrc); // jalr

  logic [1:0] ALUOp;
  logic       Branch, PCUpdate;
  logic       branchtaken; // added for other branches

  // Main FSM
```

```
  mainfsm fsm(clk, reset, op,
              ALUSrcA, ALUSrcB, ResultSrc, AdrSrc,
              IRWrite, PCUpdate, RegWrite, MemWrite,
              ALUOp, Branch);

  // ALU Decoder
  aludec  ad(op[5], funct3, funct7b5, ALUOp, ALUControl);

  // Instruction Decoder
  instrdec id(op, ImmSrc);

  // Branch logic
  lsu     lsu(funct3, LoadType, StoreType);
  bu      branchunit(Branch, Flags, funct3, branchtaken); // added for bne,
blt, etc.

  assign PCWrite = branchtaken | PCUpdate;

endmodule

module mainfsm(input  logic         clk,
               input  logic         reset,
               input  logic [6:0]   op,
               output logic [1:0]   ALUSrcA, ALUSrcB,
               output logic [1:0]   ResultSrc,
               output logic         AdrSrc,
               output logic         IRWrite, PCUpdate,
               output logic         RegWrite, MemWrite,
               output logic [1:0]   ALUOp,
               output logic         Branch);

  typedef enum logic [3:0] {FETCH, DECODE, MEMADR, MEMREAD, MEMWB, MEMWRITE,
                            EXECUTER, EXECUTEI, ALUWB,
                            BEQ, JAL, UNKNOWN} statetype;

  statetype state, nextstate;
  logic [14:0] controls;

  // state register
  always @(posedge clk or posedge reset)
    if (reset) state <= FETCH;
    else state <= nextstate;

  // next state logic
  always_comb
    case(state)
      FETCH:                    nextstate = DECODE;
      DECODE: casez(op)
                7'b0?00011:     nextstate = MEMADR;    // lw or sw
                7'b0110011:     nextstate = EXECUTER;  // R-type
                7'b0010011:     nextstate = EXECUTEI;  // addi
                7'b1100011:     nextstate = BEQ;       // beq
                7'b1101111:     nextstate = JAL;       // jal
                default:        nextstate = UNKNOWN;
              endcase
      MEMADR:
        if (op[5])              nextstate = MEMWRITE;  // sw
```

```
        else                    nextstate = MEMREAD;   // lw
      MEMREAD:                  nextstate = MEMWB;
      EXECUTER:                 nextstate = ALUWB;
      EXECUTEI:                 nextstate = ALUWB;
      JAL:                      nextstate = ALUWB;
      default:                  nextstate = FETCH;
    endcase

  // state-dependent output logic
  always_comb
    case(state)
      FETCH:        controls = 15'b00_10_10_0_1100_00_0;
      DECODE:     controls = 15'b01_01_00_0_0000_00_0;
      MEMADR:     controls = 15'b10_01_00_0_0000_00_0;
      MEMREAD:  controls = 15'b00_00_00_1_0000_00_0;
      MEMWRITE: controls = 15'b00_00_00_1_0001_00_0;
      MEMWB:      controls = 15'b00_00_01_0_0010_00_0;
      EXECUTER:   controls = 15'b10_00_00_0_0000_10_0;
      EXECUTEI: controls = 15'b10_01_00_0_0000_10_0;
      ALUWB:    controls = 15'b00_00_00_0_0010_00_0;
      BEQ:          controls = 15'b10_00_00_0_0000_01_1;
      JAL:          controls = 15'b01_10_00_0_0100_00_0;
      default:    controls = 15'bxx_xx_xx_x_xxxx_xx_x;
    endcase

  assign {ALUSrcA, ALUSrcB, ResultSrc, AdrSrc, IRWrite, PCUpdate,
  RegWrite, MemWrite, ALUOp, Branch} = controls;

endmodule

module aludec(input  logic       opb5,
              input  logic [2:0] funct3,
              input  logic       funct7b5,
              input  logic [1:0] ALUOp,
              output logic [3:0] ALUControl);   // expand to 4 bits for sra

  logic  RtypeSub;
  assign RtypeSub = funct7b5 & opb5;  // TRUE for R-type subtract instruction

  always_comb
    case(ALUOp)
      2'b00:                ALUControl = 4'b000; // addition
      2'b01:                ALUControl = 4'b001; // subtraction
      default: case(funct3) // R-type or I-type ALU
                 3'b000:  if (RtypeSub)
                            ALUControl = 4'b0001; // sub
                          else
                            ALUControl = 4'b0000; // add, addi
                 3'b001:    ALUControl = 4'b0110; // sll, slli
                 3'b010:    ALUControl = 4'b0101; // slt, slti
                 3'b100:    ALUControl = 4'b0100; // xor, xori
                 3'b101:  if (funct7b5)
                            ALUControl = 4'b1000; // sra, srai
                          else
                            ALUControl = 4'b0111; // srl, srli
                 3'b110:    ALUControl = 4'b0011; // or, ori
                 3'b111:    ALUControl = 4'b0010; // and, andi
```

```
                default:   ALUControl = 4'bxxx; // ???
              endcase
    endcase
endmodule

module instrdec (input  logic [6:0] op,
                 output logic [2:0] ImmSrc);
  always_comb
    case(op)
      7'b0110011: ImmSrc = 3'bxxx; // R-type
      7'b0010011: ImmSrc = 3'b000; // I-type ALU
      7'b0000011: ImmSrc = 3'b000; // lw / lbu
      7'b0100011: ImmSrc = 3'b001; // sw / sb
      7'b1100011: ImmSrc = 3'b010; // branches
      7'b1101111: ImmSrc = 3'b011; // jal
      7'b0110111: ImmSrc = 3'b100; // lui
      7'b1100111: ImmSrc = 3'b000; // jalr
      7'b0010111: ImmSrc = 3'b100; // auipc
      default:    ImmSrc = 3'bxxx; // ???
    endcase
endmodule

module datapath(input  logic        clk, reset,
                input  logic [2:0]  ImmSrc,
                input  logic [1:0]  ALUSrcA, ALUSrcB,
                input  logic [1:0]  ResultSrc,
                                    input  logic        AdrSrc,
                input  logic        IRWrite, PCWrite,
                input  logic        RegWrite, MemWrite,
                input  logic [3:0]  alucontrol,
                input  logic        LoadType, StoreType, // lbu, sb
                input  logic        PCTargetSrc,
                output logic [6:0]  op,
                output logic [2:0]  funct3,
                output logic        funct7b5,
                output logic [3:0]  Flags,
                output logic [31:0] Adr,
                input  logic [31:0] ReadData,
                output logic [31:0] WriteData);

  logic [31:0] PC, OldPC, Instr, immext, ALUResult;
  logic [31:0] SrcA, SrcB, RD1, RD2, A;
  logic [31:0] Result, Data, ALUOut;


  // next PC logic
  flopenr #(32) pcreg(clk, reset, PCWrite, Result, PC);
  flopenr #(32) oldpcreg(clk, reset, IRWrite, PC, OldPC);

  // memory logic
  mux2    #(32) adrmux(PC, Result, AdrSrc, Adr);
  flopenr #(32) ir(clk, reset, IRWrite, ReadData, Instr);
  flopr   #(32) datareg(clk, reset, ReadData, Data);

  // register file logic
  regfile       rf(clk, RegWrite, Instr[19:15], Instr[24:20],
                   Instr[11:7], Result, RD1, RD2);
```

```
  extend        ext(Instr[31:7], ImmSrc, immext);
  flopr  #(32)  srcareg(clk, reset, RD1, A);
  flopr  #(32)  wdreg(clk, reset, RD2, WriteData);

  // ALU logic
  mux3   #(32)  srcamux(PC, OldPC, A, ALUSrcA, SrcA);
  mux3   #(32)  srcbmux(WriteData, immext, 32'd4, ALUSrcB, SrcB);
  alu           alu(SrcA, SrcB, alucontrol, ALUResult, Flags);
  flopr  #(32)  aluoutreg(clk, reset, ALUResult, ALUOut);
  mux3   #(32)  resmux(ALUOut, Data, ALUResult, ResultSrc, Result);

  // outputs to control unit
  assign op       = Instr[6:0];
  assign funct3   = Instr[14:12];
  assign funct7b5 = Instr[30];

endmodule


module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [ 4:0] a1, a2, a3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally (A1/RD1, A2/RD2)
  // write third port on rising edge of clock (A3/WD3/WE3)
  // register 0 hardwired to 0

  always_ff @(posedge clk)
    if (we3) rf[a3] <= wd3;

  assign rd1 = (a1 != 0) ? rf[a1] : 0;
  assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule

module adder(input  [31:0] a, b,
             output [31:0] y);

  assign y = a + b;
endmodule

module extend(input  logic [31:7] instr,
              input  logic [2:0]  immsrc,  // extended to 3 bits for lui
              output logic [31:0] immext);

  always_comb
    case(immsrc)
               // I-type
      3'b000:   immext = {{20{instr[31]}}, instr[31:20]};
               // S-type (stores)
      3'b001:   immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
               // B-type (branches)
      3'b010:   immext = {{20{instr[31]}}, instr[7], instr[30:25],
```

```
instr[11:8], 1'b0};
               // J-type (jal)
      3'b011:   immext = {{12{instr[31]}}, instr[19:12], instr[20],
instr[30:21], 1'b0};
               // U-type (lui, auipc)
      3'b100:   immext = {instr[31:12], 12'b0};
      default:  immext = 32'bx; // undefined
    endcase
endmodule

// zeroextend module added for lbu
module zeroextend(input logic  [7:0]  a,
                  output logic [31:0] zeroimmext);

  assign zeroimmext = {24'b0, a};
endmodule


module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule

module flopenr #(parameter WIDTH = 8)
                (input  logic             clk, reset, en,
                 input  logic [WIDTH-1:0] d,
                 output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en) q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);

  assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module mux4 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2, d3,
              input  logic [1:0]       s,
```

```
              output logic [WIDTH-1:0] y);

  assign y = s[1] ? (s[0] ? d3: d2) : (s[0] ? d1 : d0);
endmodule

module mem(input  logic        clk, we,
           input  logic [31:0] a, wd,
           output logic [31:0] rd);

  logic [31:0] RAM[127:0];          // increased size of memory

  initial
      $readmemh("example.txt",RAM);

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule

module alu(input  logic [31:0] a,
           input  logic [31:0] b,
           input  logic [3:0]  alucontrol,  // expanded to 4 bits for sra
           output logic [31:0] result,
           output logic [3:0]  flags);      // added for blt and other
branches

  logic [31:0] condinvb, sum;
  logic        v, c, n, z;     // flags: overflow, carry out, negative, zero
  logic        cout;           // carry out of adder
  logic        isAdd;          // true if is an add operation
  logic        isSub;          // true if is a subtract operation

  assign flags = {v, c, n, z};
  assign condinvb = alucontrol[0] ? ~b : b;
  assign {cout, sum} = a + condinvb + alucontrol[0];
  assign isAddSub = ~alucontrol[3] & ~alucontrol[2] & ~alucontrol[1] |
                    ~alucontrol[3] & ~alucontrol[1] &  alucontrol[0];

  always_comb
    case (alucontrol)
      4'b0000:  result = sum;                  // add
      4'b0001:  result = sum;                  // subtract
      4'b0010:  result = a & b;                // and
      4'b0011:  result = a | b;                // or
      4'b0100:  result = a ^ b;                // xor
      4'b0101:  result = sum[31] ^ v;              // slt
      4'b0110:  result = a << b[4:0];          // sll
      4'b0111:  result = a >> b[4:0];          // srl
      4'b1000:  result = $signed(a) >>> b[4:0];  // sra
      default:  result = 32'bx;
    endcase

  // added for blt and other branches
  assign z = (result == 32'b0);
  assign n = result[31];
  assign c = cout & isAddSub;
```

```
  assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;
endmodule

// Load/store Unit (lsu) added for lbu
module lsu(input  logic [2:0] funct3,
           output logic       LoadType, StoreType);
  always_comb
    case(funct3)
      3'b000:     {LoadType, StoreType} = 2'b01;
      3'b010:     {LoadType, StoreType} = 2'b00;
      3'b100:     {LoadType, StoreType} = 2'b1x;
      default:    {LoadType, StoreType} = 2'bxx;
    endcase
endmodule

// Branch Unit (bu) added for bne, blt, bltu, bge, bgeu
module bu (input  logic       Branch,
           input  logic [3:0] Flags,
           input  logic [2:0] funct3,
           output logic       taken);
  logic  v, c, n, z;    // Flags: overflow, carry out, negative, zero
  logic  cond;          // cond is 1 when condition for branch met
  assign {v, c, n, z} = Flags;
  assign taken = cond & Branch;

  always_comb
    case (funct3)
      3'b000:  cond = z;        // beq
      3'b001:  cond = ~z;       // bne
      3'b100:  cond = (n ^ v);  // blt
      3'b101:  cond = ~(n ^ v); // bge
      3'b110:  cond = ~c;       // bltu
      3'b111:  cond = c;        // bgeu
      default: cond = 1'b0;
    endcase
endmodule
```

Test Program:

Use the same test program as shown in the solutions for Exercise 7.9.

Exercise 7.28

The code executes as follows:

Cycle	Fetch	Decode	Execute	Memory	Writeback
1	addi				
2	lw	addi			
3	add	lw	addi		
4	or	add	lw	addi	
5	lw	**or**	add	lw	**addi**

In cycle 5, the `or` instruction is in the Decode stage and `addi` is in the Writeback stage. So, `s1` is written (by `addi`) during the first half of cycle 5. `s1` and `s2` are read (by `or`) during the second half of cycle 5.
Notice that `s1` is both written and read in cycle 5. Also note that no hazards exist in this code.

Exercise 7.29

The code executes as follows:

Pipeline stages

Cycle	Fetch	Decode	Execute	Memory	Writeback
1	xor				
2	addi	xor			
3	lw	addi	xor		
4	sw	lw	addi	xor	
5	or	**sw**	lw	addi	**xor**

In cycle 5, the `sw` instruction is in the Decode stage and `xor` is in the Writeback stage. So, `s1` is written (by `xor`) during the first half of cycle 5. `s1` and `s4` are read (by the `sw`) during the second half of cycle 5.
Notice that `s1` is both written and read in cycle 5. Also note that no hazards exist in this code.

Exercise 7.30

The code executes as follows:

Pipeline stages

Cycle	Fetch	Decode	Execute	Memory	Writeback
1	addi s1,x0,11				
2	lw s2,25(s1)	addi s1,x0,11			
3	lw s5,16(s2)	lw s2,25(s1)	addi s1,x0,11		
4	add s3,s2,s5	lw s5,16(s2)	lw s2,25(s1)	addi s1,x0,11	
5	**add s3,s2,s5**	**lw s5,16(s2)**	**bubble**	lw s2,25(s1)	**addi s1,x0,11**
6	or s4,s3,t4	add s3,s2,s5	lw s5,16(s2)	bubble	lw s2,25(s1)

In cycle 4, the `lw` instruction in the Decode stage detects that it needs the result of the `lw` that is in the Execute stage (and that result won't be available until the end of the Memory stage), so the second `lw` instruction (`lw s5, 16(s2)`) and the next instruction (`add s3, s2, s5`) stall – that is, remain in the Decode and Fetch stages. So, in Cycle 5 `s1` is written (by `addi`) during the first half of the cycle. `S2` is read (by the second `lw` instruction) during the second half of cycle 5.

Exercise 7.31

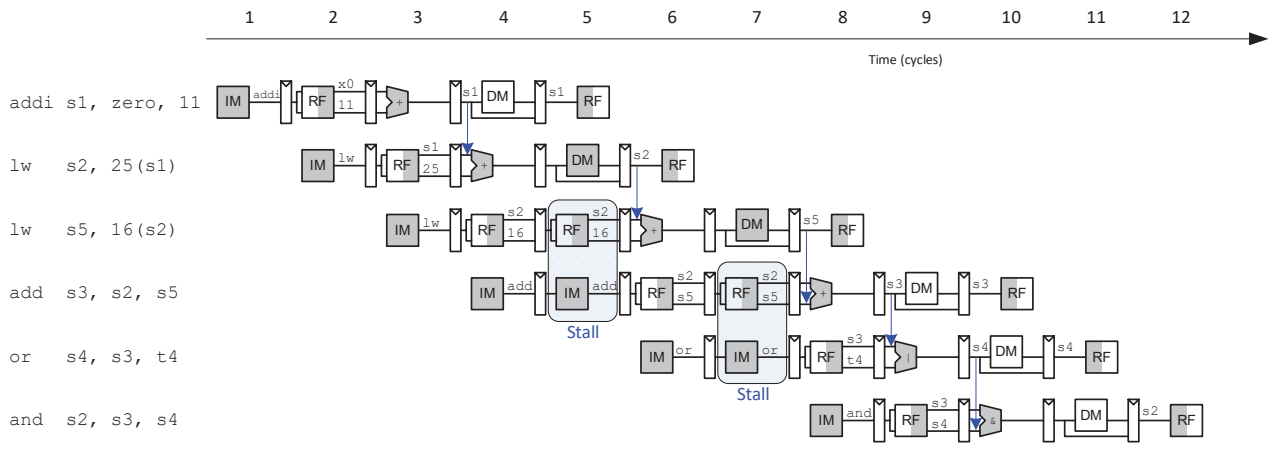
The code executes as follows:

Pipeline stages

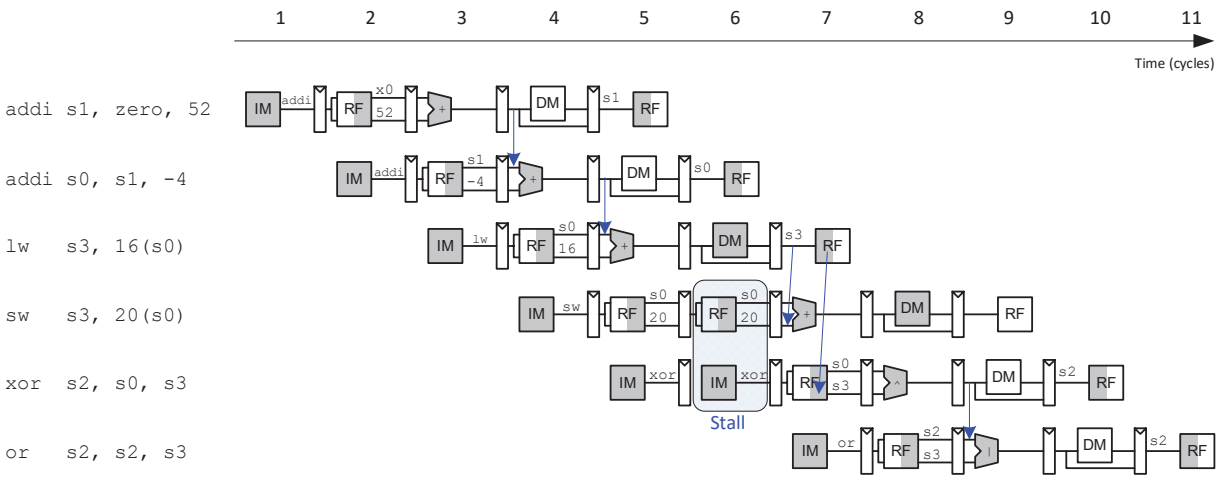
Cycle	Fetch	Decode	Execute	Memory	Writeback
1	`addi s1,x0,52`				
2	`addi s0,s1,-4`	`addi s1,x0,52`			
3	`lw s3,16(s0)`	`addi s0,s1,-4`	`addi s1,x0,52`		
4	`sw s3,20(s0)`	`lw s3,16(s0)`	`addi s0,s1,-4`	`addi s1,x0,52`	
5	`xor s2,s0,s3`	**`sw s3,20(s0)`**	`lw s3,16(s0)`	`addi s0,s1,-4`	**`addi s1,x0,52`**
6	**`xor s2,s0,s3`**	**`sw s3,20(s0)`**	**`bubble`**	`lw s3,16(s0)`	`addi s0,s1,-4`

In cycle 5, **`s1`** is being written (by `addi`) in the Decode stage, and **`s0`** and **`s3`** are being read by `sw` in the Decode stage. Note that in this cycle, `sw` detects that it needs to stall in the next cycle – because a `lw` is in the Execute stage that will produce one of its source registers (`s3`), and the `lw` won't have that operand ready until the end of the Memory stage.

Exercise 7.32



Exercise 7.33



Exercise 7.34

It takes **8 clock cycles** to issue all the instructions.
# of instructions = 6
CPI = 8 clock cycles / 6 instructions = **1.33 CPI.**

Exercise 7.35

It takes **7 clock cycles** to issue all the instructions.
# instructions = 6.
CPI = 7 clock cycles / 6 instructions = **1.17 CPI.**

Exercise 7.36

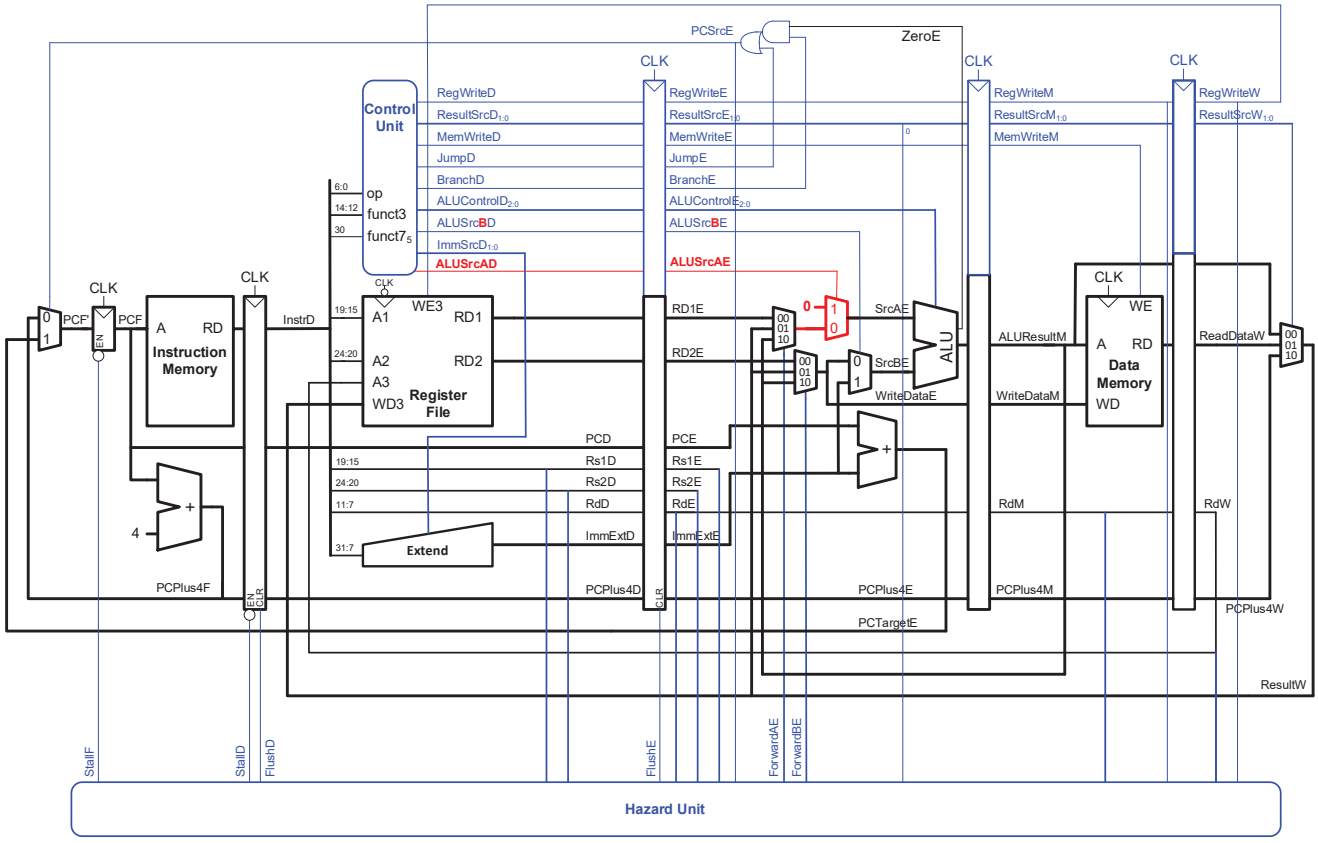
First, we update the immediate Extend unit to support `lui`.

Enhanced *ImmSrc* encoding to support `lui`

ImmSrc	*ImmExt*	Type	Description
000	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
001	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
010	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate
011	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1'b0}	J	21-bit signed immediate
100	**{*Instr*[31:12], 12'b0}**	**U**	**20-bit signed immediate**

Next, we modify the datapath by increasing the width of the *ImmSrc* control signal to 3 bits and by making 0 an option for the ALU's top input (*SrcA*).

Enhanced datapath to support `lui`



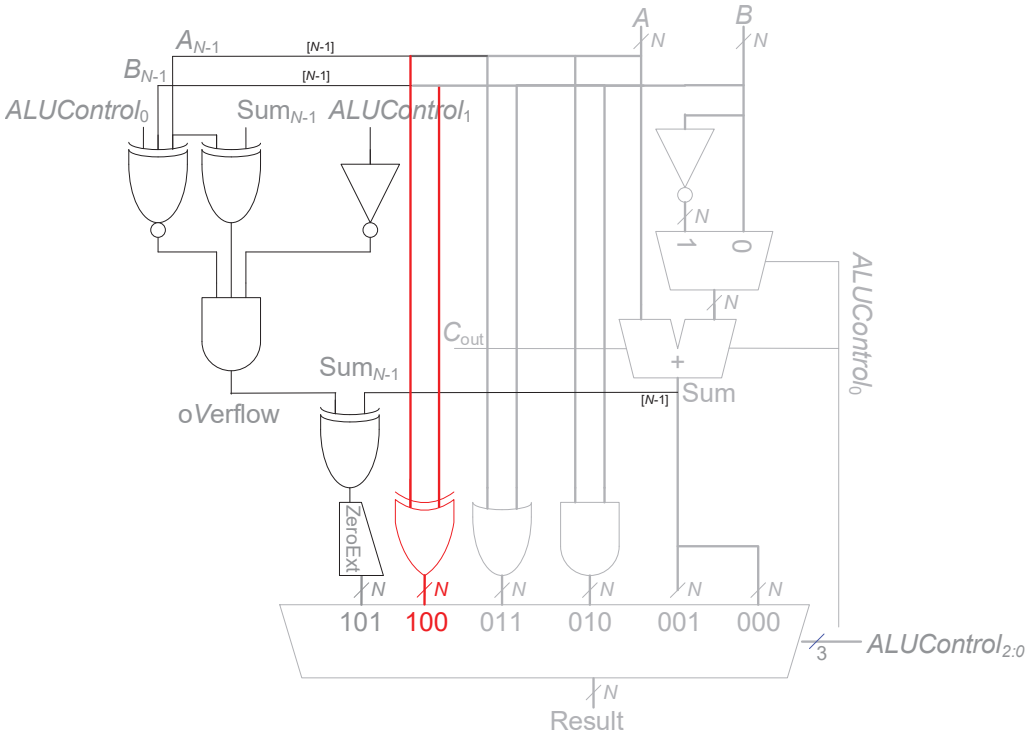
Main Decoder truth table enhanced to support `lui`

Instruction	Opcode	RegWrite	ImmSrc	ALUSrcA	ALUSrcB	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	000	0	1	0	01	0	00	0
`sw`	0100011	0	001	0	1	1	xx	0	00	0
R-type	0110011	1	xxx	0	0	0	00	0	10	0
`beq`	1100011	0	010	0	0	0	xx	1	01	0
I-type ALU	0010011	1	000	0	0	0	00	0	10	0
`jal`	1101111	1	011	x	x	0	10	0	xx	1
`lui`	**0110111**	**1**	**100**	**1**	**x**	**0**	**11**	**0**	**xx**	**0**

Exercise 7.37

The datapath already supports R-type instructions. So no changes need to be made to the datapath. Only the ALU needs to be modified: we add another input to the multiplexer and N 2-bit XOR gates within the ALU. We also update the ALU Decoder truth table / logic. The Main Decoder truth table need not be updated because it already supports R-type instructions. These changes are shown below.

Modified ALU to support `xor`



Modified ALU operations to support `xor`

$ALUControl_{2:0}$	Function
000	add
001	subtract
010	and
011	or
100	**xor**
101	SLT

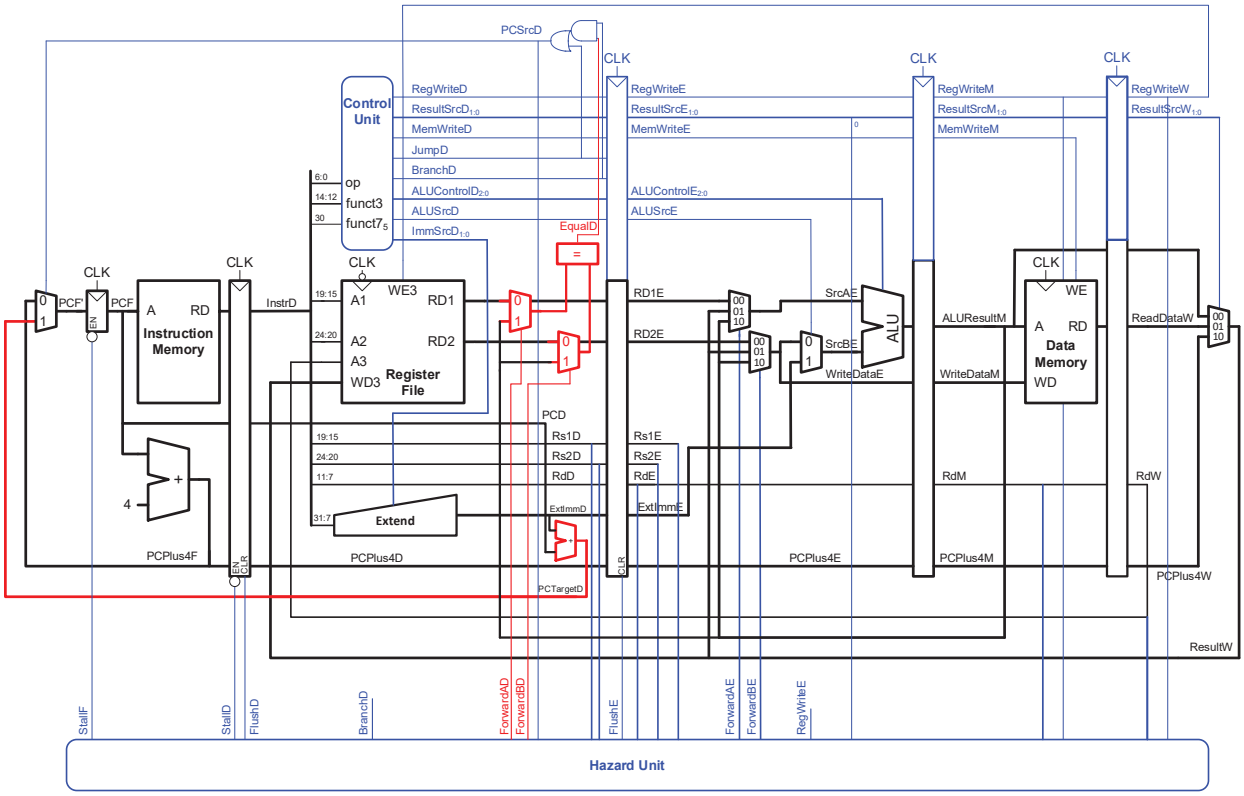
Modified ALU Decoder truth table to support `xor`

ALUOp	funct3	op_5, $funct7_5$	*ALUControl*	Instruction
00	x	x	000 (add)	`lw, sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add, addi`
	000	11	001 (subtract)	`sub`
	010	x	101 (set less than)	`slt, slti`
	100	**x**	**100 (xor)**	**`xor, xori`**
	110	x	011 (or)	`or, ori`
	111	x	010 (and)	`and, andi`

Exercise 7.38

The figure below shows the modified RISC-V pipelined processor with branch resolution moved to the Decode stage (instead of in the Execute stage).

Modified Pipelined Processor



We compare **rs1** and **rs2** using an equality comparator in the Decode stage and move the adder that calculates *PCTarget* to the Decode stage.
However, by using the operands (**rs1** and **rs2**) in the Decode stage, we also introduce additional hazards. A branch in the Decode stage cannot be determined if any instruction producing a needed operand is:

Case 1. In the Execute stage (because the operand won't be available until the end of that stage), or

Case 2. A load that is in the Memory stage (because the operand won't be available until the end of that stage).

In these two cases, the branch (and the instruction in the Fetch stage) must stall until its source operand(s) can be forwarded from the Memory stage.
However, an operand could be forwarded from the Memory stage (as long as the instruction in that stage is not a load).

Thus, we add two additional multiplexers to forward from the Memory stage when needed.

The Hazard unit generates a *branchStall* signal that is true when the branch must stall:
***branchStall* =**
BranchD AND RegWriteE AND ((RdE == Rs1D) OR (RdE == Rs2D)) // Case 1
OR

BranchD AND ResultSrcM_0 AND ((RdM == Rs1D) OR (RdM == Rs2D)) // Case 2

This stall will hold up the pipeline so that a branch instruction can have a value forwarded from the Memory stage or to wait until the register file is written during the first half of the cycle of the Writeback stage. `lwStall` covers a load instruction still in the Execute stage. This means that a load instruction now requires two cycles of stall if a branch instruction follows it.

The StallF and StallD signals require only slight updates:

StallF = StallD = lwStall OR branchStall

Because the combinational logic for *PCSrc* will have to be moved to the Decode stage, jump instructions will also be handled during the Decode stage. With this, we can eliminate the need for the BranchE and JumpE signals.

The FlushD signal will retain the same logic, but with the *PCSrcE* signal replaced with *PCSrcD*:

FlushD = PCSrcD

The FlushE requires additional logic to make sure it is asserted only when necessary:

FlushE = lwStall OR branchStall OR (PCSrcD AND BranchD)

The last part of the logic for FlushE ensures that the Execute stage isn't flushed during a jump instruction – otherwise (if Execute were flushed) PCPlus4 wouldn't be available to write to the destination register during the Writeback stage.

The Hazard Unit needs to generate two new forwarding signals, *ForwardAD* and *ForwardBD*. The multiplexers associated with these signals forward values from the Memory stage of an ALU instruction if either of the source registers for the branch instruction matches the destination register of the instruction in the Memory stage. The logic for these signals would be:

ForwardAD = (Rs1D != 0) AND (RdM == Rs1D) AND RegWriteM
ForwardBD = (Rs2D != 0) AND (RdM == Rs2D) AND RegWriteM

To calculate a branch in the Decode stage, a comparator must be added so that an *EqualD* signal is produced if the registers are equivalent. This replaces the *Zero* flag from the ALU and an AND operation is performed with *BranchD* to determine if *PCSrcD* should be asserted. Two 2:1 multiplexers select between *RD1*/*RD2* and the forwarded value of *ALUResultM* if necessary. Signals *BranchE* and *JumpE* are removed. The combinational logic for *PCSrc* also move to the Decode stage. And the adder for *PCTarget* moves to the Decode stage. *BranchD* and *RegWriteE* now also must be fed to the Hazard Unit for the stall, flush, and forwarding logic to work.

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 an average CPI of (0.6)(1) + (0.4)(2) = 1.4. Branches now only require 2 cycles when misprediction occurs, and 1 cycle otherwise. This gives an average CPI of (0.5)(1) + (0.5)(2) = 1.5 for branches. The other instructions still have a CPI of 1. So, for the benchmark given in Example 9:

Average CPI = (0.25)(1.4) + (0.1)(1) + (0.13)(1.5) + (0.52)(1) = 1.17

* It should be noted that, when branching occurs in the Decode stage, a branch instruction immediately following a load instruction that has a dependency will actually require 2 stalls (3 cycles total). This instruction mix would cause both the *lwStall* and *branchStall* signals to assert, causing the 2 stalls.

Cycle time calculation:
Both the Decode and Execute stages would now have different critical paths. The cycle time of the Decode stage's critical path would now be:

$T_c = 2(t_{RFRead} + 2t_{mux} + t_{eq} + t_{AND\text{-}OR} + t_{setup})$

The cycle time of the Execute stage's critical path would now be:

$T_c = t_{pcq} + 3t_{mux} + t_{ALU} + t_{setup}$

Because the overall cycle time of the pipelined processor corresponds to the longest stage, the Decode stage now represents the critical path overall. Using the delay times from Table 7.7 and the given delay of an equality comparator (23 ps), the new cycle time is:

$T_c = 2(100 + 2(30) + 23 + 20 + 50)$ ps = **506 ps**

The total execution time is then equal to:

$T_3 = (100\times10^9$ instructions$)(1.165$ cycles / instruction$)(506\times10^{-12}$ s / cycle$)$ = **59 seconds.**

This compares to the 43 seconds when branching takes places during the Execute stage (see Example 7.10). So, in this case, moving the branch calculation hardware earlier decreased performance.

Exercise 7.39

From **Equation 7.5**:
$T_{c3} = \max((40 + 200 + 50), \big(2(100 + 50)\big), (40 + 4(30) + 120 + 20 + 50), (40 + 200 + 50), (2(40 + 30 + 60))) = \max(290, 300, 350, 290, 260)$
The slowest stage is the Execute stage at 350ps. The next slowest stage is the Decode stage at 300ps. Thus, the execute stage should be reduced by 50 ps to make it as fast as

the next slowest stage, Decode. She should, thus reduce the **ALU** delay to: 120 - 50 = **70ps.**
The new cycle time is then **300 ps**.

Exercise 7.40

Using the delays from Table 7.7, the Execute stage represents the critical path. The cycle time of the pipelined processor before any changes to the ALU is 350 ps. This means, for the following:

If the ALU were 20% faster, the cycle time for the pipelined processor would **decrease** by **24 ps** (i.e., 120 ps × 0.2 = 24 ps). The new cycle time would then be (350 – 24) ps = **326 ps**.

If the ALU were 20% slower, the cycle time for the pipelined processor would **increase** by **24 ps**. The new cycle time would then be (350 + 24) ps = **374 ps**.

Exercise 7.41

Loads take one clock cycle when there is no dependency and seven clock cycles when there is (load plus 6 stalls), so they have an average CPI of (0.5)(1) + (0.5)(7) = 4.
Branches take one clock cycle when they are predicted properly and two when they are not, so their average CPI is (0.7)(1) + (0.3)(2) = 1.3. The remaining instructions have a CPI of 1. Hence, the average CPI for the SPECINT2000 benchmark is:
CPI = 0.25(4) + 0.10(1) + 0.13(1.3) + 0.52(1) = **1.79 CPI.**
Execution time = (100 x 10^9 instructions)(1.79 cycles/instruction)(400 x 10^{-12} s/cycle) = **71.6 seconds**

Exercise 7.42

RISC-V Pipelined Processor

```
module testbench();

  logic        clk;
  logic        reset;

  logic [31:0] WriteData, DataAdr;
  logic        MemWrite;

  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
```

```
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check results
  always @(negedge clk)
    begin
      if(MemWrite) begin
        if(DataAdr === 100 & WriteData === 25) begin
          $display("Simulation succeeded");
          $stop;
        end else if (DataAdr !== 96) begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule

module top(input  logic        clk, reset,
           output logic [31:0] WriteDataM, DataAdrM,
           output logic        MemWriteM);

  logic [31:0] PCF, InstrF, ReadDataM;

  // instantiate processor and memories
  riscv riscv(clk, reset, PCF, InstrF, MemWriteM, DataAdrM,
              WriteDataM, ReadDataM);
  imem imem(PCF, InstrF);
  dmem dmem(clk, MemWriteM, DataAdrM, WriteDataM, ReadDataM);
endmodule

module riscv(input  logic        clk, reset,
             output logic [31:0] PCF,
             input  logic [31:0] InstrF,
             output logic        MemWriteM,
             output logic [31:0] ALUResultM, WriteDataM,
             input  logic [31:0] ReadDataM);

  logic [6:0]  opD;
  logic [2:0]  funct3D;
  logic        funct7b5D;
  logic [1:0]  ImmSrcD;
  logic        ZeroE;
  logic        PCSrcE;
  logic [2:0]  ALUControlE;
  logic        ALUSrcE;
  logic        ResultSrcEb0;
  logic        RegWriteM;
  logic [1:0]  ResultSrcW;
  logic        RegWriteW;

  logic [1:0]  ForwardAE, ForwardBE;
  logic        StallF, StallD, FlushD, FlushE;

  logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW;
```

```
  controller c(clk, reset,
               opD, funct3D, funct7b5D, ImmSrcD,
               FlushE, ZeroE, PCSrcE, ALUControlE, ALUSrcE, ResultSrcEb0,
               MemWriteM, RegWriteM,
                     RegWriteW, ResultSrcW);

  datapath dp(clk, reset,
              StallF, PCF, InstrF,
                    opD, funct3D, funct7b5D, StallD, FlushD, ImmSrcD,
                    FlushE, ForwardAE, ForwardBE, PCSrcE, ALUControlE,
ALUSrcE, ZeroE,
              MemWriteM, WriteDataM, ALUResultM, ReadDataM,
              RegWriteW, ResultSrcW,
              Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW);

  hazard  hu(Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
             PCSrcE, ResultSrcEb0, RegWriteM, RegWriteW,
             ForwardAE, ForwardBE, StallF, StallD, FlushD, FlushE);

endmodule


module controller(input  logic             clk, reset,
                  // Decode stage control signals
                  input logic [6:0]  opD,
                  input logic [2:0]  funct3D,
                  input logic            funct7b5D,
                  output logic [1:0] ImmSrcD,
                  // Execute stage control signals
                  input logic            FlushE,
                  input logic            ZeroE,
                  output logic       PCSrcE,        // for datapath and
Hazard Unit
                  output logic [2:0] ALUControlE,
                  output logic           ALUSrcE,
                  output logic       ResultSrcEb0,  // for Hazard Unit
                  // Memory stage control signals
                  output logic           MemWriteM,
                  output logic       RegWriteM,     // for Hazard Unit

                  // Writeback stage control signals
                  output logic           RegWriteW,     // for datapath and
Hazard Unit
                  output logic [1:0] ResultSrcW);

  // pipelined control signals
  logic       RegWriteD, RegWriteE;
  logic [1:0] ResultSrcD, ResultSrcE, ResultSrcM;
  logic       MemWriteD, MemWriteE;
  logic             JumpD, JumpE;
  logic             BranchD, BranchE;
  logic     [1:0] ALUOpD;
  logic [2:0] ALUControlD;
  logic       ALUSrcD;

  // Decode stage logic
```

```
  maindec md(opD, ResultSrcD, MemWriteD, BranchD,
             ALUSrcD, RegWriteD, JumpD, ImmSrcD, ALUOpD);
  aludec  ad(opD[5], funct3D, funct7b5D, ALUOpD, ALUControlD);

  // Execute stage pipeline control register and logic
  floprc #(10) controlregE(clk, reset, FlushE,
                           {RegWriteD, ResultSrcD, MemWriteD, JumpD, BranchD,
ALUControlD, ALUSrcD},
                           {RegWriteE, ResultSrcE, MemWriteE, JumpE, BranchE,
ALUControlE, ALUSrcE});

  assign PCSrcE = (BranchE & ZeroE) | JumpE;
  assign ResultSrcEb0 = ResultSrcE[0];

  // Memory stage pipeline control register
  flopr #(4) controlregM(clk, reset,
                         {RegWriteE, ResultSrcE, MemWriteE},
                         {RegWriteM, ResultSrcM, MemWriteM});

  // Writeback stage pipeline control register
  flopr #(3) controlregW(clk, reset,
                         {RegWriteM, ResultSrcM},
                         {RegWriteW, ResultSrcW});
endmodule

module maindec(input  logic [6:0] op,
               output logic [1:0] ResultSrc,
               output logic       MemWrite,
               output logic       Branch, ALUSrc,
               output logic       RegWrite, Jump,
               output logic [1:0] ImmSrc,
               output logic [1:0] ALUOp);

  logic [10:0] controls;

  assign {RegWrite, ImmSrc, ALUSrc, MemWrite,
          ResultSrc, Branch, ALUOp, Jump} = controls;

  always_comb
    case(op)
    // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump
      7'b0000011: controls = 11'b1_00_1_0_01_0_00_0; // lw
      7'b0100011: controls = 11'b0_01_1_1_00_0_00_0; // sw
      7'b0110011: controls = 11'b1_xx_0_0_00_0_10_0; // R-type
      7'b1100011: controls = 11'b0_10_0_0_00_1_01_0; // beq
      7'b0010011: controls = 11'b1_00_1_0_00_0_10_0; // I-type ALU
      7'b1101111: controls = 11'b1_11_0_0_10_0_00_1; // jal
      7'b0000000: controls = 11'b0_00_0_0_00_0_00_0; // need valid values at
reset
      default:    controls = 11'bx_xx_x_x_xx_x_xx_x; // non-implemented
instruction
    endcase
endmodule

module aludec(input  logic       opb5,
              input  logic [2:0] funct3,
              input  logic       funct7b5,
```

```
              input  logic [1:0] ALUOp,
              output logic [2:0] ALUControl);

  logic  RtypeSub;
  assign RtypeSub = funct7b5 & opb5;  // TRUE for R-type subtract instruction

  always_comb
    case(ALUOp)
      2'b00:                ALUControl = 3'b000; // addition
      2'b01:                ALUControl = 3'b001; // subtraction
      default: case(funct3) // R-type or I-type ALU
                 3'b000:  if (RtypeSub)
                            ALUControl = 3'b001; // sub
                          else
                            ALUControl = 3'b000; // add, addi
                 3'b010:    ALUControl = 3'b101; // slt, slti
                 3'b110:    ALUControl = 3'b011; // or, ori
                 3'b111:    ALUControl = 3'b010; // and, andi
                 default:   ALUControl = 3'bxxx; // ???
               endcase
    endcase
endmodule

module datapath(input logic clk, reset,
                // Fetch stage signals
                input  logic        StallF,
                output logic [31:0] PCF,
                input  logic [31:0] InstrF,
                // Decode stage signals
                output logic [6:0]  opD,
                output logic [2:0]  funct3D,
                output logic        funct7b5D,
                input  logic        StallD, FlushD,
                input  logic [1:0]  ImmSrcD,
                // Execute stage signals
                input  logic        FlushE,
                input  logic [1:0]  ForwardAE, ForwardBE,
                input  logic        PCSrcE,
                input  logic [2:0]  ALUControlE,
                input  logic        ALUSrcE,
                output logic        ZeroE,
                // Memory stage signals
                input  logic        MemWriteM,
                output logic [31:0] WriteDataM, ALUResultM,
                input  logic [31:0] ReadDataM,
                // Writeback stage signals
                input  logic        RegWriteW,
                input  logic [1:0]  ResultSrcW,
                // Hazard Unit signals
                output logic [4:0]  Rs1D, Rs2D, Rs1E, Rs2E,
                output logic [4:0]  RdE, RdM, RdW);

  // Fetch stage signals
  logic [31:0] PCNextF, PCPlus4F;
  // Decode stage signals
  logic [31:0] InstrD;
  logic [31:0] PCD, PCPlus4D;
```

```
logic [31:0] RD1D, RD2D;
logic [31:0] ImmExtD;
logic [4:0]  RdD;
// Execute stage signals
logic [31:0] RD1E, RD2E;
logic [31:0] PCE, ImmExtE;
logic [31:0] SrcAE, SrcBE;
logic [31:0] ALUResultE;
logic [31:0] WriteDataE;
logic [31:0] PCPlus4E;
logic [31:0] PCTargetE;
// Memory stage signals
logic [31:0] PCPlus4M;
// Writeback stage signals
logic [31:0] ALUResultW;
logic [31:0] ReadDataW;
logic [31:0] PCPlus4W;
logic [31:0] ResultW;

// Fetch stage pipeline register and logic
mux2    #(32) pcmux(PCPlus4F, PCTargetE, PCSrcE, PCNextF);
flopenr #(32) pcreg(clk, reset, ~StallF, PCNextF, PCF);
adder         pcadd(PCF, 32'h4, PCPlus4F);

// Decode stage pipeline register and logic
flopenrc #(96) regD(clk, reset, FlushD, ~StallD,
                    {InstrF, PCF, PCPlus4F},
                    {InstrD, PCD, PCPlus4D});
assign opD       = InstrD[6:0];
assign funct3D   = InstrD[14:12];
assign funct7b5D = InstrD[30];
assign Rs1D      = InstrD[19:15];
assign Rs2D      = InstrD[24:20];
assign RdD       = InstrD[11:7];

regfile        rf(clk, RegWriteW, Rs1D, Rs2D, RdW, ResultW, RD1D, RD2D);
extend         ext(InstrD[31:7], ImmSrcD, ImmExtD);

// Execute stage pipeline register and logic
floprc #(175) regE(clk, reset, FlushE,
                   {RD1D, RD2D, PCD, Rs1D, Rs2D, RdD, ImmExtD, PCPlus4D},
                   {RD1E, RD2E, PCE, Rs1E, Rs2E, RdE, ImmExtE, PCPlus4E});

mux3   #(32)  faemux(RD1E, ResultW, ALUResultM, ForwardAE, SrcAE);
mux3   #(32)  fbemux(RD2E, ResultW, ALUResultM, ForwardBE, WriteDataE);
mux2   #(32)  srcbmux(WriteDataE, ImmExtE, ALUSrcE, SrcBE);
alu           alu(SrcAE, SrcBE, ALUControlE, ALUResultE, ZeroE);
adder         branchadd(ImmExtE, PCE, PCTargetE);

// Memory stage pipeline register
flopr  #(101) regM(clk, reset,
                   {ALUResultE, WriteDataE, RdE, PCPlus4E},
                   {ALUResultM, WriteDataM, RdM, PCPlus4M});

// Writeback stage pipeline register and logic
flopr  #(101) regW(clk, reset,
                   {ALUResultM, ReadDataM, RdM, PCPlus4M},
```

```
                    {ALUResultW, ReadDataW, RdW, PCPlus4W});
  mux3   #(32)  resultmux(ALUResultW, ReadDataW, PCPlus4W, ResultSrcW,
ResultW);
endmodule

// Hazard Unit: forward, stall, and flush
module hazard(input  logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
              input  logic       PCSrcE, ResultSrcEb0,
              input  logic       RegWriteM, RegWriteW,
              output logic [1:0] ForwardAE, ForwardBE,
              output logic       StallF, StallD, FlushD, FlushE);

  logic lwStallD;

  // forwarding logic
  always_comb begin
    ForwardAE = 2'b00;
    ForwardBE = 2'b00;
    if (Rs1E != 5'b0)
      if      ((Rs1E == RdM) & RegWriteM) ForwardAE = 2'b10;
      else if ((Rs1E == RdW) & RegWriteW) ForwardAE = 2'b01;

    if (Rs2E != 5'b0)
      if      ((Rs2E == RdM) & RegWriteM) ForwardBE = 2'b10;
      else if ((Rs2E == RdW) & RegWriteW) ForwardBE = 2'b01;
  end

  // stalls and flushes
  assign lwStallD = ResultSrcEb0 & ((Rs1D == RdE) | (Rs2D == RdE));
  assign StallD = lwStallD;
  assign StallF = lwStallD;
  assign FlushD = PCSrcE;
  assign FlushE = lwStallD | PCSrcE;
endmodule

module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [ 4:0] a1, a2, a3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally (A1/RD1, A2/RD2)
  // write third port on rising edge of clock (A3/WD3/WE3)
  // write occurs on falling edge of clock
  // register 0 hardwired to 0

  always_ff @(negedge clk)
    if (we3) rf[a3] <= wd3;

  assign rd1 = (a1 != 0) ? rf[a1] : 0;
  assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule

module adder(input  [31:0] a, b,
```

```
              output [31:0] y);

  assign y = a + b;
endmodule

module extend(input  logic [31:7] instr,
              input  logic [1:0]  immsrc,
              output logic [31:0] immext);

  always_comb
    case(immsrc)
               // I-type
      2'b00:   immext = {{20{instr[31]}}, instr[31:20]};
               // S-type (stores)
      2'b01:   immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
               // B-type (branches)
      2'b10:   immext = {{20{instr[31]}}, instr[7], instr[30:25],
instr[11:8], 1'b0};
               // J-type (jal)
      2'b11:   immext = {{12{instr[31]}}, instr[19:12], instr[20],
instr[30:21], 1'b0};
      default: immext = 32'bx; // undefined
    endcase
endmodule

module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule

module flopenr #(parameter WIDTH = 8)
                (input  logic             clk, reset, en,
                 input  logic [WIDTH-1:0] d,
                 output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en) q <= d;
endmodule

module flopenrc #(parameter WIDTH = 8)
                 (input  logic             clk, reset, clear, en,
                  input  logic [WIDTH-1:0] d,
                  output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en)
      if (clear) q <= 0;
      else       q <= d;
endmodule
```

```
module floprc #(parameter WIDTH = 8)
               (input  logic clk,
                input  logic reset,
                input  logic clear,
                input  logic [WIDTH-1:0] d,
                output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else
      if (clear) q <= 0;
      else       q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);

  assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module imem(input  logic [31:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
      $readmemh("riscvtest.txt",RAM);

  assign rd = RAM[a[31:2]]; // word aligned
endmodule

module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule

module alu(input  logic [31:0] a, b,
           input  logic [2:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);
```

```
  logic [31:0] condinvb, sum;
  logic        v;              // overflow
  logic        isAddSub;       // true when is add or subtract operation

  assign condinvb = alucontrol[0] ? ~b : b;
  assign sum = a + condinvb + alucontrol[0];
  assign isAddSub = ~alucontrol[2] & ~alucontrol[1] |
                    ~alucontrol[1] &  alucontrol[0];

  always_comb
    case (alucontrol)
      3'b000:  result = sum;         // add
      3'b001:  result = sum;         // subtract
      3'b010:  result = a & b;       // and
      3'b011:  result = a | b;       // or
      3'b100:  result = a ^ b;       // xor
      3'b101:  result = sum[31] ^ v; // slt
      3'b110:  result = a << b[4:0]; // sll
      3'b111:  result = a >> b[4:0]; // srl
      default: result = 32'bx;
    endcase

  assign zero = (result == 32'b0);
  assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;

endmodule
```

Exercise 7.43 and 7.44

RISC-V Pipelined Processor – modified to support `xor` and `lui`.

```
module testbench();

  logic        clk;
  logic        reset;

  logic [31:0] WriteData, DataAdr;
  logic        MemWrite;

  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check results
  always @(negedge clk)
    begin
      if(MemWrite) begin
        if(DataAdr === 132 & WriteData === 32'hABCDE02E) begin
          $display("Simulation succeeded");
          $stop;
        end
      end
    end
endmodule

module top(input  logic        clk, reset,
           output logic [31:0] WriteDataM, DataAdrM,
           output logic        MemWriteM);

  logic [31:0] PCF, InstrF, ReadDataM;

  // instantiate processor and memories
  riscv riscv(clk, reset, PCF, InstrF, MemWriteM, DataAdrM,
              WriteDataM, ReadDataM);
  imem imem(PCF, InstrF);
  dmem dmem(clk, MemWriteM, DataAdrM, WriteDataM, ReadDataM);
endmodule

module riscv(input  logic        clk, reset,
             output logic [31:0] PCF,
             input  logic [31:0] InstrF,
             output logic        MemWriteM,
             output logic [31:0] ALUResultM, WriteDataM,
```

```
            input  logic [31:0] ReadDataM);

  logic [6:0]  opD;
  logic [2:0]  funct3D;
  logic        funct7b5D;
  logic [2:0]  ImmSrcD;
  logic        ZeroE;
  logic        PCSrcE;
  logic [2:0]  ALUControlE;
  logic        ALUSrcAE, ALUSrcBE;
  logic        ResultSrcEb0;
  logic        RegWriteM;
  logic [1:0]  ResultSrcW;
  logic        RegWriteW;

  logic [1:0]  ForwardAE, ForwardBE;
  logic        StallF, StallD, FlushD, FlushE;

  logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW;

  controller c(clk, reset,
               opD, funct3D, funct7b5D, ImmSrcD,
               FlushE, ZeroE, PCSrcE, ALUControlE, ALUSrcAE, ALUSrcBE,
ResultSrcEb0,
               MemWriteM, RegWriteM,
                          RegWriteW, ResultSrcW);

  datapath dp(clk, reset,
              StallF, PCF, InstrF,
                          opD, funct3D, funct7b5D, StallD, FlushD, ImmSrcD,
                          FlushE, ForwardAE, ForwardBE, PCSrcE, ALUControlE,
ALUSrcAE, ALUSrcBE, ZeroE,
              MemWriteM, WriteDataM, ALUResultM, ReadDataM,
              RegWriteW, ResultSrcW,
              Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW);

  hazard  hu(Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
             PCSrcE, ResultSrcEb0, RegWriteM, RegWriteW,
             ForwardAE, ForwardBE, StallF, StallD, FlushD, FlushE);

endmodule


module controller(input  logic             clk, reset,
                  // Decode stage control signals
                  input logic [6:0]  opD,
                  input logic [2:0]  funct3D,
                  input logic            funct7b5D,
                  output logic [2:0] ImmSrcD,
                  // Execute stage control signals
                  input logic            FlushE,
                  input logic            ZeroE,
                  output logic       PCSrcE,        // for datapath and
Hazard Unit
                  output logic [2:0] ALUControlE,
                  output logic           ALUSrcAE,
                  output logic       ALUSrcBE,      // for lui
```

```
                output logic       ResultSrcEb0,  // for Hazard Unit
                // Memory stage control signals
                output logic           MemWriteM,
                output logic       RegWriteM,     // for Hazard Unit

                // Writeback stage control signals
                output logic           RegWriteW,     // for datapath and
Hazard Unit
                output logic [1:0] ResultSrcW);

  // pipelined control signals
  logic         RegWriteD, RegWriteE;
  logic [1:0] ResultSrcD, ResultSrcE, ResultSrcM;
  logic       MemWriteD, MemWriteE;
  logic               JumpD, JumpE;
  logic               BranchD, BranchE;
  logic     [1:0] ALUOpD;
  logic [2:0] ALUControlD;
  logic         ALUSrcAD;
  logic         ALUSrcBD;  // for lui

  // Decode stage logic
  maindec md(opD, ResultSrcD, MemWriteD, BranchD,
             ALUSrcAD, ALUSrcBD, RegWriteD, JumpD, ImmSrcD, ALUOpD);
  aludec  ad(opD[5], funct3D, funct7b5D, ALUOpD, ALUControlD);

  // Execute stage pipeline control register and logic
  floprc #(11) controlregE(clk, reset, FlushE,
                           {RegWriteD, ResultSrcD, MemWriteD, JumpD, BranchD,
ALUControlD, ALUSrcAD, ALUSrcBD},
                           {RegWriteE, ResultSrcE, MemWriteE, JumpE, BranchE,
ALUControlE, ALUSrcAE, ALUSrcBE});

  assign PCSrcE = (BranchE & ZeroE) | JumpE;
  assign ResultSrcEb0 = ResultSrcE[0];

  // Memory stage pipeline control register
  flopr #(4) controlregM(clk, reset,
                         {RegWriteE, ResultSrcE, MemWriteE},
                         {RegWriteM, ResultSrcM, MemWriteM});

  // Writeback stage pipeline control register
  flopr #(3) controlregW(clk, reset,
                         {RegWriteM, ResultSrcM},
                         {RegWriteW, ResultSrcW});
endmodule

module maindec(input  logic [6:0] op,
               output logic [1:0] ResultSrc,
               output logic       MemWrite,
               output logic       Branch,
               output logic       ALUSrcA,           // for lui
               output logic       ALUSrcB,
               output logic       RegWrite, Jump,
               output logic [2:0] ImmSrc,
               output logic [1:0] ALUOp);
```

```
  logic [12:0] controls;

  assign {RegWrite, ImmSrc, ALUSrcA, ALUSrcB, MemWrite,
          ResultSrc, Branch, ALUOp, Jump} = controls;

  always_comb
    case(op)
    // RegWrite_ImmSrc_ALUSrcA_ALUSrcB_MemWrite_ResultSrc_Branch_ALUOp_Jump
      7'b0000011: controls = 13'b1_000_0_1_0_01_0_00_0; // lw
      7'b0100011: controls = 13'b0_001_0_1_1_00_0_00_0; // sw
      7'b0110011: controls = 13'b1_xxx_0_0_0_00_0_10_0; // R-type
      7'b1100011: controls = 13'b0_010_0_0_0_00_1_01_0; // beq
      7'b0010011: controls = 13'b1_000_0_1_0_00_0_10_0; // I-type ALU
      7'b1101111: controls = 13'b1_011_0_0_0_10_0_00_1; // jal
      7'b0110111: controls = 13'b1_100_1_1_0_00_0_00_0; // lui
      7'b0000000: controls = 13'b0_000_0_0_0_00_0_00_0; // need valid values
at reset
      default:    controls = 13'bx_xxx_x_x_x_xx_x_xx_x; // non-implemented
instruction
    endcase
endmodule

module aludec(input  logic       opb5,
              input  logic [2:0] funct3,
              input  logic       funct7b5,
              input  logic [1:0] ALUOp,
              output logic [2:0] ALUControl);

  logic  RtypeSub;
  assign RtypeSub = funct7b5 & opb5;  // TRUE for R-type subtract instruction

  always_comb
    case(ALUOp)
      2'b00:                ALUControl = 3'b000; // addition
      2'b01:                ALUControl = 3'b001; // subtraction
      default: case(funct3) // R-type or I-type ALU
                 3'b000:  if (RtypeSub)
                            ALUControl = 3'b001; // sub
                          else
                            ALUControl = 3'b000; // add, addi
                 3'b010:    ALUControl = 3'b101; // slt, slti
                 3'b100:    ALUControl = 3'b100; // xor
                 3'b110:    ALUControl = 3'b011; // or, ori
                 3'b111:    ALUControl = 3'b010; // and, andi
                 default:   ALUControl = 3'bxxx; // ???
               endcase
    endcase
endmodule

module datapath(input logic clk, reset,
                // Fetch stage signals
                input  logic        StallF,
                output logic [31:0] PCF,
                input  logic [31:0] InstrF,
                // Decode stage signals
                output logic [6:0]  opD,
                output logic [2:0]  funct3D,
```

```
                output logic        funct7b5D,
                input  logic        StallD, FlushD,
                input  logic [2:0]  ImmSrcD,
                // Execute stage signals
                input  logic        FlushE,
                input  logic [1:0]  ForwardAE, ForwardBE,
                input  logic        PCSrcE,
                input  logic [2:0]  ALUControlE,
                input  logic        ALUSrcAE,      // needed for lui
                input  logic        ALUSrcBE,
                output logic        ZeroE,
                // Memory stage signals
                input  logic        MemWriteM,
                output logic [31:0] WriteDataM, ALUResultM,
                input  logic [31:0] ReadDataM,
                // Writeback stage signals
                input  logic        RegWriteW,
                input  logic [1:0]  ResultSrcW,
                // Hazard Unit signals
                output logic [4:0]  Rs1D, Rs2D, Rs1E, Rs2E,
                output logic [4:0]  RdE, RdM, RdW);

  // Fetch stage signals
  logic [31:0] PCNextF, PCPlus4F;
  // Decode stage signals
  logic [31:0] InstrD;
  logic [31:0] PCD, PCPlus4D;
  logic [31:0] RD1D, RD2D;
  logic [31:0] ImmExtD;
  logic [4:0]  RdD;
  // Execute stage signals
  logic [31:0] RD1E, RD2E;
  logic [31:0] PCE, ImmExtE;
  logic [31:0] SrcAE, SrcBE;
  logic [31:0] SrcAEforward;
  logic [31:0] ALUResultE;
  logic [31:0] WriteDataE;
  logic [31:0] PCPlus4E;
  logic [31:0] PCTargetE;
  // Memory stage signals
  logic [31:0] PCPlus4M;
  // Writeback stage signals
  logic [31:0] ALUResultW;
  logic [31:0] ReadDataW;
  logic [31:0] PCPlus4W;
  logic [31:0] ResultW;

  // Fetch stage pipeline register and logic
  mux2    #(32) pcmux(PCPlus4F, PCTargetE, PCSrcE, PCNextF);
  flopenr #(32) pcreg(clk, reset, ~StallF, PCNextF, PCF);
  adder         pcadd(PCF, 32'h4, PCPlus4F);

  // Decode stage pipeline register and logic
  flopenrc #(96) regD(clk, reset, FlushD, ~StallD,
                      {InstrF, PCF, PCPlus4F},
                      {InstrD, PCD, PCPlus4D});
  assign opD       = InstrD[6:0];
```

```
  assign funct3D   = InstrD[14:12];
  assign funct7b5D = InstrD[30];
  assign Rs1D      = InstrD[19:15];
  assign Rs2D      = InstrD[24:20];
  assign RdD       = InstrD[11:7];

  regfile        rf(clk, RegWriteW, Rs1D, Rs2D, RdW, ResultW, RD1D, RD2D);
  extend         ext(InstrD[31:7], ImmSrcD, ImmExtD);

  // Execute stage pipeline register and logic
  floprc #(175) regE(clk, reset, FlushE,
                     {RD1D, RD2D, PCD, Rs1D, Rs2D, RdD, ImmExtD, PCPlus4D},
                     {RD1E, RD2E, PCE, Rs1E, Rs2E, RdE, ImmExtE, PCPlus4E});

  mux3   #(32)  faemux(RD1E, ResultW, ALUResultM, ForwardAE, SrcAEforward);
  mux2   #(32)  srcamux(SrcAEforward, 32'b0, ALUSrcAE, SrcAE);    // for lui
  mux3   #(32)  fbemux(RD2E, ResultW, ALUResultM, ForwardBE, WriteDataE);
  mux2   #(32)  srcbmux(WriteDataE, ImmExtE, ALUSrcBE, SrcBE);
  alu           alu(SrcAE, SrcBE, ALUControlE, ALUResultE, ZeroE);
  adder         branchadd(ImmExtE, PCE, PCTargetE);

  // Memory stage pipeline register
  flopr  #(101) regM(clk, reset,
                     {ALUResultE, WriteDataE, RdE, PCPlus4E},
                     {ALUResultM, WriteDataM, RdM, PCPlus4M});

  // Writeback stage pipeline register and logic
  flopr  #(101) regW(clk, reset,
                     {ALUResultM, ReadDataM, RdM, PCPlus4M},
                     {ALUResultW, ReadDataW, RdW, PCPlus4W});
  mux3   #(32)  resultmux(ALUResultW, ReadDataW, PCPlus4W, ResultSrcW,
ResultW);
endmodule

// Hazard Unit: forward, stall, and flush
module hazard(input  logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
              input  logic       PCSrcE, ResultSrcEb0,
              input  logic       RegWriteM, RegWriteW,
              output logic [1:0] ForwardAE, ForwardBE,
              output logic       StallF, StallD, FlushD, FlushE);

  logic lwStallD;

  // forwarding logic
  always_comb begin
    ForwardAE = 2'b00;
    ForwardBE = 2'b00;
    if (Rs1E != 5'b0)
      if      ((Rs1E == RdM) & RegWriteM) ForwardAE = 2'b10;
      else if ((Rs1E == RdW) & RegWriteW) ForwardAE = 2'b01;

    if (Rs2E != 5'b0)
      if      ((Rs2E == RdM) & RegWriteM) ForwardBE = 2'b10;
      else if ((Rs2E == RdW) & RegWriteW) ForwardBE = 2'b01;
  end

  // stalls and flushes
```

```
  assign lwStallD = ResultSrcEb0 & ((Rs1D == RdE) | (Rs2D == RdE));
  assign StallD = lwStallD;
  assign StallF = lwStallD;
  assign FlushD = PCSrcE;
  assign FlushE = lwStallD | PCSrcE;
endmodule

module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [ 4:0] a1, a2, a3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally (A1/RD1, A2/RD2)
  // write third port on rising edge of clock (A3/WD3/WE3)
  // write occurs on falling edge of clock
  // register 0 hardwired to 0

  always_ff @(negedge clk)
    if (we3) rf[a3] <= wd3;

  assign rd1 = (a1 != 0) ? rf[a1] : 0;
  assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule

module adder(input  [31:0] a, b,
             output [31:0] y);

  assign y = a + b;
endmodule

module extend(input  logic [31:7] instr,
              input  logic [2:0]  immsrc,  // extended to 3 bits for lui
              output logic [31:0] immext);

  always_comb
    case(immsrc)
               // I-type
      3'b000:   immext = {{20{instr[31]}}, instr[31:20]};
               // S-type (stores)
      3'b001:   immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
               // B-type (branches)
      3'b010:   immext = {{20{instr[31]}}, instr[7], instr[30:25],
instr[11:8], 1'b0};
               // J-type (jal)
      3'b011:   immext = {{12{instr[31]}}, instr[19:12], instr[20],
instr[30:21], 1'b0};
               // U-type (lui, auipc)
      3'b100:   immext = {instr[31:12], 12'b0};
      default:  immext = 32'bx; // undefined
    endcase
endmodule

module flopr #(parameter WIDTH = 8)
```

```
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule

module flopenr #(parameter WIDTH = 8)
                (input  logic             clk, reset, en,
                 input  logic [WIDTH-1:0] d,
                 output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en) q <= d;
endmodule

module flopenrc #(parameter WIDTH = 8)
                (input  logic             clk, reset, clear, en,
                 input  logic [WIDTH-1:0] d,
                 output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en)
      if (clear) q <= 0;
      else       q <= d;
endmodule

module floprc #(parameter WIDTH = 8)
              (input  logic clk,
               input  logic reset,
               input  logic clear,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else
      if (clear) q <= 0;
      else       q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);
```

```
  assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module imem(input  logic [31:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
      $readmemh("example.txt",RAM);

  assign rd = RAM[a[31:2]]; // word aligned
endmodule

module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule

module alu(input  logic [31:0] a, b,
           input  logic [2:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum;
  logic        v;              // overflow
  logic        isAddSub;       // true when is add or subtract operation

  assign condinvb = alucontrol[0] ? ~b : b;
  assign sum = a + condinvb + alucontrol[0];
  assign isAddSub = ~alucontrol[2] & ~alucontrol[1] |
                    ~alucontrol[1] &  alucontrol[0];

  always_comb
    case (alucontrol)
      3'b000:  result = sum;          // add
      3'b001:  result = sum;          // subtract
      3'b010:  result = a & b;        // and
      3'b011:  result = a | b;        // or
      3'b100:  result = a ^ b;        // xor
      3'b101:  result = sum[31] ^ v;  // slt
      default: result = 32'bx;
    endcase

  assign zero = (result == 32'b0);
  assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;

endmodule
```

Test Program:

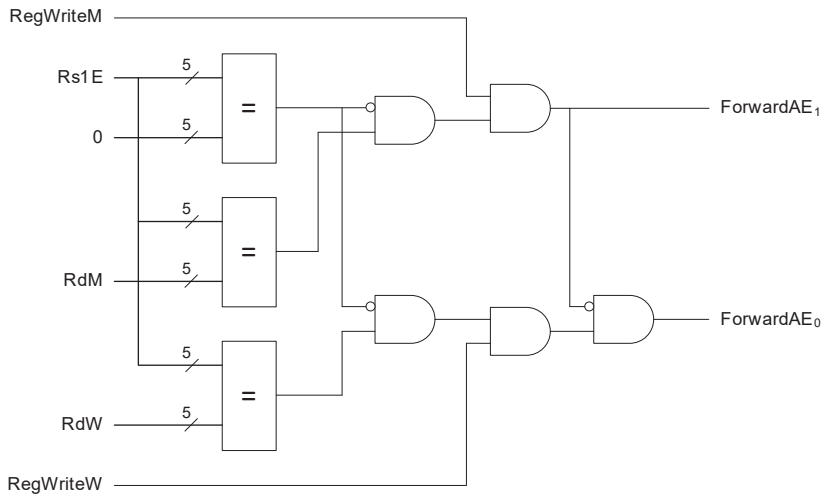
```
# If successful, it should write the value 0xABCDE02E to address 132 (0x84)
#       RISC-V Assembly         Description               Address   Machine
Code
main:   addi x2, x0, 5          # x2 = 5                  0         00500113
        addi x3, x0, 12         # x3 = 12                 4         00C00193
        addi x7, x3, -9         # x7 = (12 - 9) = 3       8         FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7       C         0023E233
        xor  x5, x3, x4         # x5 = (12 XOR 7) = 11    10        0041C2B3
        add  x5, x5, x4         # x5 = (11 + 7) = 18      14        004282B3
        beq  x5, x7, end        # shouldn't be taken      18        02728863
        slt  x4, x3, x4         # x4 = (12 < 7) = 0       1C        0041A233
        beq  x4, x0, around     # should be taken         20        00020463
        addi x5, x0, 0          # shouldn't happen        24        00000293
around: slt  x4, x7, x2         # x4 = (3 < 5)  = 1       28        0023A233
        add  x7, x4, x5         # x7 = (1 + 18) = 19      2C        005203B3
        sub  x7, x7, x2         # x7 = (19 - 5) = 14      30        402383B3
        sw   x7, 84(x3)         # [96] = 14               34        0471AA23
        lw   x2, 96(x0)         # x2 = [96] = 14          38        06002103
        add  x9, x2, x5         # x9 = (14 + 18) = 32     3C        005104B3
        jal  x3, end            # jump to end, x3 = 0x44  40        008001EF
        addi x2, x0, 1          # shouldn't happen        44        00100113
end:    add  x2, x2, x9         # x2 = (14 + 32)  = 46    48        00910133
        addi x4, x0, 1          # x4 = 1                  4C        00100213
        lui  x5, 0x80000        # x5 = 0x80000000         50        800002b7
        slt  x6, x5, x4         # x6 = 1                  54        0042a333
wrong:  beq  x6, x0, wrong      # shouldn’t be taken      58        00030063
        lui  x9, 0xABCDE        # x3 = 0xABCDE000         5C        ABCDE4B7
        add  x2, x2, x9         # x2 = 0xABCDE02E         60        00910133
        sw   x2, 0x40(x3)       # mem[132] = 0xABCDE02E   64        0421a023
done:   beq  x2, x2, done       # infinite loop           68        00210063
```

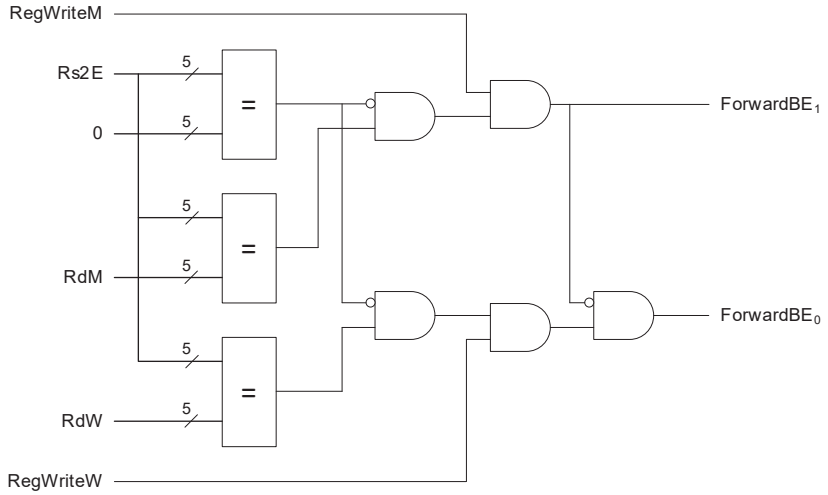
Exercise 7.45

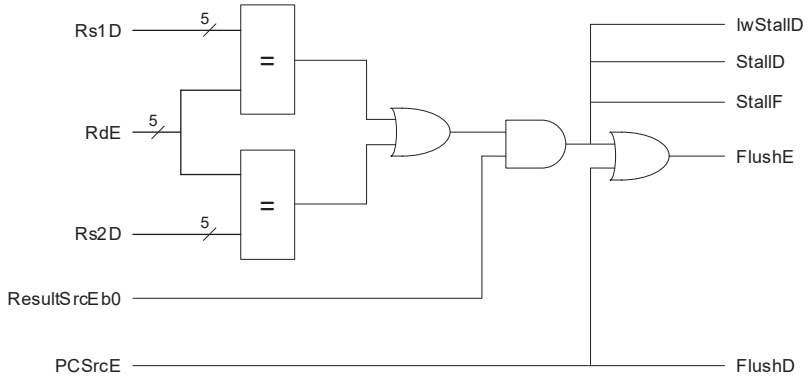
SystemVerilog

```
module hazard(input  logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
              input  logic       PCSrcE, ResultSrcEb0,
              input  logic       RegWriteM, RegWriteW,
              output logic [1:0] ForwardAE, ForwardBE,
              output logic       StallF, StallD, FlushD, FlushE);

  logic lwStallD;

  // forwarding logic
  always_comb begin
    ForwardAE = 2'b00;
    ForwardBE = 2'b00;
    if (Rs1E != 5'b0)
      if      ((Rs1E == RdM) & RegWriteM) ForwardAE = 2'b10;
      else if ((Rs1E == RdW) & RegWriteW) ForwardAE = 2'b01;

    if (Rs2E != 5'b0)
      if      ((Rs2E == RdM) & RegWriteM) ForwardBE = 2'b10;
      else if ((Rs2E == RdW) & RegWriteW) ForwardBE = 2'b01;
  end

  // stalls and flushes
  assign lwStallD = ResultSrcEb0 & ((Rs1D == RdE) | (Rs2D == RdE));
  assign StallD   = lwStallD;
  assign StallF   = lwStallD;
  assign FlushD   = PCSrcE;
  assign FlushE   = lwStallD | PCSrcE;
endmodule
```

Pipelined RISC-V Processor Hazard Unit


RegWriteM
Rs1E
0
RdM
RdW
RegWriteW
5
=
ForwardAE1
ForwardAE0

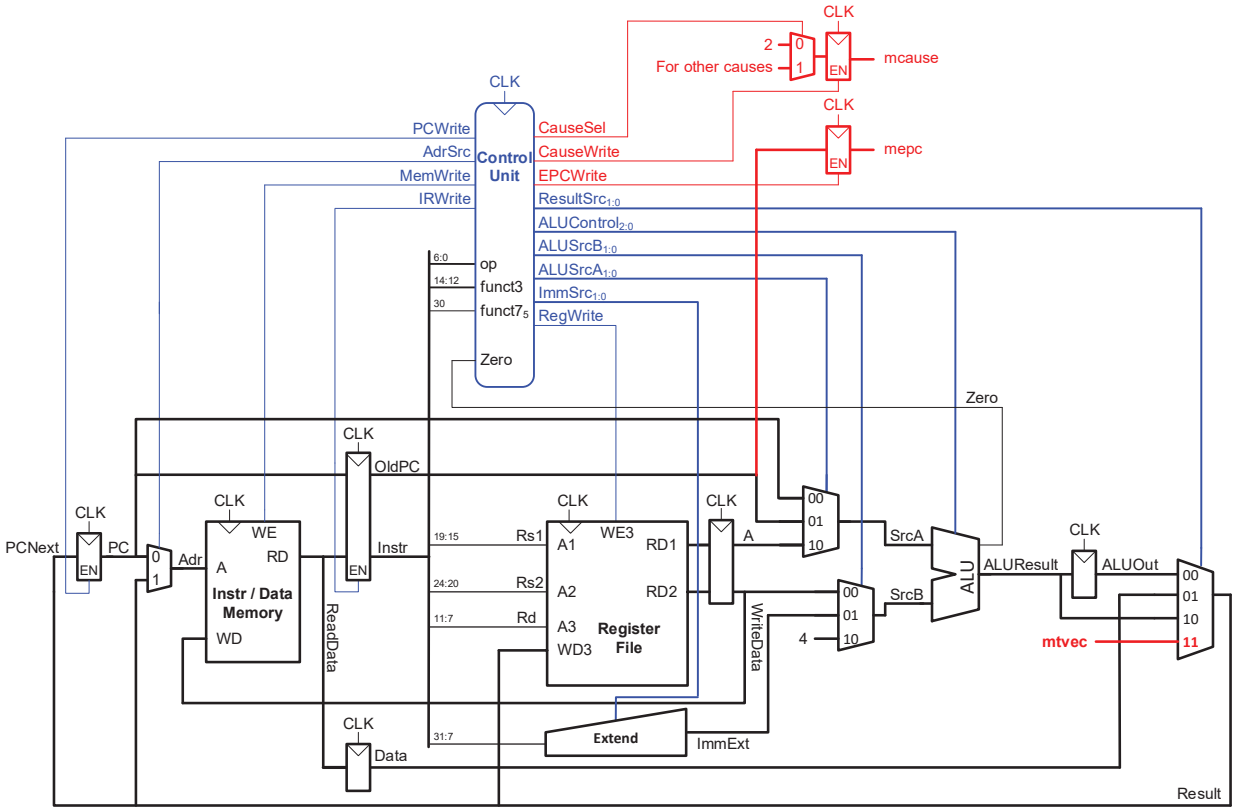

RegWriteM
Rs2E
0
RdM
RdW
RegWriteW
5
=
ForwardBE1
ForwardBE0


Rs1D
RdE
Rs2D
ResultSrcEb0
PCSrcE
5
=
lwStallD
StallD
StallF
FlushE
FlushD

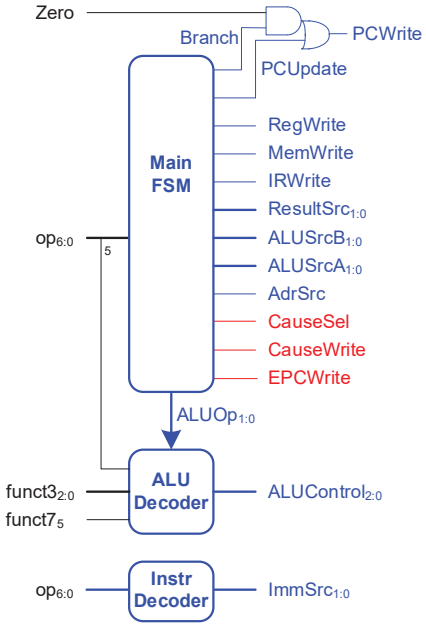
Exercise 7.46

We expand the Result multiplexer so that it can select `mtvec` (the instruction address of the exception handler). We also add enabled registers to hold `mcause` and `mepc`, the cause code of the exception (`mcause`) and the PC where the exception occurred (`mepc`). We expand the control unit to produce enables for these registers (*CauseWrite* and *EPCWrite*). A select signal, *CauseSel*, chooses amongst exception causes (i.e., cause codes). In this case, we are only supporting a single exception cause, the undefined instruction exception (cause code = 2), so the multiplexer isn't actually needed. But adding it allows us to expand to support other exception causes in the future.

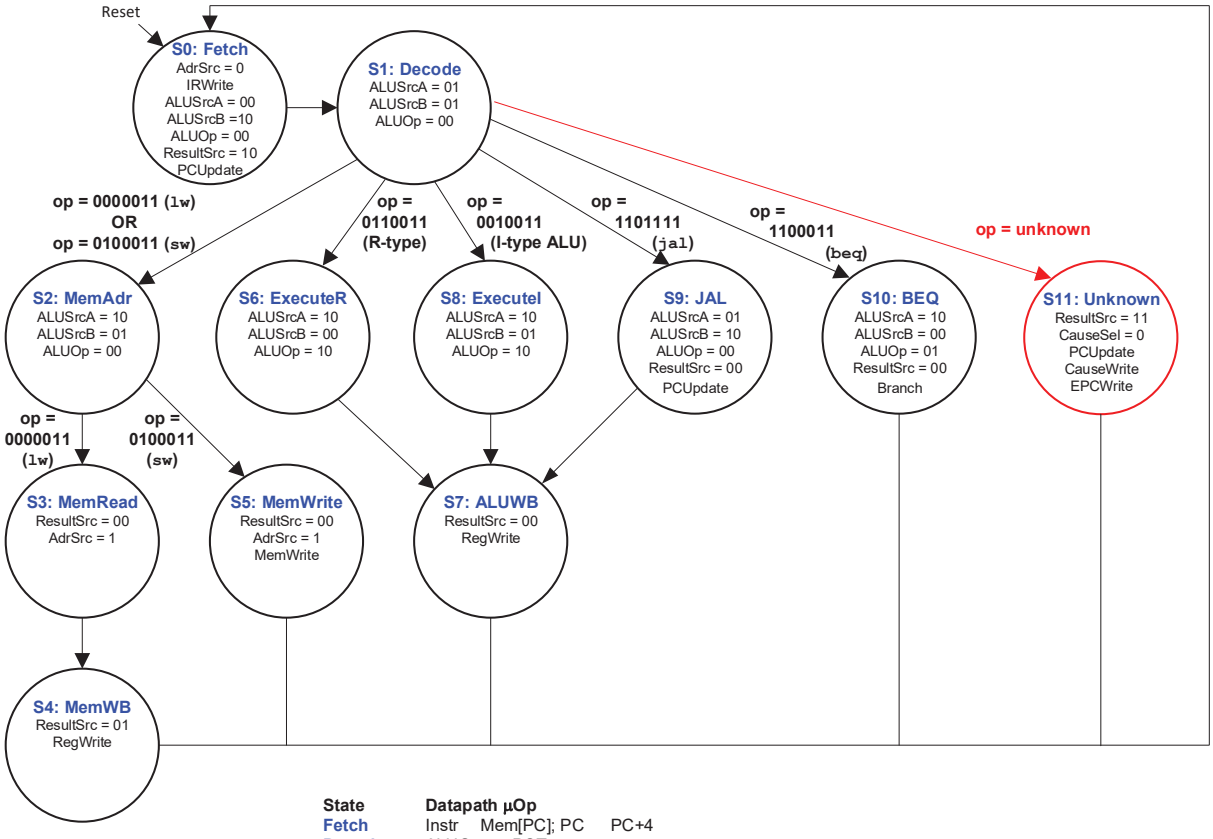
RISC-V multicycle processor modified to support the undefined instruction exception



RISC-V multicycle control modified to support the undefined instruction exception



Main FSM modified to support the undefined instruction exception

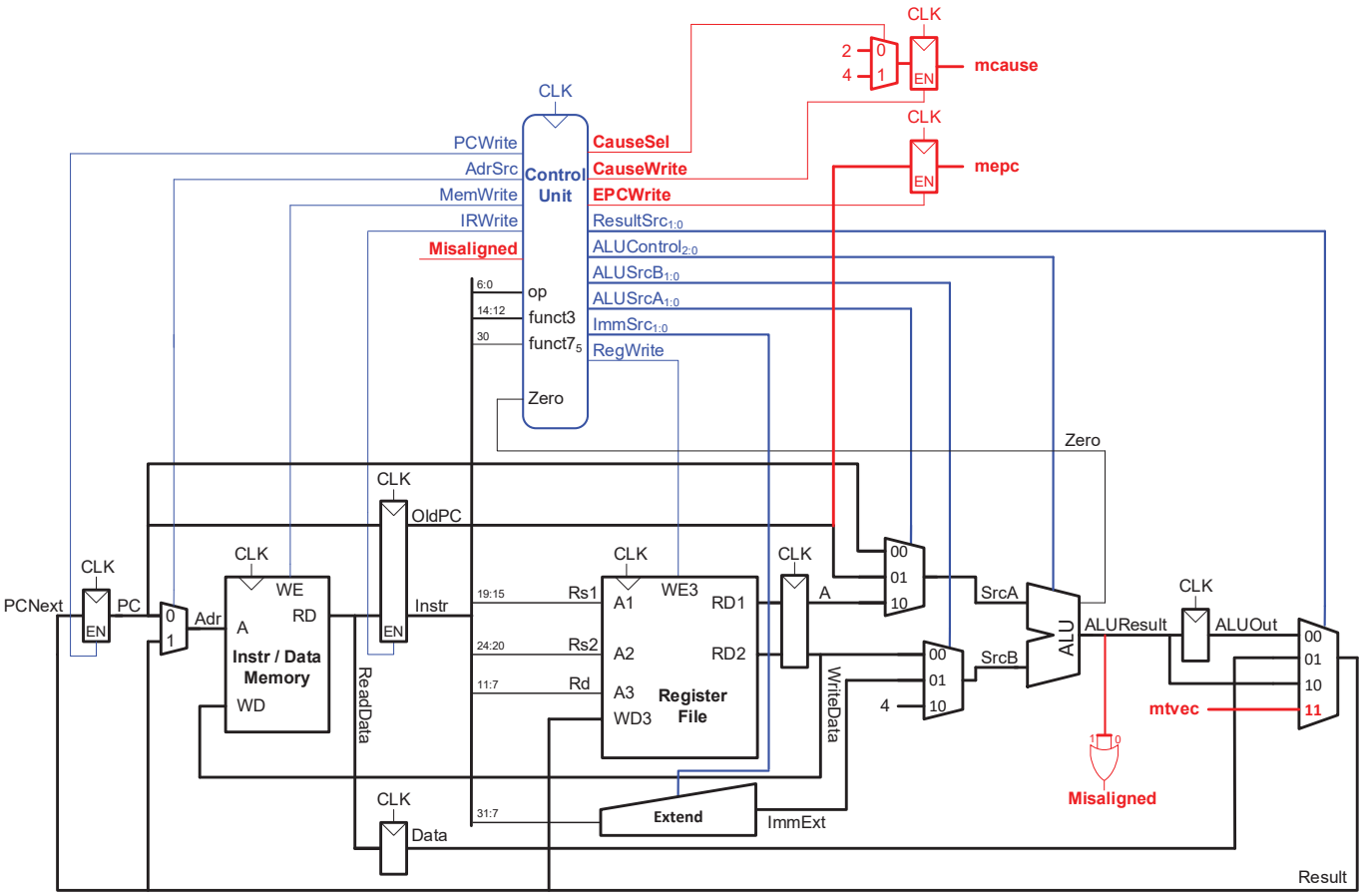


State	Datapath µOp
Fetch	Instr Mem[PC]; PC PC+4
Decode	ALUOut PCTarget
MemAdr	ALUOut rs1 + imm
MemRead	Data Mem[ALUOut]
MemWB	rd Data
MemWrite	Mem[ALUOut] rd
ExecuteR	ALUOut rs1 op rs2
ExecuteI	ALUOut rs1 op imm
ALUWB	rd ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC ALUOut; ALUOut PC+4
Unknown	PC mtvec; mcause mepc PC

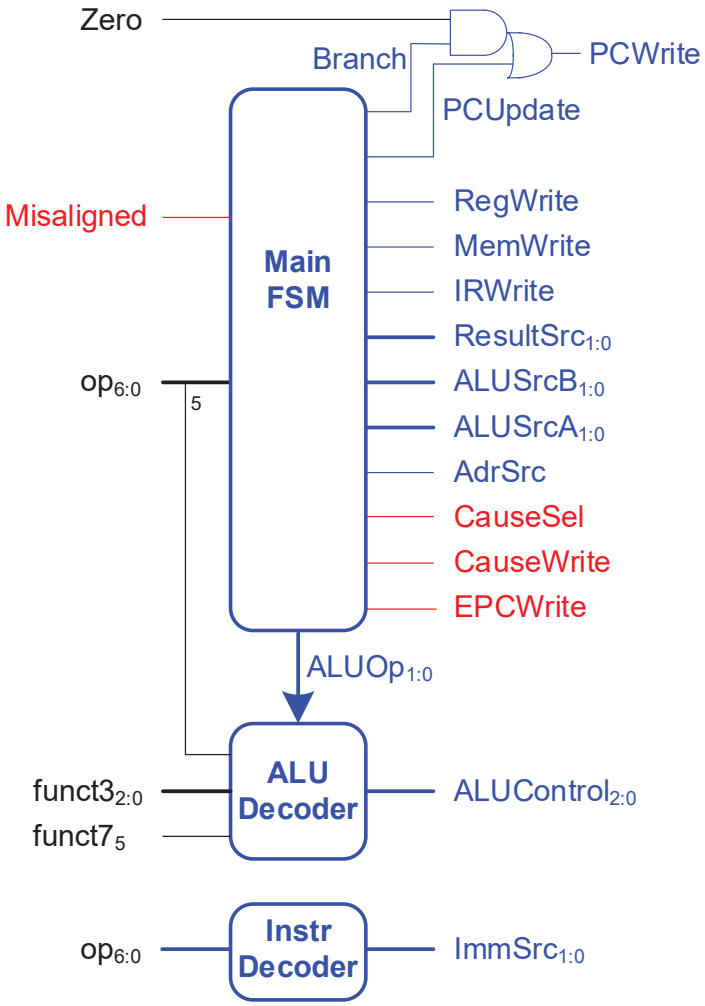
Exercise 7.47

We add a datapath signal, *Misaligned*, that is true when any of the two least significant bits of the address ($ALUResult_{1:0}$) is 1. *Misaligned* feeds into the Control Unit.
We also expand the Result multiplexer so that it can select `mtvec` (the instruction address of the exception handler). We also add enabled registers to hold `mcause` and `mepc`, the cause code of the exception (`mcause`) and the PC where the exception occurred (`mepc`). We expand the control unit to produce enables for these registers (*CauseWrite* and *EPCWrite*). A select signal, *CauseSel*, chooses amongst exception causes (i.e., cause codes). In this case, we are only supporting a single exception cause, the misaligned load exception (cause code = 4), so the multiplexer isn't actually needed. But adding it allows us to expand to support other exception causes in the future.

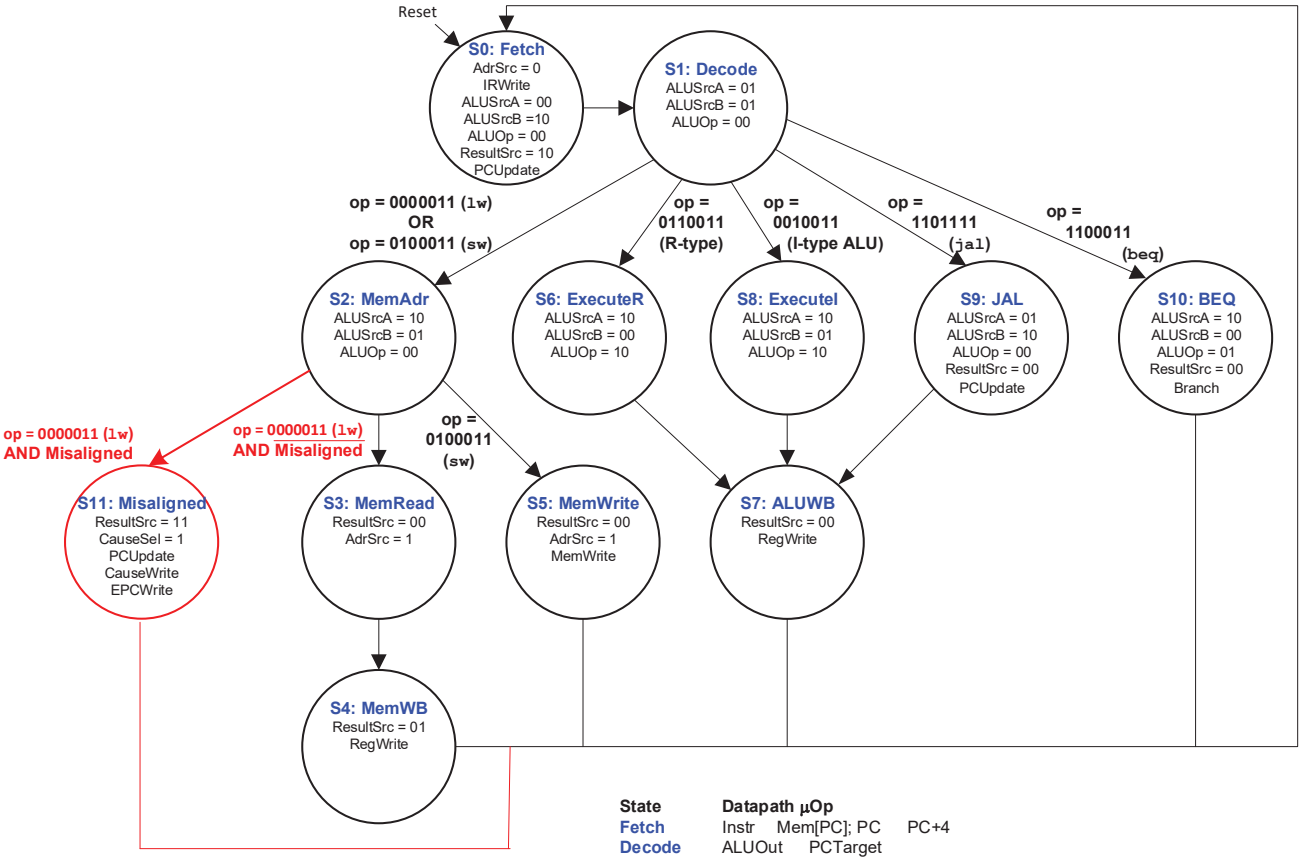
RISC-V multicycle processor modified to support the misaligned load exception



RISC-V multicycle control modified to support the misaligned load exception



Main FSM modified to support the misaligned load exception



State	Datapath μOp
Fetch	Instr Mem[PC]; PC PC+4
Decode	ALUOut PCTarget
MemAdr	ALUOut rs1 + imm
MemRead	Data Mem[ALUOut]
MemWB	rd Data
MemWrite	Mem[ALUOut] rd
ExecuteR	ALUOut rs1 op rs2
ExecuteI	ALUOut rs1 op imm
ALUWB	rd ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4
Misaligned	PC mtvec; mcause mepc PC

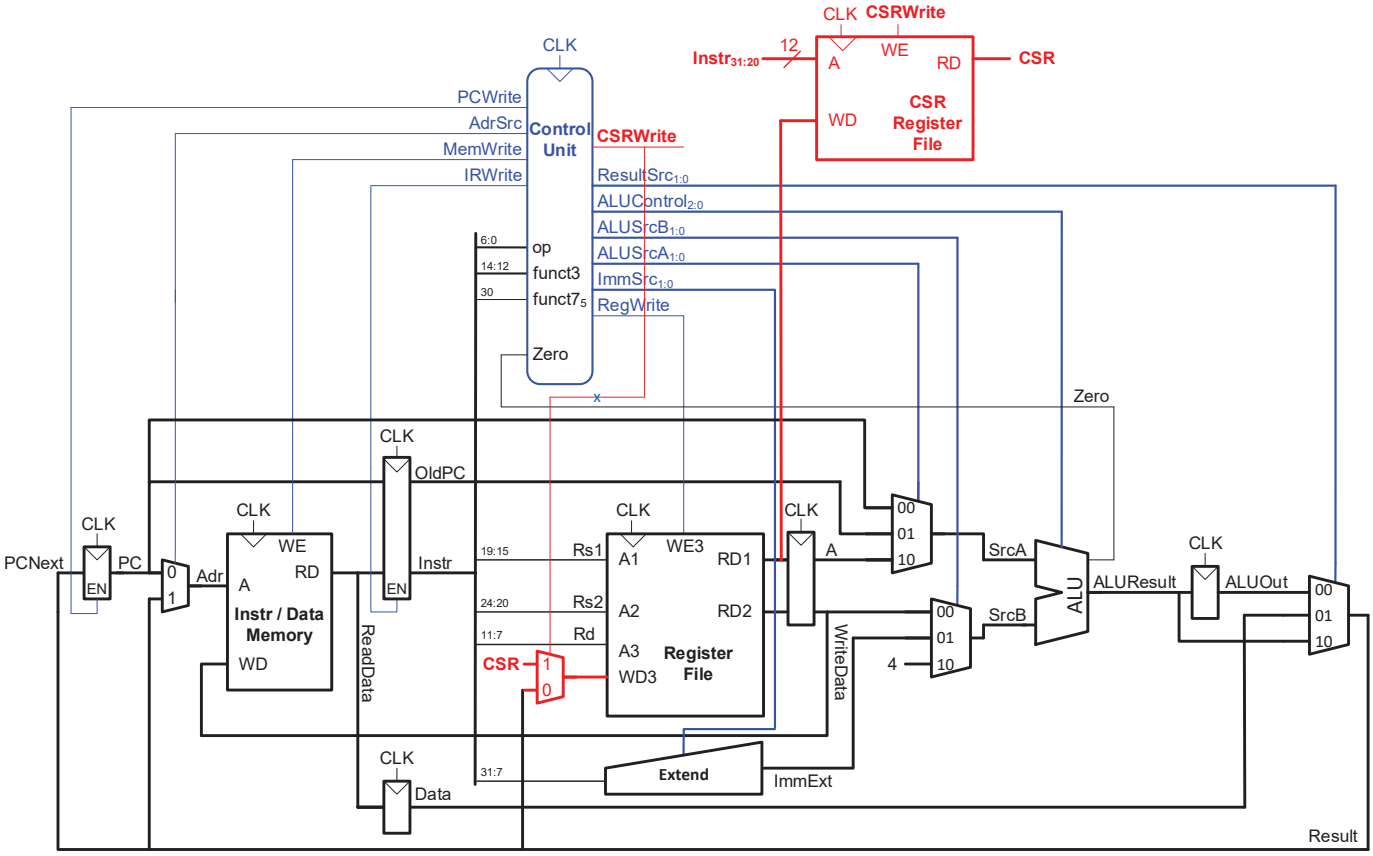
Exercise 7.48

We add a control signal, *CSRWrite*, that is true to both write the CSR with **rs1** and write the CSR to **rd**.

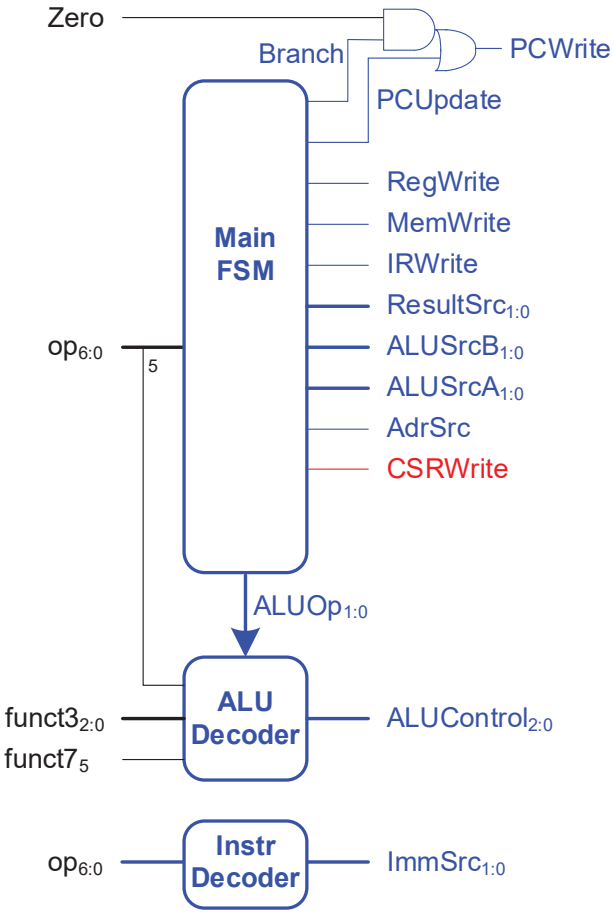
We add the following hardware:

- a multiplexer in front of the WD3 port to select either *Result* or the CSR contents to write to the register file.
- A CSR register file. The CSR number is given in the immediate field ($instr_{31:20}$) of the instruction. The CSR register file has a single read/write port.

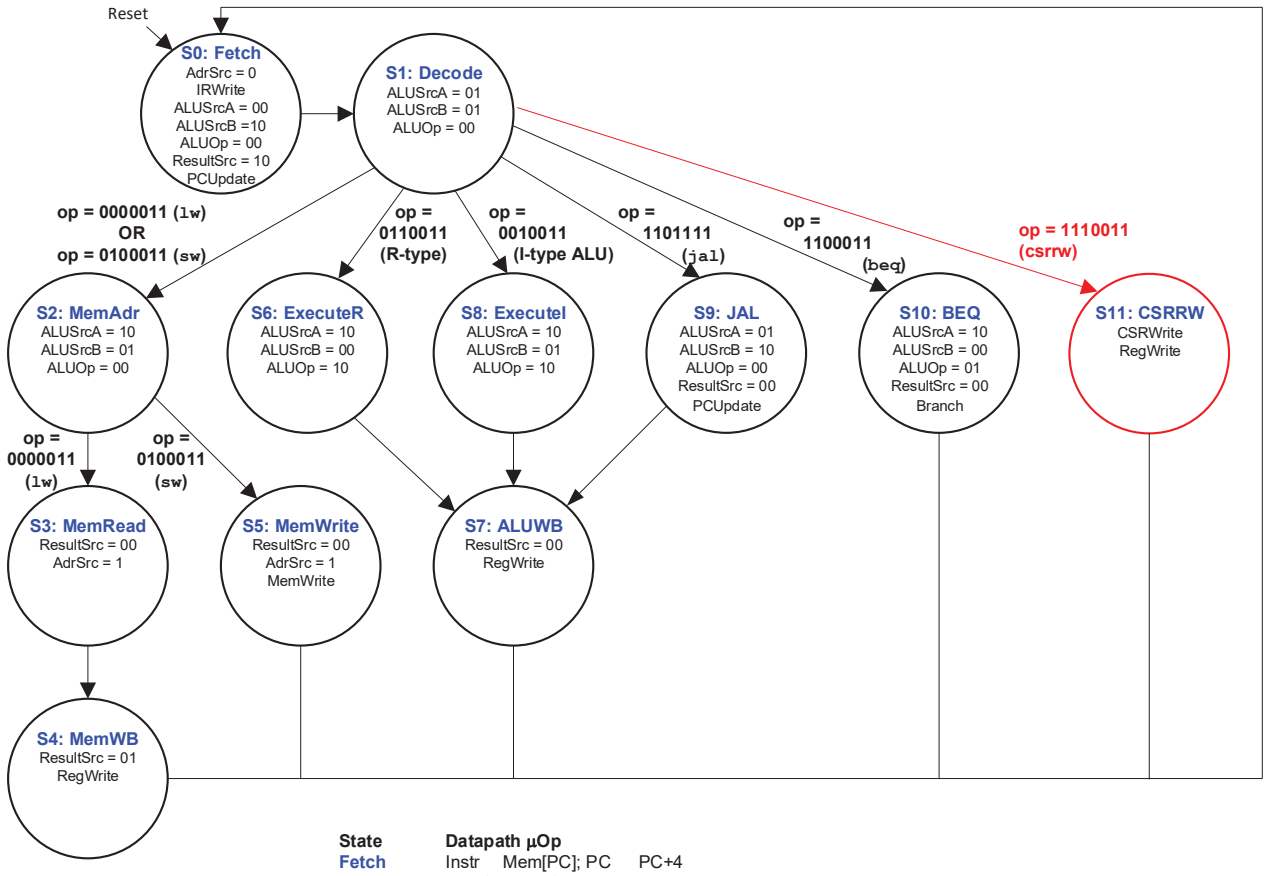
RISC-V multicycle processor modified to support the `csrrw` instruction



RISC-V multicycle control modified to support the `csrrw` instruction



Main FSM modified to support the `csrrw` instruction



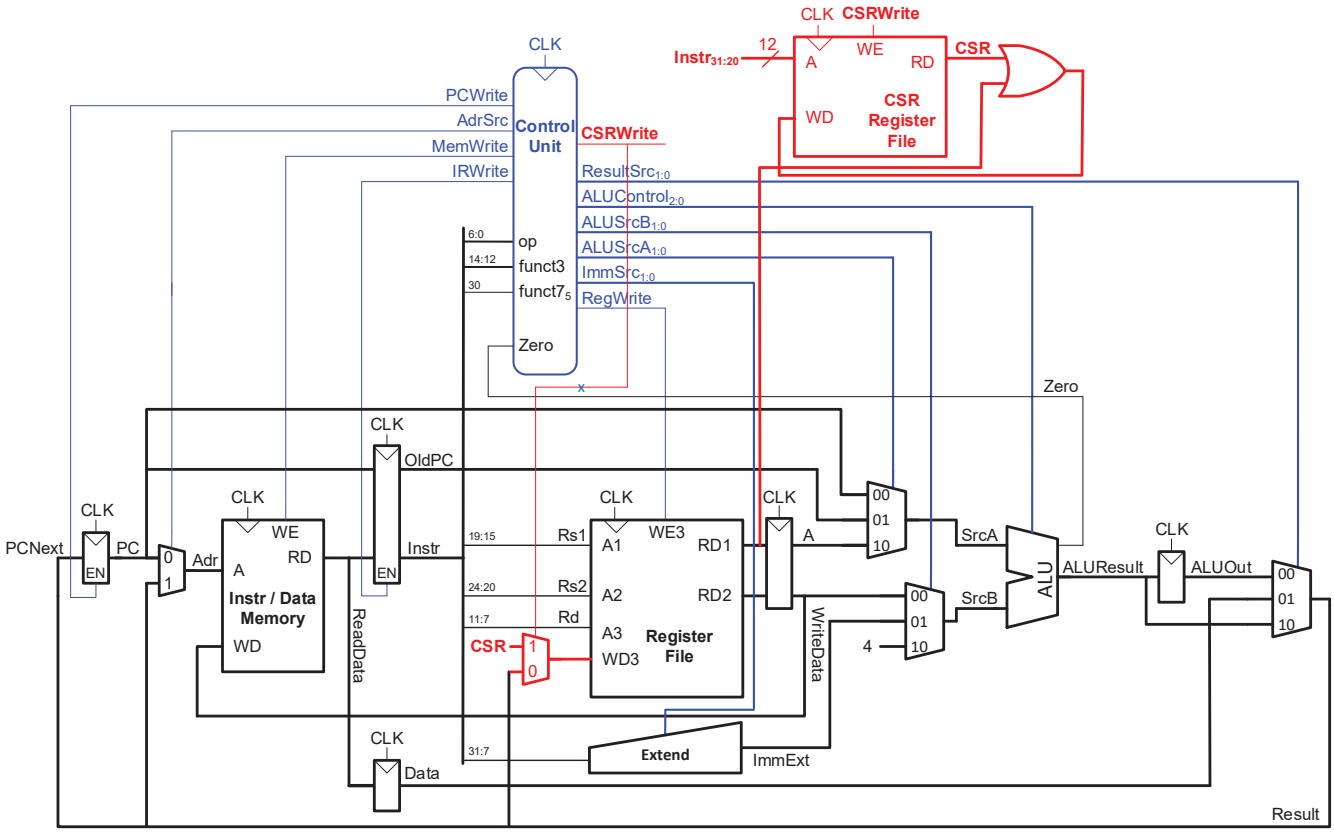
State	Datapath µOp
Fetch	Instr Mem[PC]; PC PC+4
Decode	ALUOut PCTarget
MemAdr	ALUOut rs1 + imm
MemRead	Data Mem[ALUOut]
MemWB	rd Data
MemWrite	Mem[ALUOut] rd
ExecuteR	ALUOut rs1 op rs2
ExecuteI	ALUOut rs1 op imm
ALUWB	rd ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC ALUOut; ALUOut PC+4
CSRRW	**CSR rs1; rd CSR**

Exercise 7.49

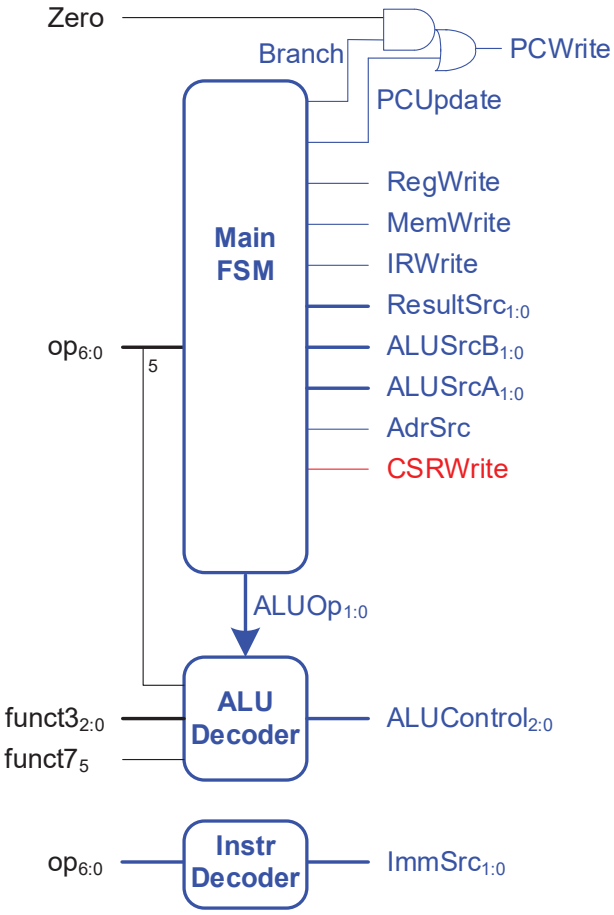
We add a control signal, *CSRWrite*, that is true to both write the CSR with **rs1** | CSR and write the CSR to **rd**.
We add the following hardware:

- a multiplexer in front of the WD3 port to select either *Result* or (**rs1** | CSR) to write to the register file.
- A CSR register file. The CSR number is given in the immediate field ($instr_{31:20}$) of the instruction. The CSR register file has a single read/write port. The OR gate perfroms **rs1** | CSR) and feeds the result to the CSR register file's WD (writedata) port.

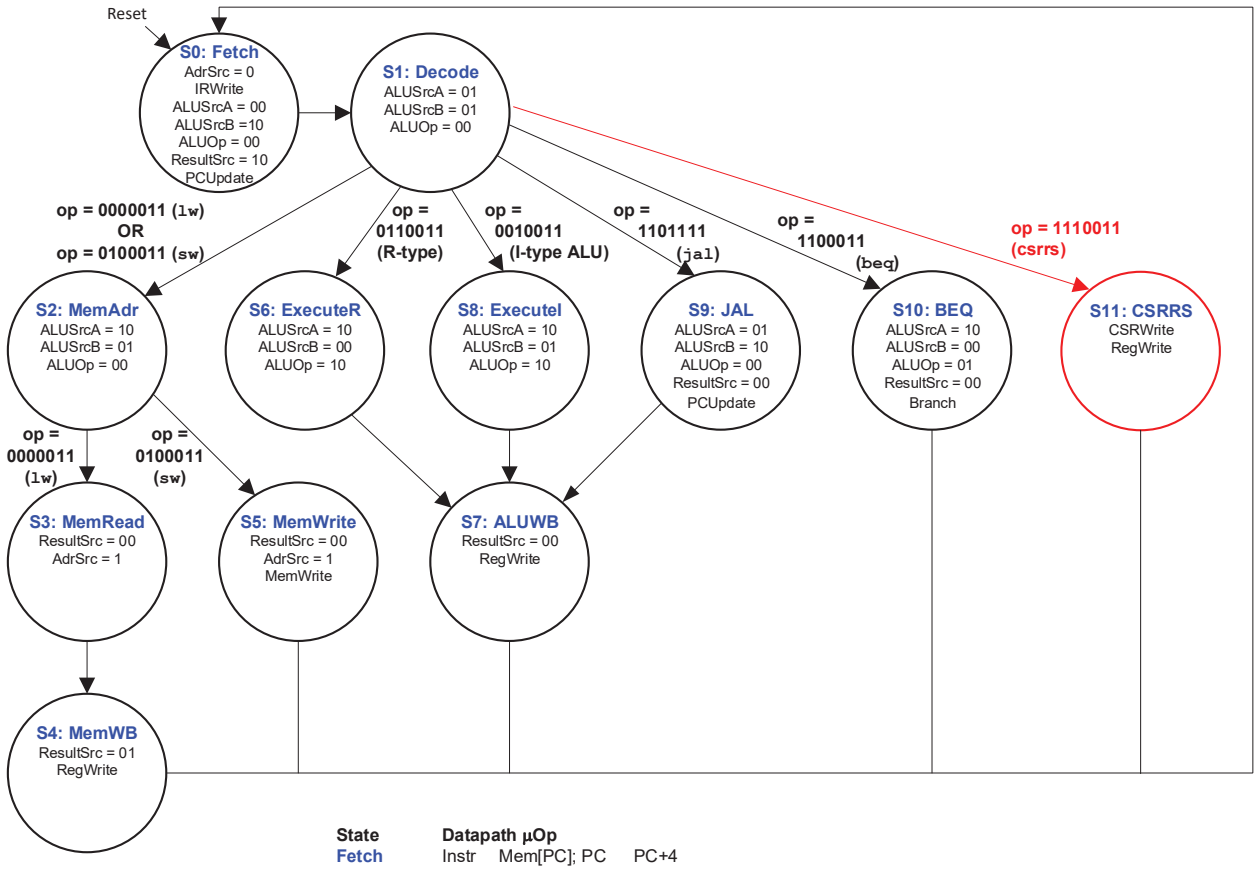
RISC-V multicycle processor modified to support the `csrrs` instruction



RISC-V multicycle control modified to support the `csrrs` instruction



Main FSM modified to support the `csrrs` instruction



State	Datapath µOp
Fetch	Instr Mem[PC]; PC PC+4
Decode	ALUOut PCTarget
MemAdr	ALUOut rs1 + imm
MemRead	Data Mem[ALUOut]
MemWB	rd Data
MemWrite	Mem[ALUOut] rd
ExecuteR	ALUOut rs1 op rs2
ExecuteI	ALUOut rs1 op imm
ALUWB	rd ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC ALUOut; ALUOut PC+4
CSRRS	CSR (CSR | rs1); rd CSR

Question 7.1

The primary advantages of pipelined processors are faster cycle time and temporal parallelism. Under ideal conditions, an *N*-stage pipelined processor is *N* times faster than a nonpipelined processor. The speedup comes at the cost of additional hardware, mainly pipelined registers and a hazard unit.

Question 7.2

Additional pipeline stages come at the cost of additional sequencing overhead. The more pipeline stages a processor has, the more dependencies arise that lead to additional stalls and flushes. This means that even with a lower cycle time, the CPI could continually go up until the additional stages provide diminishing returns or even decrease the performance of the processor. Additional pipeline stages also lead to more pipeline registers, which consume more power and increase cost. The Hazard Unit also becomes more complex with deeper pipelines. This can also affect performance and increase costs.

Question 7.3

A hazard in a pipelined microprocessor occurs when one instruction is dependent on the results of another that has not yet completed. The hazard is either a data hazard or a control hazard. A data hazard happens when an instruction tries to read a register that has not yet been written back by a previous instruction. A control hazard happens when the decision of what instruction to fetch has not been made before the first fetch takes place. Several options exist for dealing with these hazards:

1.) **Require the programmer/complier to insert nop instructions or reorder the code to eliminate dependencies.**

Pros: No additional hardware or hazard unit is needed, which reduces cost and power usage.

Cons: This complicates programming and degrades the performance of the microprocessor.

2.) **Have the hardware stall (or flush the pipeline for branches/jumps) when a dependency exists.**

Pros: Requires minimal added hardware.

Cons: Performance is not maximized (cases where forwarding can be used instead of stalling).

3.) **Forward the result from the Memory/Writeback Stage to the dependent instruction in the Execute stage and stall/flush when not possible.**

Pros: Greatest performance advantage.

Cons: Requires the most additional hardware (Hazard Unit and multiplexers).

Question 7.4

Pros:

- Superscalar processors can achieve a CPI < 1. Because of this, with superscalar processors, we measure the number of instructions per cycle (IPC)
- Superscalar processors can achieve higher throughput than a scalar processor.

Cons:

- It is difficult to fully use additional datapaths due to dependencies
- Superscalar processors require more hardware than scalar processors (i.e., x`multiple copies of the datapath, more ports on the register file/memory, more complex Hazard Unit, etc.).
- Superscalar processors require more power and are more expensive, due to the amount and complexity of the hardware.

CHAPTER 8

Exercise 8.1

Answers will vary.
Temporal locality: (1) making phone calls (if you called someone recently, you're likely to call them again soon). (2) using a textbook (if you used a textbook recently, you will likely use it again soon).

Spatial locality: (1) reading a magazine (if you looked at one page of the magazine, you're likely to look at next page soon). (2) walking to locations on campus - if a student is visiting a professor in the engineering department, she or he is likely to visit another professor in the engineering department soon.

Exercise 8.2

Answers will vary.
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.

Exercise 8.3

Repeat data accesses to the following addresses:

0x0 0x10 0x20 0x30 0x40

The miss rate for the fully associative cache is: 100%. Miss rate for the direct-mapped cache is 2/5 = 40%.

Exercise 8.4

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%.

Exercise 8.5

(a) Increasing block size will increase the cache's ability to take advantage of spatial locality. This will reduce the miss rate for applications with spatial locality. However, it also decreases the number of locations to map an address, possibly increasing conflict misses. Also, the miss penalty (the amount of time it takes to fetch the cache block from memory) increases.
(b) Increasing the associativity increases the amount of necessary hardware but in most cases decreases the miss rate. Associativities above 8 usually show only incremental decreases in miss rate.
(c) Increasing the cache size will decrease capacity misses and could decrease conflict misses. It could also, however, increase access time.

Exercise 8.6

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.

Exercise 8.7

(a) **False.**
Counterexample: A 2-word cache with block size of 1 word and access pattern:

0 4 8

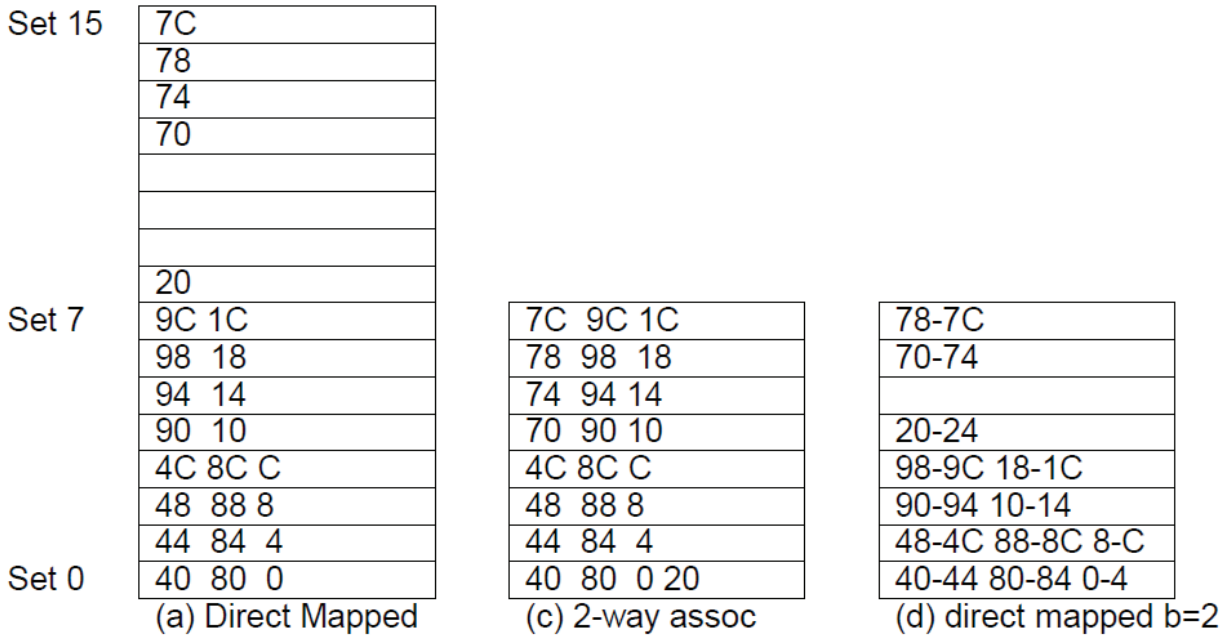
This has a 50% miss rate with a direct-mapped cache, and a100% miss rate with a 2-way set associative cache.
(b) **True.**
The 16KB cache is a superset of the 8KB cache. (Note: it's possible that they have the same miss rate.)
(c) **Usually true.**
Instruction memory accesses display great spatial locality, so a large block size reduces the miss rate.

Exercise 8.8

(a) b × S × *N* × 4 bytes
(b) [A - ($\log_2$(*S*) + $\log_2$(*b*) + 2)] × S × N
(c) *S* = 1, *N* = *C*/*b*
(d) *S* = *C*/*b*

Exercise 8.9

The figure below shows where each address maps for each cache configuration.



(a) **80% miss rate.** Addresses 70-7C and 20 use unique cache blocks and are not removed once placed into the cache. Miss rate is 20/25 = 80%.
(b) **100% miss rate.** A repeated sequence of length greater than the cache size produces no hits for a fully-associative cache using LRU.
(c) **100% miss rate.** The repeated sequence makes at least three accesses to each set during each pass. Using LRU replacement, each value must be replaced each pass through.
(d) **40% miss rate.** Data words from consecutive locations are stored in each cache block. The larger block size is advantageous since accesses in the given sequence are made primarily to consecutive word addresses. A block size of two cuts the number of block fetches in half since two words are obtained per block fetch. The address of the second word in the block will always hit in this type of scheme (e.g. address 44 of the 40-44 address pair). Thus, the second

consecutive word accesses always hit: 44, 4C, 74, 7C, 84, 8C, 94, 9C, 4, C, 14, 1C. Tracing block accesses (see Figure 8.1) shows that three of the eight blocks (70-74, 78-7C, 20-24) also remain in memory. Thus, the hit rate is: 15/25 = 60% and miss rate is 40%.

Exercise 8.10

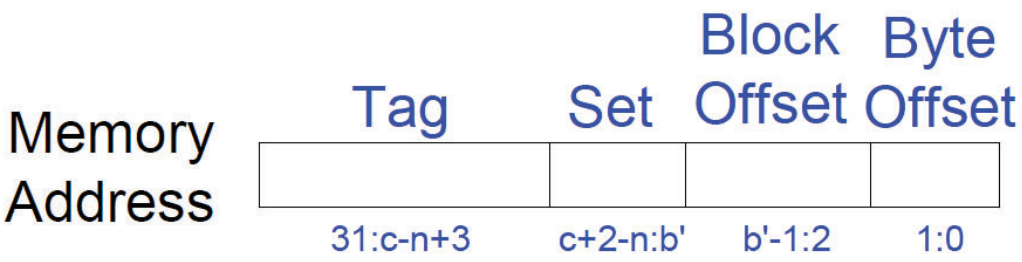
(a) 11/14 = **79%** miss rate
(b) 12/14 = **86%** miss rate
(c) 6/14 = **43%** miss rate
(d) 7/14 = **50%** miss rate

Exercise 8.11

(a) 128
(b) 100%
(c) ii

Exercise 8.12

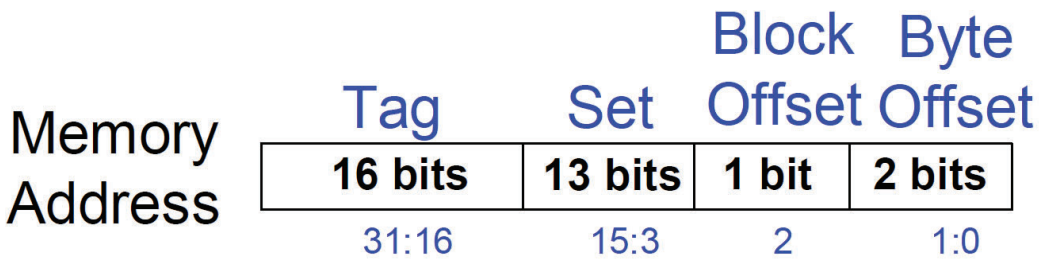
(a - b)



(c) Each tag is 32 - (c+2-n) bits = (30 - (c-n)) bits
(d) # tag bits × # blocks = (30 - (c-n)) × $2^{c+2-b'}$

Exercise 8.13

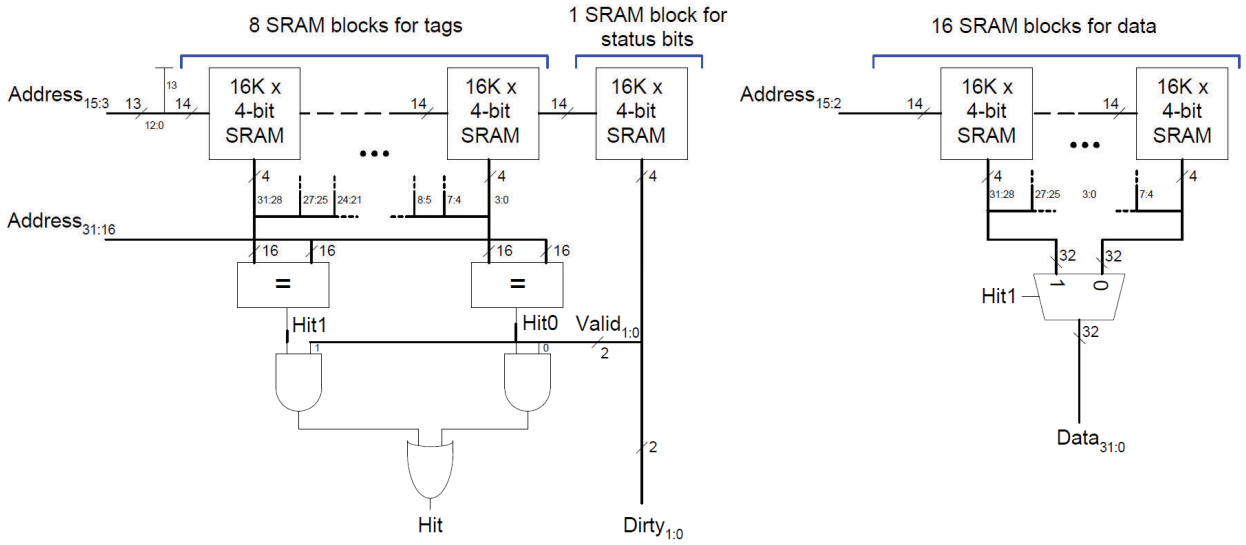
(a)



(b) Each tag is 16 bits. There are 32Kwords / (2 words / block) = 16K blocks and each block needs a tag: 16 × 16K = 218 = **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) See figure below. The design must use enough RAM chips to handle both the total capacity and the number of bits that must be read on each cycle. For the data, the SRAM must provide a capacity of 128 KB and must read 64 bits per cycle (one 32-bit word from each way). Thus the design needs at least 128KB / (8KB/RAM) = 16 RAMs to hold the data and 64 bits / (4 pins/RAM) = 16 RAMs to supply the number of bits. These are equal, so the design needs exactly 16 RAMs for the data.

For the tags, the total capacity is 32 KB, from which 32 bits (two 16-bit tags) must be read each cycle. Therefore, only 4 RAMs are necessary to meet the capacity, but 8 RAMs are needed to supply 32 bits per cycle. Therefore, the design will need 8 RAMs, each of which is being used at half capacity.

With 8K sets, the status bits require another 8K × 4-bit RAM. We use a 16K × 4-bit RAM, using only half of the entries.



Bits 15:2 of the address select the word within a set and block. Bits 15-3 select the set. Bits 31:16 of the address are matched against the tags to find a hit in one (or none) of the two blocks with each set.

Exercise 8.14

(a) The word in memory might be found in two locations, one in the on-chip cache, and one in the off-chip cache.

(b) For the first-level cache, the number of sets, S = 512 / 4 = 128 sets. Thus, 7 bits of the address are set bits. The block size is 16 bytes / 4 bytes/word = 4 words, so there are 2 block offset bits. Thus, the number of tag bits for the first-level cache is 32 - (7+2+2) = **21 bits.**

For the second-level cache, the number of sets is equal to the number of blocks, S = 256 Ksets. Thus, 18 bits of the address are set bits. The block size is 16 bytes / 4 bytes/word = 4 words, so there are 2 block offset bits. Thus, the number of tag bits for the second-level cache is 32 - (18+2+2) = **10 bits.**

(c) From Equation 8.2, $AMAT = t_{cache} + MR_{cache}(t_{MM} + MR_{MM}\, t_{VM})$. In this case, there is no virtual memory but there is an L2 cache. Thus,

$$AMAT = t_{cache} + MR_{cache}(t_{L2cache} + MR_{L2cache}\, t_{MM})$$

where, MR is the miss rate. In terms of hit rate, $MR_{cache} = 1 - HR_{cache}$, and $MR_{L2cache} = 1 - HR_{L2cache}$. Using the values given in Table 8.6,

$AMAT = t_a + (1 - A)(t_b + (1 - B)\, t_m)$

(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.

Exercise 8.15

(a) **FIFO:** FIFO replacement approximates LRU replacement by discarding data that has been in the cache longest (and is thus least likely to be used again). A FIFO cache can be stored as a queue, so the cache need not keep track of the least recently used way in an N-way set-associative cache. It simply loads a new cache block into the next way upon a new access. FIFO replacement doesn't work well when the least recently used data is not also the data fetched longest ago.
Random: Random replacement requires less overhead (storage and hardware to update status bits). However, a random replacement policy might randomly evict recently used data. In practice random replacement works quite well.

(b) FIFO replacement would work well for an application that accesses a first set of data, then the second set, then the first set again. It then accesses a third set of data and finally goes back to access the second set of data. In this case, FIFO would replace the first set with the third set, but LRU would replace the second set. The LRU replacement would require the cache to pull in the second set of data twice.

Exercise 8.16

(a) $AMAT = t_{\text{cache}} + MR_{\text{cache}}\, t_{\text{MM}}$

With a cycle time of 1/1 GHz = 1 ns,
AMAT = 1 ns + 0.05(60 ns) = **4 ns**

(b) CPI = 4 + 4 = **8 cycles** (for a load)
CPI = 4 + 3 = **7 cyles** (for a store)

(c) Average CPI = (0.11 + 0.02)(3) + (0.52)(4) + (0.1)(7) + (0.25)(8) = **5.17**

(d) Average CPI = 5.17 + 0.07(60) = **9.37**

Exercise 8.17

(a) $AMAT = t_{cache} + MR_{cache}\, t_{MM}$

With a cycle time of 1/1 GHz = 1 ns,
AMAT = 1 ns + 0.15(200 ns) = **31 ns**

(b) CPI = 31 + 4 = **35 cycles** (for a load)
CPI = 31 + 3 = **34 cyles** (for a store)

(c) Average CPI = (0.11 + 0.02)(3) + (0.52)(4) + (0.1)(34) + (0.25)(35) = **14.6**

(d) Average CPI = 14.6 + 0.1(200) = **34.6**

Exercise 8.18

2^{64} bytes = 2^4 exabytes = **16 exabytes**

Exercise 8.19

From Figure 8.4, $1 million will buy about ($1 million / ($0.05/GB)) = 20 million GB of hard disk:
20 million GB ≈ $2^{25} \times 2^{30}$ bytes = 2^{55} bytes = 2^5 petabytes = **32 petabytes**

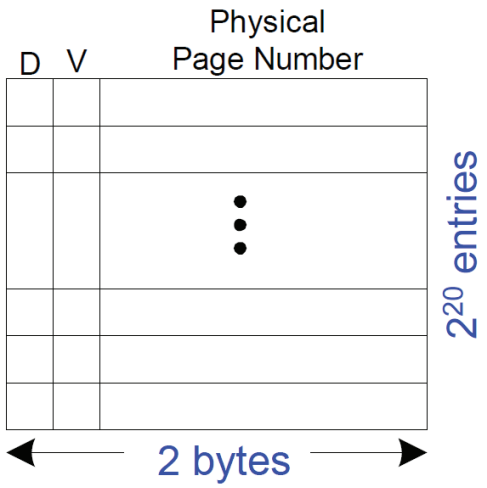
$1 million will buy about ($1,000,000 / ($7/GB)) ≈143,000 GB of DRAM.
143,000 GB ≈ $2^7 \times 2^{10} \times 2^{30} = 2^{47}$ bytes= 2^7 terabytes = **128 terabytes**

Thus, the system would need **47 bits** for the physical address and **55 bits** for the virtual address.

Exercise 8.20

(a) **23 bits**
(b) $2^{32}/2^{12}$ = **2^{20} virtual pages**
(c) 8 MB / 4 KB = $2^{23}/2^{12}$ = **2^{11} physical pages**
(d) virtual page number: **20 bits**; physical page number = **11 bits**
(e) # virtual pages / # physical pages = **29** 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**.



Exercise 8.21

(a) **31 bits**
(b) $2^{50}/2^{12} = \mathbf{2^{38}}$ **virtual pages**
(c) 2 GB / 4 KB = $2^{31}/2^{12} = \mathbf{2^{19}}$ **physical pages**
(d) virtual page number: **38 bits**; physical page number = **19 bits**
(e) $\mathbf{2^{38}}$ page table entries (one for each virtual page).
(f) Each entry uses 19 bits of physical page number and 2 bits of status information. Thus, **3 bytes** are needed for each entry (rounding 21 bits up to thevnearest number of bytes).
(h)The total table size is $\mathbf{3 \times 2^{38}}$ **bytes**.

Exercise 8.22

(a) From Equation 8.2, $AMAT = t_{cache} + MR_{cache}\,(t_{MM} + MR_{MM}\,t_{VM})$.
However, each data access now requires an address translation (page table or TLB lookup). Thus,

Without the TLB:

$AMAT = t_{MM} + [t_{cache} + MR_{cache}\,(t_{MM} + MR_{MM}\,t_{VM})]$

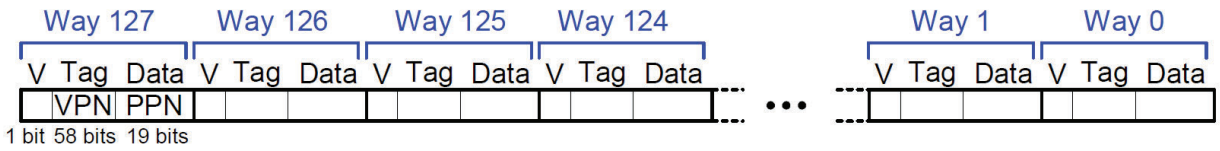
$AMAT = 100 + [1 + 0.02(100 + 0.000003(1{,}000{,}000))]$ cycles = **103.06 cycles**

With the TLB:
$AMAT = [t_{TLB} + MR_{TLB}(t_{MM})] + [t_{cache} + MR_{cache}\,(t_{MM} + MR_{MM}\,t_{VM})]$
$AMAT = [1 + 0.0005(100)] + [1 + 0.02(100 + 0.000003 \times 1{,}000{,}000)]$ 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**

(c)

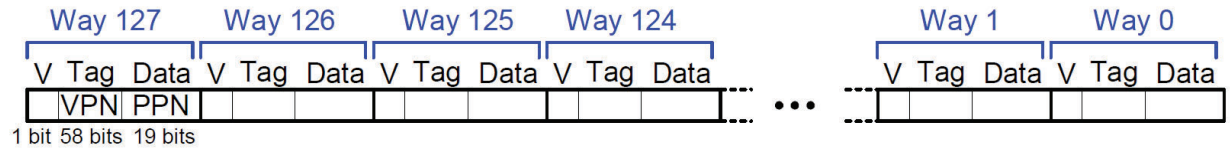


(d) **1 × 2048 bit SRAM**

Exercise 8.23

(a) 1 valid bit + 19 data bits (PPN) + 38 tag bits (VPN) x 128 entries = 58 × 128 bits = **7424 bits**

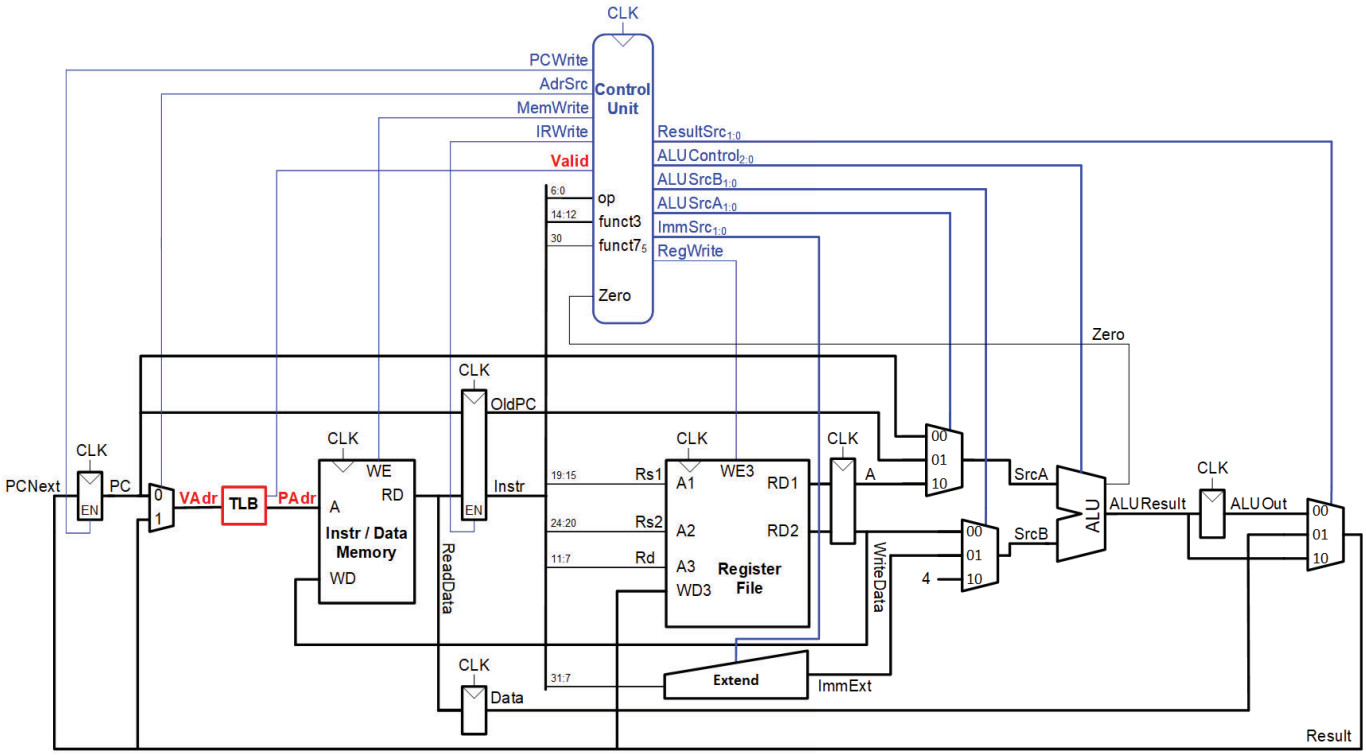
(b)



(c) 128 × 58-bit SRAM

Exercise 8.24

(a)



(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.

Exercise 8.25

(a) Each entry in the page table has 2 status bits (*V* and *D*), and a physical page number (22-16 = 6 bits). The page table has $2^{25-16} = 2^9$ entries.
Thus, the total page table size is $2^9 \times 8$ bits = **4096 bits**

(b) This would increase the virtual page number to 25 - 14 = 11 bits, and the physical page number to 22 - 14 = 8 bits. This would increase the page table size to:

$2^{11} \times 10$ bits = **20480 bits**

This increases the page table by 5 times, wasted valuable hardware to store the extra page table bits.

(c) Yes, this is possible. In order for concurrent access to take place, the number of set + block offset + byte offset bits must be less than the page offset bits.

(d) It is impossible to perform the tag comparison in the on-chip cache concurrently with the page table access because the upper (most significant) bits of the physical address are unknown until after the page table lookup (address translation) completes.

Exercise 8.26

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.

Exercise 8.27

(a) 2^{32} bytes = 4 gigabytes
(b) The amount of the hard disk devoted to virtual memory determines how many applications can run and how much virtual memory can be devoted to each application.
(c) The amount of physical memory affects how many physical pages can be accessed at once. With a small main memory, if many applications run at once or a single application accesses addresses from many different pages, thrashing can occur. This can make the applications dreadfully slow.

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.