

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 re-specify them when he uses the same windows in another room. This will save him both design time and, thus, money. He could also save by buying some items (like windows) in bulk.

Exercise 1.4

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

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

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) 1110110100111010; (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

$(5 \times 10^9 \text{ bits/second})(60 \text{ seconds/minute})(1 \text{ byte/8 bits}) = 3.75 \times 10^{10}$ bytes

Exercise 1.47

46.566 gigabytes

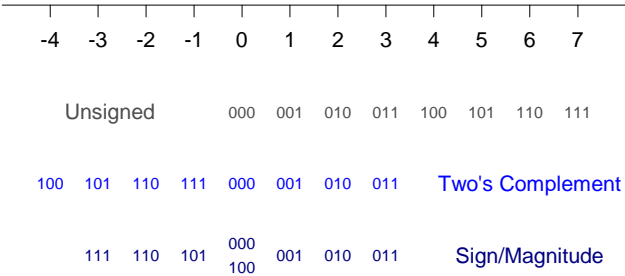
Exercise 1.48

2 billion

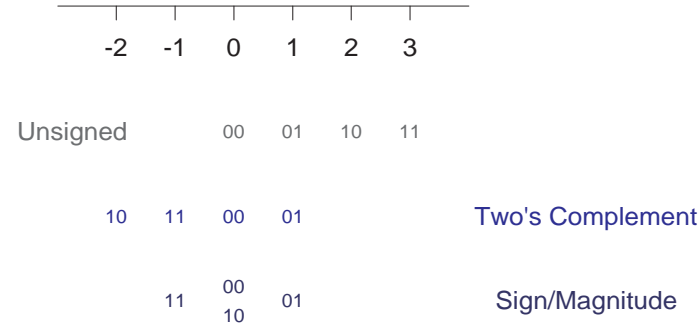
Exercise 1.49

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) $00100 - 11000 = 01100$

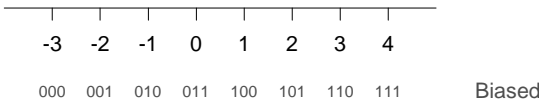
Exercise 1.61

(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;
    }

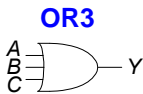
    while (num > 0) {
        digit = num % b2;
        num = num / b2;
        num2[digits2++] = digit < 10 ? digit + '0' : digit + 'A' -
10;
    }
    num2[digits2] = 0;

    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);
}

```

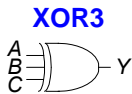
Exercise 1.71



$Y = A + B + C$

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

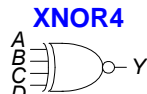
(a)



$Y = A \oplus B \oplus C$

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

(b)



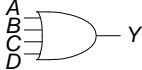
$Y = \overline{A \oplus B \oplus C \oplus D}$

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

(c)

Exercise 1.72

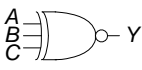
OR4



$Y = A+B+C+D$

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
(a) 1	1	1	1	1

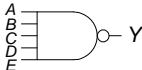
XNOR3



$Y = A \oplus B \oplus C$

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
(b) 1	1	1	0

NAND5



$Y = \overline{ABCDE}$

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
(c) 1	1	1	1	1	0

Exercise 1.73

<i>A</i>	<i>B</i>	<i>C</i>	<i>Y</i>
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

<i>A</i>	<i>B</i>	<i>C</i>	<i>Y</i>
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

<i>A</i>	<i>B</i>	<i>C</i>	<i>Y</i>
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	A	B	Y	A	B	Y	A	B	Y
0	0	0	0	0	1	0	0	0	0	0	1
0	1	0	0	1	0	0	1	1	0	1	1
1	0	0	1	0	0	1	0	0	1	0	0
1	1	0	1	1	0	1	1	0	1	1	0
Zero			A NOR B			\overline{AB}			NOT A		
A	B	Y	A	B	Y	A	B	Y	A	B	Y
0	0	0	0	0	1	0	0	0	0	0	1
0	1	0	0	1	0	0	1	1	0	1	1
1	0	1	1	0	1	1	0	1	1	0	1
1	1	0	1	1	0	1	1	0	1	1	0
$A\overline{B}$			NOT B			XOR			NAND		
A	B	Y	A	B	Y	A	B	Y	A	B	Y
0	0	0	0	0	1	0	0	0	0	0	1
0	1	0	0	1	0	0	1	1	0	1	1
1	0	0	1	0	0	1	0	0	1	0	0
1	1	1	1	1	1	1	1	1	1	1	1
AND			XNOR			B			$\overline{A} + B$		
A	B	Y	A	B	Y	A	B	Y	A	B	Y
0	0	0	0	0	1	0	0	0	0	0	1
0	1	0	0	1	0	0	1	1	0	1	1
1	0	1	1	0	1	1	0	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	1
A			$A + \overline{B}$			OR			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

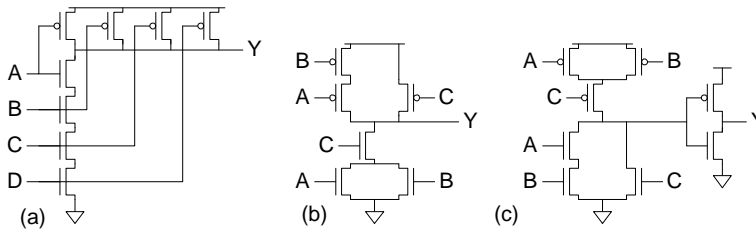
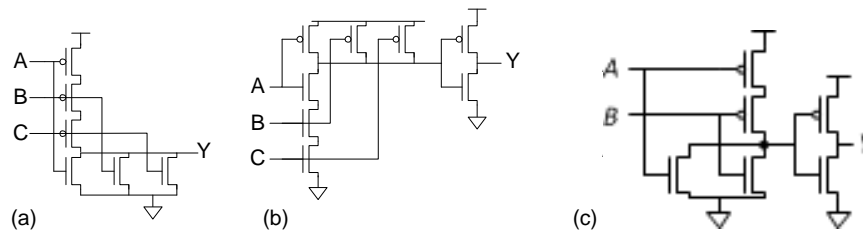
The circuit functions as a buffer with logic levels $V_{IL} = 1.5$; $V_{IH} = 1.8$; $V_{OL} = 1.2$; $V_{OH} = 3.0$. It can receive inputs from LVCMOS and LVTTL gates because their output logic levels are compatible with this gate's input levels. However, it cannot drive LVCMOS or LVTTL gates because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

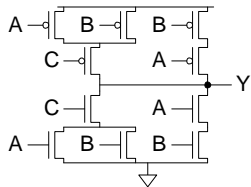
Exercise 1.82

(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

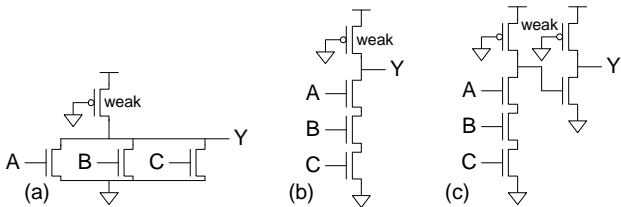
XOR

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

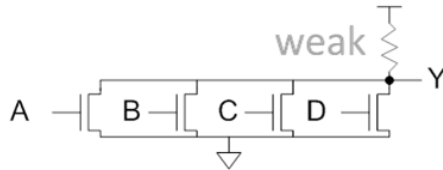
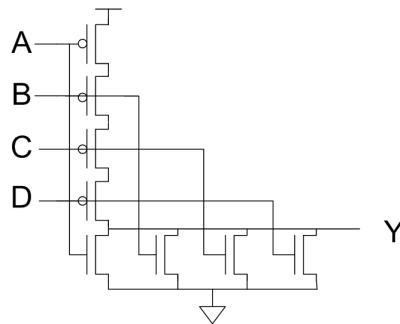
Exercise 1.88

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

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

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

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

$$(d)$$

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

$$(e)$$

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

Exercise 2.2

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

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

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

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

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

Exercise 2.3

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

(b)

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

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

(d)

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

(e)

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

Exercise 2.4

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

(d)

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

(e)

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

Exercise 2.5

(a) $Y = A + \bar{B}$ (b) $Y = \bar{A}\bar{B}\bar{C} + ABC$ (c) $Y = \bar{A}\bar{C} + A\bar{B} + AC$ (d) $Y = \bar{A}\bar{B} + \bar{B}\bar{D} + ACD$

(e)

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

This can also be expressed as:

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

Exercise 2.6

(a) $Y = A + B$

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

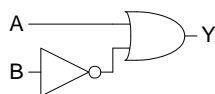
(c) $Y = AB + \bar{A}\bar{B}C$

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

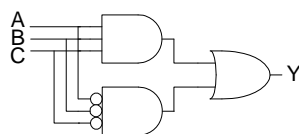
(e) $Y = A\bar{B} + \bar{A}BC + \bar{A}CD$ or $Y = A\bar{B} + \bar{A}BC + \bar{B}CD$

Exercise 2.7

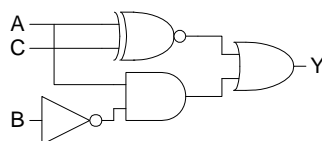
(a)



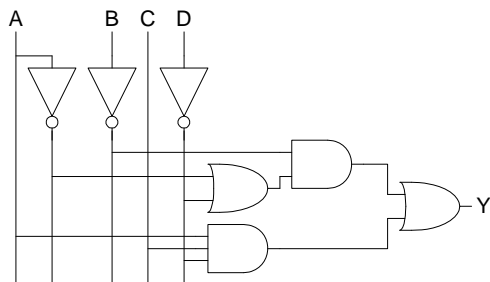
(b)



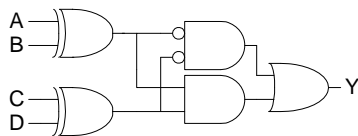
(c)



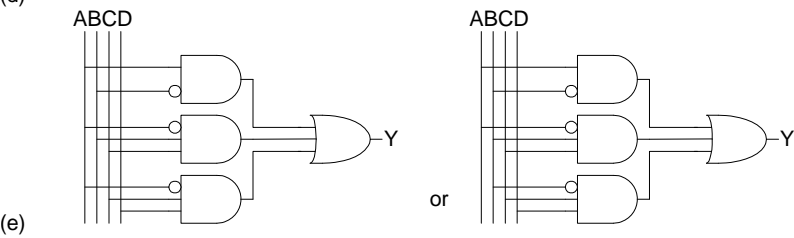
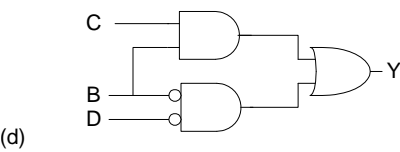
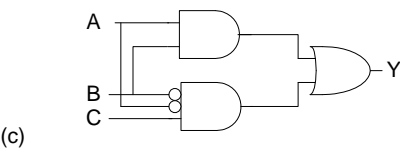
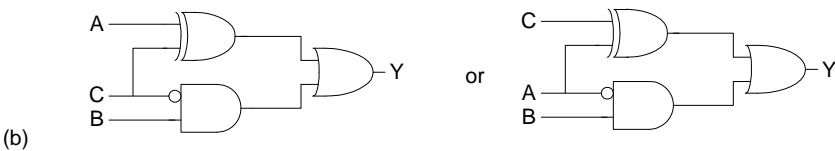
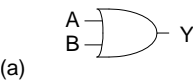
(d)



(e)



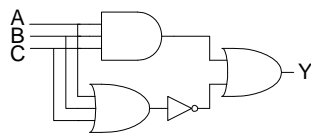
Exercise 2.8



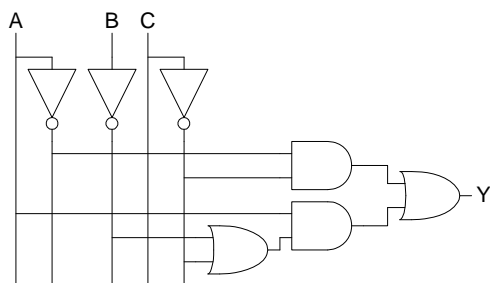
Exercise 2.9

(a) Same as 2.7(a)

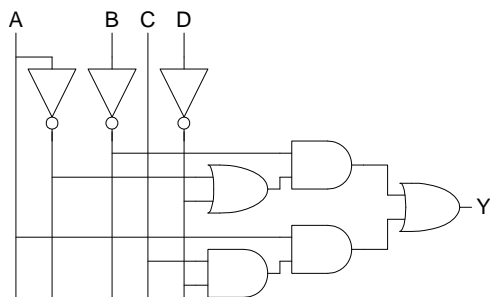
(b)



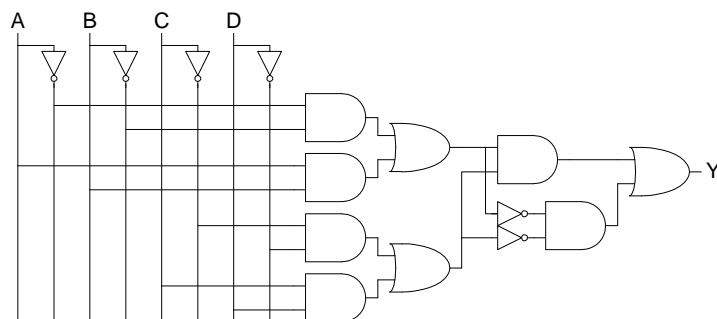
(c)

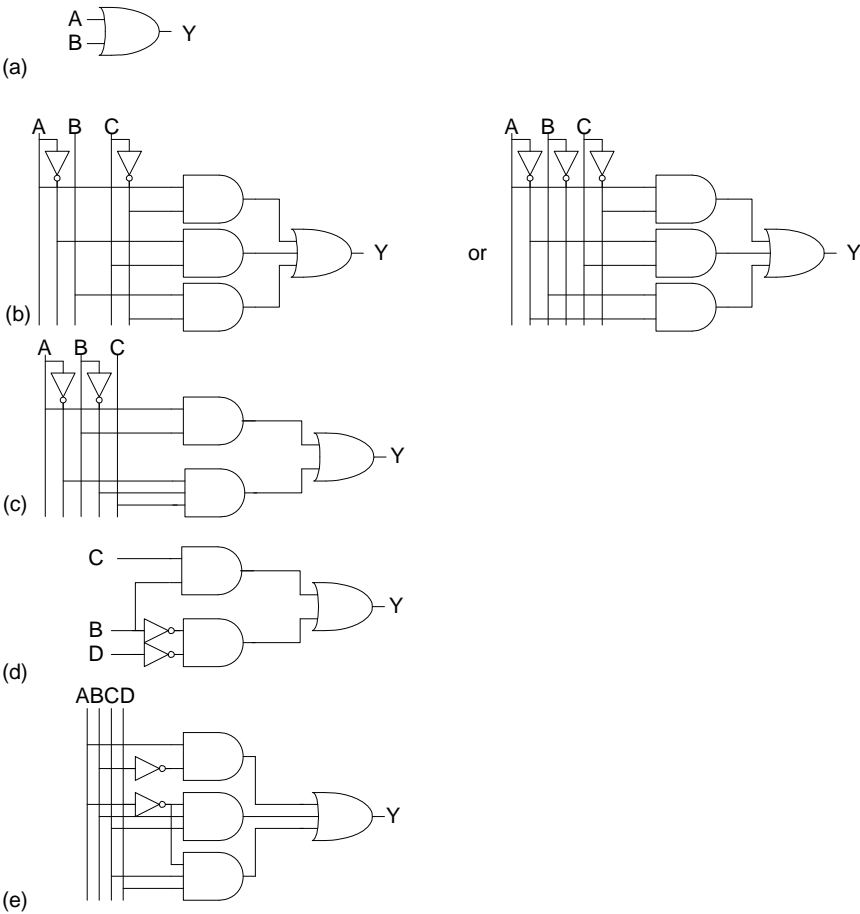


(d)

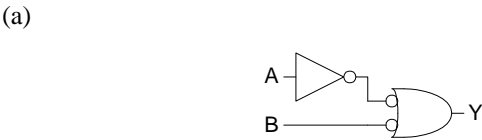


(e)

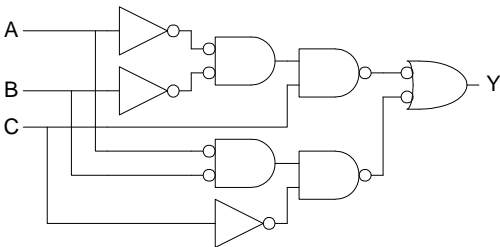
**Exercise 2.10**



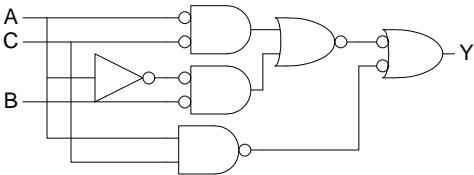
Exercise 2.11



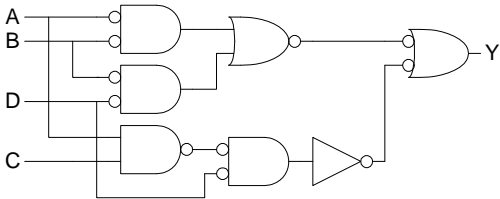
(b)



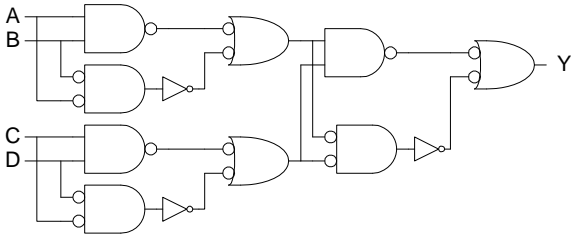
(c)



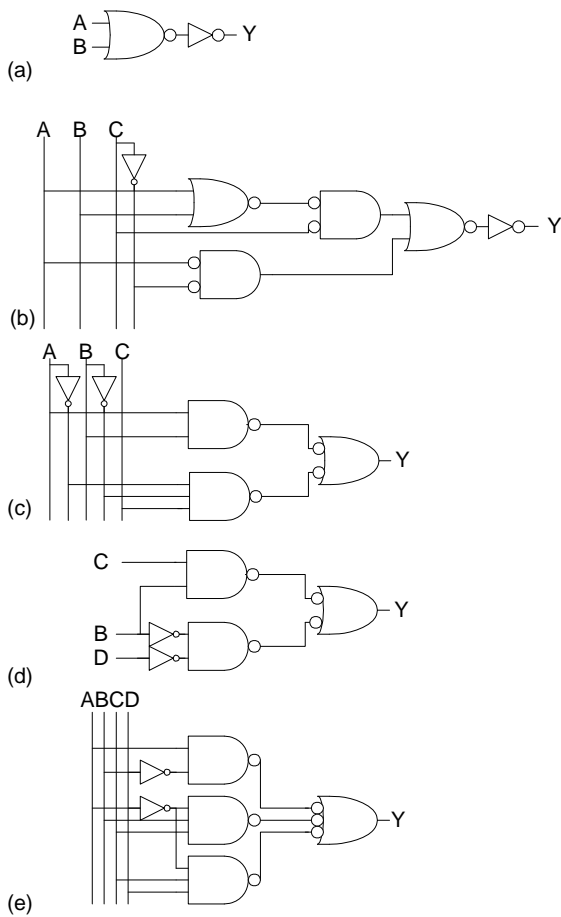
(d)



(e)



Exercise 2.12



Exercise 2.13

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

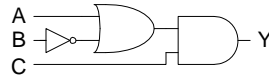
Exercise 2.14

- $$\begin{aligned} \text{(a)} \quad Y &= \bar{A}B \\ \text{(b)} \quad Y &= \bar{A} + \bar{B} + \bar{C} = \overline{ABC} \end{aligned}$$

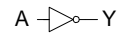
$$(c) Y = A(\bar{B} + \bar{C} + \bar{D}) + \bar{B}\bar{C}\bar{D} = \overline{ABCD} + \bar{B}\bar{C}\bar{D}$$

Exercise 2.15

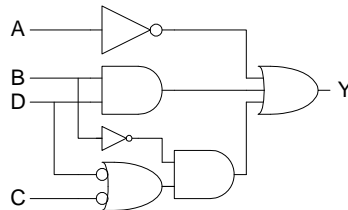
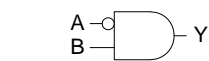
(a)



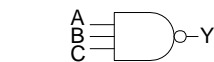
(b)



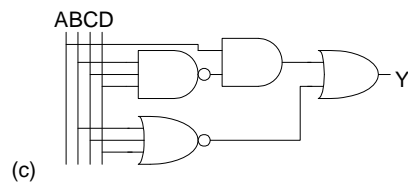
(c)

**Exercise 2.16**

(a)



(b)



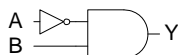
(c)

Exercise 2.17

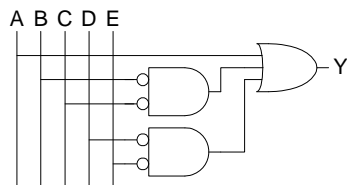
(a) $Y = B + \overline{A}\overline{C}$



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



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



Exercise 2.18

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

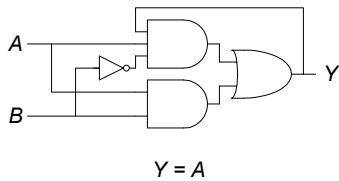
(b) $Y = (A + \overline{C})D + B$

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

Exercise 2.19

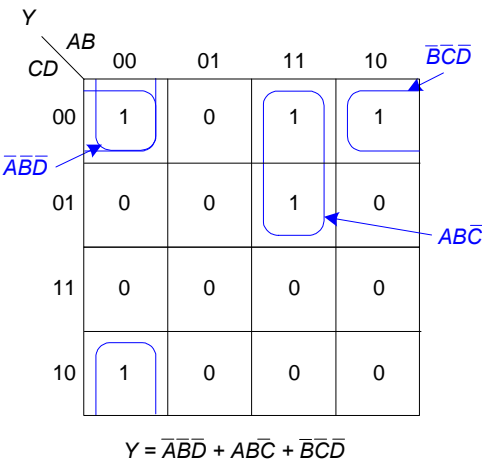
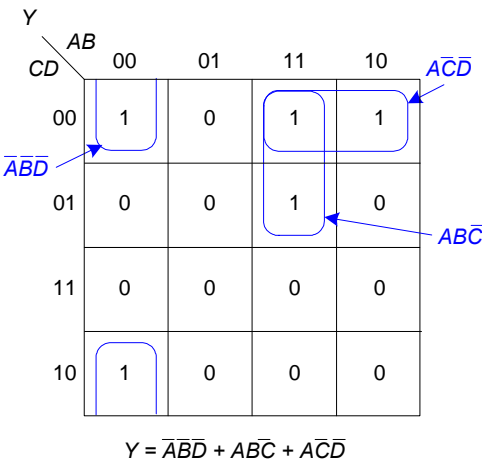
4 gigarows = 4×2^{30} rows = 2^{32} rows, so the truth table has 32 inputs.

Exercise 2.20



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



Exercise 2.22

(a)

B	B • B
0	0
1	1

(b)

<i>B</i>	<i>C</i>	<i>D</i>	$(B \bullet C) + (B \bullet D)$	$B \bullet (C + D)$
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)

<i>B</i>	<i>C</i>	$(B \bullet C) + (B \bullet \overline{C})$
0	0	0
0	1	0
1	0	1
1	1	1

Exercise 2.23

<i>B</i> ₂	<i>B</i> ₁	<i>B</i> 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

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

Exercise 2.25

Y

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

D (points to row 01)
 $\bar{A}\bar{B}C$ (points to cell 11, 10)

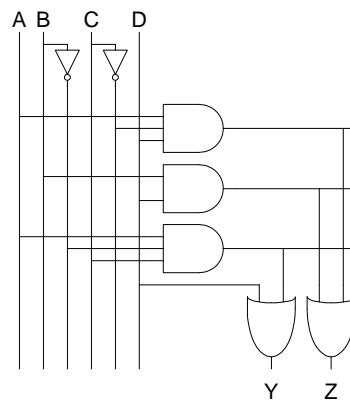
$$Y = \bar{A}\bar{B}C + D$$

Z

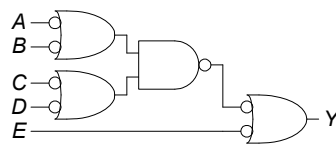
	AB	00	01	11	10
CD	00	0	0	1	0
	01	0	1	1	1
	11	0	1	1	0
	10	0	0	0	0

$\bar{A}\bar{C}D$ (points to cell 01, 11)
 BD (points to cell 11, 11)

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

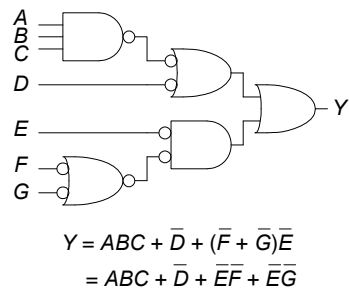


Exercise 2.26



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

Exercise 2.27



Exercise 2.28

Two possible options are shown below:

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

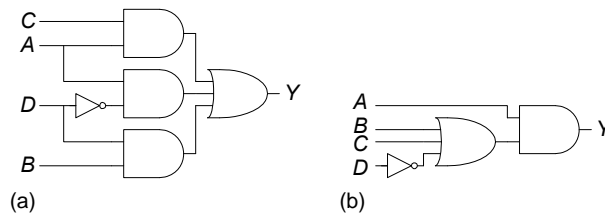
(a) $Y = A\bar{D} + AC + BD$

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

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

Exercise 2.29

Two possible options are shown below:



Exercise 2.30

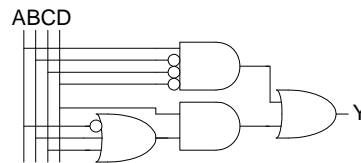
Option (a) could have a glitch when $A=1$, $B=1$, $C=0$, and D transitions from 1 to 0. The glitch could be removed by instead using the circuit in option (b).

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

Exercise 2.31

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

Exercise 2.32



Exercise 2.33

The equation can be written directly from the description:

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

Exercise 2.34

(a)

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

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

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

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

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

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

S_f		$D_{3:2}D_{1:0} + D_3D_2D_1D_0$			
		$D_{3:2}$	00	01	11
$D_{1:0}$	00	1	1	0	1
	01	0	1	0	1
	11	0	0	0	0
	10	0	1	0	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 $D_{1:0}$		$D_{3:2}$			
		00	01	11	10
00	0	1	0	1	
01	0	1	0	1	
11	1	0	0	0	
10	1	1	0	0	

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

(b)

S_a

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

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

S_b

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

$$S_b = \bar{D}_1 \bar{D}_0 + D_1 D_0 + \bar{D}_2$$

S_c

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

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

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

S_e

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

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

S_f

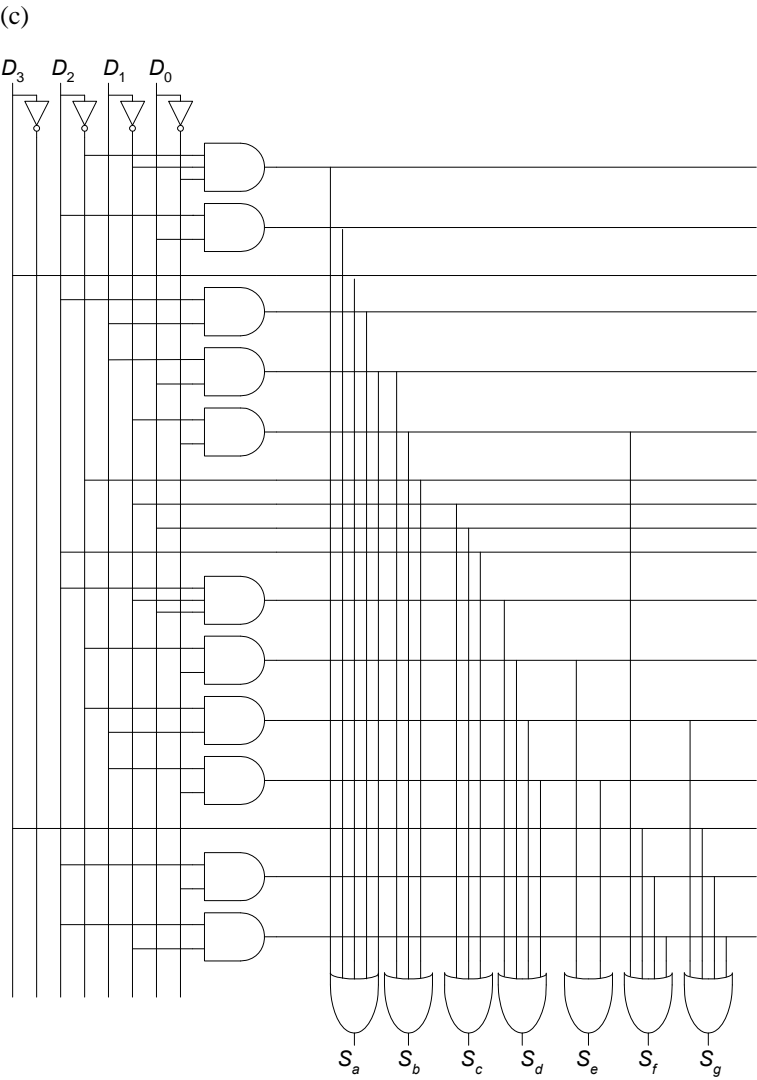
$D_{1:0} \backslash D_{3:2}$	00	01	11	10
00	1	1	X	1
01	0	1	X	1
11	0	0	X	X
10	0	1	X	X

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

S_g

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

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



Exercise 2.35

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

P has two possible minimal solutions:

D

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

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

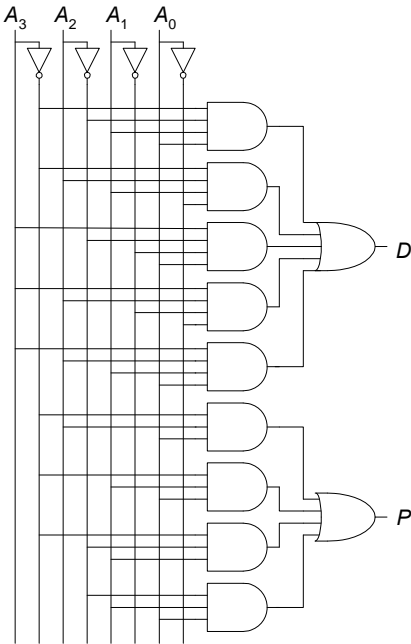
P

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

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

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

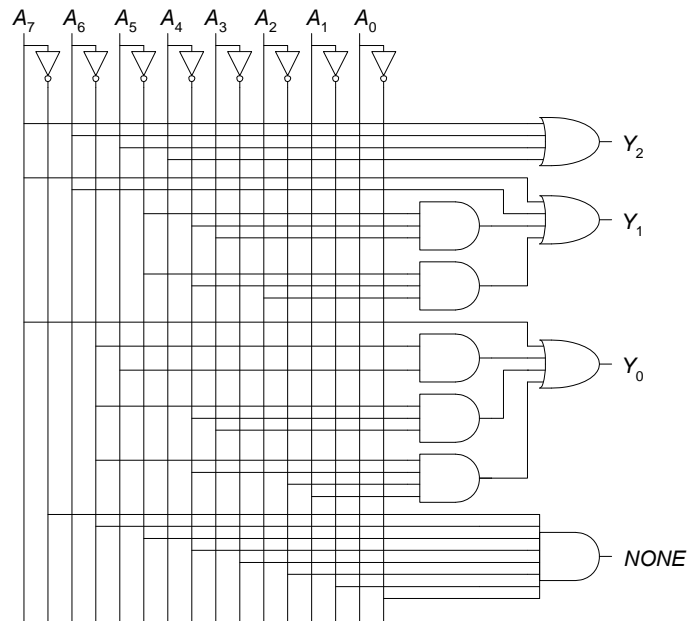
Hardware implementations are below (implementing the first minimal equation given for P).



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

$$Y_2 = A_7 + A_6 + A_5 + A_4$$
$$Y_1 = A_7 + A_6 + \overline{A_5}\overline{A_4}A_3 + \overline{A_5}\overline{A_4}A_2$$
$$Y_0 = A_7 + \overline{A_6}A_5 + \overline{A_6}\overline{A_4}A_3 + \overline{A_6}\overline{A_4}\overline{A_2}A_1$$
$$NONE = \overline{A_7}\overline{A_6}\overline{A_5}\overline{A_4}\overline{A_3}\overline{A_2}\overline{A_1}\overline{A_0}$$



Exercise 2.37

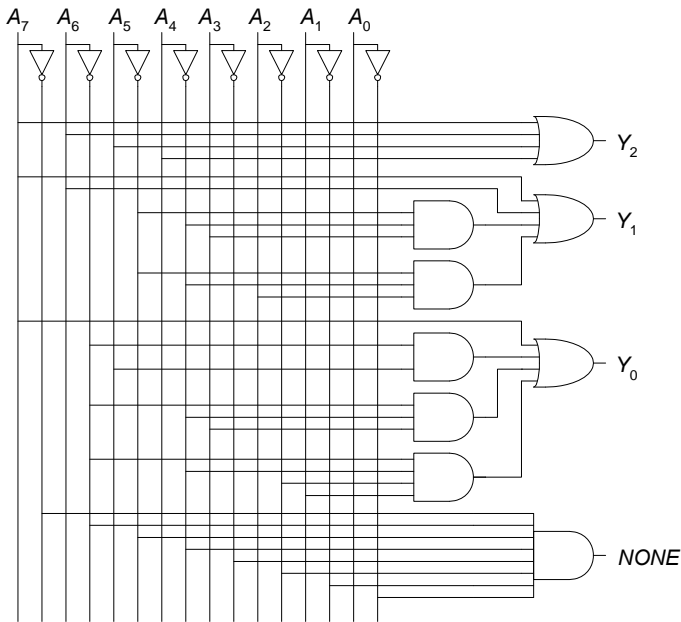
The equations and circuit for $Y_{2:0}$ is the same as in Exercise 2.25, repeated here for convenience.

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

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

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

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



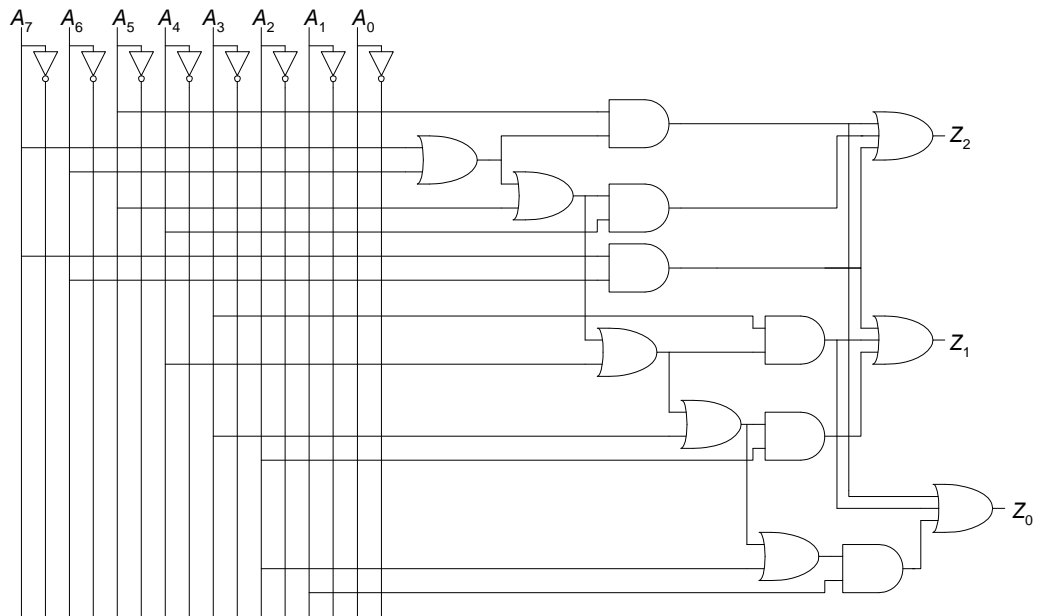
The truth table, equations, and circuit for $Z_{2:0}$ are as follows.

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

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

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

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

**Exercise 2.38**

$$Y_6 = A_2A_1A_0$$

$$Y_5 = A_2A_1$$

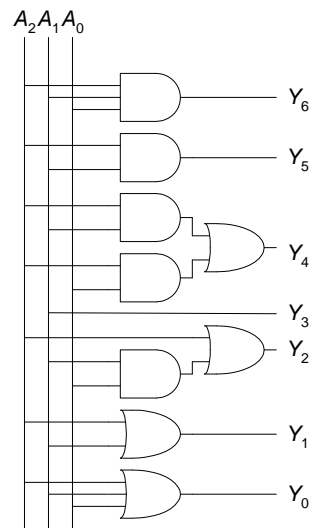
$$Y_4 = A_2A_1 + A_2A_0$$

$$Y_3 = A_2$$

$$Y_2 = A_2 + A_1A_0$$

$$Y_1 = A_2 + A_1$$

$$Y_0 = A_2 + A_1 + A_0$$

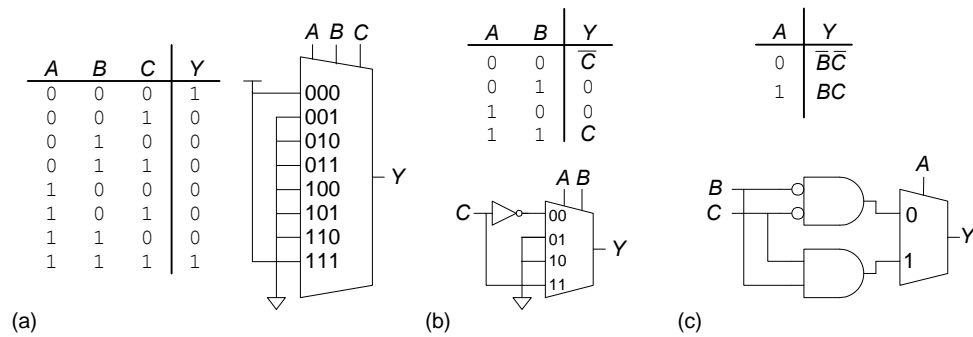
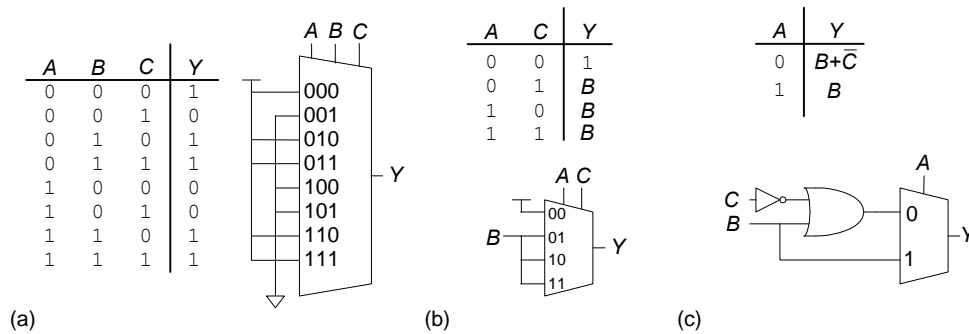
**Exercise 2.39**

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

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**

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

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

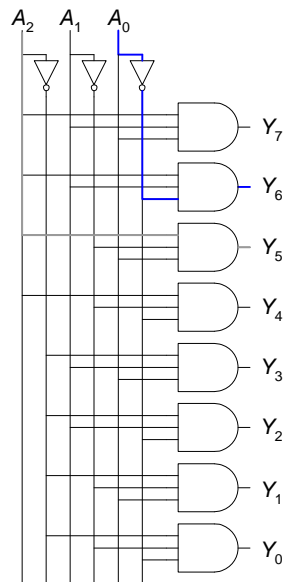
Exercise 2.44

$$\begin{aligned} t_{pd} &= t_{pd_AND2} + 2t_{pd_NOR2} + t_{pd_NAND2} \\ &= [30 + 2(30) + 20]\ ps \\ &= \mathbf{110\ ps} \end{aligned}$$

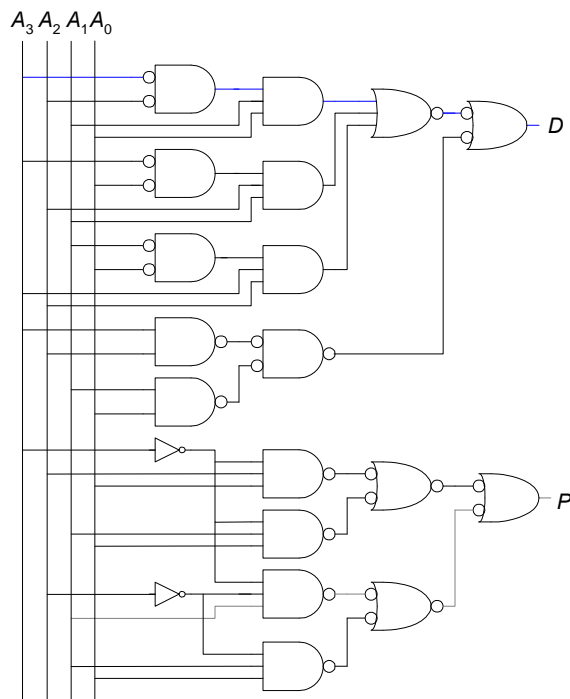
$$\begin{aligned} t_{cd} &= 2t_{cd_NAND2} + t_{cd_NOR2} \\ &= [2(15) + 25]\ ps \\ &= \mathbf{55\ ps} \end{aligned}$$

Exercise 2.45

$$\begin{aligned} t_{pd} &= t_{pd_NOT} + t_{pd_AND3} \\ &= 15 \text{ ps} + 40 \text{ ps} \\ &= \mathbf{55 \text{ ps}} \\ t_{cd} &= t_{cd_AND3} \\ &= \mathbf{30 \text{ ps}} \end{aligned}$$



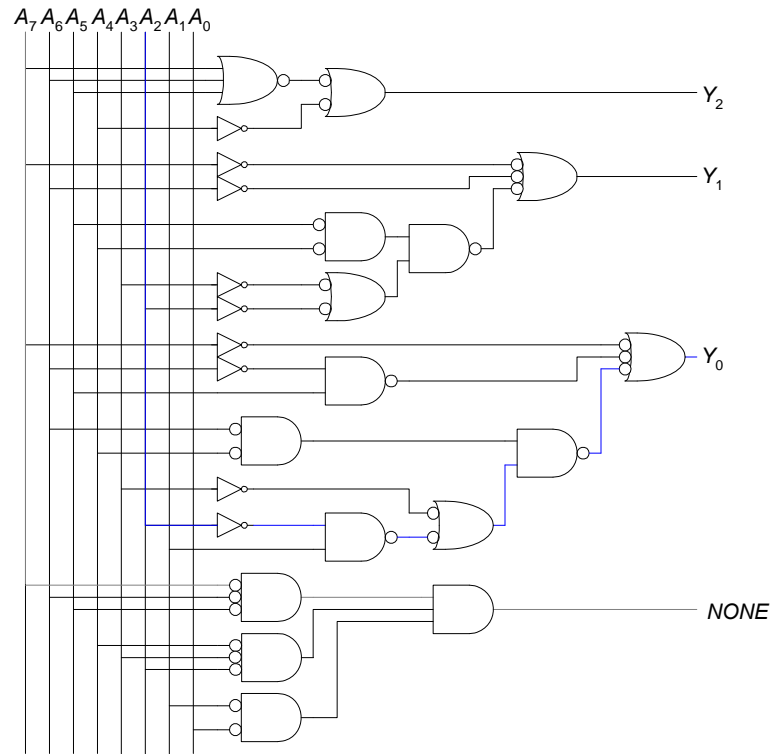
Exercise 2.46



$$\begin{aligned}
 t_{pd} &= t_{pd_NOR2} + t_{pd_AND3} + t_{pd_NOR3} + t_{pd_NAND2} \\
 &= [30 + 40 + 45 + 20] \text{ ps} \\
 &= \mathbf{135 \text{ ps}}
 \end{aligned}$$

$$\begin{aligned}
 t_{cd} &= 2t_{cd_NAND2} + t_{cd_OR2} \\
 &= [2(15) + 30] \text{ ps} \\
 &= \mathbf{60 \text{ ps}}
 \end{aligned}$$

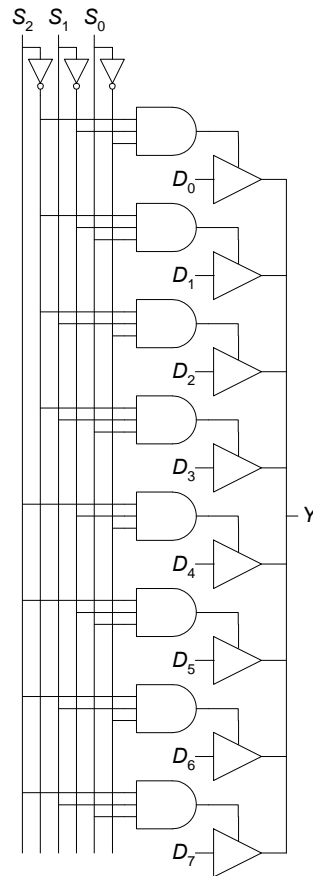
Exercise 2.47



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

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

Exercise 2.48



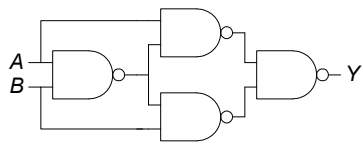
$$t_{pd_dy} = t_{pd_TRI_AY} \\ = \mathbf{50 \text{ ps}}$$

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

$$t_{pd_sy} = t_{pd_NOT} + t_{pd_AND3} + t_{pd_TRI_SY} \\ = [30 + 80 + 35] \text{ ps} \\ = \mathbf{145 \text{ ps}}$$

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

Question 2.1



Question 2.2

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

Y

$A_{3:2}$

$A_{1:0}$

	00	01	11	10
00	X	0	1	1
01	1	1	X	0
11	1	1	X	0
10	0	0	X	1

A_3

A_0

Y

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

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.

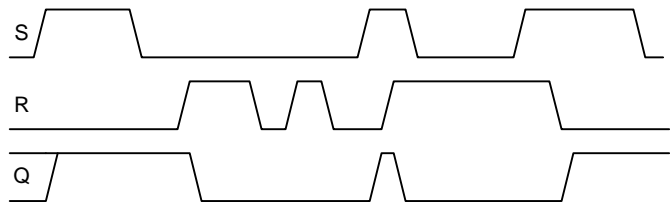
(c) The NAND gate by itself is universal. A NAND gate with its inputs tied together performs the NOT operation. A NAND gate with a NOT gate on its output performs AND. And a NAND gate with NOT gates on its inputs performs OR. Thus, a NAND gate is equivalent to the set {AND, OR, NOT} and is universal.

Question 2.5

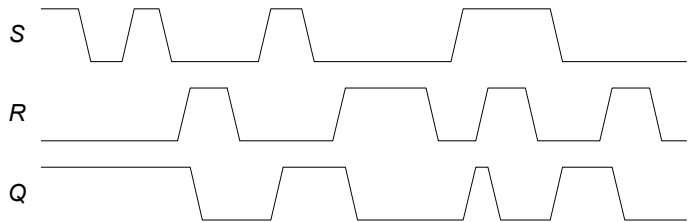
A circuit's contamination delay might be less than its propagation delay because the circuit may operate over a range of temperatures and supply voltages, for example, 3–3.6 V for LVCMOS (low voltage CMOS) chips. As temperature increases and voltage decreases, circuit delay increases. Also, the circuit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

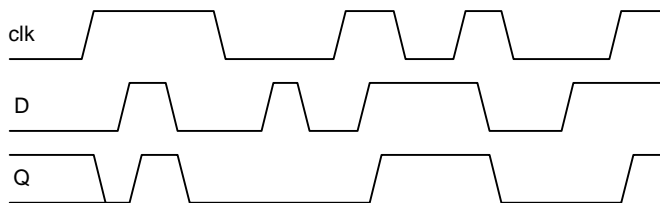
Exercise 3.1



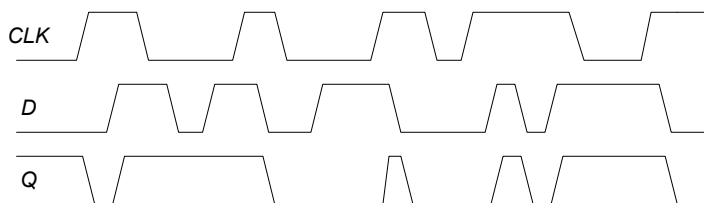
Exercise 3.2



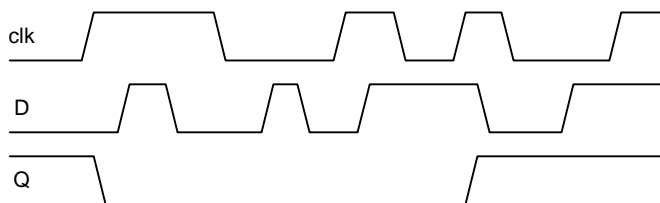
Exercise 3.3



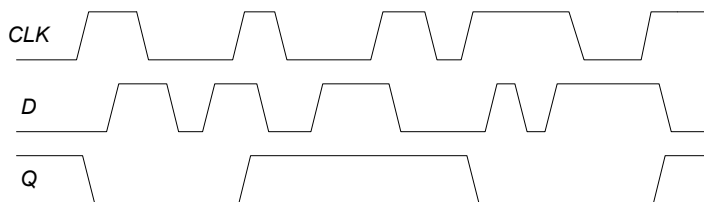
Exercise 3.4



Exercise 3.5



Exercise 3.6



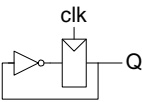
Exercise 3.7

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.

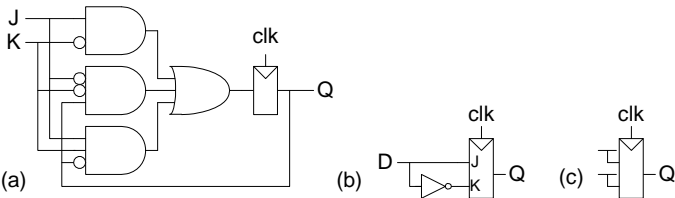
Exercise 3.8

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

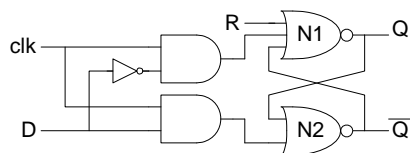


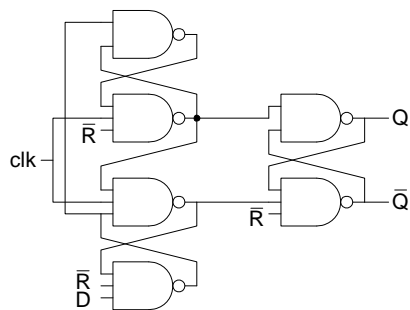
Exercise 3.11

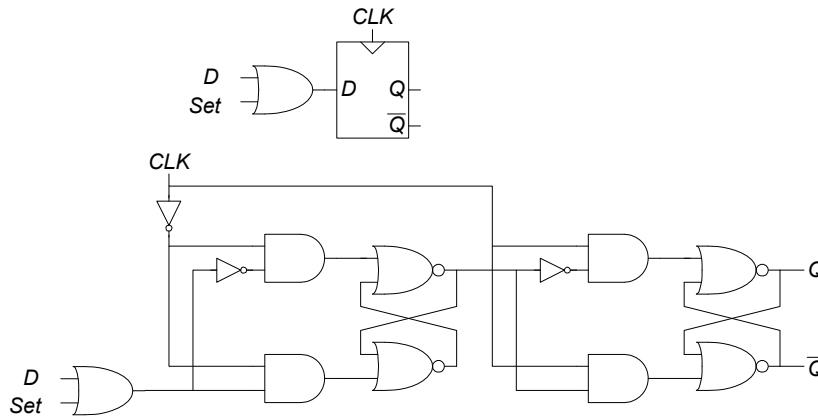
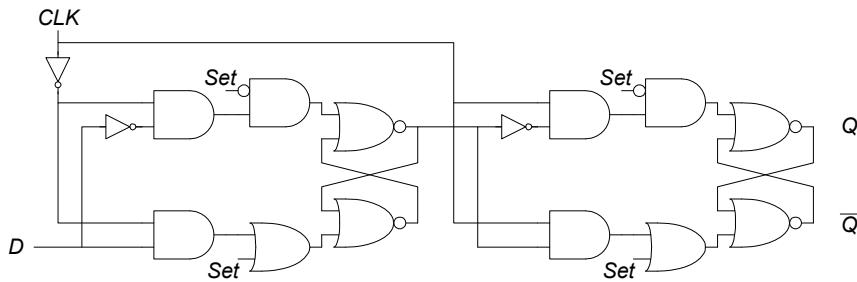
If A and B have the same value, C takes on that value. Otherwise, C retains its old value.

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

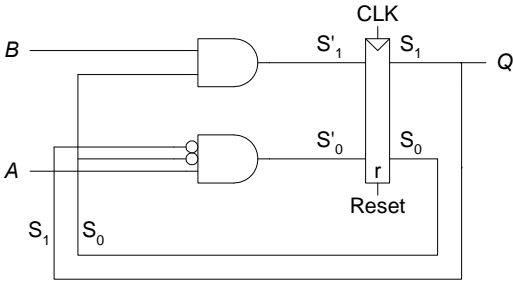
TABLE 3.2 State transition table with binary encodings for Exercise 3.22

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

$$S'_1 = S_0B$$
$$S'_0 = \overline{S_1}\overline{S_0}A$$

$$Q = S_1$$



Exercise 3.23

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

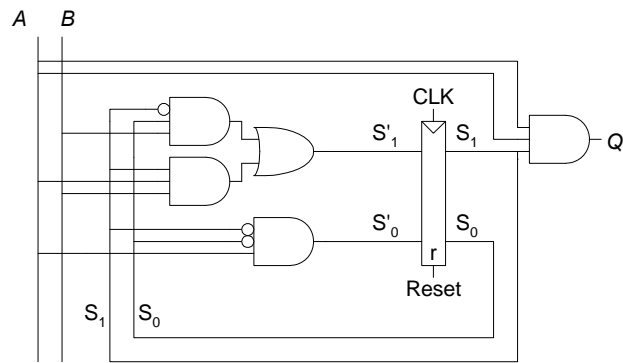
state	encoding $s_{1:0}$
S0	00
S1	01
S2	10

TABLE 3.4 State encoding for Exercise 3.23

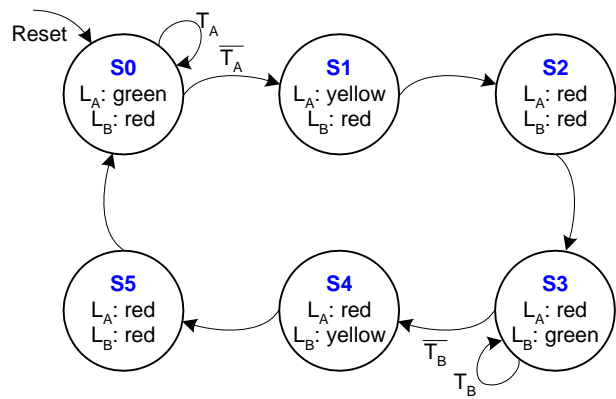
current state		inputs		next state		output
s_1	s_0	a	b	s'_1	s'_0	q
0	0	0	X	0	0	0
0	0	1	X	0	1	0
0	1	X	0	0	0	0
0	1	X	1	1	0	0
1	0	1	1	1	0	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0

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

$$S_1 = \overline{S_1}S_0B + S_1AB$$
$$S_0 = \overline{S_1}\overline{S_0}A$$
$$Q' = S_1AB$$



Exercise 3.24



state	encoding s _{1:0}
S ₀	000
S ₁	001
S ₂	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

$$S_2^{\prime} = S_2 \oplus S_1$$
$$S_1^{\prime} = \overline{S_1} S_0$$
$$S_0^{\prime} = \overline{S_1} \overline{S_0} (\overline{S_2} \overline{t_a} + S_2 \overline{t_b})$$

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

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

(3.1)

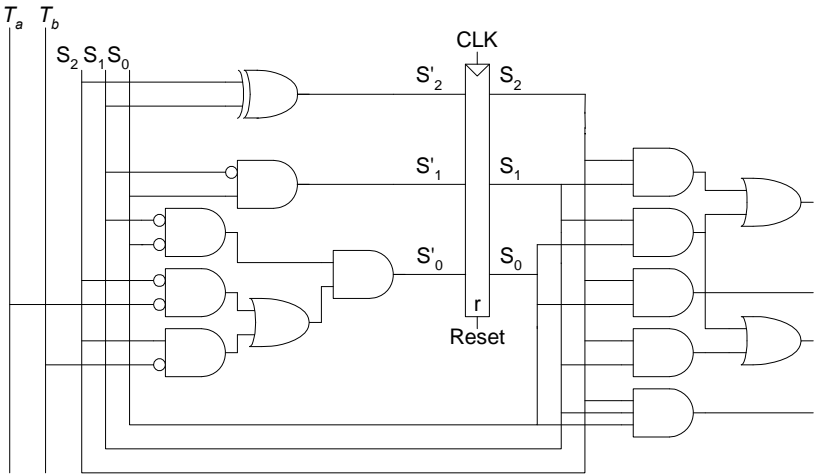
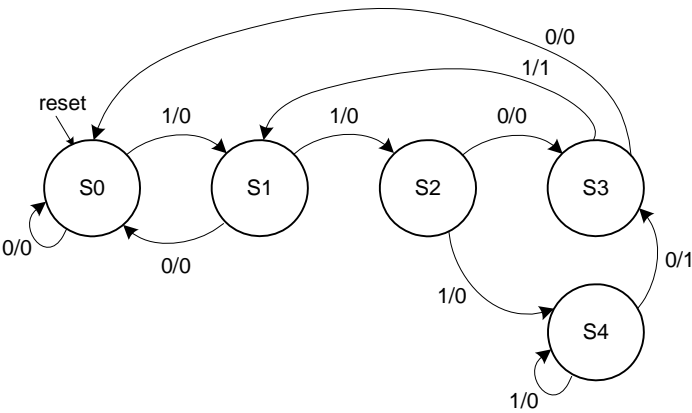


FIGURE 3.1 State machine circuit for traffic light controller for Exercise 3.21

Exercise 3.25



state	encoding s_1s_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

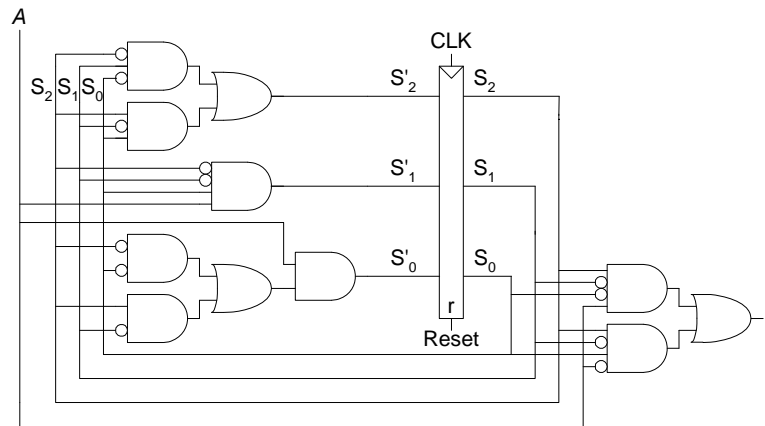
TABLE 3.10 Combined state transition and output table with binary encodings for Exercise 3.25

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

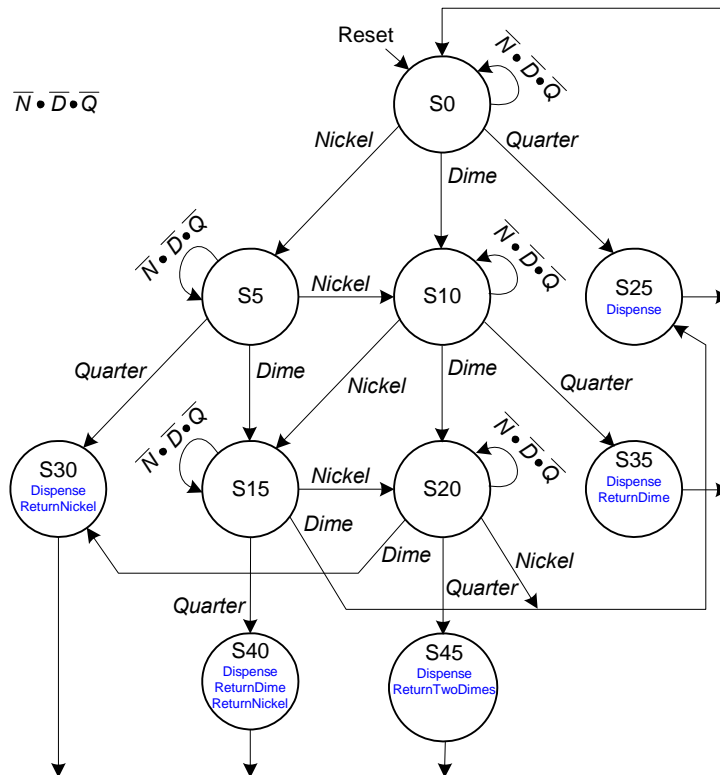
$$S'_1 = \overline{S_2}\overline{S_1}S_0A$$

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

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



Exercise 3.26



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

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'
	<i>nickel</i>	<i>dime</i>	<i>quarter</i>	
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'
	<i>nickel</i>	<i>dime</i>	<i>quarter</i>	
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'
	<i>nickel</i>	<i>dime</i>	<i>quarter</i>	
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 <i>s</i>	inputs			next state <i>s'</i>
	<i>nickel</i>	<i>dime</i>	<i>quarter</i>	
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

$$s_9 = s_6Q$$

$$s_8 = s_5Q$$

$$S_7 = S_2Q$$

$$S_6 = S_2D + S_5N + S_6\overline{ND}\overline{Q}$$

$$S_5 = S_1D + S_2N + S_5NDQ$$

$$S_4 = S_1Q + S_6D$$

$$S_3 = S_0Q + S_5D + S_6N$$

$$S_2 = S_0D + S_1N + S_2\overline{ND}\overline{Q}$$

$$S_1 = S_0N + S_1NDQ$$

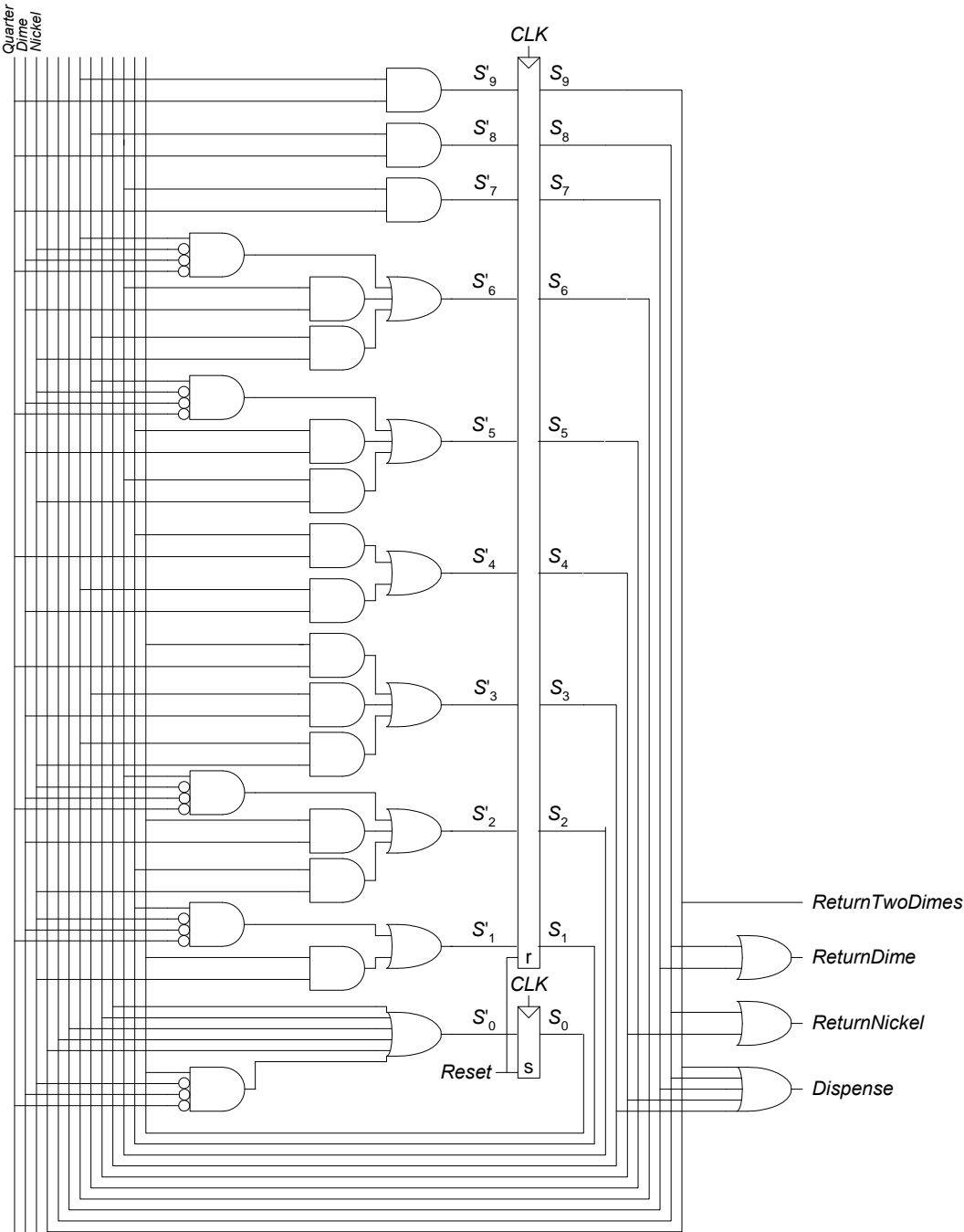
$$S_0 = S_0\overline{ND}\overline{Q} + S_3 + S_4 + S_7 + S_8 + S_9$$

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

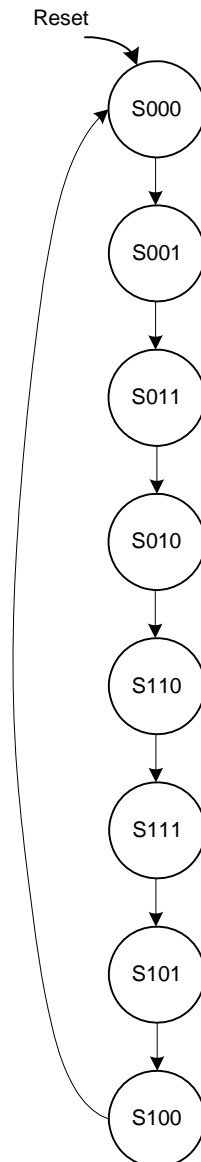
$$ReturnNickel = S_4 + S_8$$

$$ReturnDime = S_7 + S_8$$

$$ReturnTwoDimes = S_9$$



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

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

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

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

$$Q_2 = S_2$$

$$Q_1 = S_1$$

$$Q_0 = S_0$$

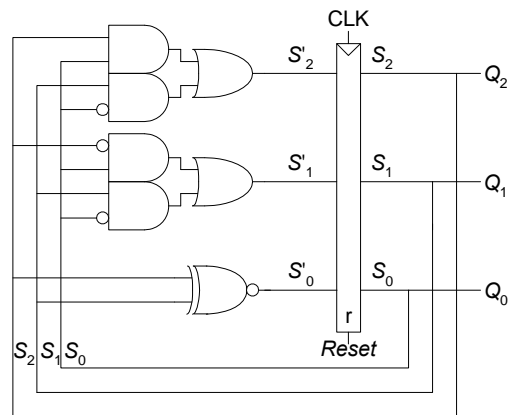


FIGURE 3.5 Hardware for Gray code counter FSM for Exercise 3.27

Exercise 3.28

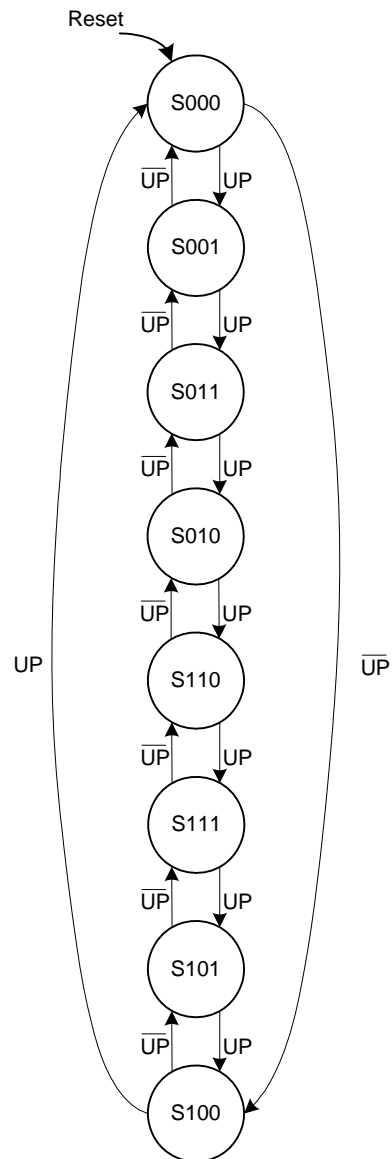


FIGURE 3.6 State transition diagram for Exercise 3.28

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

$$\begin{aligned} S_2 &= UPS_1\overline{S_0} + \overline{UP}\overline{S_1}\overline{S_0} + S_2S_0 \\ S_1 &= S_1\overline{S_0} + UP\overline{S_2}S_0 + \overline{UP}S_2S_1 \\ S_0 &= UP \oplus S_2 \oplus S_1 \\ Q_2 &= S_2 \\ Q_1 &= S_1 \\ Q_0 &= S_0 \end{aligned}$$

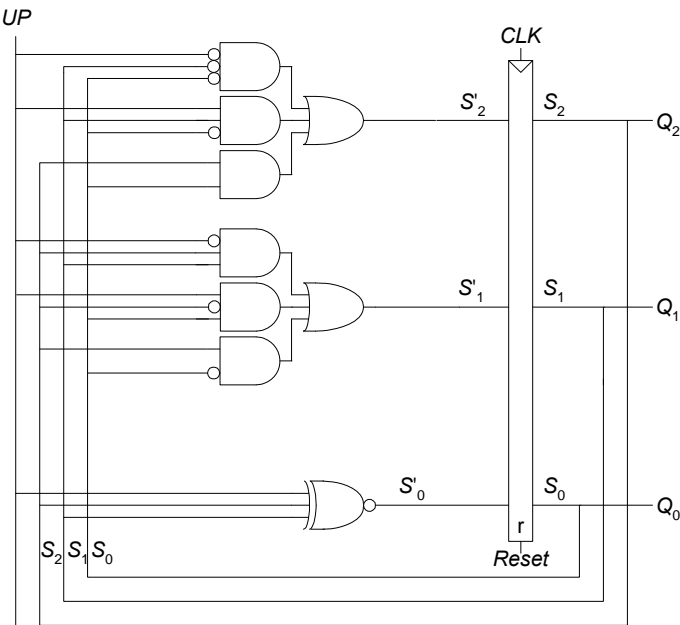


FIGURE 3.7 Finite state machine hardware for Exercise 3.28

Exercise 3.29

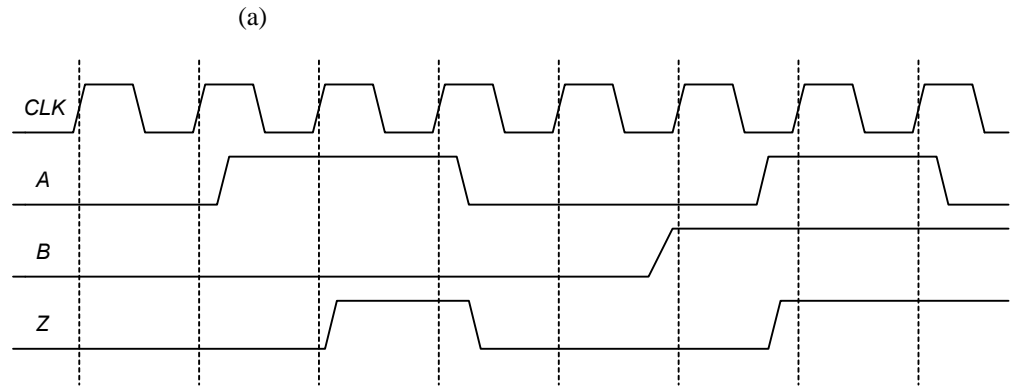


FIGURE 3.8 Waveform showing Z output for Exercise 3.29

(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)

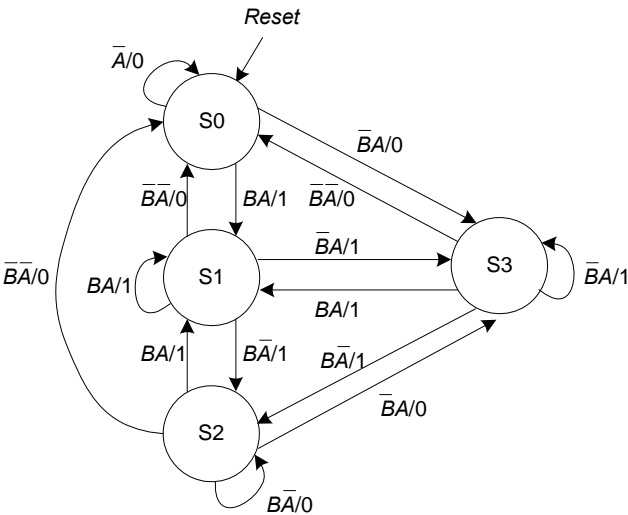


FIGURE 3.9 State transition diagram for Exercise 3.29

(Note: another viable solution would be to allow the state to transition from S0 to S1 on $\overline{B}\overline{A}/0$. 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 z
	b	a		
10	1	1	01	1
11	0	0	00	0
11	0	1	11	1
11	1	0	10	1
11	1	1	01	1

TABLE 3.15 State transition table for Exercise 3.29

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

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

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

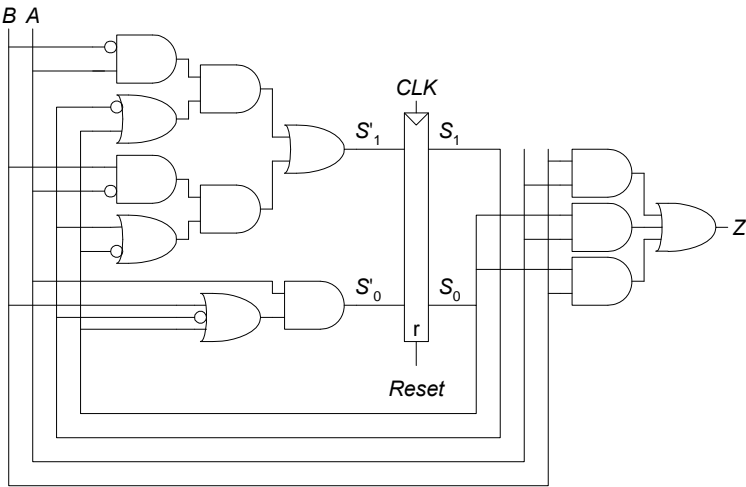


FIGURE 3.10 Hardware for FSM of Exercise 3.26

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

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

Exercise 3.30

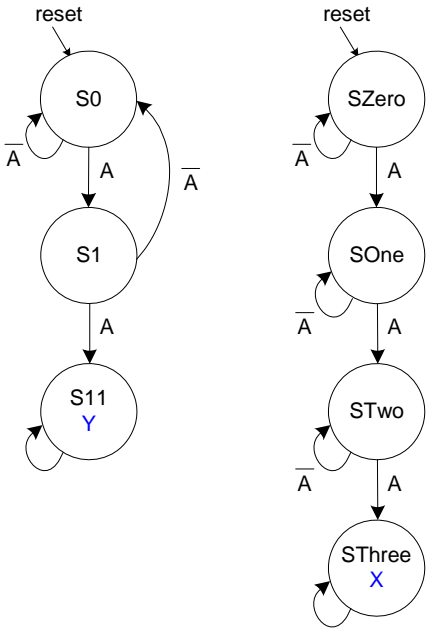


FIGURE 3.11 Factored state transition diagram for Exercise 3.30

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

$$\begin{aligned} S_1 &= S_0(S_1 + A) \\ S_0 &= \overline{S_1}A + S_0(S_1 + A) \\ Y &= S_1 \\ \\ T_1 &= T_1 + T_0A \\ T_0 &= A(T_1 + \overline{T_0}) + \overline{A}T_0 + T_1T_0 \\ X &= T_1T_0 \end{aligned}$$

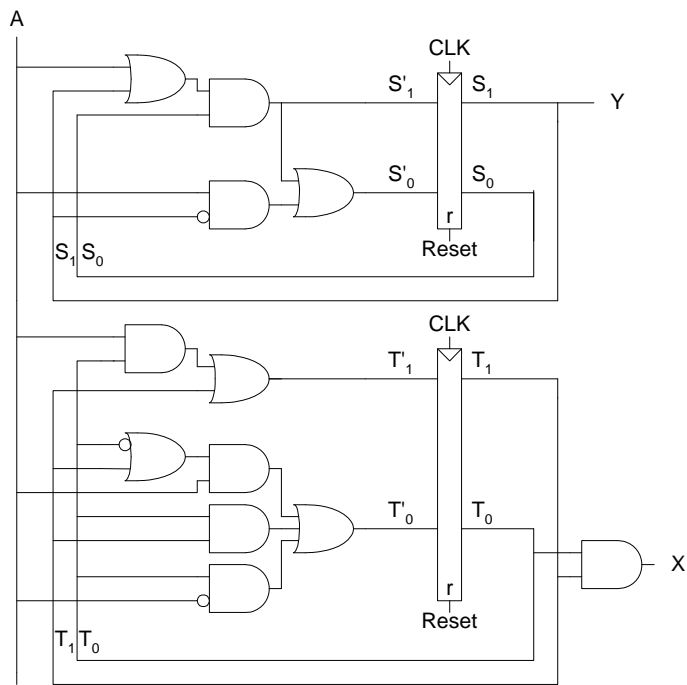


FIGURE 3.12 Finite state machine hardware for Exercise 3.30

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

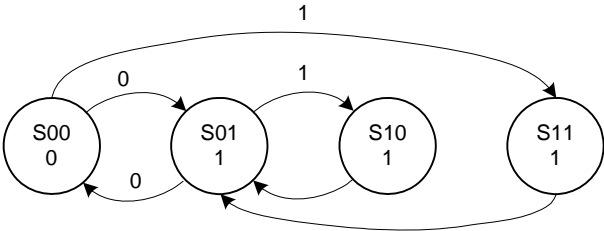
TABLE 3.18 State transition table with binary encodings for Exercise 3.31

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

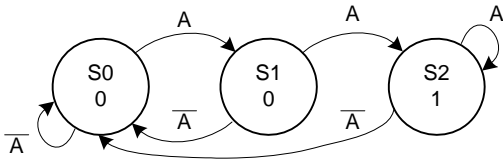


FIGURE 3.13 State transition diagram for Exercise 3.32

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:

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

Next, we calculate the cycle time:

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

$$f = 1 / 430 \text{ ps} = \mathbf{2.33 \text{ GHz}}$$

(b)

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

Thus,

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

(c)

Exercise 3.34

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

Exercise 3.35

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

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

Exercise 3.36

1.138 ns

Exercise 3.37

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

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

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

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

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

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

Exercise 3.38

(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,

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

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

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

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:

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

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

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

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 T_0 (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

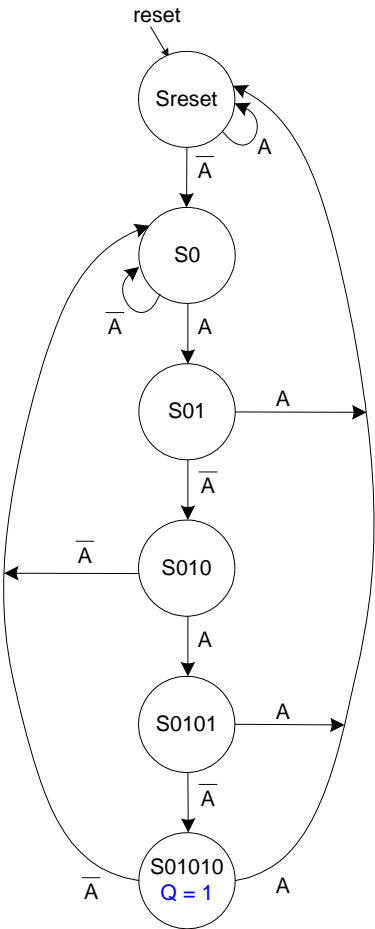


FIGURE 3.15 State transition diagram for Question 3.1

current state $s_5;0$	input	next state $s';s;0$
	a	
000001	0	000010
000001	1	000001

TABLE 3.21 State transition table for Question 3.1

current state <i>s</i> _{5:0}	input <i>a</i>	next state <i>s'</i> _{5:0}
000010	0	000010
000010	1	000100
000100	0	001000
000100	1	000001
001000	0	000010
001000	1	010000
010000	0	100000
010000	1	000001
100000	0	000010
100000	1	000001

TABLE 3.21 State transition table for Question 3.1

$S_5' = S_4A$
 $S_4' = S_3A$
 $S_3' = S_2A$
 $S_2' = S_1A$
 $S_1' = A(S_1 + S_3 + S_5)$
 $S_0' = A(S_0 + S_2 + S_4 + S_5)$
 $Q = S_5$

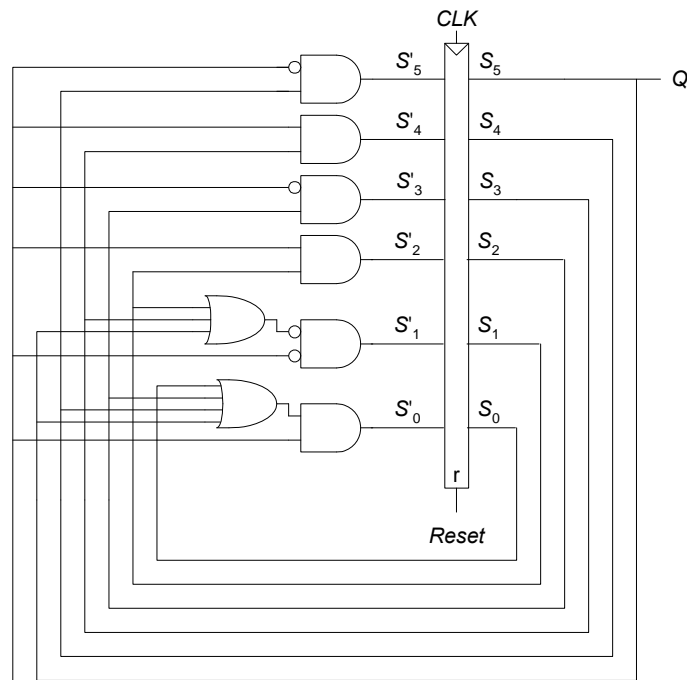


FIGURE 3.16 Finite state machine hardware for Question 3.1

Question 3.2

The FSM should output the value of A until after the first 1 is received. It then should output the inverse of A . For example, the 8-bit two's complement of the number 6 (00000110) is (1111010). 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 1111010.

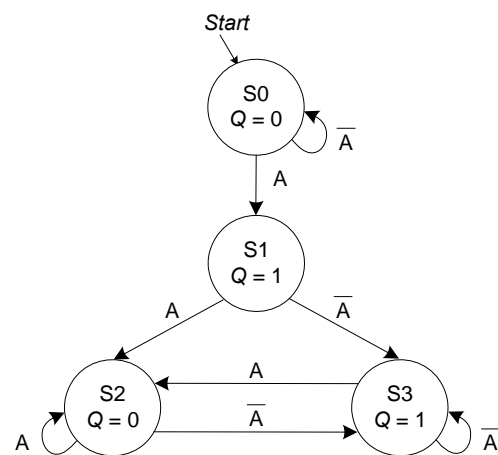


FIGURE 3.17 State transition diagram for Question 3.2

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

$$S'_1 = S_1 + S_0$$
$$S'_0 = A \oplus (S_1 + S_0)$$
$$Q = S_0$$

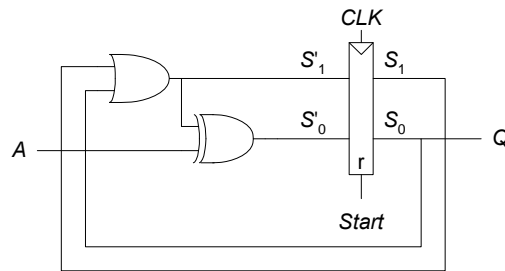


FIGURE 3.18 Finite state machine hardware for Question 3.2

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

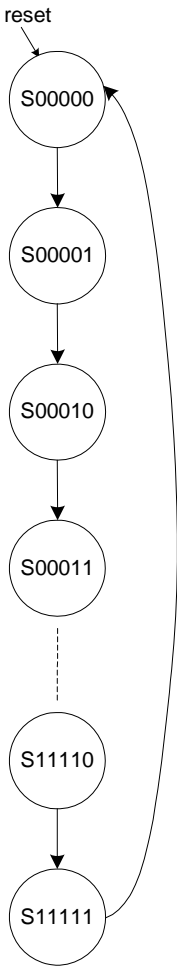


FIGURE 3.19 State transition diagram for Question 3.4

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

$$s'_4 = s_4 \oplus s_3 s_2 s_1 s_0$$

$$s'_3 = s_3 \oplus s_2 s_1 s_0$$

$$s'_2 = s_2 \oplus s_1 s_0$$

$$s'_1 = s_1 \oplus s_0$$

$$s'_0 = \overline{s_0}$$

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

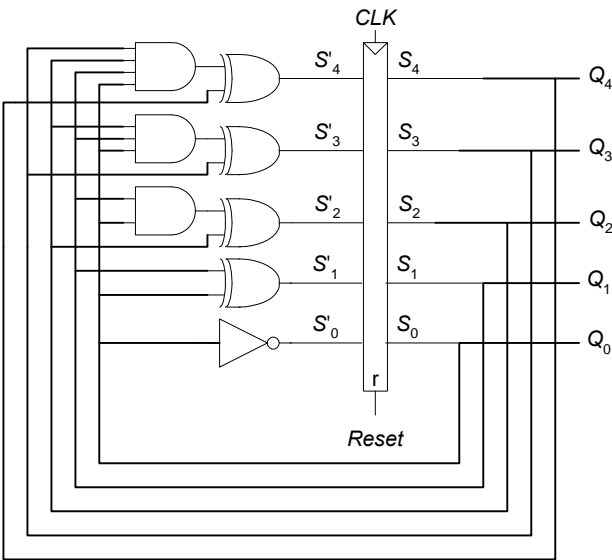


FIGURE 3.20 Finite state machine hardware for Question 3.4

Question 3.5

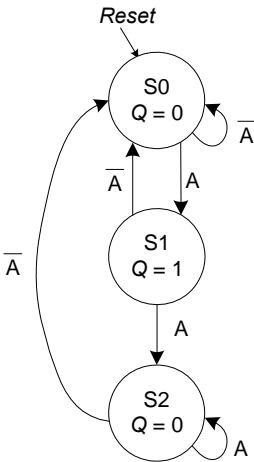


FIGURE 3.21 State transition diagram for edge detector circuit of Question 3.5

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

$S_1 = AS_1$

$S_0 = AS_1S_0$

$Q = S_1$

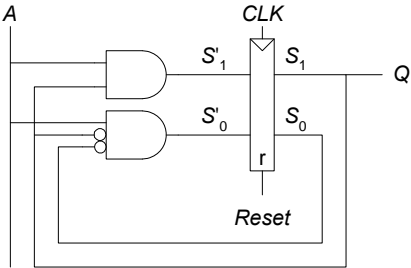


FIGURE 3.22 Finite state machine hardware for Question 3.5

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, $t_{\text{setup}} + t_{\text{pcq}}$. (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 text-book) 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.

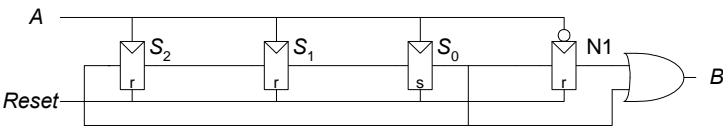


FIGURE 3.23 Hardware for Question 3.8

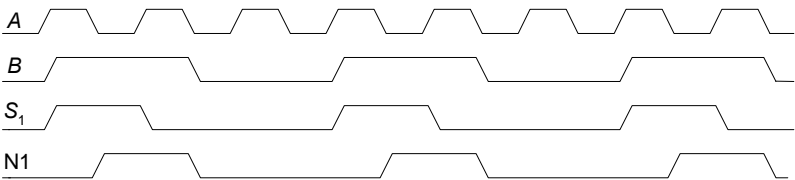


FIGURE 3.24 Waveforms for Question 3.8

Question 3.9

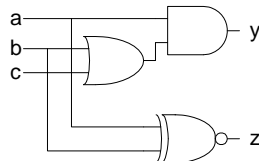
Without the added buffer, the propagation delay through the logic, t_{pd} , must be less than or equal to $T_c - (t_{pcq} + t_{setup})$. However, if you add a buffer to the clock input of the receiver, the clock arrives at the receiver later. The earliest that the clock edge arrives at the receiver is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

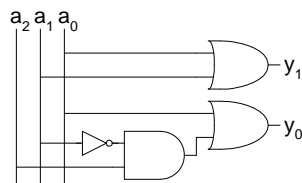
Note: the HDL files given in the following solutions are available on the textbook's companion website at:

<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'bxx) 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 tarray is array(MEMSIZE downto 0) of
        STD_LOGIC_VECTOR(4 downto 0);
    signal testvectors: tarray;
    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 tarray is array(MEMSIZE downto 0) of
        STD_LOGIC_VECTOR(10 downto 0);
    signal testvectors: tarray;
    shared variable vectornum, errors: integer;
begin
    -- instantiate device under test
    dut: seven_seg_decoder port map(data, s);

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

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

```

(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 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 tarray is array(MEMSIZE downto 0) of
    STD_LOGIC_VECTOR(10 downto 0);
  signal testvectors: tarray;
  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';
  end process;
end;

```

```

    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 process;
    end;
  end;
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;

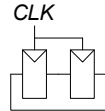
entity mux4 is
    generic(width: integer := 4);
    port(d0,
         d1,
         d2,
         d3: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s: in  STD_LOGIC_VECTOR(1 downto 0);
         y: out STD_LOGIC_VECTOR(width-1 downto 0));
end;

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'b000000001: y = 8'b000000001;
        default:      y = 8'b000000000;
    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****VHDL**

```

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

```

```

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

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

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

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'b0000001?: z = 3'b001;
        8'b0000010?: z = 3'b001;
        8'b0001001?: z = 3'b001;
        8'b0010001?: z = 3'b001;
        8'b0100001?: z = 3'b001;
        8'b1000001?: z = 3'b001;
        8'b0000011?: 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;
    endcase
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 entity;

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 "0000010?" => 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 "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 "001011---" => z <= "011";
            when "010011---" => z <= "011";
            when "100011---" => 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'b00000000;
            1: y = 7'b00000001;
            2: y = 7'b00000011;
            3: y = 7'b00000111;
            4: y = 7'b00001111;
            5: y = 7'b00011111;
            6: y = 7'b00111111;
            7: y = 7'b01111111;
        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 <= "00000000";
            when "001" => y <= "00000001";
            when "010" => y <= "00000011";
            when "011" => y <= "00000111";
            when "100" => y <= "00001111";
            when "101" => y <= "00011111";
            when "110" => y <= "00111111";
            when "111" => y <= "01111111";
            when others => y <= "00000000";
        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
```

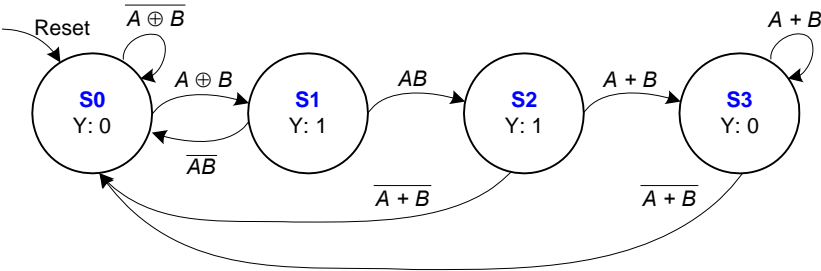
VHDL

```
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

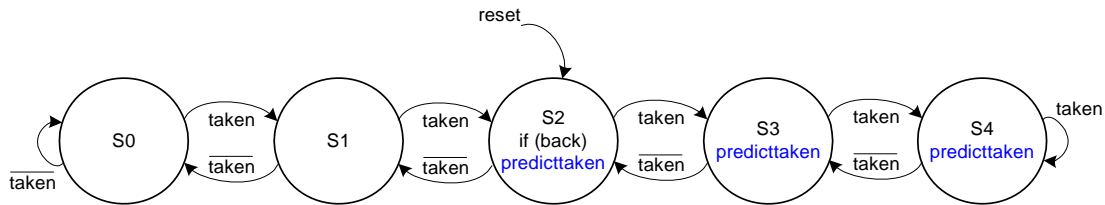


FIGURE 4.1 State transition diagram for Exercise 4.25

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

architecture synth of daughterterfsm 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
endmodule

```

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;
end architecture synth;

```

*(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

```

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;
    end case;
  end process;
end architecture synth;

```

(continued on next page)

*(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;
end;

```

*(continued from previous page)***VHDL**

```

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

```

Option 2**SystemVerilog**

```

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

  logic aprev;

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

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

```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

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

```

Exercise 4.40

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 <= S0;
                        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;

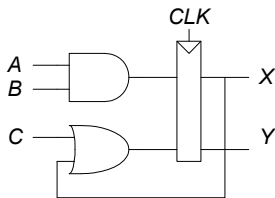
entity syncbad is
    port(clk: in  STD_LOGIC;
          d:  in  STD_LOGIC;
          q:  out STD_LOGIC);
end;

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;

```

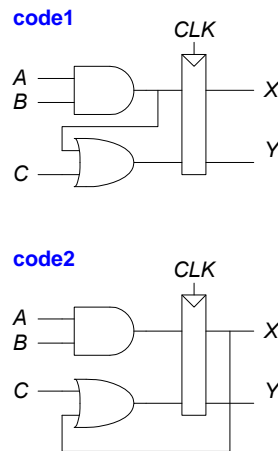
Exercise 4.48

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;
    end process;
end;

```

```

    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:

$$t_{CLA} = t_{\text{pg}} + t_{\text{pg_block}} + \left(\frac{N}{k} - 1\right)t_{\text{AND_OR}} + kt_{\text{FA}} \quad 150$$

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

(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:

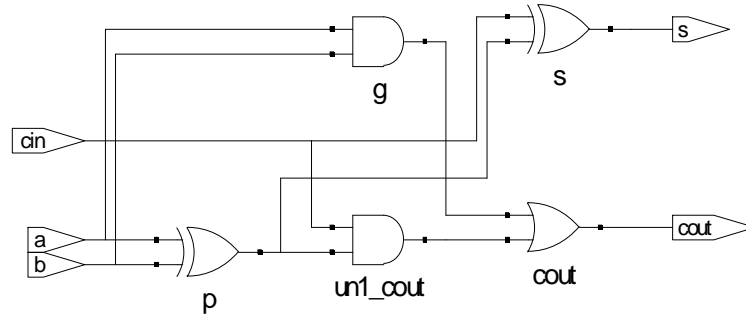


FIGURE 5.1 Full adder implementation

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:

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

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

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

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,

$$P_{\text{dynamic_ripple}} = \frac{1}{2} C V_{DD}^2 f = \frac{1}{2} (6.4 \text{ pF}) (1.2 \text{ V})^2 (100 \text{ MHz}) = 0.461 \text{ mW}$$

$$P_{\text{dynamic_CLA}} = \frac{1}{2} C V_{DD}^2 f = \frac{1}{2} (12.48 \text{ pF}) (1.2 \text{ V})^2 (100 \text{ MHz}) = 0.899 \text{ mW}$$

	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 prefixaddl6(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 prefixaddl6 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 prefixaddl6 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 previous page)***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

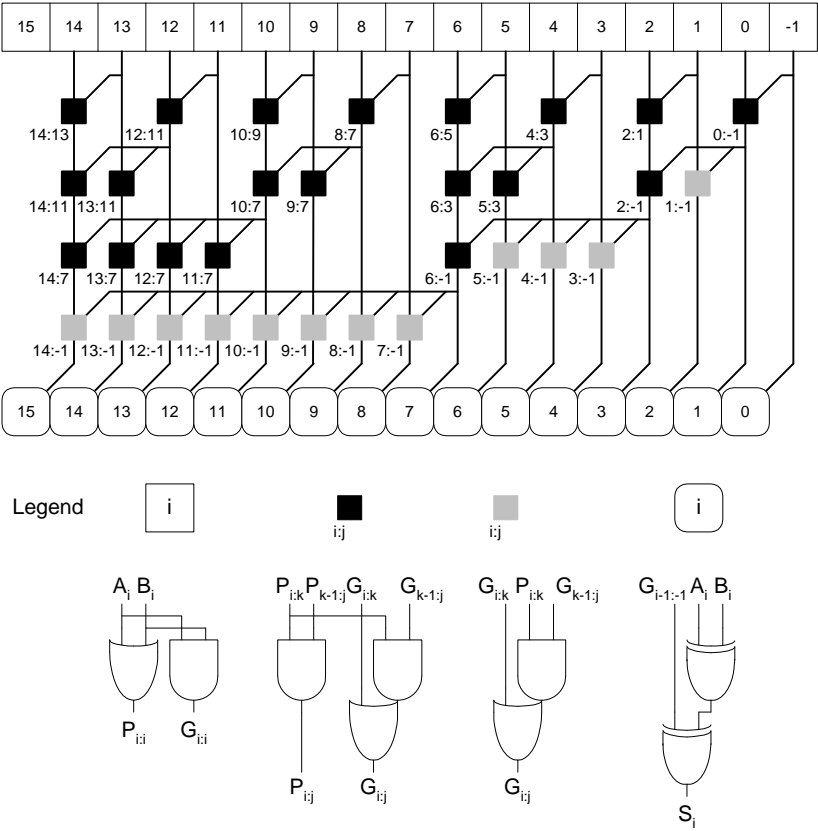


FIGURE 5.2 16-bit prefix adder with “gray cells”

Exercise 5.6

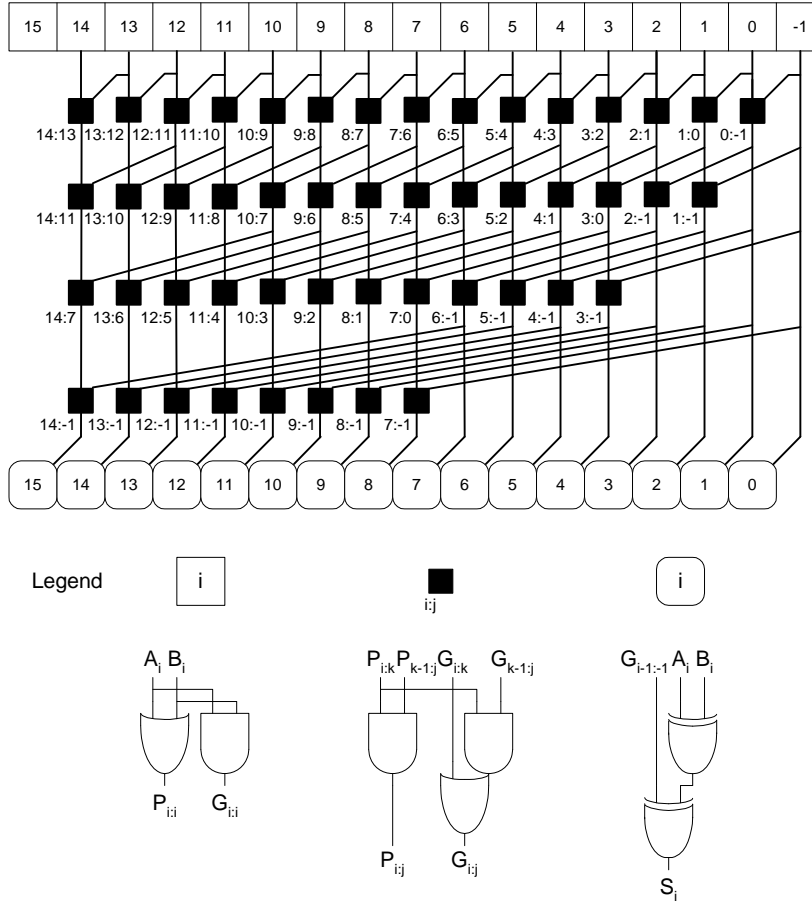


FIGURE 5.3 Schematic of a 16-bit Kogge-Stone adder

Exercise 5.7

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

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

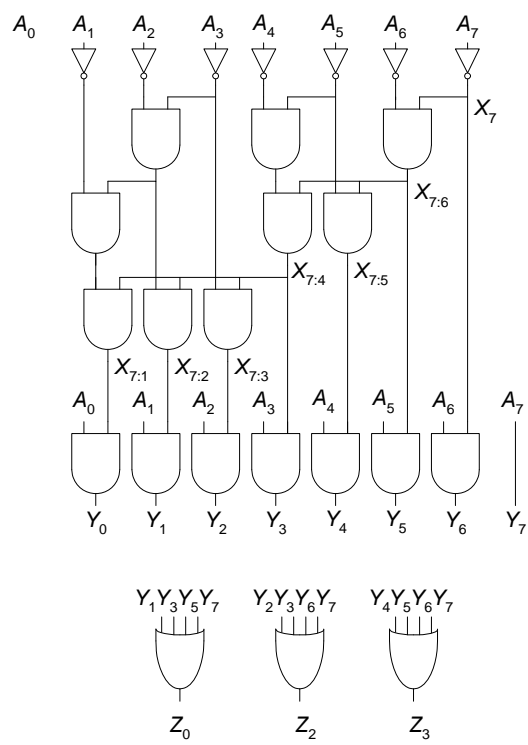


FIGURE 5.4 8-input priority encoder

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));

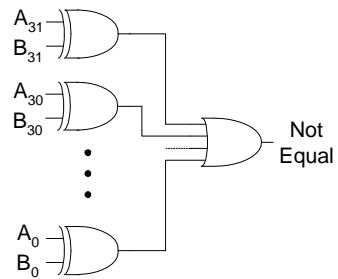
    -- row of OR gates
    z <= ( (y(7) or y(6) or y(5) or y(4)) &
          (y(7) or y(6) or y(3) or y(2)) &
          (y(1) or y(3) or y(5) or y(7)) );

end;

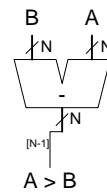
```

Exercise 5.8

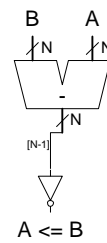
(a)



(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: $-3 - 6 = 1101 - 0110 = 1101 + 1001 + 1 = 01112 (= -7, \text{ but overflow occurred} - \text{ the result should be } -9)$. 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.

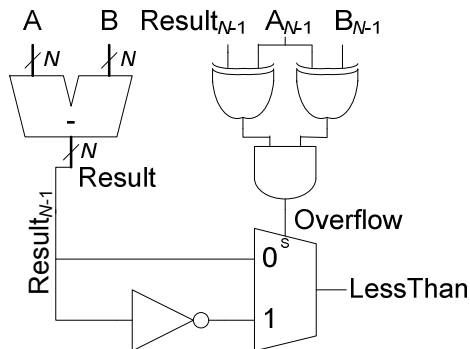
(c) In the general, the N -bit signed comparator of Figure 5.12 operates incorrectly upon overflow.

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, b,
          input  logic [1:0] ALUControl,
          output logic [31:0] Result);

    logic [31:0] condinvb;
    logic [32:0] sum;

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

    always_comb
        casex (ALUControl[1:0])
```

```

    2'b0?: Result = sum;
    2'b10: Result = a & b;
    2'b11: Result = a | b;
endcase

```

```
endmodule
```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity alu is
    port(a, b:          in  STD_LOGIC_VECTOR(31 downto 0);
         ALUControl: in  STD_LOGIC_VECTOR(1 downto 0);
         Result:        buffer STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of alu is
    signal condinvb: STD_LOGIC_VECTOR(31 downto 0);
    signal sum:      STD_LOGIC_VECTOR(32 downto 0);
begin
    condinvb <= not b when ALUControl(0) else b;
    sum <= ('0', a) + ('0', condinvb) + ALUControl(0);

    process(all) begin
        case? ALUControl(1 downto 0) is
            when "0-" => result <= sum(31 downto 0);
            when "10" => result <= a and b;
            when "11" => result <= a or b;
            when others => result <= (others => '-');
        end case?;
    end process;
end;

```

Exercise 5.12

SystemVerilog

```

module alu(input  logic [31:0] a, b,
           input  logic [1:0] ALUControl,
           output logic [31:0] Result,
           output logic [3:0] ALUFlags);

    logic      neg, zero, carry, overflow;
    logic [31:0] condinvb;
    logic [32:0] sum;

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

    always_comb
        casex (ALUControl[1:0])
            2'b0?: Result = sum;
            2'b10: Result = a & b;
            2'b11: Result = a | b;
        endcase

    assign neg      = Result[31];
    assign zero     = (Result == 32'b0);
    assign carry    = (ALUControl[1] == 1'b0) & sum[32];
    assign overflow = (ALUControl[1] == 1'b0) &
        ~(a[31] ^ b[31] ^ ALUControl[0]) &
        (a[31] ^ sum[31]);

```



```

    assign ALUFlags = {neg, zero, carry, overflow};
endmodule

```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity alu is
    port(a, b:          in      STD_LOGIC_VECTOR(31 downto 0);
          ALUControl: in      STD_LOGIC_VECTOR(1 downto 0);
          Result:       buffer STD_LOGIC_VECTOR(31 downto 0);
          ALUFlags:     out     STD_LOGIC_VECTOR(3 downto 0));
end;

architecture behave of alu is
    signal condinvb: STD_LOGIC_VECTOR(31 downto 0);
    signal sum:      STD_LOGIC_VECTOR(32 downto 0);
    signal neg, zero, carry, overflow: STD_LOGIC;
begin
    condinvb <= not b when ALUControl(0) else b;
    sum <= ('0', a) + ('0', condinvb) + ALUControl(0);

    process(all) begin
        case? ALUControl(1 downto 0) is
            when "0-" => result <= sum(31 downto 0);
            when "10" => result <= a and b;
            when "11" => result <= a or b;
            when others => result <= (others => '-');
        end case?;
    end process;

    neg      <= Result(31);
    zero     <= '1' when (Result = 0) else '0';
    carry    <= (not ALUControl(1)) and sum(32);
    overflow <= (not ALUControl(1)) and
                (not (a(31) xor b(31) xor ALUControl(0))) and
                (a(31) xor sum(31));
    ALUFlags <= (neg, zero, carry, overflow);
end;

```

Exercise 5.13

SystemVerilog

```

module testbench();
    logic clk;
    logic [31:0] a, b, y, y_expected;
    logic [1:0] ALUControl;

    logic [31:0] vectornum, errors;
    logic [99:0] testvectors[10000:0];

    // instantiate device under test
    alu dut(a, b, ALUControl, y);

    // generate clock
    always begin
        clk = 1; #50; clk = 0; #50;
    end

    // at start of test, load vectors
    initial begin
        $readmemh("ex5.13_alu.tv", testvectors);
        vectornum = 0; errors = 0;
    end
end

```

```

// apply test vectors at rising edge of clock
always @(posedge clk)
begin
    #1;
    ALUControl = testvectors[vectornum][97:96];
    a = testvectors[vectornum][95:64];
    b = testvectors[vectornum][63:32];
    y_expected = testvectors[vectornum][31:0];
end

// check results on falling edge of clock
always @(negedge clk)
begin
    if (y != y_expected) begin
        $display("Error in vector %d", vectornum);
        $display(" Inputs : a = %h, b = %h, ALUControl = %b", a, b, ALUControl);
        $display(" Outputs: y = %h (%h expected)",
            y, y_expected);
        errors = errors+1;
    end
    vectornum = vectornum + 1;
    if (testvectors[vectornum][0] === 1'bx) begin
        $display("%d tests completed with %d errors", vectornum, errors);
        $stop;
    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 testbench is -- no inputs or outputs
end;

architecture sim of testbench is
    component alu
        port(a, b:          in      STD_LOGIC_VECTOR(31 downto 0);
              ALUControl: in      STD_LOGIC_VECTOR(1 downto 0);
              Result:       buffer STD_LOGIC_VECTOR(31 downto 0));
    end component;
    signal a, b, Result, Result_expected: STD_LOGIC_VECTOR(31 downto 0);
    signal ALUControl: STD_LOGIC_VECTOR(1 downto 0);
    signal clk, reset: STD_LOGIC;
    constant MEMSIZE: integer := 99;
    type tarray is array(MEMSIZE downto 0) of STD_LOGIC_VECTOR(99 downto 0);
    shared variable testvectors: tarray;
    shared variable vectornum, errors: integer;
begin
    -- instantiate device under test
    dut: alu port map(a, b, ALUControl, Result);

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

    -- at start of test, pulse reset

```

```

process begin
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
end process;

-- run tests
-- at start of test, load vectors
process is
    file tv: TEXT;
    variable i, index, count: integer;
    variable L: line;
    variable ch: character;
    variable readvalue: integer;
begin
    -- read file of test vectors
    i := 0;
    index := 0;
    FILE_OPEN(tv, "ex5.13_alu.tv", READ_MODE);
    report "Opening file\n";
    while (not endfile(tv)) loop
        readline(tv, L);
        readvalue := 0;
        count := 3;
        for i in 1 to 28 loop
            read(L, ch);
            report "Line: " & integer'image(index) & " i = " &
                integer'image(i) & " char = " &
                character'image(ch)
                severity error;

            if '0' <= ch and ch <= '9' then
                readvalue := readvalue*16 + character'pos(ch)
                    - character'pos('0');
            elsif 'a' <= ch and ch <= 'f' then
                readvalue := readvalue*16 + character'pos(ch)
                    - character'pos('a')+10;
            else report "Format error on line " &
                integer'image(index) & " i = " &
                integer'image(i) & " char = " &
                character'image(ch)
                severity error;
            end if;

            -- load vectors
            -- assign first 4 bits (will be used for ALUControl)
            if (i = 1) then
                testvectors(index)( 99 downto 96) := CONV_STD_LOGIC_VECTOR(readvalue, 4);
                count := count - 1;
                readvalue := 0; -- reset readvalue

                -- assign a, b, and Result (in testvectors) in
                -- 32-bit increments
                elsif ((i = 10) or (i = 19) or (i = 28)) then
                    testvectors(index)( (count*32 + 31) downto (count*32)) :=
CONV_STD_LOGIC_VECTOR(readvalue, 32);
                    count := count - 1;
                    readvalue := 0; -- reset readvalue
                end if;
            end loop;
            index := index + 1;
        end loop;

        vectornum := 0; errors := 0;
    end process;

```

```

    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
        ALUControl <= testvectors(vectornum)(97 downto 96)
            after 1 ns;
        a <= testvectors(vectornum)(95 downto 64)
            after 1 ns;
        b <= testvectors(vectornum)(63 downto 32)
            after 1 ns;
        Result_expected <= testvectors(vectornum)(31 downto 0)
            after 1 ns;
    end if;
end process;

-- check results on falling edge of clk
process (clk) begin
    if (clk'event and clk = '0' and reset = '0') then
        if (is_x(testvectors(vectornum))) then
            if (errors = 0) then
                report "Just kidding -- " & integer'image(vectornum) & " tests completed
                    successfully. NO ERRORS." severity failure;
            else
                report integer'image(vectornum) & " tests completed, errors = " &
                    integer'image(errors) severity failure;
            end if;
        end if;

        assert Result = Result_expected
        report "Error: vectornum = " &
            integer'image(vectornum) &
            ", a = " & integer'image(CONV_INTEGER(a)) &
            ", b = " & integer'image(CONV_INTEGER(b)) &
            ", Result = " & integer'image(CONV_INTEGER(Result)) &
            ", ALUControl = " & integer'image(CONV_INTEGER(ALUControl));
        if (Result /= Result_expected) then
            errors := errors + 1;
        end if;
        vectornum := vectornum + 1;
    end if;
end process;
end;

```

Testvector file (ex5.13_alu.tv)

```

0_00000000_00000000_00000000
0_00000000_ffffffff_ffffffff
0_00000001_ffffffff_00000000
0_000000ff_00000001_00000100
1_00000000_00000000_00000000
1_00000000_ffffffff_00000001
1_00000001_00000001_00000000
1_00000100_00000001_000000ff
2_ffffffff_ffffffff_ffffffff
2_ffffffff_12345678_12345678
2_12345678_87654321_02244220
2_00000000_ffffffff_00000000
3_ffffffff_ffffffff_ffffffff
3_12345678_87654321_97755779

```

```
3_00000000_ffffffff_ffffffff
3_00000000_00000000_00000000
```

Exercise 5.14

SystemVerilog

```
module testbench();
    logic clk;
    logic [31:0] a, b, y, y_expected;
    logic [1:0] ALUControl;
    logic [3:0] ALUFlags, ALUFlags_expected;

    logic [31:0] vectornum, errors;
    logic [103:0] testvectors[10000:0];

    // instantiate device under test
    alu dut(a, b, ALUControl, y, ALUFlags);

    // generate clock
    always begin
        clk = 1; #50; clk = 0; #50;
    end

    // at start of test, load vectors
    initial begin
        $readmemh("ex5.14_alu.tv", testvectors);
        vectornum = 0; errors = 0;
    end

    // apply test vectors at rising edge of clock
    always @(posedge clk)
        begin
            #1;
            ALUControl = testvectors[vectornum][101:100];
            a = testvectors[vectornum][99:68];
            b = testvectors[vectornum][67:36];
            y_expected = testvectors[vectornum][35:4];
            ALUFlags_expected = testvectors[vectornum][3:0];
        end

    // check results on falling edge of clock
    always @(negedge clk)
        begin
            if (y != y_expected || ALUFlags != ALUFlags_expected) begin
                $display("Error in vector %d", vectornum);
                $display(" Inputs : a = %h, b = %h, ALUControl = %b", a, b, ALUControl);
                $display(" Outputs: y = %h (%h expected), ALUFlags = %h (%h expected)",
                    y, y_expected, ALUFlags, ALUFlags_expected);
                errors = errors+1;
            end
            vectornum = vectornum + 1;
            if (testvectors[vectornum][0] === 1'bx) begin
                $display("%d tests completed with %d errors", vectornum, errors);
                $stop;
            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 testbench is -- no inputs or outputs
end;

architecture sim of testbench is
  component alu
    port(a, b:          in      STD_LOGIC_VECTOR(31 downto 0);
         ALUControl: in      STD_LOGIC_VECTOR(1 downto 0);
         Result:       buffer STD_LOGIC_VECTOR(31 downto 0);
         ALUFlags:     out     STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal a, b, Result, Result_expected: STD_LOGIC_VECTOR(31 downto 0);
  signal ALUControl: STD_LOGIC_VECTOR(1 downto 0);
  signal ALUFlags, ALUFlags_expected: STD_LOGIC_VECTOR(3 downto 0);
  signal clk, reset: STD_LOGIC;
  constant MEMSIZE: integer := 99;
  type tarray is array(MEMSIZE downto 0) of STD_LOGIC_VECTOR(103 downto 0);
  shared variable testvectors: tarray;
  shared variable vectornum, errors: integer;
begin
  -- instantiate device under test
  dut: alu port map(a, b, ALUControl, Result, ALUFlags);

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

  -- at start of test, pulse reset
  process begin
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
  end process;

  -- run tests
  -- at start of test, load vectors
  process is
    file tv: TEXT;
    variable i, index, count: integer;
    variable L: line;
    variable ch: character;
    variable readvalue: integer;
  begin
    -- read file of test vectors
    i := 0;
    index := 0;
    FILE_OPEN(tv, "ex5.14_alu.tv", READ_MODE);
    report "Opening file\n";
    while (not endfile(tv)) loop
      readline(tv, L);
      readvalue := 0;
      count := 3;
      for i in 1 to 30 loop
        read(L, ch);
        report "Line: " & integer'image(index) & " i = " &
          integer'image(i) & " char = " &
          character'image(ch)
          severity error;

        if '0' <= ch and ch <= '9' then
          readvalue := readvalue*16 + character'pos(ch)
            - character'pos('0');
        end if;
      end loop;
    end loop;
  end process;
end;

```

```

    elsif 'a' <= ch and ch <= 'f' then
        readvalue := readvalue*16 + character'pos(ch)
        - character'pos('a')+10;
    else report "Format error on line " &
        integer'image(index) & " i = " &
        integer'image(i) & " char = " &
        character'image(ch)
        severity error;
    end if;

    -- load vectors
    -- assign first 4 bits (will be used for ALUControl)
    if (i = 1) then
        testvectors(index)( 103 downto 100) := CONV_STD_LOGIC_VECTOR(readvalue, 4);
        count := count - 1;
        readvalue := 0; -- reset readvalue

        -- assign a, b, and Result (in testvectors) in
        -- 32-bit increments
        elsif ((i = 10) or (i = 19) or (i = 28)) then
            testvectors(index)( (count*32 + 35) downto (count*32+4)) :=
CONV_STD_LOGIC_VECTOR(readvalue, 32);
            count := count - 1;
            readvalue := 0; -- reset readvalue
        -- assign ALUFlags (in testvectors)
        elsif (i=30) then
            testvectors(index)(3 downto 0) := CONV_STD_LOGIC_VECTOR(readvalue, 4);
        end if;
    end loop;
    index := index + 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
        ALUControl <= testvectors(vectornum)(101 downto 100)
        after 1 ns;
        a <= testvectors(vectornum)(99 downto 68)
        after 1 ns;
        b <= testvectors(vectornum)(67 downto 36)
        after 1 ns;
        Result_expected <= testvectors(vectornum)(35 downto 4)
        after 1 ns;
        ALUFlags_expected <= testvectors(vectornum)(3 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
        if (is_x(testvectors(vectornum))) then
            if (errors = 0) then
                report "Just kidding -- " & integer'image(vectornum) & " tests completed
successfully. NO ERRORS." severity failure;
            else
                report integer'image(vectornum) & " tests completed, errors = " &
integer'image(errors) severity failure;
            end if;
        end if;
    end if;
end process;

```

```

    end if;
end if;

assert Result = Result_expected
report "Error: vectornum = " &
integer'image(vectornum) &
", a = " & integer'image(CONV_INTEGER(a)) &
", b = " & integer'image(CONV_INTEGER(b)) &
", Result = " & integer'image(CONV_INTEGER(Result)) &
", ALUControl = " & integer'image(CONV_INTEGER(ALUControl));
assert ALUFlags = ALUFlags_expected
report "Error: ALUFlags = " & integer'image(CONV_INTEGER(ALUFlags));
if ( (Result /= Result_expected) or
    (ALUFlags /= ALUFlags_expected) ) then
    errors := errors + 1;
end if;
vectornum := vectornum + 1;
end if;
end process;
end;
```

Testvectors file (ex5.14_alu.tv)

```

0_00000000_00000000_00000000_4
0_00000000_ffffffff_ffffffff_8
0_00000001_ffffffff_00000000_6
0_000000ff_00000001_00000100_0
1_00000000_00000000_00000000_6
1_00000000_ffffffff_00000001_0
1_00000001_00000001_00000000_6
1_00000100_00000001_000000ff_2
2_ffffffff_ffffffff_ffffffff_8
2_ffffffff_12345678_12345678_0
2_12345678_87654321_02244220_0
2_00000000_ffffffff_00000000_4
3_ffffffff_ffffffff_ffffffff_8
3_12345678_87654321_97755779_8
3_00000000_ffffffff_ffffffff_8
3_00000000_00000000_00000000_4
```

Exercise 5.15

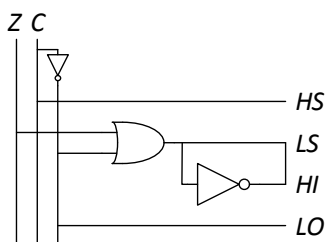
(a) $HS = C$

$LS = Z + \bar{C}$

$HI = \bar{Z}C = \overline{LS}$

$LO = \bar{C} = \overline{HS}$

(b)



Exercise 5.16

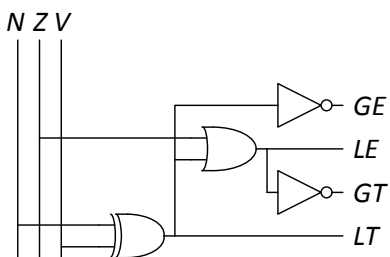
(a) $GE = \overline{N \oplus V}$

$$LE = Z + (N \oplus V)$$

$$GT = \overline{LE} = \overline{Z(N \oplus V)}$$

$$LT = \overline{GE} = N \oplus V$$

(b)



Exercise 5.17

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.

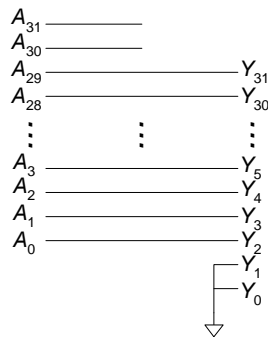


FIGURE 5.6 2-bit left shifter, 32-bit input and output

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
```

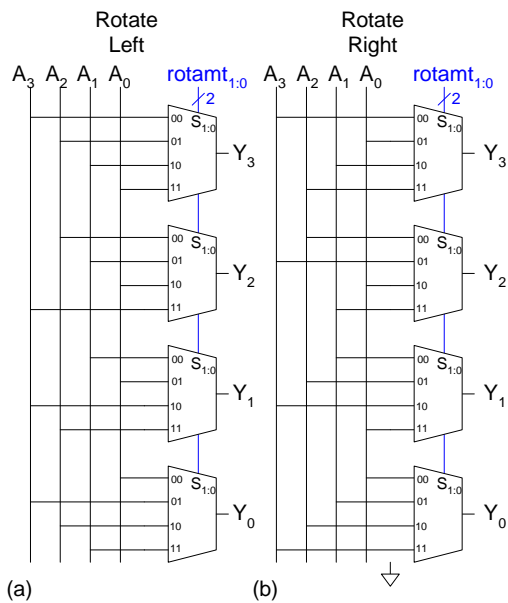
VHDL

```
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 5.18



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 5.19

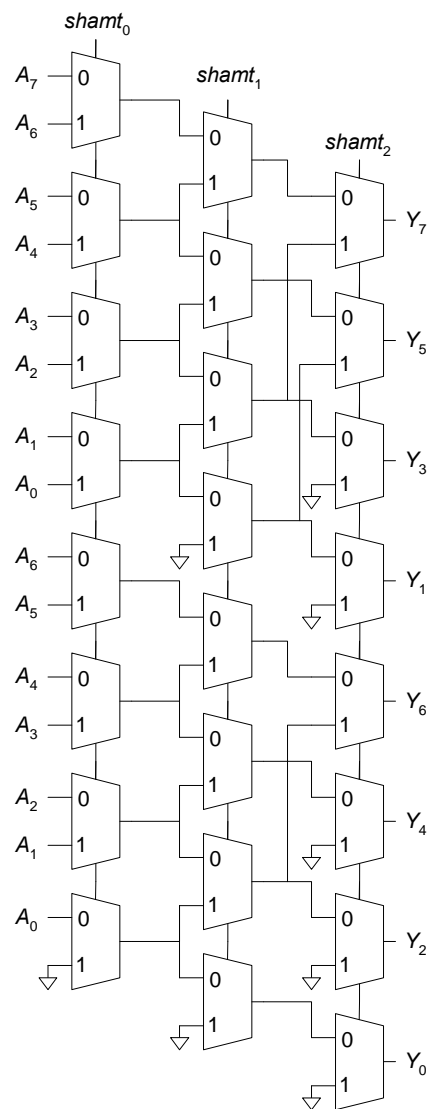


FIGURE 5.7 8-bit left shifter using 24 2:1 multiplexers

Exercise 5.20

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 5.21

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

Exercise 5.22

$$t_{pd_MULT4} = t_{AND} + 8t_{FA}$$

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:

$$t_{pd_MULTN} = t_{AND} + [N + 2(N-1)]t_{FA}$$

Exercise 5.23

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

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

Exercise 5.24

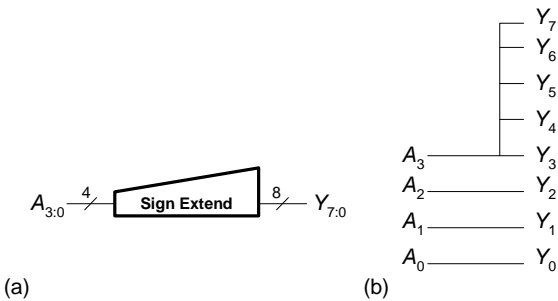


FIGURE 5.9 Sign extension unit (a) symbol, (b) underlying hardware

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 5.26

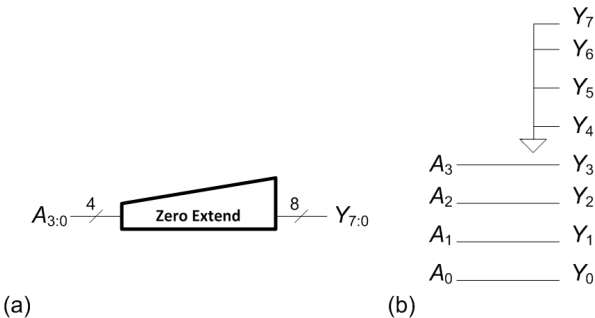


FIGURE 5.10 Zero extension unit (a) symbol, (b) underlying hardware

Exercise 5.29

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

Exercise 5.30

- (a) $111110.100000 = 0xFA0$
- (b) $010000.010000 = 0x410$
- (c) $101000.000101 = 0xA05$

Exercise 5.31

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

Exercise 5.32

- (a) $100001.100000 = 0x860$
- (b) $010000.010000 = 0x410$
- (c) $110111.111011 = 0xDFB$

Exercise 5.33

- (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:
 $1\ 1000\ 0010\ 101\ 1001\ 0000\ 0000\ 0000\ 0000 = \mathbf{0xC1590000}$
- (b) $101010.0101 = 1.010100101 \times 2^5$
Thus, the biased exponent $= 127 + 5 = 132 = 1000\ 0100_2$
In IEEE 754 single-precision floating-point format:
 $0\ 1000\ 0100\ 010\ 1001\ 0100\ 0000\ 0000\ 0000 = \mathbf{0x42294000}$
- (c) $-10001.00101 = -1.000100101 \times 2^4$
Thus, the biased exponent $= 127 + 4 = 131 = 1000\ 0011_2$
In IEEE 754 single-precision floating-point format:
 $1\ 1000\ 0011\ 000\ 1001\ 0100\ 0000\ 0000\ 0000 = \mathbf{0xC1894000}$

Exercise 5.34

(a) $-11110.1 = -1.111101 \times 2^4$

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) $10000.01 = 1.000001 \times 2^4$

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) $-1000.000101 = -1.000000101 \times 2^3$

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 5.35

(a) 5.5

(b) $-0000.0001_2 = -0.0625$

(c) -8

Exercise 5.36

(a) 29.65625

(b) -25.1875

(c) -23.875

Exercise 5.37

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 5.38

- (a) C0123456
- (b) D1E072C3
- (c) 5F19659A

Exercise 5.39

(a)

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

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

$$0x72407020$$

(b)

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

$$\begin{aligned}
 &1.101\ 1100\ 0000\ 0000\ 0000\ 01 \times 2^2 \\
 &- 1.101\ 0010\ 0000\ 0000\ 0000\ 01 \times 2^2 \\
 &= 0.000\ 1010 \qquad \qquad \qquad \times 2^2 \\
 &= 1.010 \times 2^{-2} \\
 &= 0\ 0111\ 1101\ 010\ 0000\ 0000\ 0000\ 0000\ 0000 \\
 &= 0x3EA00000
 \end{aligned}$$

(c)

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

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

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

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 5.40

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 5.41

$$(a) 2(2^{31} - 1 - 2^{23}) = 2^{32} - 2 - 2^{24} = 4,278,190,078$$

$$(b) 2(2^{31} - 1) = 2^{32} - 2 = 4,294,967,294$$

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

Exercise 5.42

$$\begin{aligned}
 \text{(a) } 245 &= 11110101 = 1.1110101 \times 2^7 \\
 &= 0\ 1000\ 0110\ 111\ 0101\ 0000\ 0000\ 0000\ 0000 \\
 &= 0x43750000 \\
 0.0625 &= 0.0001 = 1.0 \times 2^{-4} \\
 &= 0\ 0111\ 1011\ 000\ 0000\ 0000\ 0000\ 0000\ 0000 \\
 &= 0x3D800000
 \end{aligned}$$

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

$$\begin{aligned}
 \text{(c)} \\
 1.1110101 \times 2^7 &= 0\ 0000\ 0111\ 111\ 0101\ 0000\ 0000\ 0000\ 0000 \\
 &= 0x03F50000 \\
 1.0 \times 2^{-4} &= 0\ 1111\ 1100\ 000\ 0000\ 0000\ 0000\ 0000\ 0000 \\
 &= 0x7E000000
 \end{aligned}$$

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

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

Exercise 5.43

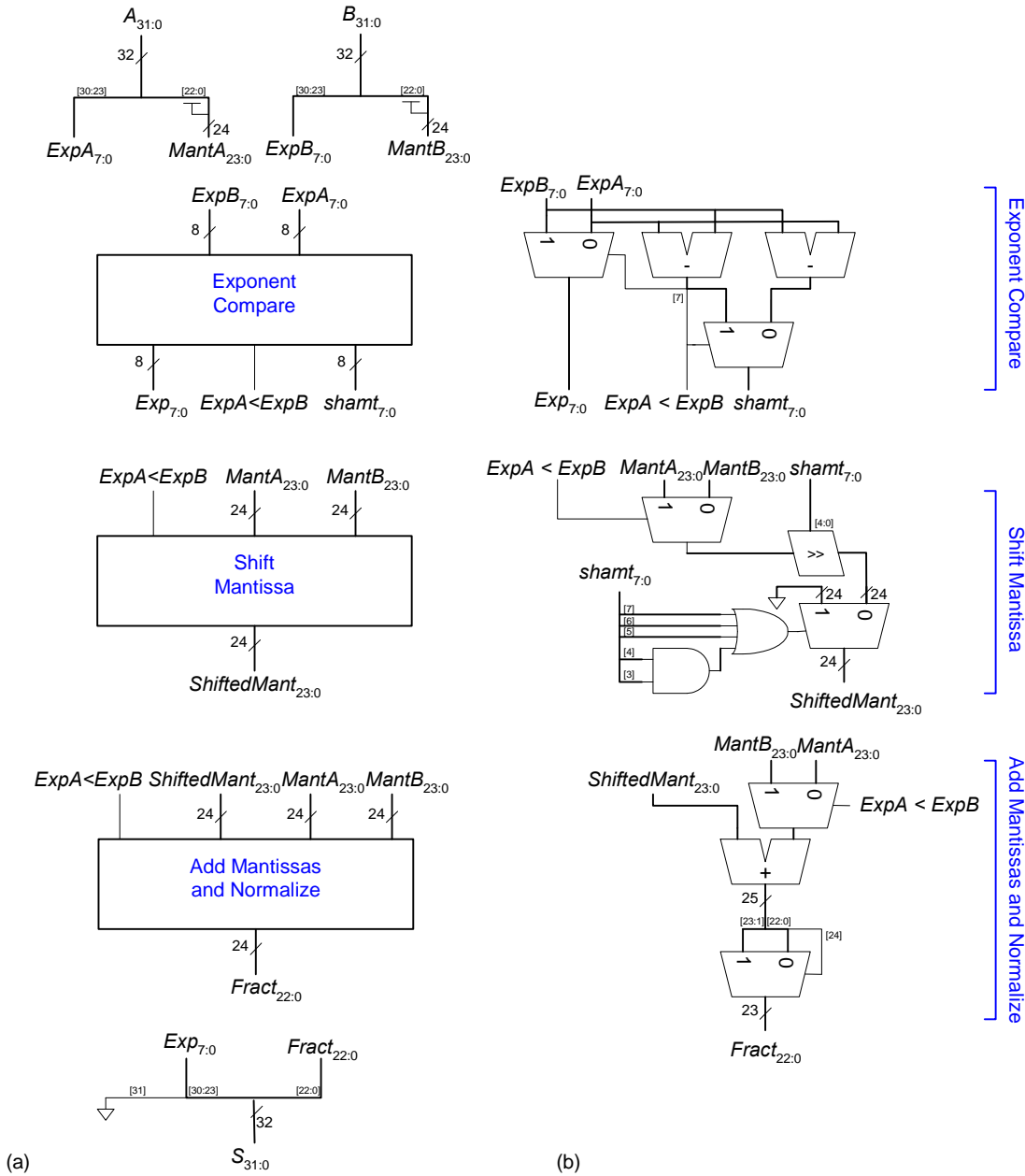


FIGURE 5.11 Floating-point adder hardware: (a) block diagram, (b) underlying hardware

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    expcompl(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;

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

end;

```


*(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
    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,
               input  logic [23:0] 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 5.44

(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

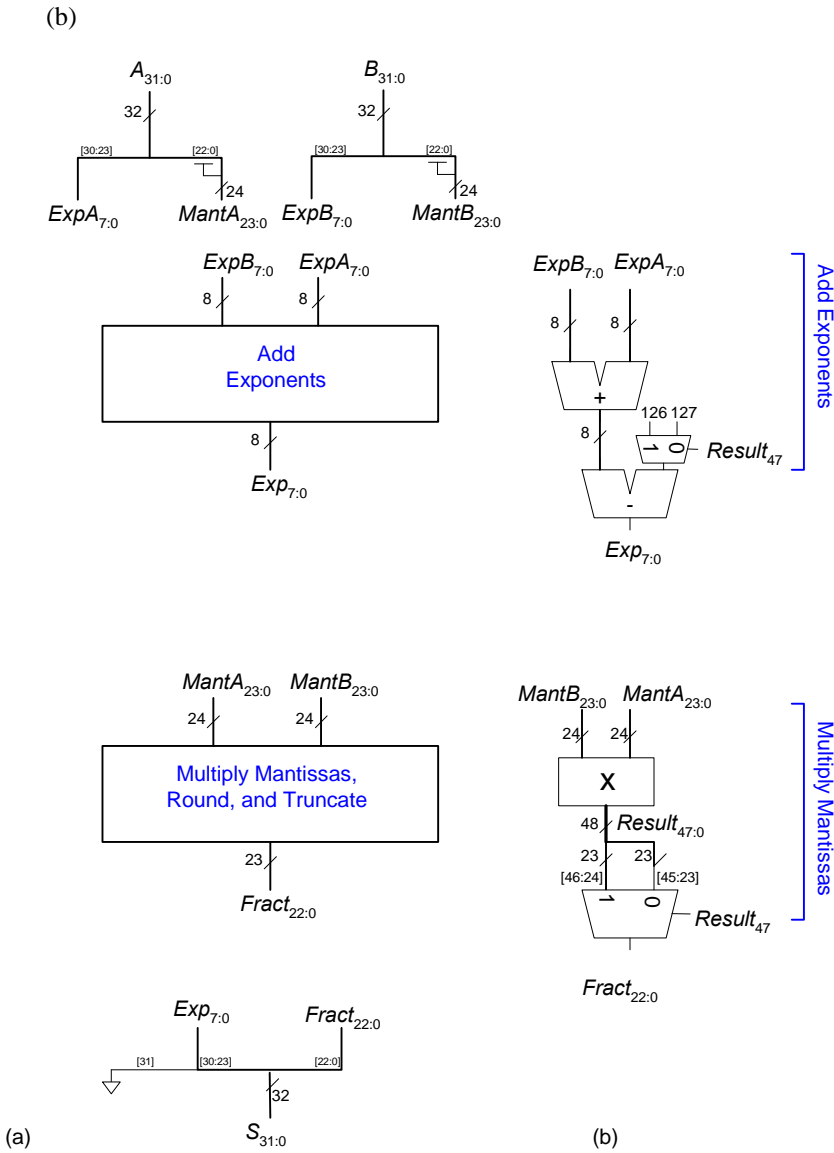


FIGURE 5.12 Floating-point multiplier block diagram

(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);

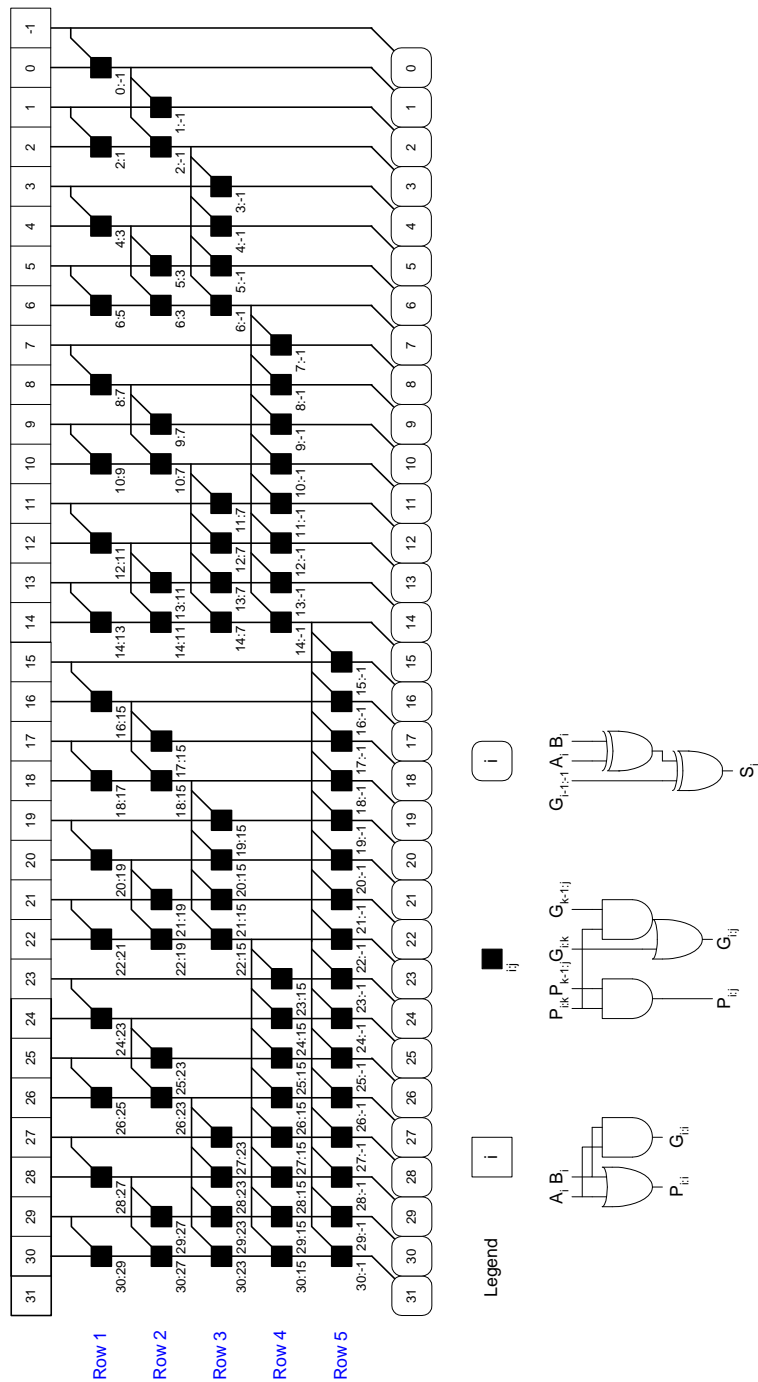
    m      <= '0' & exp & fract;
    result <= manta * mantb;
    fract  <= result(46 downto 24)
        when (result(47) = '1')
        else result(45 downto 23);
    exp    <= (expa + expb - 126)
        when (result(47) = '1')
        else (expa + expb - 127);

end;

```

Exercise 5.45

(a) Figure on next page



5.45 (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 subblock 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;

```

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

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

We find the delays for each block:

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

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

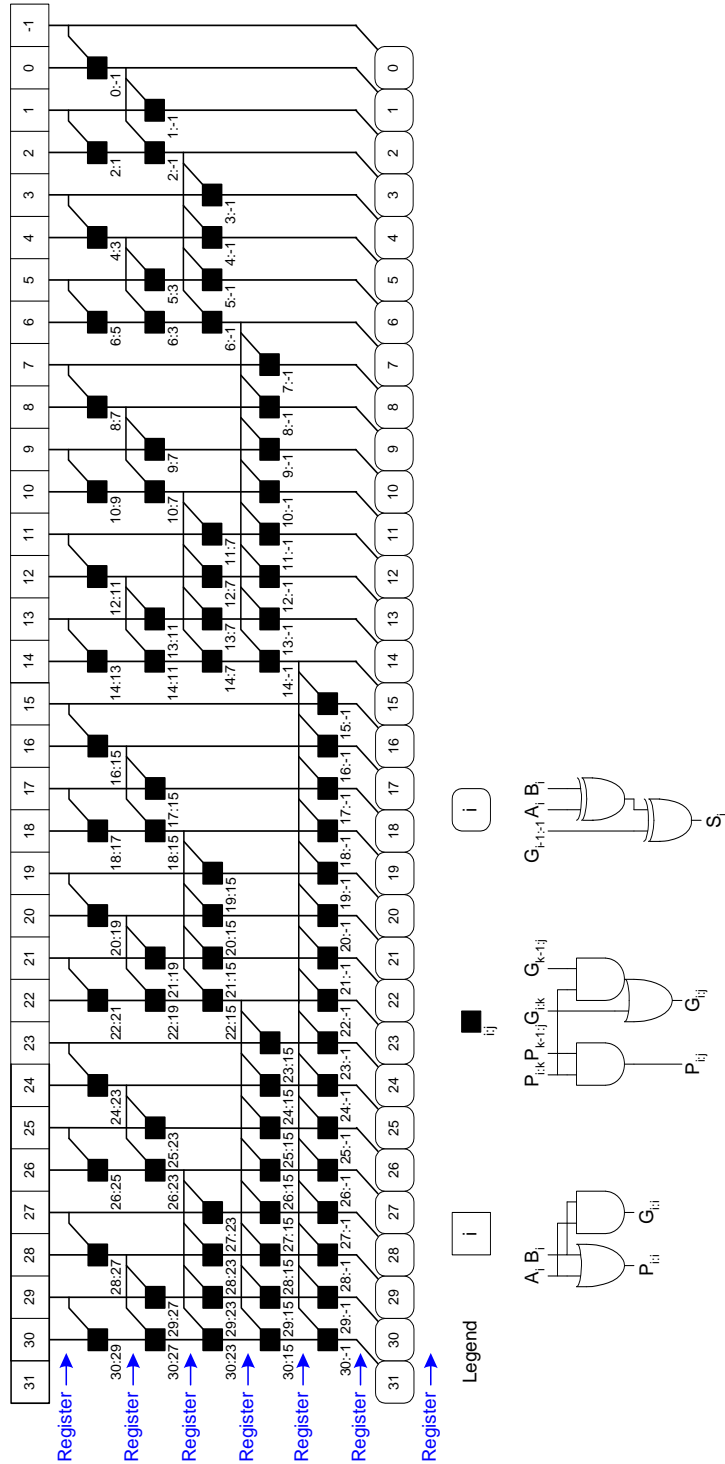
$$t_{XOR} = 100 \text{ ps}$$

Thus,

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

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



5.45 (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

```


5.45 (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, p_1_1, g_1_1, p2, g2);
row3_reg: row3 port map(clk, p2, g2, p_1_2, g_1_2, p3, g3);
row4_reg: row4 port map(clk, p3, g3, p_1_3, g_1_3, p4, g4);
row5_reg: row5 port map(clk, p4, g4, p_1_4, g_1_4, p5, g5);

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

-- pipeline p and g for column -1
p_1_0 <= '0'; flop1_g0: flop1 port map (clk, cin, g_1_0);
flop1_p1: flop1 port map (clk, p_1_0, p_1_1);
flop1_g1: flop1 port map (clk, g_1_0, g_1_1);
flop1_p2: flop1 port map (clk, p_1_1, p_1_2);
flop1_g2: flop1 port map (clk, g_1_1, g_1_2);
flop1_p3: flop1 port map (clk, p_1_2, p_1_3); flop1_g3:
flop1 port map (clk, g_1_2, g_1_3);
flop1_p4: flop1 port map (clk, p_1_3, p_1_4);
flop1_g4: flop1 port map (clk, g_1_3, g_1_4);
flop1_p5: flop1 port map (clk, p_1_4, p_1_5);
flop1_g5: flop1 port map (clk, g_1_4, g_1_5);

-- generate sum and cout
g5_all <= (g5&g_1_5);
row6: 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;

```

```

        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) <= pgl_out(5); g1(9) <= pgl_out(4); g1(7) <= pgl_out(3);
g1(5) <= pgl_out(2); g1(3) <= pgl_out(1); g1(1) <= pgl_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 pgl_in, pg2_out: STD_LOGIC_VECTOR(29 downto 0);

begin
  pgl_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, pgl_in, pg2_out);

```

```

p2(28 downto 27) <= pg2_out(29 downto 28);
p2(24 downto 23) <= pg2_out(27 downto 26);
p2(20 downto 19) <= pg2_out(25 downto 24);
p2(16 downto 15) <= pg2_out(23 downto 22);
p2(12 downto 11) <= pg2_out(21 downto 20);
p2(8 downto 7) <= pg2_out(19 downto 18);
p2(4 downto 3) <= pg2_out(17 downto 16);
p2(0) <= pg2_out(15);
g2(28 downto 27) <= pg2_out(14 downto 13);
g2(24 downto 23) <= pg2_out(12 downto 11);
g2(20 downto 19) <= pg2_out(10 downto 9);
g2(16 downto 15) <= pg2_out(8 downto 7);
g2(12 downto 11) <= pg2_out(6 downto 5);
g2(8 downto 7) <= pg2_out(4 downto 3);
g2(4 downto 3) <= pg2_out(2 downto 1); g2(0) <= pg2_out(0);

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

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

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

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

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

```

```

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

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

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

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

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

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

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

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

```

```

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

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

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

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

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

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

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

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

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

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

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

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

```



```

begin

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

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

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

end;

```

Exercise 5.46

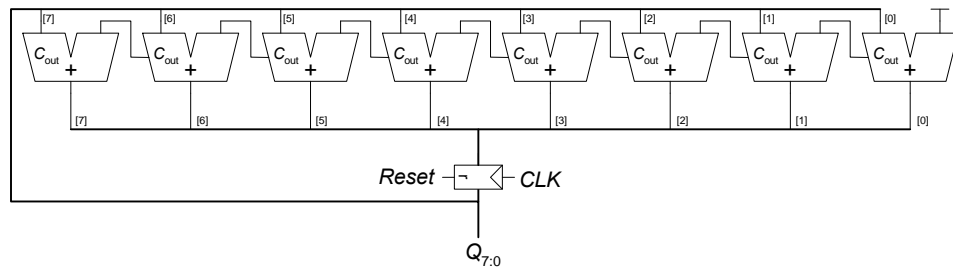


FIGURE 5.13 Incrementer built using half adders

Exercise 5.47

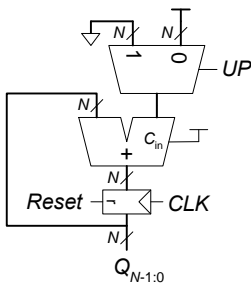


FIGURE 5.14 Up/Down counter

Exercise 5.48

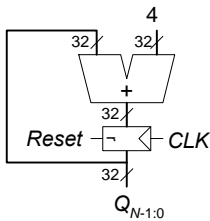


FIGURE 5.15 32-bit counter that increments by 4 on each clock edge

Exercise 5.49

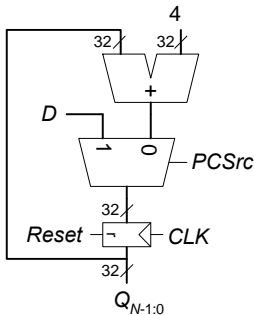


FIGURE 5.16 32-bit counter that increments by 4 or loads a new value, D

Exercise 5.50

(a)

0000

1000

1100

1110

1111

0111

0011

0001

(repeat)

(b)

$2N$. 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.

5.50 (c)

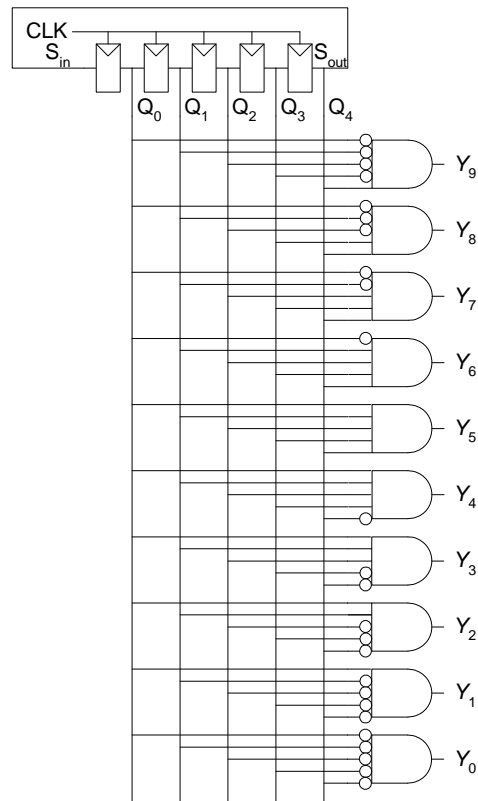


FIGURE 5.17 10-bit decimal counter using a 5-bit Johnson counter

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

Exercise 5.51

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 5.52

(a)

value <i>a</i> 1:0	encoding <i>y</i> 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.

5.52 (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

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 5.53

<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.

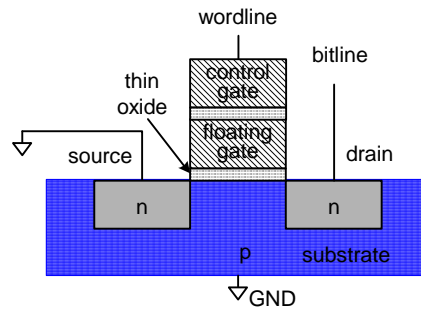


FIGURE 5.18 Flash EEPROM

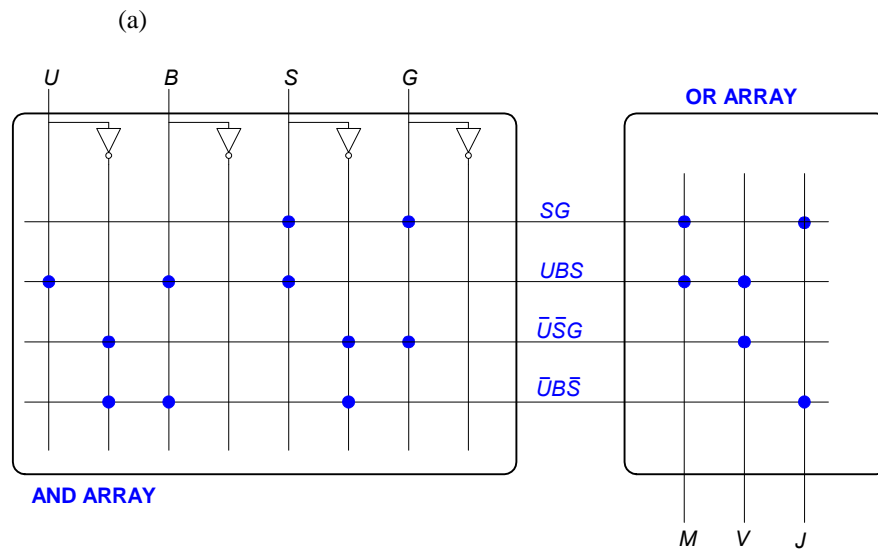
Exercise 5.54

FIGURE 5.19 4 x 4 x 3 PLA implementing Exercise 5.44

5.54 (b)

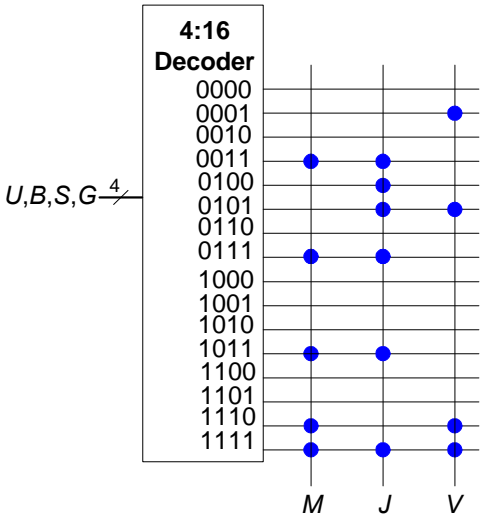


FIGURE 5.20 16 x 3 ROM implementation of Exercise 5.44

(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
```

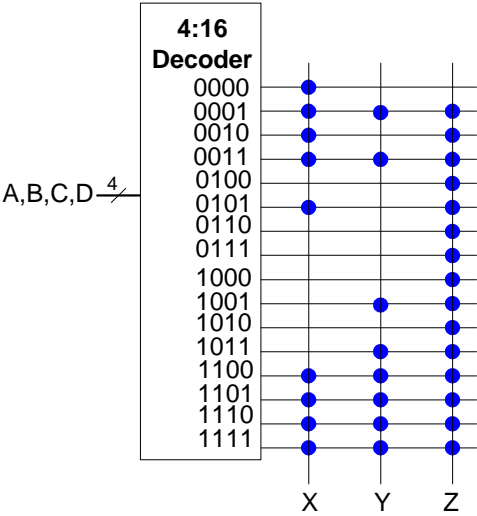
VHDL

```
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 5.55



Exercise 5.56

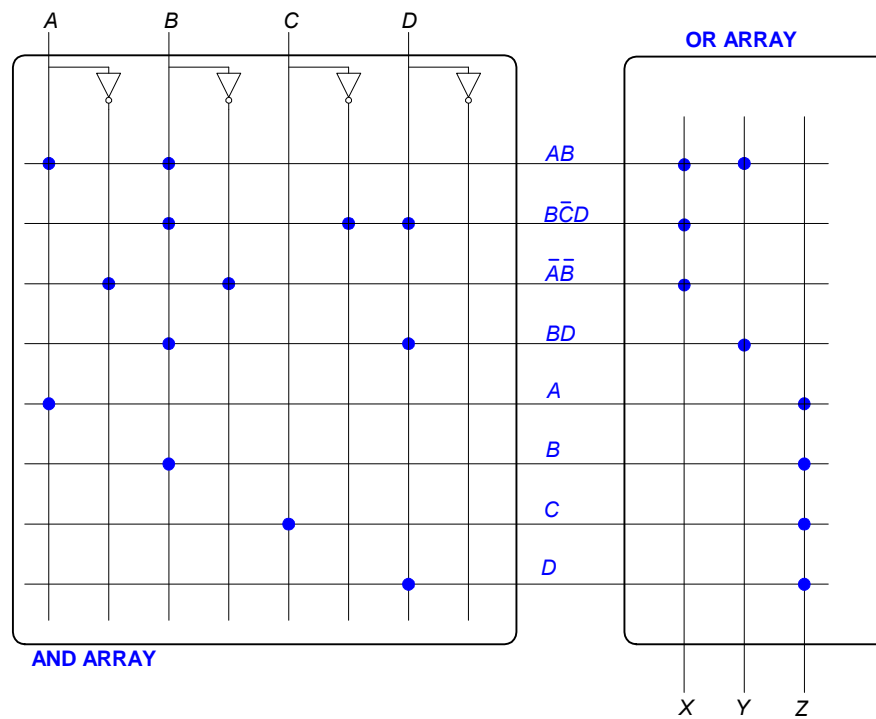


FIGURE 5.21 4 x 8 x 3 PLA for Exercise 5.52

Exercise 5.57

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

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

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

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

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

Thus, this would require a 2^{16} 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.

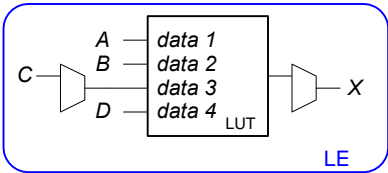
Exercise 5.58

- (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 5.59

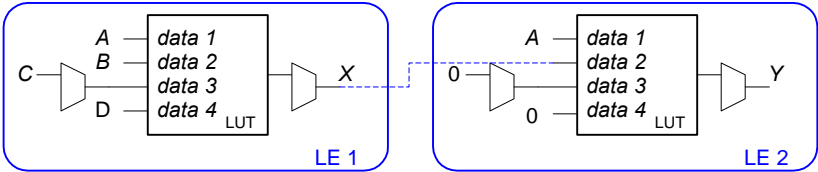
- (a) 1 LE

(A)	(B)	(C)	(D)	(Y)
data 1	data 2	data 3	data 4	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



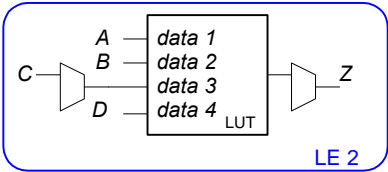
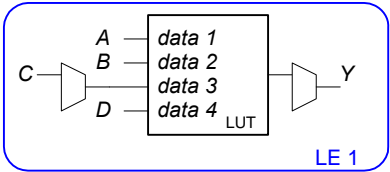
(b) 2 LEs

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



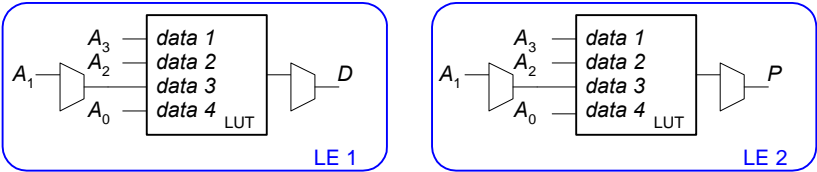
(c) 2 LEs

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



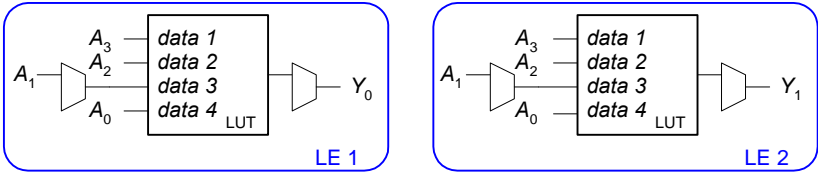
(d) 2 LEs

(A ₃)	(A ₂)	(A ₁)	(A ₀)	(D)	(A ₃)	(A ₂)	(A ₁)	(A ₀)	(P)
data 1	data 2	data 3	data 4	LUT output	data 1	data 2	data 3	data 4	LUT output
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1	0
0	0	1	0	0	0	0	1	0	1
0	0	1	1	1	0	0	1	1	1
0	1	0	0	0	0	1	0	0	0
0	1	0	1	0	0	1	0	1	1
0	1	1	0	1	0	1	1	0	0
0	1	1	1	0	0	1	1	1	1
1	0	0	0	0	1	0	0	0	0
1	0	0	1	1	1	0	0	1	0
1	0	1	0	0	1	0	1	0	0
1	0	1	1	0	1	0	1	1	1
1	1	0	0	1	1	1	0	0	0
1	1	0	1	0	1	1	0	1	1
1	1	1	0	0	1	1	1	0	0
1	1	1	1	1	1	1	1	1	0



(e) 2 LEs

(A ₃)	(A ₂)	(A ₁)	(A ₀)	(Y ₀)	(A ₃)	(A ₂)	(A ₁)	(A ₀)	(Y ₁)
data 1	data 2	data 3	data 4	LUT output	data 1	data 2	data 3	data 4	LUT output
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1	0
0	0	1	0	1	0	0	1	0	0
0	0	1	1	1	0	0	1	1	0
0	1	0	0	0	0	1	0	0	1
0	1	0	1	0	0	1	0	1	1
0	1	1	0	0	0	1	1	0	1
0	1	1	1	0	0	1	1	1	1
1	0	0	0	1	1	0	0	0	1
1	0	0	1	1	1	0	0	1	1
1	0	1	0	1	1	0	1	0	1
1	0	1	1	1	1	0	1	1	1
1	1	0	0	1	1	1	0	0	1
1	1	0	1	1	1	1	0	1	1
1	1	1	0	1	1	1	1	0	1
1	1	1	0	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1



Exercise 5.60

(a) 8 LEs (see next page for figure)

LE 1

data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) 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

LE 2

data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) 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 3

data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) 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 4

data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) 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 5

data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) 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 6

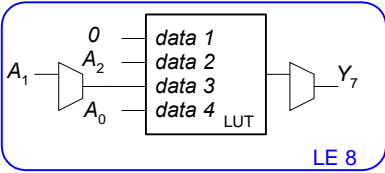
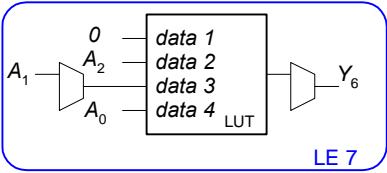
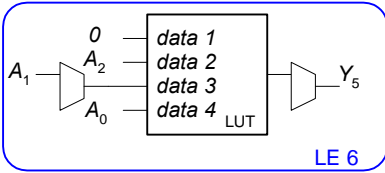
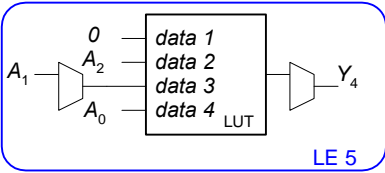
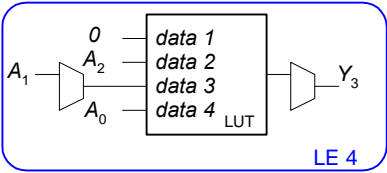
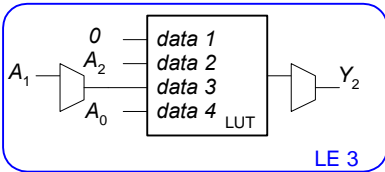
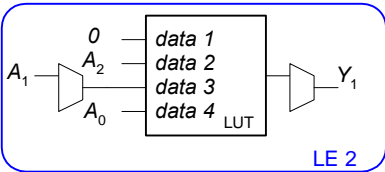
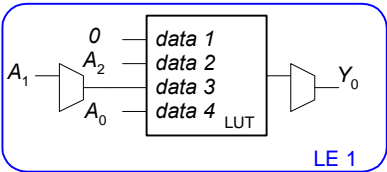
data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) data 4	(Y ₅) LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	1
X	1	1	0	0
X	1	1	1	0

LE 7

data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) 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 8

data 1	(A ₂) data 2	(A ₁) data 3	(A ₀) 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



(b) 8 LEs (see next page for figure)

LE 7

	(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

LE 6

	(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 5

	(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 4

	(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 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	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

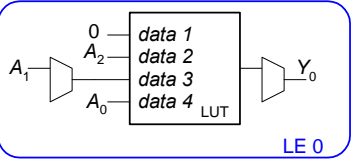
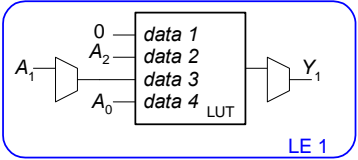
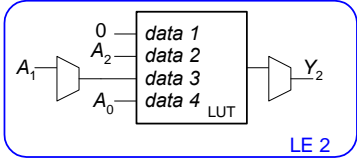
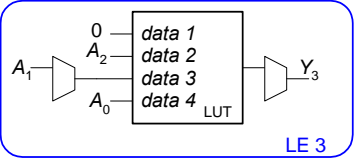
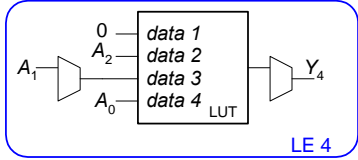
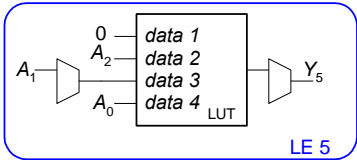
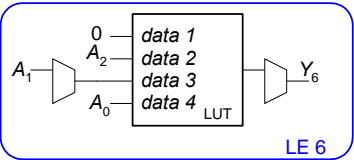
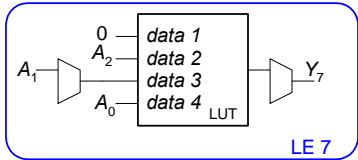
	(A ₂)	(A ₁)	(A ₀)	(Y ₂)
data 1	data 2	data 3	data 4	LUT output
X	0	0	0	0
X	0	0	1	0
X	0	1	0	1
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	0
X	1	1	1	0

LE 1

	(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 0

	(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



(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

data 1	data 2	(A ₀) data 3	(B ₀) 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

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

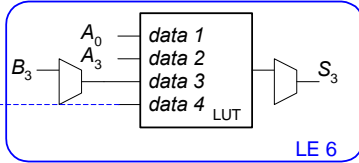
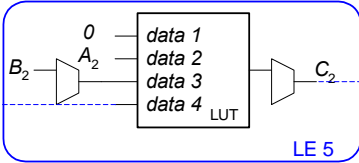
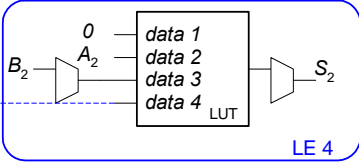
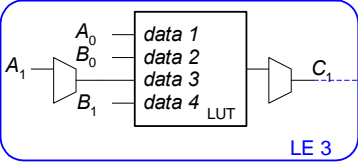
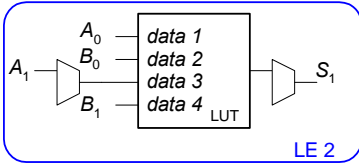
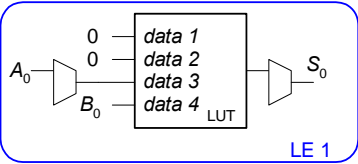
data 1	(A ₂) data 2	(B ₂) data 3	(C ₁) data 4	(S ₂) LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	1
X	0	1	1	0
X	1	0	0	1
X	1	0	1	0
X	1	1	0	0
X	1	1	1	1

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

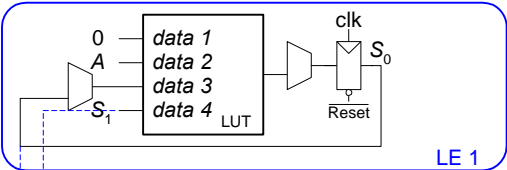
LE 6

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

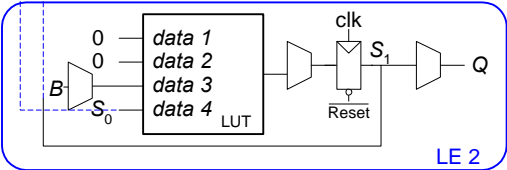


(d) 2 LEs

	(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

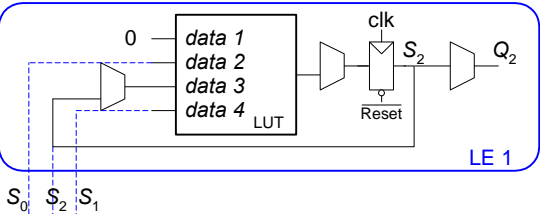


	(B)	(S ₀)	(S ₁)	(S ₁)
data 1	data 2	data 3	data 4	LUT output
X	X	0	0	0
X	X	0	1	0
X	X	1	0	0
X	X	1	1	1

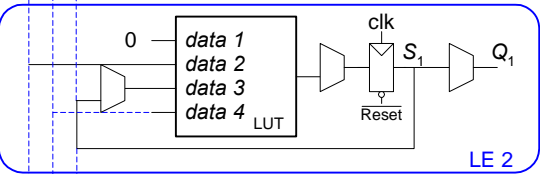


(e) 3 LEs

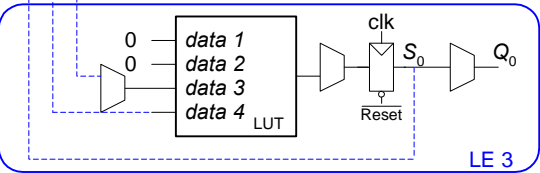
	(S ₀)	(S ₂)	(S ₁)	(S ₂)
data 1	data 2	data 3	data 4	LUT output
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	1
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
X	1	1	1	1



	(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 5.61

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

(b)

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

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

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

(c)

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

$$\begin{aligned}
 t_{cd_LE} &= t_{pd_LE} = 381 \text{ ps} \\
 t_{cd} &= t_{cd_LE} + t_{wire} = 627 \text{ ps}
 \end{aligned}$$

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

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

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

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

$$f = 1 / 3.902 \text{ ns} = \mathbf{256 \text{ MHz}}$$

Exercise 5.62

(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 5.63

First, we find the cycle time:

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

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

Thus, $N \leq 15.5$

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

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

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

Question 5.1

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

Question 5.2

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

Question 5.3

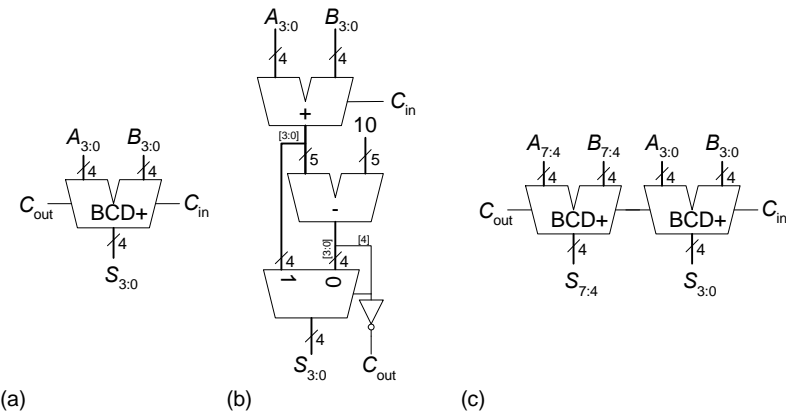


FIGURE 5.22 BCD adder: (a) 4-bit block, (b) underlying hardware, (c) 8-bit BCD adder

*(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

(1) Regularity supports simplicity

- Each instruction has a 2-bit opcode.
- Each instruction has a 4-bit condition code.
- ARM has 3 instruction formats for the most common instructions (Data-processing format, Memory format, and Branch format).
- The Data-processing and Memory instruction formats have a similar number and order of operands.
- Each instruction is the same size, making decoding hardware simple.

(2) Make the common case fast

- Registers make the access to most recently accessed variables fast.
- The RISC (reduced instruction set computer) architecture, makes the common/simple instructions fast because the computer must handle only a small number of simple instructions.
- Most instructions require all 32 bits of an instruction, so all instructions are 32 bits (even though some would have an advantage of a larger instruction size and others a smaller instruction size). The instruction size is chosen to make the common instructions fast.

(3) Smaller is faster

- The register file has only 16 registers.
- The ISA (instruction set architecture) includes only a small number of commonly used instructions. This keeps the hardware small and, thus, fast.
- The instruction size is kept small to make instruction fetch fast.

(4) Good design demands good compromises

- ARM uses three instruction formats (instead of just one).
- Ideally all accesses would be as fast as a register access, but ARM architecture also supports main memory accesses to allow for a compromise between fast access time and a large amount of memory.
- Because ARM is a RISC architecture, it includes only a set of simple instructions, but it provides pseudocode to the user and compiler for commonly used operations, like NOP.
- ARM provides three formats to encode immediate values (and four if you count the 5-bit immediate encoding for a shift, shamt5):
 - **{rot_{3:0}, imm8_{7:0}}** for data-processing instructions
 - **imm12_{11:0}** for memory instructions

- **imm24**_{23:0} for branch instructions

Exercise 6.2

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

ADD 0x10, 0x20, 0x24

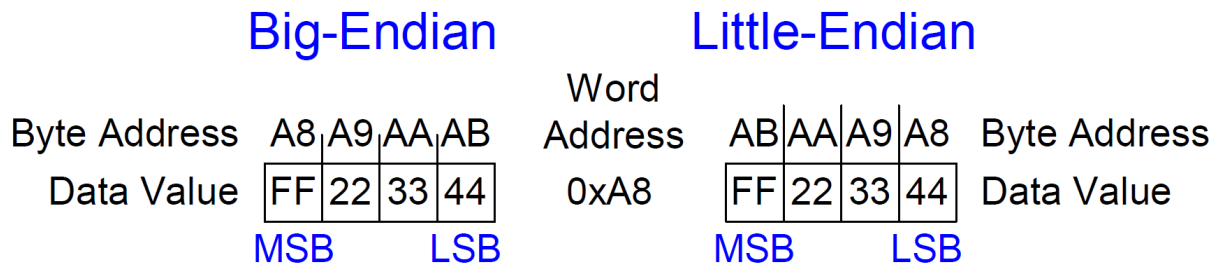
This would add the values stored at memory addresses 0x20 and 0x24 and place the result in memory address 0x10. Other instructions would follow the same pattern, accessing memory instead of registers. Some advantages of the architecture are that it would require fewer instructions. Load and store operations are now unnecessary. This would make the decoding hardware simpler and faster. Some disadvantages of this architecture over the MIPS architecture is that each operation would require a memory access. Thus, either the processor would need to be slow or the memory small. Also, because the instructions must encode memory addresses instead of register numbers, the instruction size would be large in order to access all memory addresses. Or, alternatively, each instruction can only access a smaller number of memory addresses. For example, the architecture might require that one of the source operands is also a destination operand, reducing the number of memory addresses that must be encoded.

Exercise 6.3

(a) $42 \times 4 = 42 \times 22 = 1010102 \ll 2 = 101010002 = 0xA8$

(b) 0xA8 through 0xAB

(c)

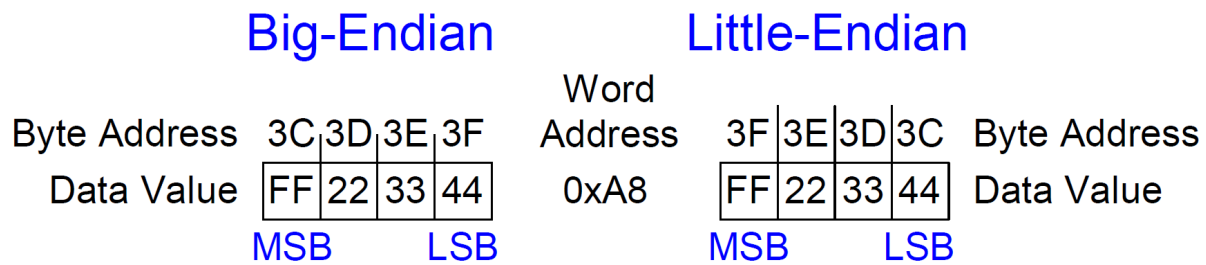


Exercise 6.4

(a) $15 \times 4 = 42 \times 22 = 11112 \ll 2 = 1111002 = 0x3C$

(b) 0x3C through 0x3F

(c)

**Exercise 6.5**

In big-endian format, the bytes are numbered from 100 to 103 from left to right. In little-endian format, the bytes are numbered from 100 to 103 from right to left. Thus, the load byte instruction (LDRB) returns a different value depending on the endianness of the machine. At the end of the program R2 contains 0xBC on a big-endian machine and 0xD8 on a little-endian machine.

Exercise 6.6

-
- (a) 0x53 4F 53 00
 - (b) 0x43 6F 6F 6C 21 00
 - (c) 0x41 6C 79 73 73 61 00 (depends on the person's name)

Exercise 6.7

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

Exercise 6.8

Little-Endian Memory

Word Address	Data
⋮	⋮
1000100C	00 53 4F 53
⋮	⋮
	Byte 3 ⋮ Byte 0

(a)

Word Address	Data
⋮	⋮
10001010	00 21
1000100C	6C 6F 6F 43
⋮	⋮
	Byte 3 ⋮ Byte 0

(b)

Word Address	Data
⋮	⋮
10001010	00 61 73
1000100C	73 79 6C 41
⋮	⋮
	Byte 3 ⋮ Byte 0

(c)

Big-Endian Memory

Word Address	Data
⋮	⋮
1000100C	53 4F 53 00
⋮	⋮
	Byte 0 ⋮ Byte 3

(a)

Word Address	Data
⋮	⋮
10001010	21 00
1000100C	43 6F 6F 6C
⋮	⋮
	Byte 0 ⋮ Byte 3

(b)

Word Address	Data
⋮	⋮
10001010	73 61 00
1000100C	41 6C 79 73
⋮	⋮
	Byte 0 ⋮ Byte 3

(c)

Exercise 6.9

Little-Endian Memory

Word Address	Data
⋮	⋮
10001010	00 79
1000100C	64 77 6F 68
⋮	⋮
	Byte 3 ⋮ Byte 0

(a)

Word Address	Data
⋮	⋮
10001010	00 73
1000100C	6E 6F 69 6C
⋮	⋮
	Byte 3 ⋮ Byte 0

(b)

Word Address	Data
⋮	⋮
10001018	00 21 65
10001014	75 63 73 65
10001010	72 20 65 68
1000100C	74 20 6F 54
⋮	⋮
	Byte 3 ⋮ Byte 0

(c)

Big-Endian Memory

Word Address	Data
⋮	⋮
10001010	79 00
1000100C	68 6F 77 64
⋮	⋮
	Byte 0 ⋮ Byte 3

(a)

Word Address	Data
⋮	⋮
10001010	73 00
1000100C	6C 69 6F 6E
⋮	⋮
	Byte 0 ⋮ Byte 3

(b)

Word Address	Data
⋮	⋮
10001018	65 21 00
10001014	65 73 63 75
10001010	68 65 20 72
1000100C	54 6F 20 74
⋮	⋮
	Byte 0 ⋮ Byte 3

(c)

Exercise 6.10

0xE3A0AB3E

```

0xE1A09386
0xE78B4008
0xE1A06357

```

Exercise 6.11

```

0xE0808001
0xE593B004
0xE2475058
0xE1A03702

```

Exercise 6.12

- (a) MOV R10, #63488
 (b) rot = 11, imm8 = 0x3E (binary: 00111110), 32-bit immediate = 0x0000F800

Exercise 6.13

- (a) SUB R5, R7, #0x58
 (b) rot = 0, imm8 = 0x58

Exercise 6.14

ARM Assembly

```

MOV R2, #0
MOV R3, R1
L1
    CMP R1, R0
    BHI DONE
    ADD R2, R2, #1
    ADD R1, R1, R3
    B L1
DONE
    MOV R0, R2

```

C Code

```

// R0 = A (dividend) and quotient, R1 = B (divisor)
// R2 = i, R3 = temp
int i = 0;
int quotient;
int temp = divisor;

while (dividend >= temp) {
    i = i + 1;
    temp = temp + divisor;
}

```

```

}
quotient = i;

```

In words

This code performs integer division: $\text{quotient} = A/B$.

Exercise 6.15

ARM Assembly

```

; R0 = decimal number, R1 = base address of array,
; R2 = val, R3 = tmp
    MOV    R2, #31
L1   LSR    R3, R0, R2
    AND    R3, R3, #1
    STRB   R3, [R1], #1
    SUBS   R2, R2, #1
    BPL    L1
L2   MOV    PC, LR

```

C Code

```

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

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

```

In words

This program converts an unsigned integer (R0) from decimal to binary and stores it in an array pointed to by R1.

Exercise 6.16

```

    ORR    R0, R1, R2
    MVN    R0, R0

```

Exercise 6.17

```

AND R0, R1, R2
MVN R0, R0

```

Exercise 6.18

(a)

(i)

```

    CMP R0, R1          ; g >= h?
    BLT ELSE
    ADD R0, R0, R1      ; g = g + h
    B    DONE
ELSE SUB R0, R0, R1     ; g = g - h
DONE

```

(ii)

```

    CMP R0, R1          ; g < h?
    BGE ELSE
    ADD R1, R1, #1      ; h = h + 1
    B    DONE
ELSE LSL R1, R1, #1     ; h = h * 2
DONE

```

(b)

(i)

```

    CMP    R0, R1          ; g >= h?
    ADDGE  R0, R0, R1      ; g = g + h
    SUBLT  R0, R0, R1      ; g = g - h

```

(ii)

```

    CMP    R0, R1          ; g < h?
    ADDLT  R1, R1, #1      ; h = h + 1
    LSLGE  R1, R1, #1      ; h = h * 2

```

(c) When conditional execution is available for all instructions, it takes 3 instructions, compared to 5 instructions when conditional execution is allowed only for branch instructions. So, in this case, allowing conditional execution for all instructions results in a 40% decrease in the number of instructions.

Thus, the advantages of conditional execution are (1) 40% less memory required for instruction storage, and (2) potentially decreased execution time. The execution time of the code in part (a)

is 3-4 instructions, whereas it is 3 instructions in part (b). As will be seen in Chapter 7, the number of instructions fetched in part (a) can be even higher when using a pipelined processor.

A disadvantage of (b) over (a) is that all instructions require a condition code, which uses four bits of encoding that could be used for something else. However, as shown, this cost in bits used for encoding the condition is usually well worth it.

Exercise 6.19

(a)

(i)

```

    CMP R0, R1          ; g > h?
    BLE ELSE
    ADD R0, R0, #1      ; g = g + 1
    B    DONE
ELSE SUB R0, R1, #1     ; g = h - 1
DONE

```

(ii)

```

    CMP R0, R1          ; g <= h?
    BGT ELSE
    MOV R0, #0          ; g = 0
    B    DONE
ELSE MOV R1, #0         ; h = 0
DONE

```

(b)

(i)

```

    CMP    R0, R1          ; g > h?
    ADDGT  R0, R0, #1      ; g = g + 1
    SUBLE  R1, R1, #1      ; h = h - 1

```

(ii)

```

    CMP    R0, R1          ; g <= h?
    MOVLE  R0, #1          ; g = 0
    MOVGT  R1, #0          ; h = 0

```

(c) When conditional execution is available for all instructions, it takes 3 instructions, compared to 5 instructions when conditional execution is allowed only for branch instructions. So, in this case, allowing conditional execution for all instructions results in a 40% decrease in the number of instructions.

Thus, the advantages of conditional execution are (1) 40% less memory required for instruction storage, and (2) potentially decreased execution time. The execution time of the code in part (a) is 3-4 instructions, whereas it is 3 instructions in part (b). As will be seen in Chapter 7, the number of instructions fetched in part (a) can be even higher when using a pipelined processor.

A disadvantage of (b) over (a) is that all instructions require a condition code, which uses four bits of encoding that could be used for something else. However, as shown, this cost in bits used for encoding the condition is usually well worth it.

Exercise 6.20

(a)

```

      ADD R3, R1, #0x190    ; R3 = end of array1
FOR   CMP R1, R3           ; reached end of array1?
      BGE DONE
      LDR R0, [R1]         ; R0 = array1[i]
      STR R0, [R2]         ; array2[i] = array1[i]
      ADD R1, R1, #4       ; R1 points to next array1 entry
      ADD R2, R2, #4       ; R2 points to next array2 entry
      B    FOR
DONE
```

(b)

```

      ADD R3, R1, #0x190    ; R3 = end of array1
FOR   CMP R1, R3           ; reached end of array1?
      BGE DONE
      LDR R0, [R1], #4     ; R0 = array1[i] and R1 update
      STR R0, [R2], #4     ; array2[i] = array1[i] and R2 update
      B    FOR
DONE
```

(c) part (a) has 8 instructions and part (b) has 6 instructions. The loop code particularly decreases from 7 instructions to 5 instructions. This is a 25% decrease in the number of instructions and a 29% decrease in loop instructions. The advantages are: (1) 25% lower memory requirements for code storage and (2) decreased execution time (approximately 29% decrease because most of the execution time is spent in the loop). The disadvantage is the number of bits required for encoding the indexing mode.

Exercise 6.21

(a)

```

        ADD R2, R3, #0x190    ; R2 = end of array
FOR    CMP R3, R2            ; reached end of array?
        BGE DONE
        LDR R1, [R3]          ; R1 = array[i]
        LSL R1, R1, #5        ; R1 = array[i] * 32
        STR R1, [R3]          ; array[i] = array[i] * 32
        ADD R3, R3, #4        ; R3 points to next array entry
        B    FOR
DONE

```

(b)

```

        ADD R2, R3, #0x190    ; R2 = end of array
FOR    CMP R3, R2            ; reached end of array?
        BGE DONE
        LDR R1, [R3]          ; R1 = array[i]
        LSL R1, R1, #7        ; R1 = array[i] * 128
        STR R1, [R3], #4      ; array[i] = array[i] * 128
                                ; R3 points to next array entry
        B    FOR
DONE

```

(c) part (a) has 8 instructions and part (b) has 7 instructions. The loop code particularly decreases from 7 instructions to 6 instructions. This is a 12.5% decrease in the number of instructions and a 14% decrease in loop instructions. The advantages are: (1) 12.5% lower memory requirements for code storage, and (2) decreased execution time (approximately 14% decrease because most of the execution time is spent in the loop). The disadvantage is the number of bits required for encoding the indexing mode.

Exercise 6.22

(a) Yes.

(b)

(i)

```

        MOV R1, #0            ; i = 0
FOR    CMP R1, #200           ; reached end of array?
        BGE DONE
        STR R1, [R0, R1, LSL #2] ; array[i] = i
        ADD R1, R1, #1        ; i = i + 1
        B    FOR

```

(ii)

```

MOV R1, #199 ; i = 199
FOR STR R1, [R0, R1, LSL #2] ; array[i] = i
SUBS R1, R1, #1 ; i = i - 1 and set flags
BPL FOR

```

(c) The second code snippet (ii), the decremented loop, uses fewer instructions and is faster. Each loop iteration in code snippet (ii) requires 3 instead of the 5 instructions required for code snippet (i). Code snippet ii combines checking the loop condition with updating the loop variable, i.

Exercise 6.23

(a) Yes.

(b)

(i)

```

MOV R1, #0 ; i = 0
FOR CMP R1, #10 ; reached end of array?
BGE DONE
LDR R2, [R0, R1, LSL #2] ; R2 = nums[i]
LSR R2, R2, #1 ; R2 = nums[i]/2
STR R2, [R0, R1, LSL #2] ; nums[i] = nums[i]/2
ADD R1, R1, #1 ; i = i + 1
B FOR
DONE

```

(ii)

```

MOV R1, #9 ; i = 9
FOR LDR R2, [R0, R1, LSL #2] ; R2 = nums[i]
LSR R2, R2, #1 ; R2 = nums[i]/2
STR R2, [R0, R1, LSL #2] ; nums[i] = nums[i]/2
SUBS R1, R1, #1 ; i = i - 1 and set flags
BPL FOR

```

(c) The second code snippet (ii), the decremented loop, uses fewer instructions and is faster. Each loop iteration in code snippet (ii) requires 5 instead of the 7 instructions required for code snippet (i). Code snippet ii combines checking the loop condition with updating the loop variable, i.

Exercise 6.24

```

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

```

```

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

return -1;
}

```

Exercise 6.25

(a)

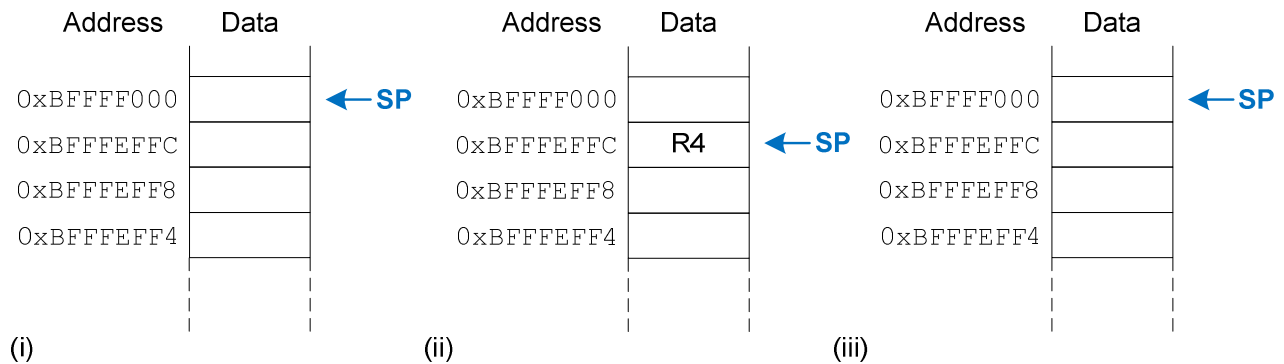
```

; ARM assembly code
; base address of array dst = R0
; base address of array src = R1
; i = R4

STRCPY
    PUSH {R4}                ; save R4 on stack
    MOV  R4, #0              ; i = 0
LOOP
    LDRB R2, [R1, R4]        ; R2 = src[i]
    STRB R2, [R0, R4]        ; dst[i] = src[i]
    CMP  R2, #0              ; array[i] == 0? (end of string?)
    ADD  R4, R4, #1          ; i++
    BNE  LOOP                ; if not, repeat
DONE
    POP  {R4}                ; restore R4
    MOV  PC, LR              ; return

```

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



Exercise 6.26

```

; ARM assembly
; R0 = base address of array
; R1 = number of elements in array
; R2 = i
    MOV  R2, #0                ; i = 0
LOOP
    CMP  R2, R1                ; i < size?
    BGE  DONE

```

```

        LDR    R3, [R0, R2, LSL #2]    ; R3 = array[i]
        CMP    R3, #42                ; array[i] == 42?
        ADDNE  R2, R2, #1              ; if not equal, increment i
        BNE    LOOP                    ;                and repeat loop
        MOV    R0, R2                  ; if EQ return i
        MOV    PC, LR

DONE
        MOV    R0, #0                  ; return -1
        SUB    R0, R0, #1
        MOV    PC, LR                  ; return

```

Exercise 6.27

(a)

func1: 8 words (for R4-R10 and LR)

func2: 3 words (for R4-R5 and LR)

func3: 4 words (for R7-R9 and LR)

func4: 1 word (for R11)

(b)

Address		Data
⋮		⋮
stack frame func1	BFFFFFF00	LR
	BFFFFFFFC	R10
	BFFFFFFF8	R9
	BFFFFFFF4	R8
	BFFFFFFF0	R7
	BFFFFFFEC	R6
	BFFFFFFE8	R5
	BFFFFFFE4	R4
stack frame func2	BFFFFFFE0	LR = 0x91024
	BFFFFFFDC	R5
	BFFFFFFD8	R4
stack frame func3	BFFFFFFD4	LR = 0x91180
	BFFFFFFD0	R9
	BFFFFFFEC	R8
	BFFFFFFC8	R7
stack frame func4	BFFFFFFC4	R11 ← SP
⋮		⋮

Exercise 6.28

(a)

fib(0) = 0

fib(-1) = 1

(b)

```

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

    // Calculate Fibonacci numbers from 0 - n
    while (n != 0) {
        result = result + prevresult; // fib(n) = fib(n-1) + fib(n-2)
        prevresult = result - prevresult; // fib(n-1) = fib(n) - fib(n-2)
        n = n - 1;
    }
    return result;
}

```


(c)

```
; fib.s
; The fib() function computes the nth Fibonacci number.
; n is passed to fib() in R0, and fib() returns the result in R0.
```

MAIN

```
MOV R0, #9      ; n = 9
BL  FIB         ; call Fibonacci function
```

...

; R1 = result; R2 = prevresult

FIB

```
MOV R1, #1      ; R1 = result = fib(0)
MOV R2, #0      ; R2 = prevresult = fib(-1)
CMP R0, #0      ; n == 0?
BEQ DONE
```

LOOP

```
ADD R1, R1, R2  ; result = result + prevresult
SUB R2, R1, R2  ; prevresult = result - prevresult
SUBS R0, R0, #1 ; n = n - 1
BPL LOOP
```

DONE

```
MOV R0, R2      ; return result
MOV PC, LR
```

Exercise 6.29

(a) 120

(b) (2)

- (c) (i) (3) returned value is $R1^4$
 (ii) (3) returned value is $R1^4$
 (iii) (4)

Exercise 6.30(a) 19. Yes, it correctly computes $2a + 3b$.

(b) (2)

(c) (i) (3) $R0 = 17$

(ii) (4)

(iii) (4) But the calling function may have a problem because R4 doesn't hold the value it had when it was called. Instead it holds the value 5.

(iv) (1)

(v) (2)

(vi) (3) $R0 = 17$

(vii) (1)

Exercise 6.31

- (a) 0xa0000001
 (b) 0xaa00000e
 (c) 0x8afff841
 (d) 0xeb00391d
 (e) 0xeaffe3fc

Exercise 6.32

(a)

Machine Code	Address/ARM Assembly
E1A04001	0x000A0028 FUNC1 MOV R4, R1
E0835125	0x000A002C ADD R5, R3, R5, LSR #2
E0404473	0x000A0030 SUB R4, R0, R3, ROR R4
EBFFFFFF	0x000A0034 BL FUNC2
E5902004	0x000A0038 FUNC2 LDR R2, [R0, #4]
E7012002	0x000A003C STR R2, [R1, -R2]
E3530000	0x000A0040 CMP R3, #0
1A000000	0x000A0044 BNE ELSE
E1A0F00E	0x000A0048 MOV PC, LR
E2433001	0x000A004C ELSE SUB R3, R3, #1
EAffFFFF8	0x000A0050 B FUNC2
	...

(b)

Addressing Mode	Address/ARM Assembly
Register (Register only)	0x000A0028 FUNC1 MOV R4, R1
Register (Immediate-shifted reg)	0x000A002C ADD R5, R3, R5, LSR #2
Register (Register-shifted reg)	0x000A0030 SUB R4, R0, R3, ROR R4
PC-Relative	0x000A0034 BL FUNC2
Base (Immediate offset)	0x000A0038 FUNC2 LDR R2, [R0, #4]
Base (Register offset)	0x000A003C STR R2, [R1, -R2]
Immediate	0x000A0040 CMP R3, #0
PC-Relative	0x000A0044 BNE ELSE
Register (Register only)	0x000A0048 MOV PC, LR
Immediate	0x000A004C ELSE SUB R3, R3, #1
PC-Relative	0x000A0050 B FUNC2
	...

Exercise 6.33

(a)

```

; R4 = i, R5 = num
SETARRAY
    PUSH {R4, R5, LR}          ; save R4, R5, and LR on the stack
    SUB SP, SP, #40           ; allocate space on stack for array

```

```

        MOV    R4, #0                ; i = 0
        MOV    R5, R0                ; R5 = num

LOOP    MOV    R1, R4                ; set up input arguments
        BL     COMPARE               ; call compare function
        STR    R0, [SP, R4, LSL #2] ; array[i] = return value
        ADD    R4, R4, #1            ; increment i
        MOV    R0, R5                ; arg0 = num
        CMP    R4, #10               ; i < 10?
        BLT    LOOP

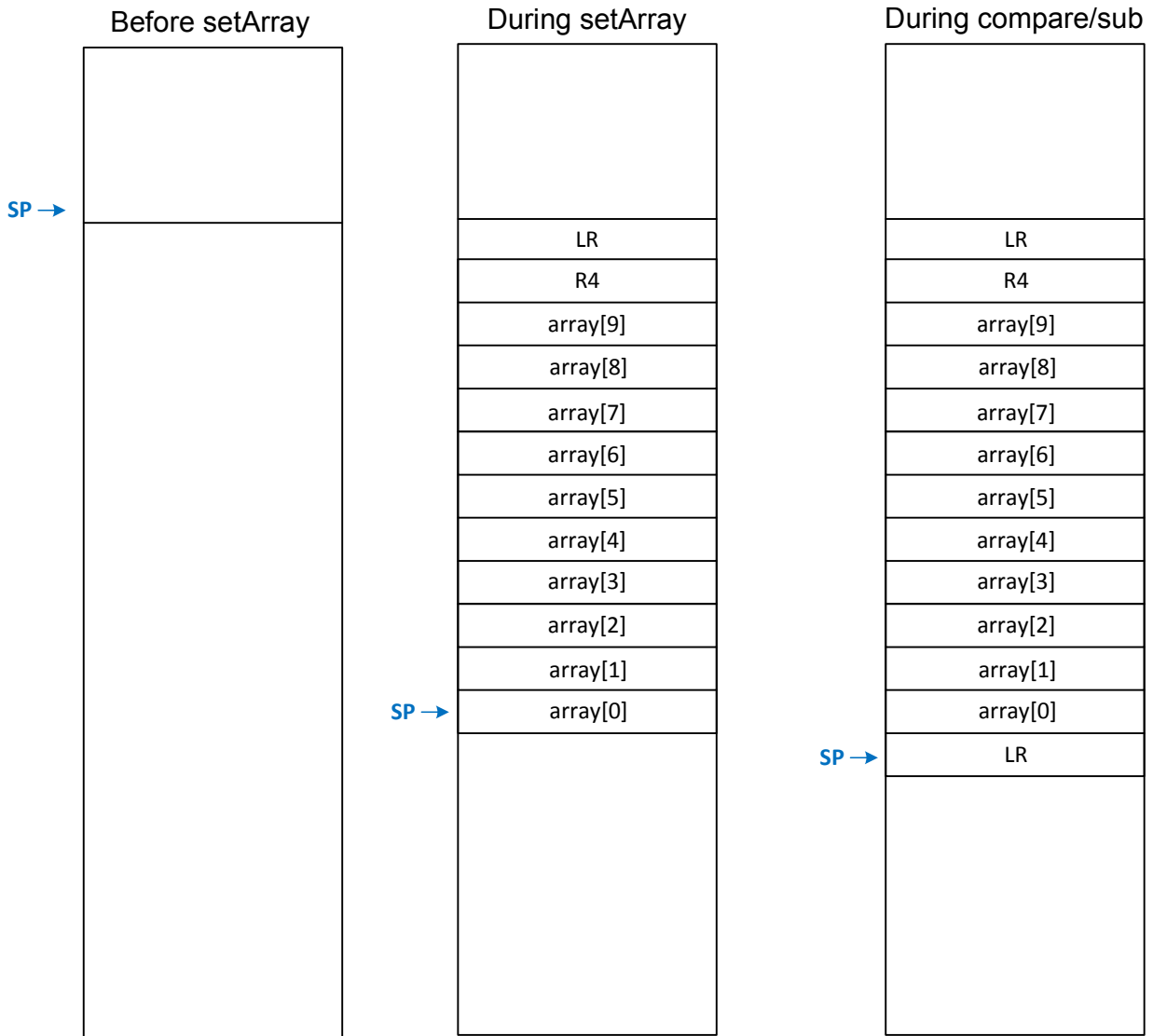
        ADD    SP, SP, #40           ; deallocate space on stack for array
        POP    {R4, R5, LR}          ; restore registers
        MOV    PC, LR                ; return to point of call

COMPARE
        PUSH   {LR}                  ; save LR
        BL     SUBFUNC               ; call sub function
        CMP    R0, #0                ; returned value >= 0?
        MOVGE  R0, #1                ; if yes, R0 = 1
        MOVLT  R0, #0                ; if no, R0 = 0
        POP    {LR}                  ; restore LR
        MOV    PC, LR                ; return to point of call

SUBFUNC
        SUB    R0, R0, R1             ; return a-b
        MOV    PC, LR                ; return to point of call

```

(b)



(c) The code would enter an infinite loop and eventually crash. When the `compare` function returns (`MOV PC, LR`), instead of returning to its point of call in the `setArray` function, the `compare` function would continue executing at the instruction just after the call to `sub` (`BL SUBFUNC`). Because of the `POP {LR}` instruction, the program would eventually crash when it went beyond the stack space available (i.e., the stack pointer was decremented past the allocated dynamic data segment).

Exercise 6.34

(a)

```
;R4 = b
;Address    ARM Assembly
```


farthest it can branch forward is to instruction address $16,777,217 * 4 = 67,108,868 = 0x4000004$.

Exercise 6.36

(a) Because branches in the ARM architecture are relative to $PC + 8$, the limitation on branch range is independent of the current instruction address. The range of a forward branch is $(2^{24}-1) + 2 = 16,777,217$ instructions. So a forward branch can branch from **0 to 16,777,217 instructions** relative to the branch instruction. So, if the current instruction address is **0x0**. The farthest it can branch forward is to instruction address $16,777,217 * 4 = 67,108,868 = 0x4000004$.

(b) Same as (a).

(c) Again, the limitation on branch range is independent of the current instruction address. The range of a backward branch is $(2^{24}) - 2 = 16,777,214$ instructions. So a backward branch can branch from **0 to 16,777,214 instructions** relative to the branch instruction. For example, a branch at address $0x3ffffff8$ ($16,777,214 * 4$) could branch back to instruction address $0x0$.

(d) Same as (c).

Exercise 6.37

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

Exercise 6.38

To branch to an instruction 2^{20} instructions from the branch instruction, the target address will be at: $0x8000 + (2^{20}*4) = 0x8000 + (2^{22}) = 0x8000 + 0x400000 = 0x408000$. (imm24 must have the value $2^{20} - 8 = 0x00FFF8$.)

```
;Address          ;ARM Assembly
0x00008000        B      DEST
...
0x00408000 DEST ...
```

Exercise 6.39

```
// High-Level Code
void little2big(int[] array) {
    int i;

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

```

        ((array[i] >> 24) & 0xFF));
    }
}

; ARM Assembly Code
; R0 = base address of array, R12 = i
little2BIG
    MOV    R12, #0                ; i = 0
LOOP
    CMP    R12, #10              ; i < 10?
    BGE    DONE
    LDR    R2, [R0, R12, LSL #2] ; R2 = array[i]
    LSL    R3, R2, #24           ; R3 = array[i] << 24
    AND    R4, R2, #0xFF00      ; R4 = (array[i] & 0xFF00)
    ORR    R3, R3, R4, LSL #8   ; R3 = top two bytes
    AND    R4, R2, #0xFF0000    ; R4 = (array[i] & 0xFF0000)
    ORR    R3, R3, R4, LSR #8   ; R3 = top three bytes
    ORR    R3, R3, R2, LSR #24  ; R3 = all four bytes
    STR    R3, [R0, R12, LSL #2] ; array[i] = R3
    ADD    R12, R12, #1         ; increment i
    B      LOOP
DONE
    MOV    PC, LR

```

Exercise 6.40

(a)

```

void concat(char[] string1, char[] string2, char[] stringconcat) {
    int i, j;

    i = 0;
    j = 0;

    while (string1[i] != 0) {
        stringconcat[i] = string1[i];
        i = i + 1;
    }

    while (string2[j] != 0) {
        stringconcat[i] = string2[j];
        i = i + 1;
        j = j + 1;
    }

    stringconcat[i] = 0; // append null at end of string
}

```

(b)

CONCAT

```

        LDRB R3, [R0], #1      ; R3 = string1[i]
        CMP  R3, #0           ; string1[i] != 0?
        BEQ  STR2
        STRB R3, [R2], #1      ; stringconcat[i] = string1[i]
        B    CONCAT

STR2
        LDRB R3, [R1], #1      ; R3 = string2[j]
        CMP  R3, #0           ; string2[j] != 0?
        BEQ  DONE
        STRB R3, [R2], #1      ; stringconcat[i] = string2[j]
        B    STR2
        MOV  R3, #0
        STRB R3, [R2]          ; append null at end of string
        MOV  PC, LR            ; return to point of call

```

Exercise 6.41

; R4, R5 = mantissas of a, b, R6, R7 = exponents of a, b

```

FLPADDD
        PUSH {R4, R5, R6, R7, R8} ; save registers that will be used
        LDR  R2, =0x007fffff      ; load mantissa mask
        LDR  R3, =0x7f800000      ; load exponent mask
        AND  R4, R0, R2           ; extract mantissa from R0 (a)
        AND  R5, R1, R2           ; extract mantissa from R1 (b)
        ORR  R4, R4, #0x800000    ; insert implicit leading 1
        ORR  R5, R5, #0x800000    ; insert implicit leading 1
        AND  R6, R0, R3           ; extract exponent from R0 (a)
        LSR  R6, R6, #23          ; shift exponent right
        AND  R7, R1, R3           ; extract exponent from R1 (b)
        LSR  R7, R7, #23          ; shift exponent right

MATCH
        CMP  R6, R7               ; compare exponents
        BEQ  ADDMANTISSA          ; if equal, skip to adding mantissas
        BHI  SHIFTB               ; if a's exponent is bigger, shift b

SHIFTA
        SUB  R8, R7, R6           ; R8 = b's exponent - a's exponent
        ASR  R4, R4, R8           ; right-shift a's mantissa
        ADD  R6, R6, R8           ; update a's exponent
        B    ADDMANTISSA          ; now add the mantissas

SHIFTB
        SUB  R8, R6, R7           ; R8 = a's exponent - b's exponent
        ASR  R5, R5, R8           ; right-shift b's mantissa

ADDMANTISSA
        ADD  R4, R4, R5           ; R4 = sum of mantissas

```


NORMALIZE

```

ANDS R5, R4, #0x1000000    ; extract overflow bit
BEQ  DONE                  ; branch to DONE if bit 24 == 0
LSR  R4, R4, #1             ; right-shift mantissa by 1 bit
ADD  R6, R6, #1             ; increment exponent

```

DONE

```

AND  R4, R4, R2             ; mask fraction
LSL  R6, R6, #23            ; shift exponent into place
ORR  R0, R4, R6             ; combine mantissa and exponent
POP  {R4, R5, R6, R7, R8}   ; restore registers
MOV  PC, LR                 ; return to caller

```

Exercise 6.42

(a)

```

; ARM Assembly Code
0x08400    MAIN  PUSH {LR}
0x08404          LDR R2, =L1
0x0840c          LDR R0, [R2]
0x08410          LDR R1, [R2, #4]
0x08414          BL DIFF
0x08418          POP {LR}
0x0841c          MOV PC, LR
0x08420    DIFF  SUB R0, R0, R1
0x08424          MOV PC, LR

...
0x9024     L1

```

(b)

Symbol Table

Address	Label
0x8400	MAIN
0x8420	DIFF
0x9024	L1

(c)

```

; machine code      ;address      ARM assembly
e52de004            ;0x08400    MAIN  PUSH {LR}  ; STR R14,[R13,#-4]!
e59f2c18            ;0x08404          LDR R2, =L1
e5920000            ;0x08408          LDR R0, [R2]
e5921004            ;0x0840c          LDR R1, [R2, #4]
eb000001            ;0x08410          BL DIFF
e49de004            ;0x08414          POP {LR}    ; LDR R14,[R13],#4
e1a0f00e            ;0x08418          MOV PC, LR
e0400001            ;0x0841c    DIFF  SUB R0, R0, R1
e1a0f00e            ;0x08420          MOV PC, LR

...
;0x09024    L1      ; holds address of the data

```

(d)

Text Segment: 10*4 = 40 bytes**Data segment:** 4 bytes**Exercise 6.43**

(a)

```

; ARM assembly code
0x8534      MAIN      PUSH {R4,LR}
0x8538                      MOV R4, #15
0x853c                      LDR R3, =L2
0x8540                      STR R4, [R3]
0x8544                      MOV R1, #27
0x8548                      STR R1, [R3, #4]
0x854c                      LDR R0, [R3]
0x8550                      BL GREATER
0x8554                      POP {R4,LR}
0x8558                      MOV PC, LR
0x855c      GREATER   CMP R0, R1
0x8560                      MOV R0, #0
0x8564                      MOVGT R0, #1
0x8568                      MOV PC, LR
...
0x9305      L2

```

(b)

Symbol Table

Address	Label
0x8534	MAIN
0x8550	GREATER
0x9305	L2

(c)

;	machine code	;	address		ARM assembly
E92D4010		;	0x8534	MAIN	PUSH {R4,LR}
		;			STMDB R13!, {R4,R14}
E3A0400F		;	0x8538		MOV R4, #15
E59F3DC1		;	0x853c		LDR R3, =L2
E5834000		;	0x8540		STR R4, [R3]
E3A0101B		;	0x8544		MOV R1, #27
E5831004		;	0x8548		STR R1, [R3, #4]

```

E5930000          ;0x854c          LDR R0, [R3]
EB000001          ;0x8550          BL GREATER
E8BD4010          ;0x8554          POP {R4,LR}
                  ;                LDMIA R13!, {R4,R14}
E1A0F00E          ;0x8558          MOV PC, LR
E1500001          ;0x855c          GREATER CMP R0, R1
E3A00000          ;0x8560          MOV R0, #0
C3A00001          ;0x8564          MOVGT R0, #1
E1A0F00E          ;0x8568          MOV PC, LR
...
                  ;0x9305          L2

```

(d)

Text Segment: 15*4 = 60 bytes**Data segment:** 4 bytes**Exercise 6.44**

1. Scaled register offset for accessing memory:

Accessing an array of integers using an index in R3 starting at a base address in R0:

Without scaled register offset:

```

    LSL R4, R3, #2          ; multiply index i by 4
    LDR R5, [R0, R4]        ; access array

```

With scaled register offset:

```

    LDR R5, [R0, R3, LSL #2] ; access array

```

2. Pre-indexing or Post-indexing:

Accessing an array of characters at base address in R0:

Without pre-indexing:

```

    REPEAT    LDR R5, [R0, #1]          ; access array
              ADD R0, R0, #1
    ...
    BLT REPEAT

```

Without pre-indexing:

```

    REPEAT    LDR R5, [R0, #1]!         ; access array
    ...
    BLT REPEAT

```

3. Conditional execution

Executing an if statement that sets R4 to 10 when R2 and R3 are equal

Without conditional execution:

```

        CMP R2, R3      ; R2 == R3?
        BNE L3
        MOV R4, #10     ; R4 = 10
L3      ...

```

With conditional execution:

```

        CMP R2, R3      ; R2 == R3?

        MOVEQ R4, #10   ; R4 = 10 when R2 == R3
L3      ...

```

Exercise 6.45

Advantages of conditional execution:

- Potentially decreased code size (increased code density)
- Potentially decreased execution time (improved performance)

Disadvantages:

- More complex hardware required to implement it
- Requires 4 instruction bits to encode

Question 6.1

```

EOR R0, R0, R1  ; R0 = R0 XOR R1
EOR R1, R0, R1  ; R1 = original value of R0
EOR R0, R0, R1  ; R0 = original value of R1

```

Question 6.2

C Code

```
// Find subset of array with largest sum
```

```

int max = -2,147,483,648; // -2^31
int start = 0;
int end = 0;

for (i=0; i<length; i++) {
    sum = 0;
    for (j=i; j<length; j++) {
        sum = sum + array[j];
        if (sum > max) {

```

```

        max = sum;
        start = i;
        end = j;
    }
}

count = 0;
for (i = start; i <= end; i++) {
    array2[count] = array[i];
    count = count + 1;
}

```

ARM Assembly Code

```

; R0 = base address of array, R1 = length of array
; R2 = base address of resulting array
; R3 = max, R4 = start, R5 = end
; R6 = i, R7 = j and count, R8 = sum
    PUSH {R4,R5,R6,R7,R8,R9}      ; save registers
    MOV R3, #0x80000000            ; R3 = large negative number
    MOV R4, #0                    ; start = 0
    MOV R5, #0                    ; end = 0

    MOV R6, #0                    ; i = 0
    LSL R1, R1, #2                ; length = length * 4
LOOPFORI
    CMP R6, R1                    ; i < length?
    BGE ENDLOOP
    MOV R8, #0                    ; reset sum
    MOV R7, R6                    ; j = i
LOOPFORJ
    CMP R7, R1                    ; j < length?
    BGE INCREMENTI
    LDR R9, [R0, R7]              ; R9 = array[j]
    ADD R8, R8, R9                ; sum = sum + array[j]
    CMP R3, R8                    ; max < sum?
    BGE INCREMENTJ
    MOV R3, R8                    ; max = sum
    MOV R4, R6                    ; start = i
    MOV R5, R7                    ; end = j
INCREMENTJ
    ADD R7, R7, #4                ; j = j + 4
    B    LOOPFORJ
INCREMENTI
    ADD R6, R6, #4                ; i = i + 4
    B    LOOPFORI
ENDLOOP

```

```

        MOV R6, R4                ; i = start
        MOV R7, #0                ; count = 0
LOOP3
        CMP R5, R6                ; end < i?
        BLT RETURN
        LDR R9, [R0, R6]          ; R9 = array[i]
        STR R9, [R2, R7]          ; array2[count] = array[i]
        ADD R7, R7, #4            ; count = count + 4
        ADD R6, R6, #4            ; i = i + 4
        B LOOP3
RETURN
        POP {R4,R5,R6,R7,R8,R9}  ; restore registers
        MOV PC, LR

```

Question 6.3

C Code

```

void reversewords(char[] array) {
    int i, j, length;

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

    length = i;

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

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

void reverse(char[] array, int i, int j) {
    char tmp;

    while (i > j) {
        tmp = array[i];

```

```

    array[i] = array[j];
    array[j] = tmp;
    i = i-1;
    j = j+1;
}
}

```

ARM Assembly

```

; R4 = i, R5 = j, R6 = length
REVERSEWORDS
    PUSH    {R4,R5,R6}        ; save registers on stack
    MOV     R4, #0             ; i = 0
GETLENGTH
    LDRB    R1, [R0, R4]      ; R1 = array[i]
    CMP     R1, #0             ; end of string?
    ADDNE   R4, R4, #1        ; i = i + 1
    BNE     GETLENGTH
STRINGREVERSE
    MOV     R6, R4             ; length = i
    SUB     R1, R6, #1        ; arg1 = length-1
    MOV     R2, #0             ; arg2 = 0
    BL      REVERSE           ; call reverse function
    MOV     R4, #0             ; i = 0
    MOV     R5, #0             ; j = 0
WHILE19
    CMP     R4, R6             ; i <= length?
    BGT     DONE19             ; if at end of string, return
    BEQ     ELSE19             ; if (i == length), do else block
    LDRB    R1, [R0, R4]      ; R1 = array[i]
    CMP     R1, #0x20          ; array[i] != 0x20?
    BEQ     ELSE19             ; if (array[i] == 0x20), do else block
    ADD     R4, R4, #1
    B       WHILE19           ; repeat while loop
ELSE19
    SUB     R1, R4, #1         ; arg1 = i-1
    MOV     R2, R5             ; arg2 = j
    BL      REVERSE           ; call reverse function
    ADD     R4, R4, #1         ; i = i+1
    MOV     R5, R4             ; j = i
    B       WHILE19           ; repeat while loop
DONE19
    POP     {R4,R5,R6}        ; restore registers from stack
    MOV     PC, LR            ; retn to calling function

REVERSE
    CMP     R1, R2             ; i > j?
    BLE     RETURN19
    LDRB    R3, [R0, R1]      ; R3 = array[i]
    LDRB    R12, [R0, R2]     ; R12 = array[j]
    STRB    R12, [R0, R1]     ; array[i] = array[j]

```

```

        STRB  R3, [R0, R2]      ; array[j] = tmp
        SUB   R1, R1, #1        ; i = i - 1
        ADD   R2, R2, #1        ; j = j + 1
        B     REVERSE           ; continue while loop
RETURN19
        MOV   PC, LR

```

Question 6.4

C Code

```

int count = 0;

while (num != 0) {
    if (num & 1)
        count = count + 1;
    num = num >> 1;
}

```

ARM Assembly Code

```

; R0 = num, R1 = count
        LDR   R3, =0x345
        MOV   R1, #0                ; count = 0
        CMP   R3, #0                ; num == 0?
        BEQ   DONE
COUNTONES
        ANDS  R2, R3, #1            ; R2 = num & 1, set flags
        BEQ   SHIFT                ; if result of AND is 0, shift only
        ADD   R1, R1, #1            ; else increment count
SHIFT
        LSRS  R3, R3, #1            ; shift num right by 1 bit, set flags
        BNE   COUNTONES            ; continue counting ones if num != 0
DONE

```

Question 6.5

C Code

```

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

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

```

ARM Assembly Code

```

        MOV   R0, R3                ; arg0 = num
        MOV   R1, #1                ; arg1 = 1

```



```

LDR R2, =0x55555555      ; arg2 = 0x55555555
BL  SWAP                  ; call swap function

MOV R1, #2                ; arg1 = 1
LDR R2, =0x33333333      ; arg2 = 0x33333333
BL  SWAP                  ; call swap function

MOV R1, #4                ; arg1 = 1
LDR R2, =0x0F0F0F0F      ; arg2 = 0x0F0F0F0F
BL  SWAP                  ; call swap function

MOV R1, #8                ; arg1 = 1
LDR R2, =0x00FF00FF      ; arg2 = 0x00FF00FF
BL  SWAP                  ; call swap function

MOV R1, #16               ; arg1 = 1
LDR R2, =0xFFFFFFFF      ; arg2 = 0xFFFFFFFF
BL  SWAP                  ; call swap function
MOV R3, R0                ; num = returned value
...

SWAP
LSR  R3, R0, R1           ; R3 = num >> shamt
AND  R3, R3, R2           ; R3 = (num >> shamt) & mask
AND  R0, R0, R2           ; R0 = num & mask
LSL  R0, R0, R1           ; R0 = (num & mask) << shamt
ORR  R0, R3, R0           ; return val = R3 | R0
MOV  PC, LR              ; return to caller

```

Question 6.6

```

ADDs R0, R2, R3           ; R0 = R2 + R3, set flags
BVS  OVERFLOW
NOOVERFLOW
...
OVERFLOW
...

```

Question 6.7**C Code**

```

bool palindrome(char* array) {
    int i, j; // array indices

    // find length of string
    for (j = 0; array[j] != 0; j=j+1) ;
    j = j-1; // j is index of last char

    i = 0;
    while (j > i) {
        if (array[i] != array[j])

```

```

        return false;
    j = j-1;
    i = i+1;
}
return true;
}

```

MIPS Assembly Code

```

; R1 = i, R2 = j, R0 = base address of string
PALINDROME
    PUSH    {R4}                ; save R4 on stack
    MOV     R2, #0              ; j = 0
GETLENGTH
    LDRB    R3, [R0, R2]        ; R3 = array[j]
    CMP     R3, #0              ; end of string?
    ADDNE   R2, R2, #1          ; j = j + 1
    BNE     GETLENGTH
    SUB     R2, R2, #1          ; j = j - 1
    MOV     R1, #0              ; i = 0
WHILE
    CMP     R2, R1              ; j > i?
    BLE     RETURNTRUE
    LDRB    R3, [R0, R1]        ; R3 = array[i]
    LDRB    R4, [R0, R2]        ; R4 = array[j]
    CMP     R3, R4              ; array[i] == array[j]?
    BNE     RETURNFALSE
    SUB     R2, R2, #1          ; j = j-1
    ADD     R1, R1, #1          ; i = i+1
    B       WHILE
RETURNTRUE
    MOV     R0, #1              ; return TRUE
    B       DONE
RETURNFALSE
    MOV     R0, #0              ; return FALSE
DONE
    POP     {R4}                ; restore R4
    MOV     PC, LR              ; return to caller

```

CHAPTER 7

Exercise 7.1

- (a) ADD, SUB, AND, ORR, LDR: the result never gets written to the register file.
- (b) SUB, AND, ORR: the ALU only performs addition
- (c) STR: the data memory never gets written

Exercise 7.2

- (a) STR, B: these instructions write to the register file when they shouldn't.
- (b) LDR, STR, B: the ALU looks at the cmd field to determine the operation to perform. However, for these instructions, the ALU should always perform addition.
- (b) ADD, SUB, AND, ORR, LDR, B: these instructions inadvertently write to the data memory.

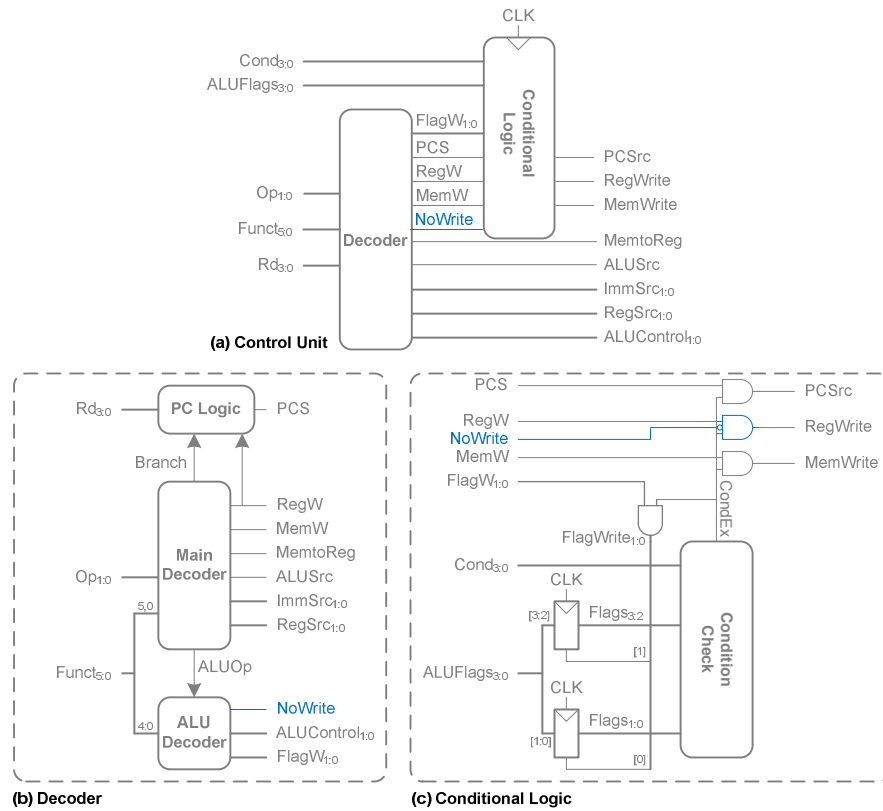
Exercise 7.3

- (a) TST

ALU Decoder truth table

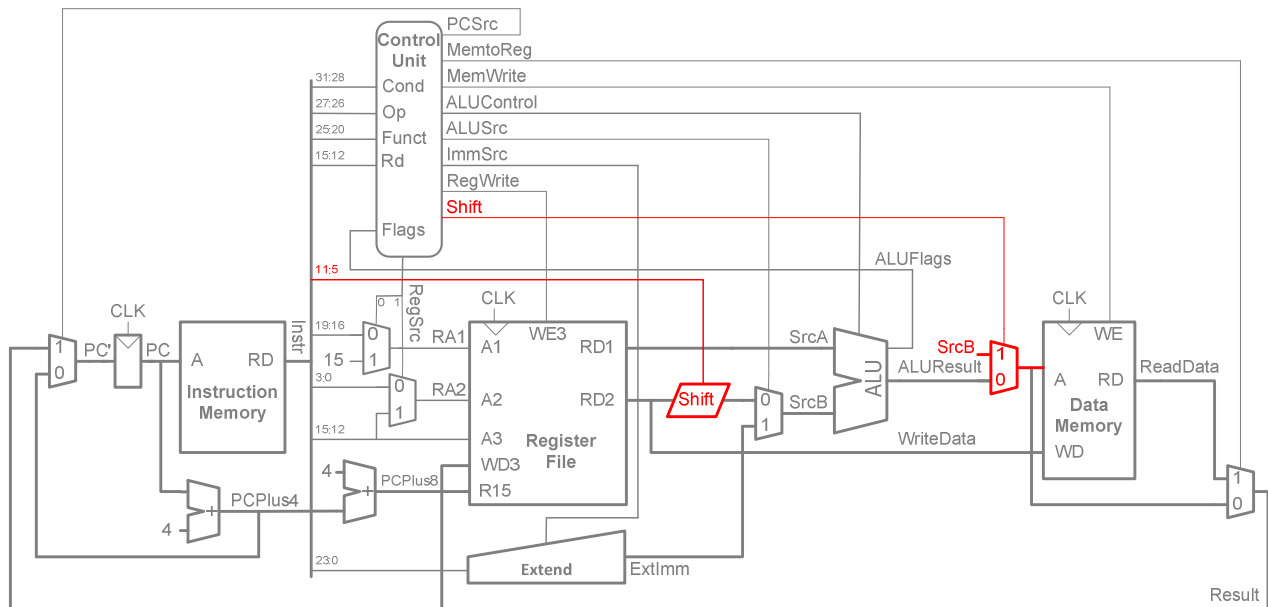
ALUOp	Funct _{4:1} (cmd)	Funct ₀ (S)	Notes	ALUControl _{1:0}	FlagW _{1:0}	NoWrite
0	X	X	Not DP	00	00	0
1	0100	0	ADD	00	00	0
		1			11	0
	0010	0	SUB	01	00	0
		1			11	0
	0000	0	AND	10	00	0
		1			10	0
	1100	0	ORR	11	00	0
		1			10	0
	1000	1	TST	10	10	1

Control Unit Schematic

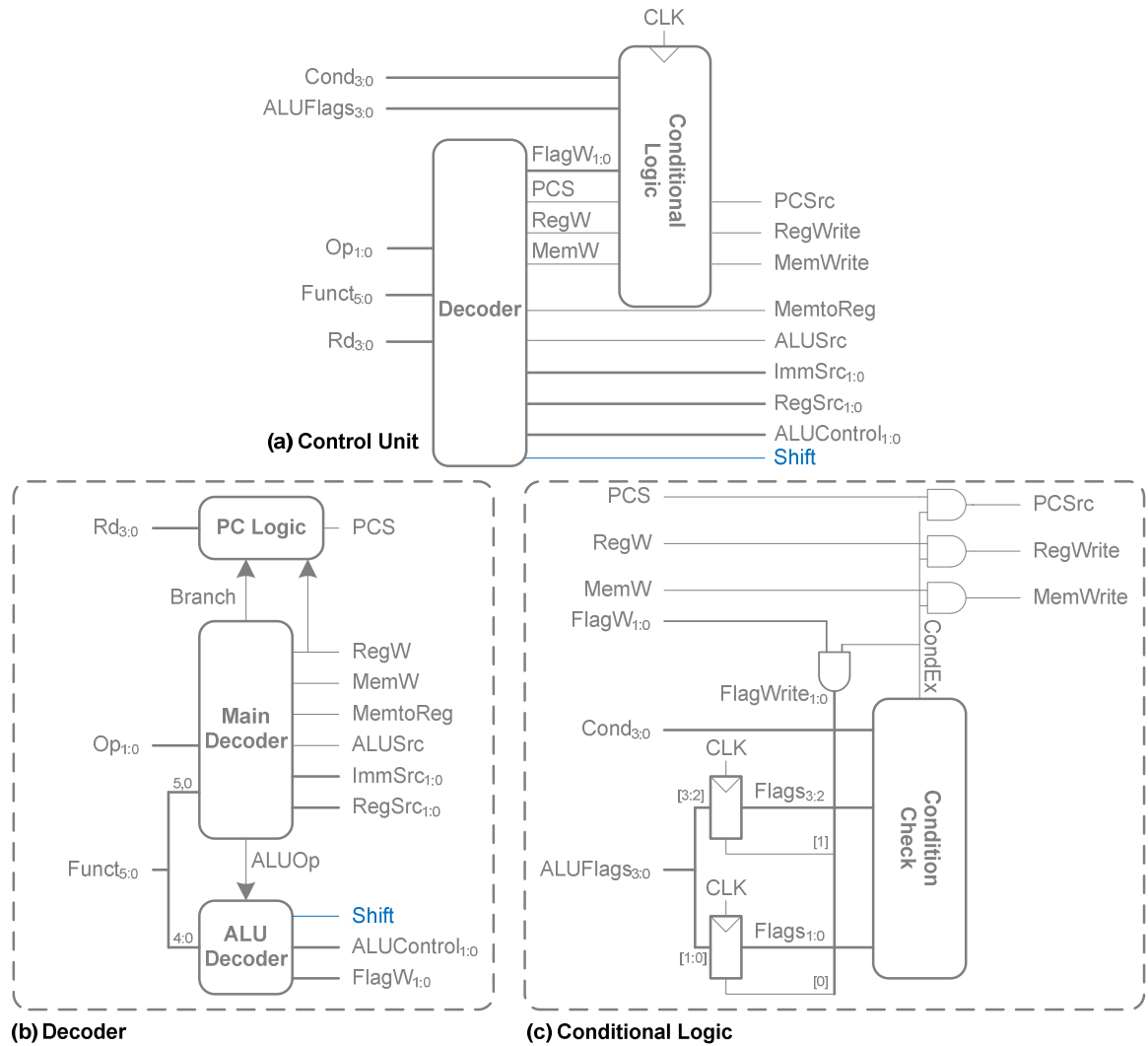


(b) LSL

Single-cycle datapath



Control unit



ALU Decoder truth table

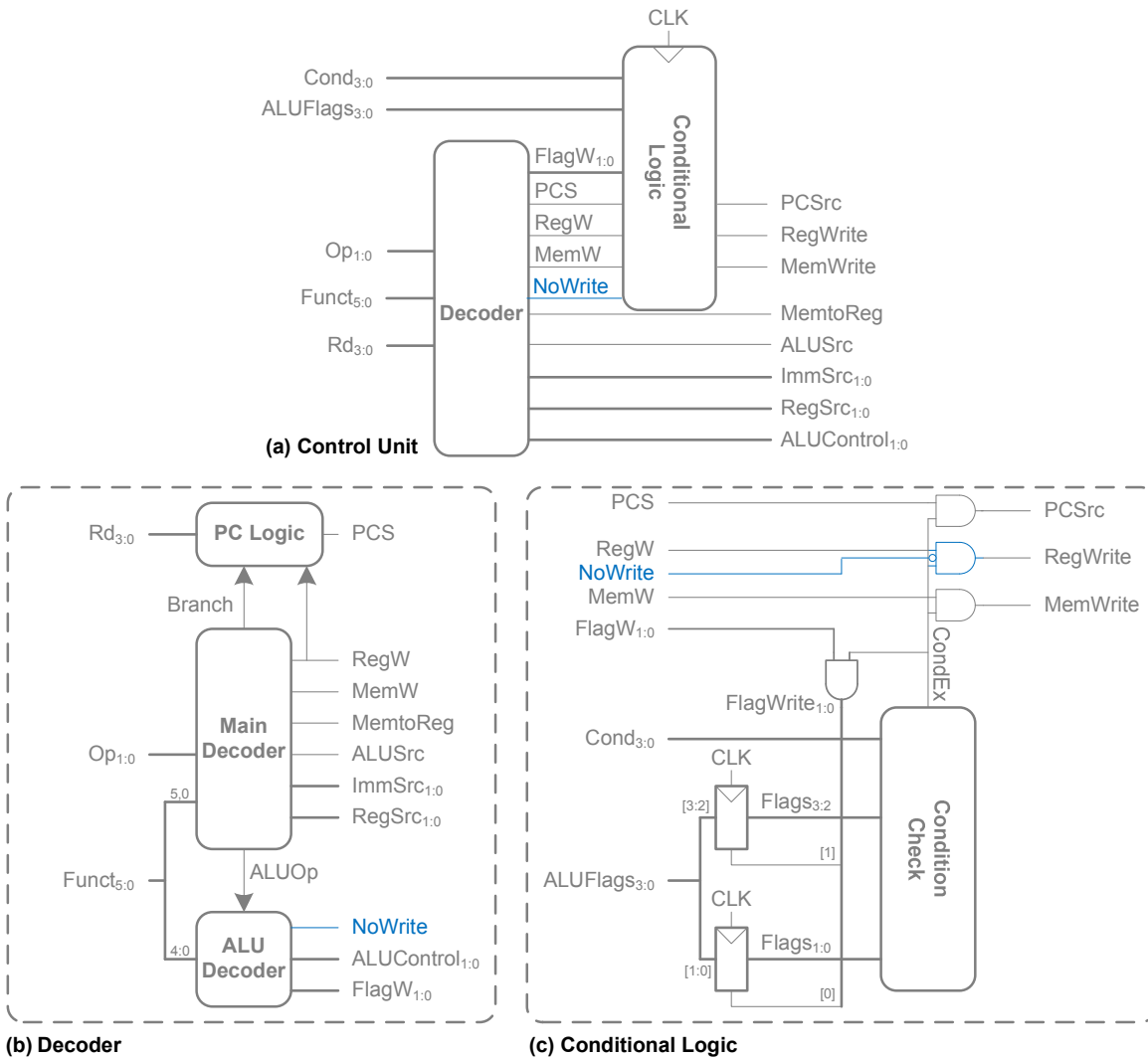
<i>ALUOp</i>	<i>Funct</i> _{4:1} (<i>cmd</i>)	<i>Funct</i> ₀ (<i>S</i>)	Notes	<i>ALUControl</i> _{1:0}	<i>FlagW</i> _{1:0}	<i>Shift</i>
0	X	X	Not DP	00	00	0
1	0100	0	ADD	00	00	0
		1			11	0
	0010	0	SUB	01	00	0
		1			11	0
	0000	0	AND	10	00	0
		1			10	0
	1100	0	ORR	11	00	0
		1			10	0
	1101	0	LSL	XX	00	1
		1			10	1

(c) CMN

ALU Decoder truth table

<i>ALUOp</i>	<i>Funct</i> _{4:1} (<i>cmd</i>)	<i>Funct</i> ₀ (<i>S</i>)	Notes	<i>ALUControl</i> _{1:0}	<i>FlagW</i> _{1:0}	<i>NoWrite</i>
0	X	X	Not DP	00	00	0
1	0100	0	ADD	00	00	0
		1			11	0
	0010	0	SUB	01	00	0
		1			11	0
	0000	0	AND	10	00	0
		1			10	0
	1100	0	ORR	11	00	0
		1			10	0
	1011	1	CMN	00	11	1

Control Unit schematic



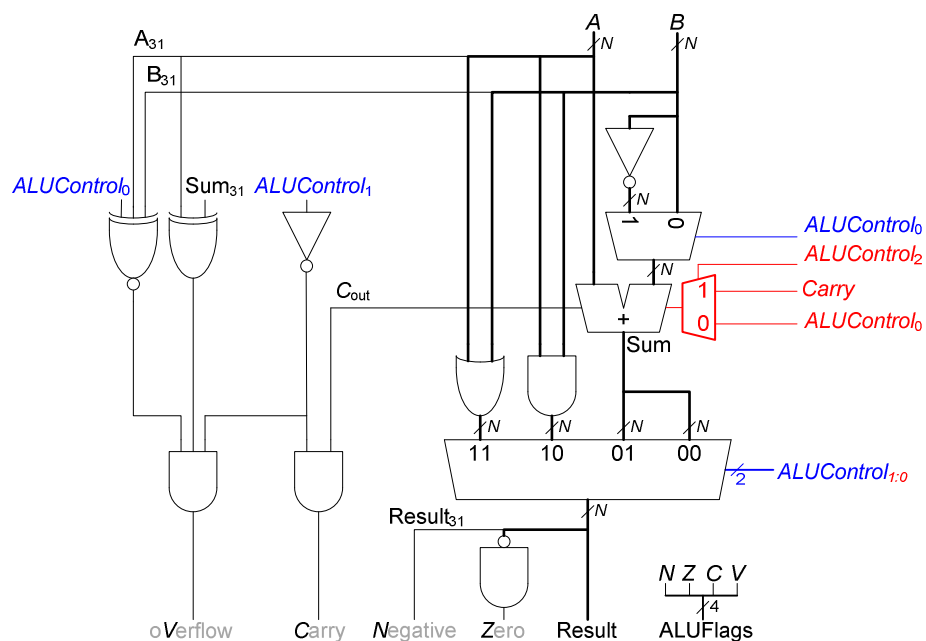
(d) ADC

ALU Decoder truth table

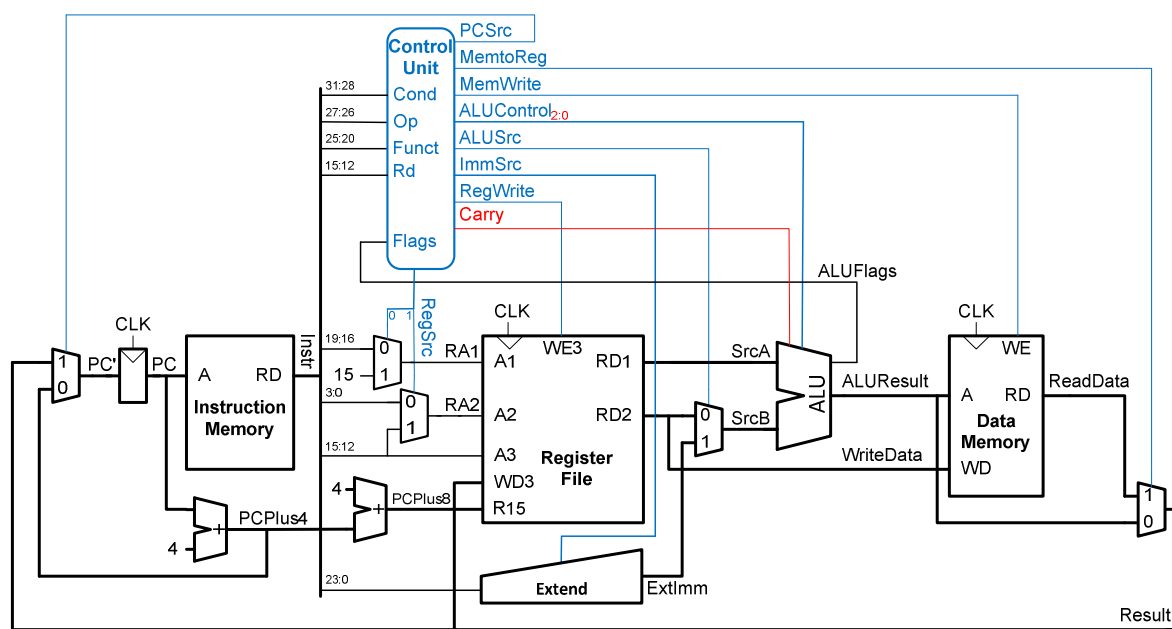
$ALUOp$	$Funct_{4:1} (cmd)$	$Funct_0 (S)$	Notes	$ALUControl_{2:0}$	$FlagW_{1:0}$
0	X	X	Not DP	000	00
1	0100	0	ADD	000	00
		1			11
	0010	0	SUB	001	00
		1			11
	0000	0	AND	010	00
		1			10
	1100	0	ORR	011	00
		1			10

0101	0	ADC	100	00
	1			11

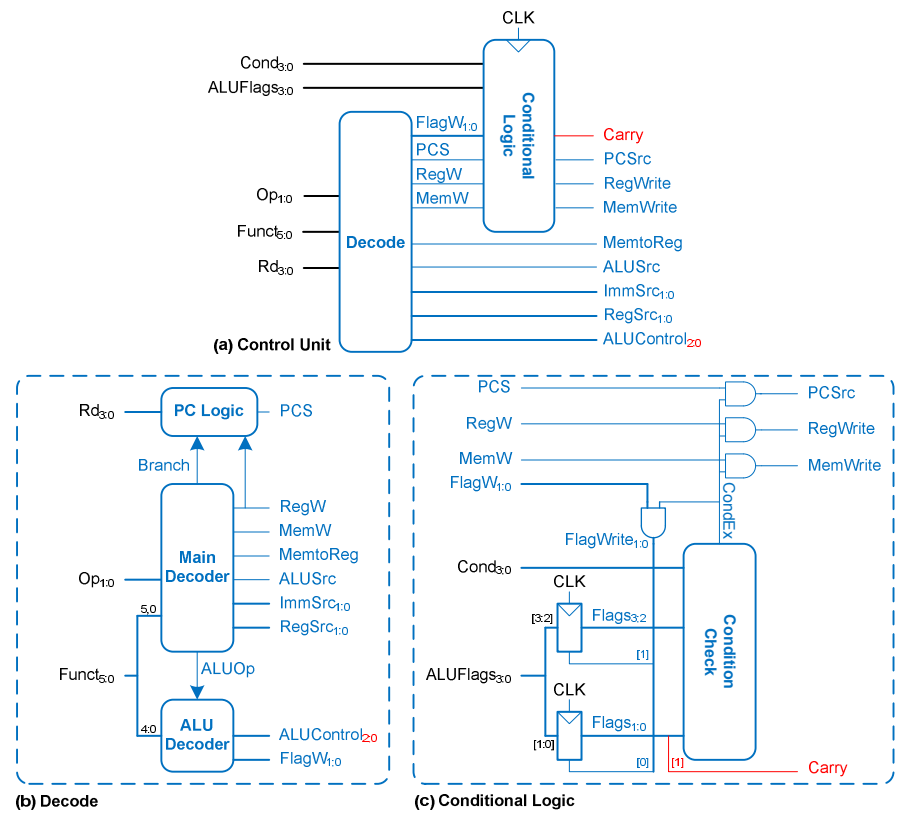
Single-cycle ARM processor ALU



Single-cycle ARM processor datapath



Single-cycle ARM processor control unit



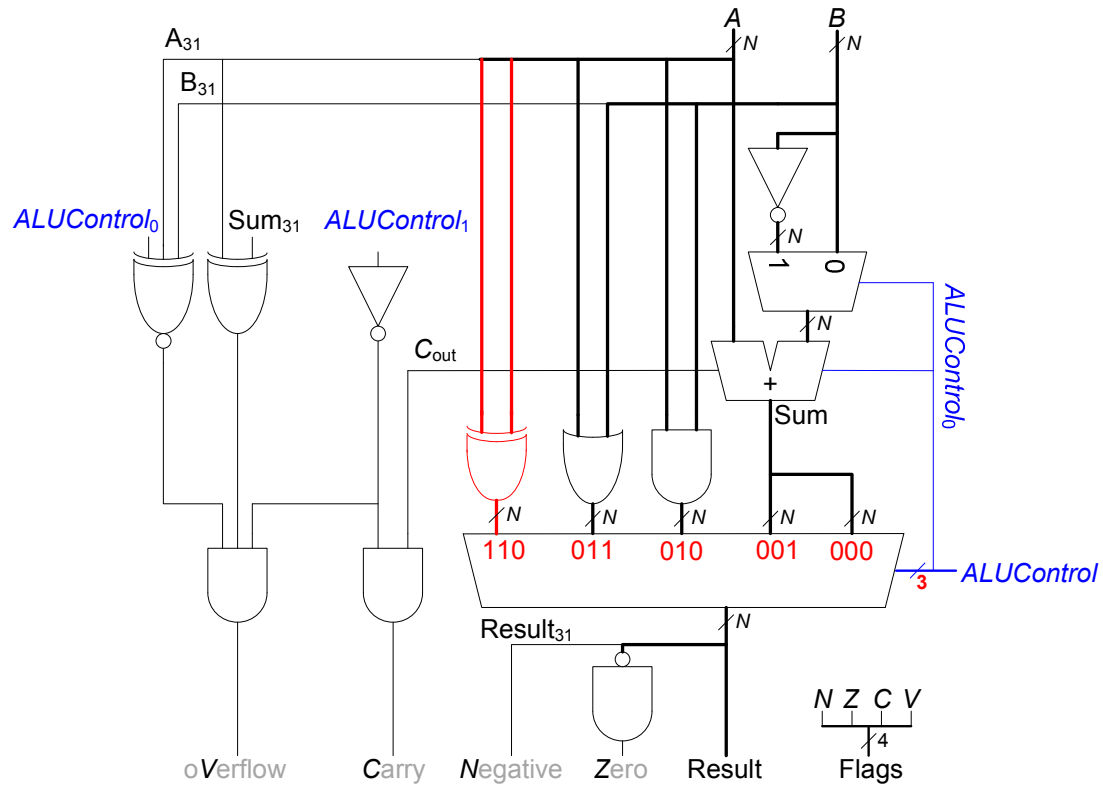
Exercise 7.4

(a) EOR

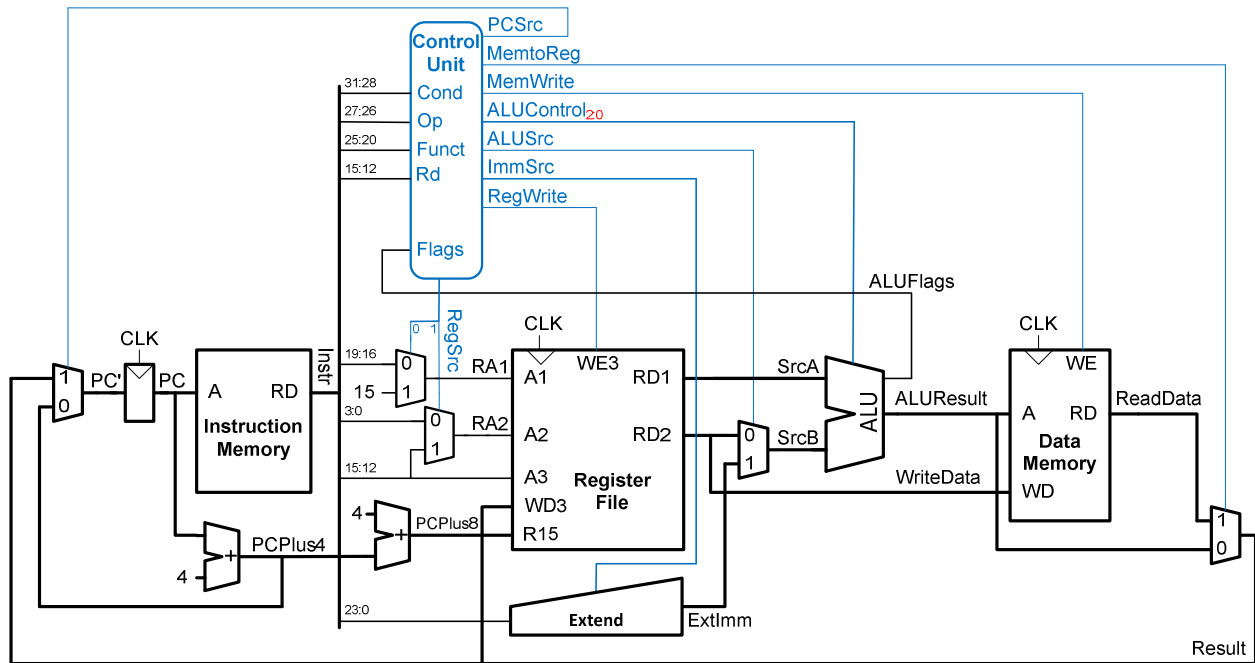
ALU Decoder truth table

$ALUOp$	$Funct_{4:1} (cmd)$	$Funct_0 (S)$	Notes	$ALUControl_{2:0}$	$FlagW_{1:0}$
0	X	X	Not DP	000	00
1	0100	0	ADD	000	00
		1			11
	0010	0	SUB	001	00
		1			11
	0000	0	AND	010	00
		1			10
	1100	0	ORR	011	00
		1			10
	0001	0	EOR	110	00
		1			10

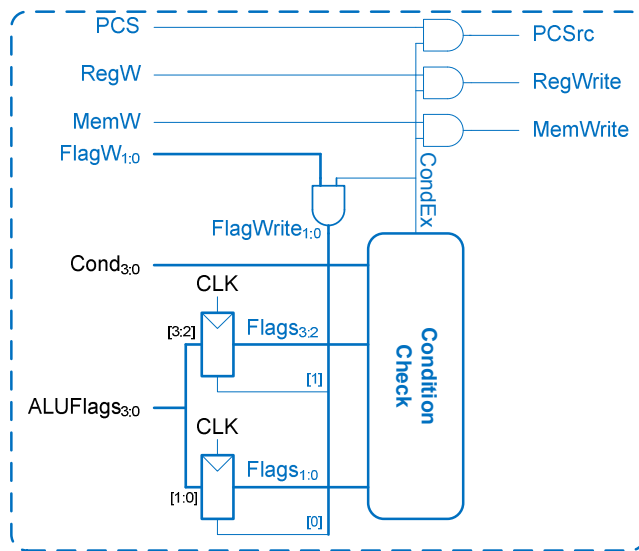
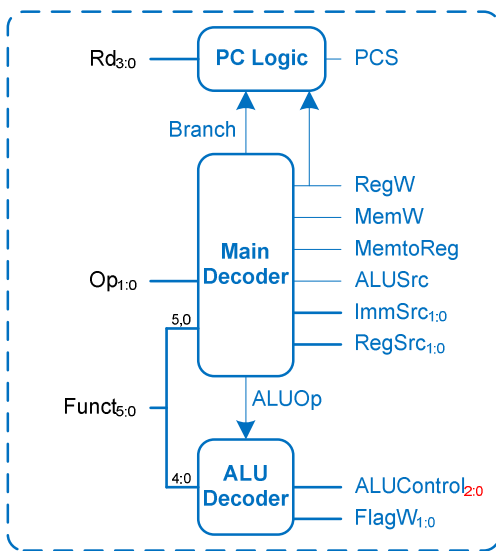
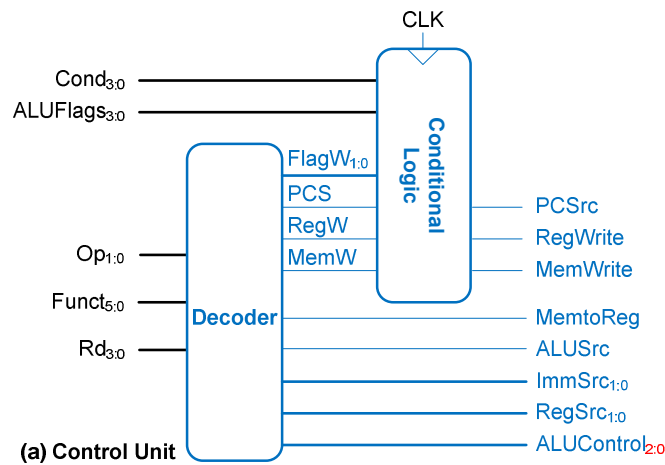
ALU



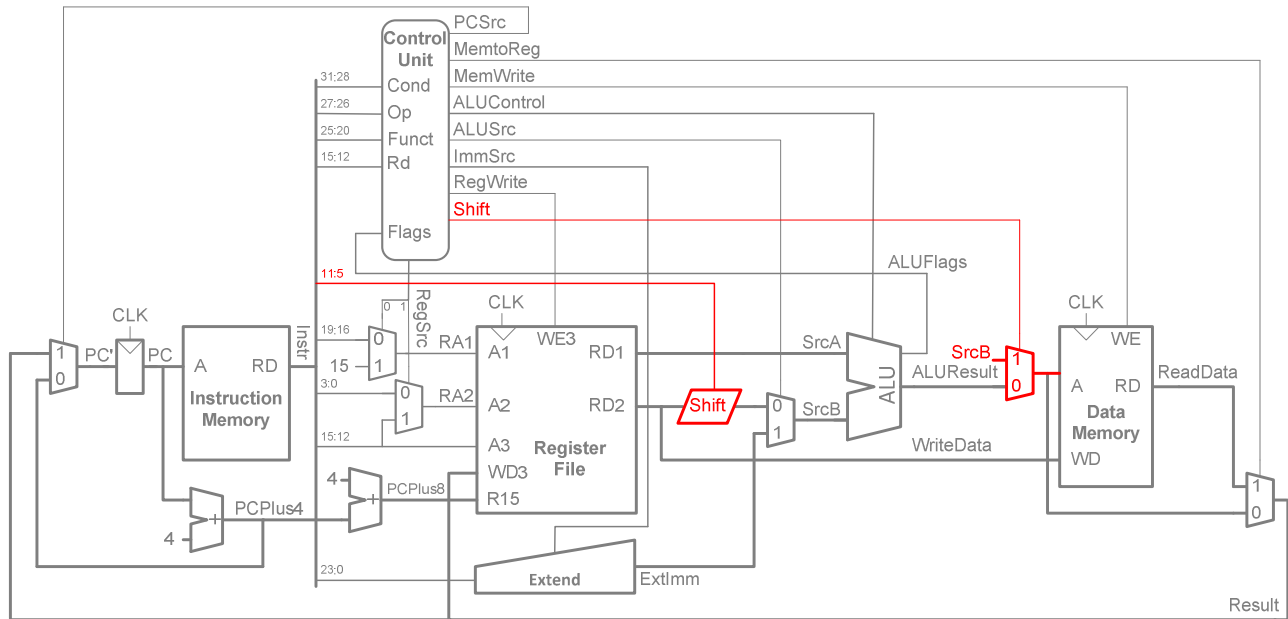
Datapath



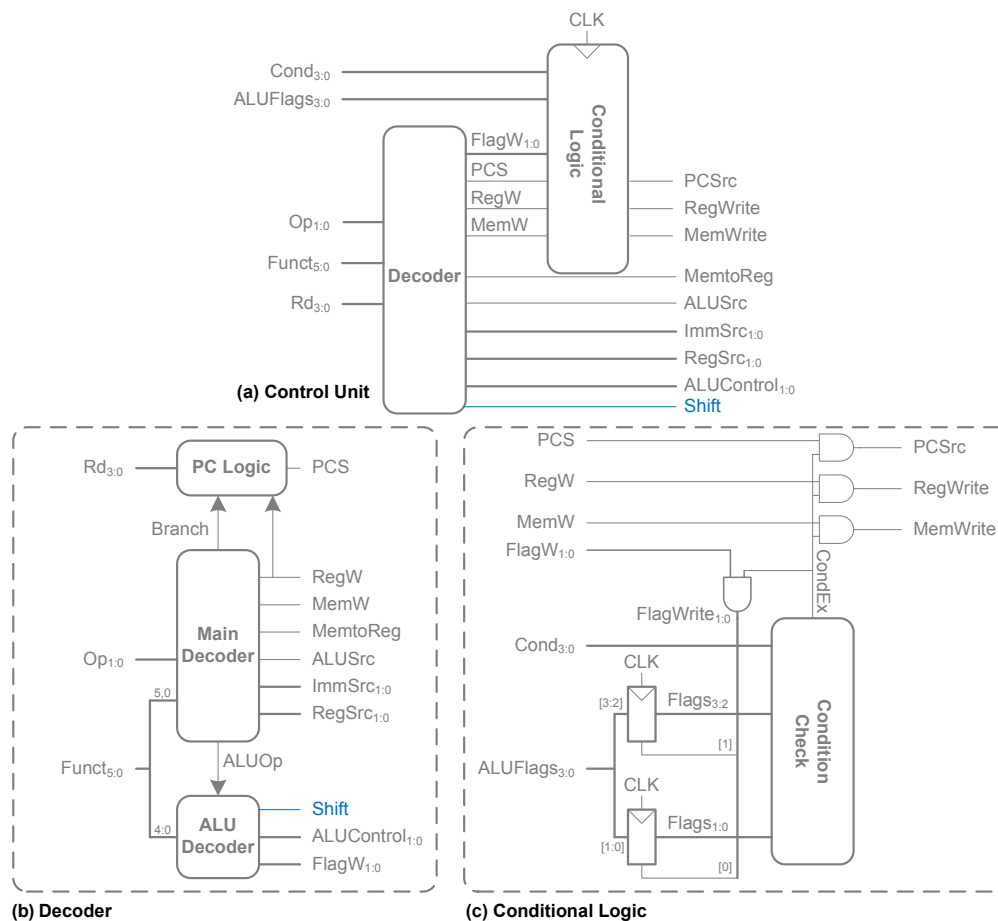
Control



(b) LSR (with immediate shift amount)



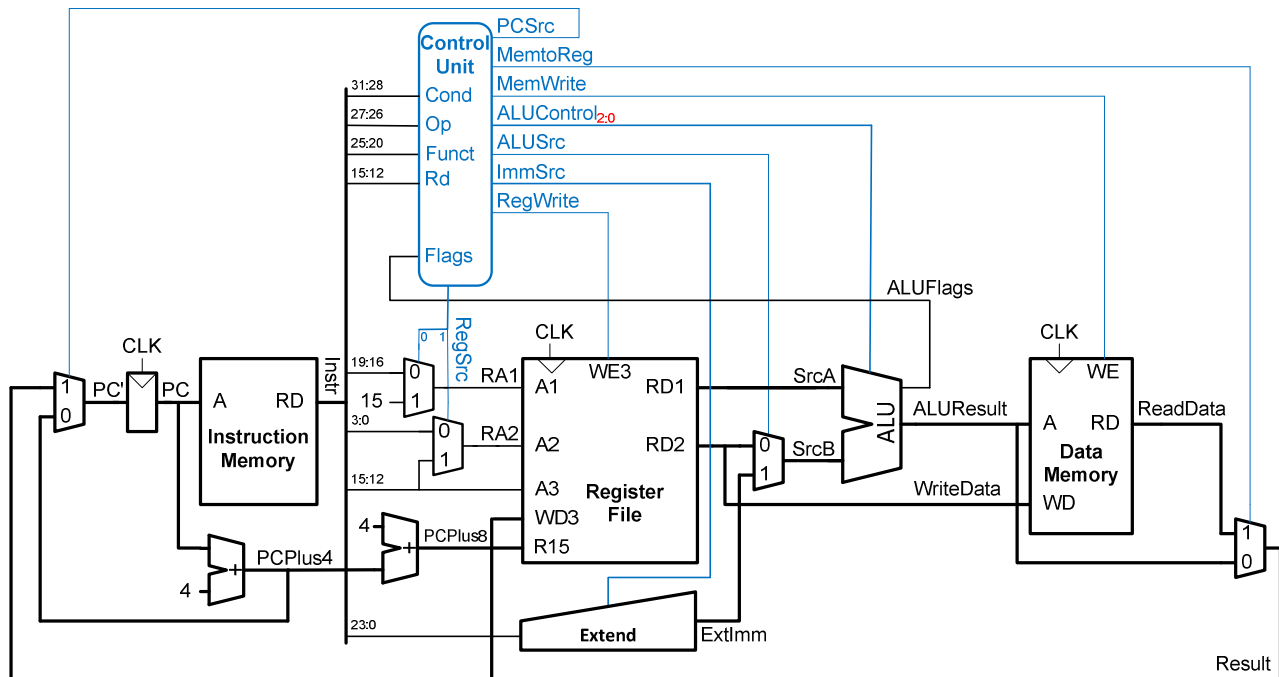
Control



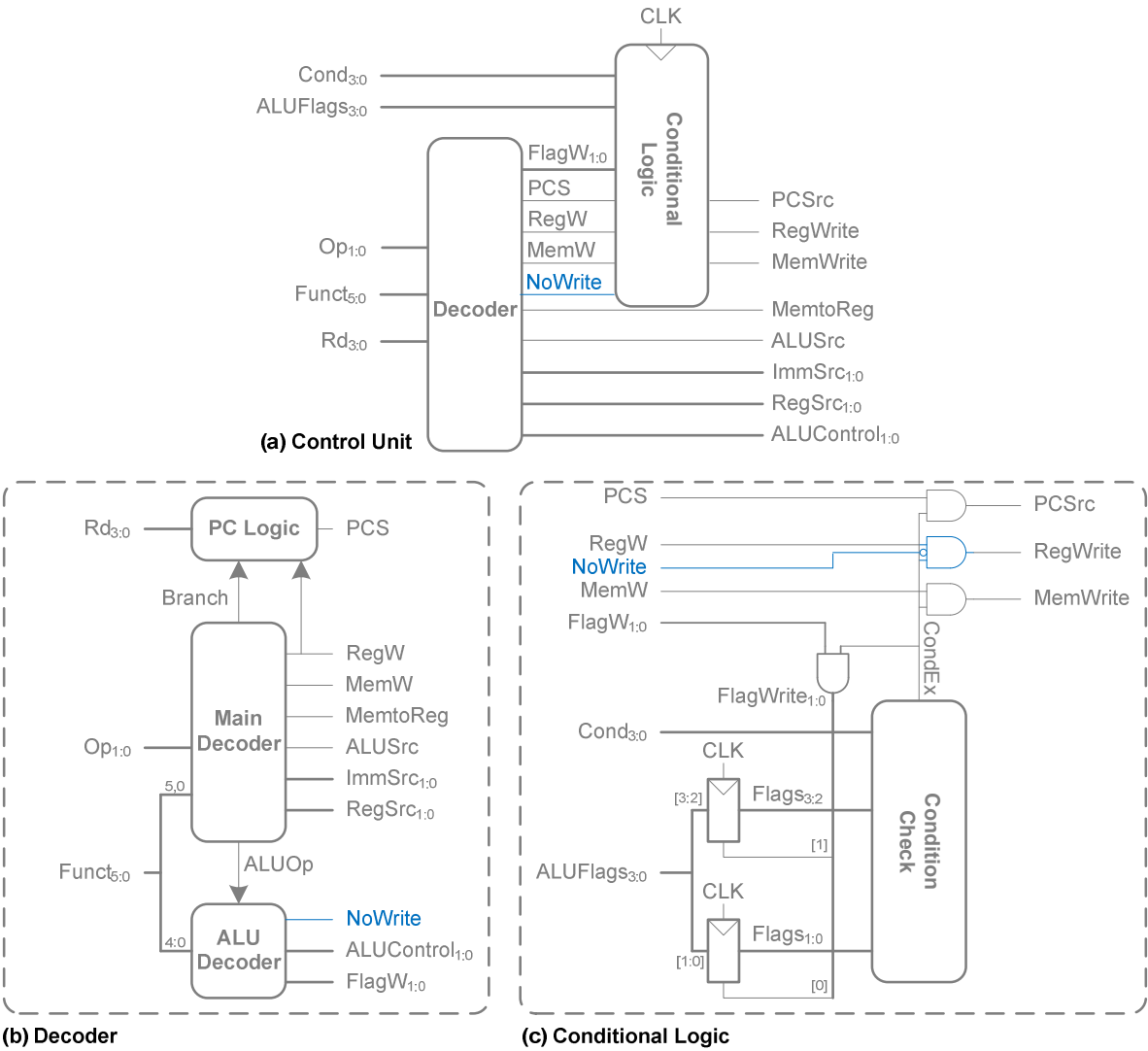
ALU Decoder truth table

<i>ALUOp</i>	<i>Funct</i> _{4:1} (<i>cmd</i>)	<i>Funct</i> ₀ (<i>S</i>)	Notes	<i>ALUControl</i> _{1:0}	<i>FlagW</i> _{1:0}	<i>Shift</i>
0	X	X	Not DP	00	00	0
1	0100	0	ADD	00	00	0
		1			11	0
	0010	0	SUB	01	00	0
		1			11	0
	0000	0	AND	10	00	0
		1			10	0
	1100	0	ORR	11	00	0
		1			10	0
	1101	0	LSR	XX	00	1
		1			10	1

(c) TEQ

Datapath

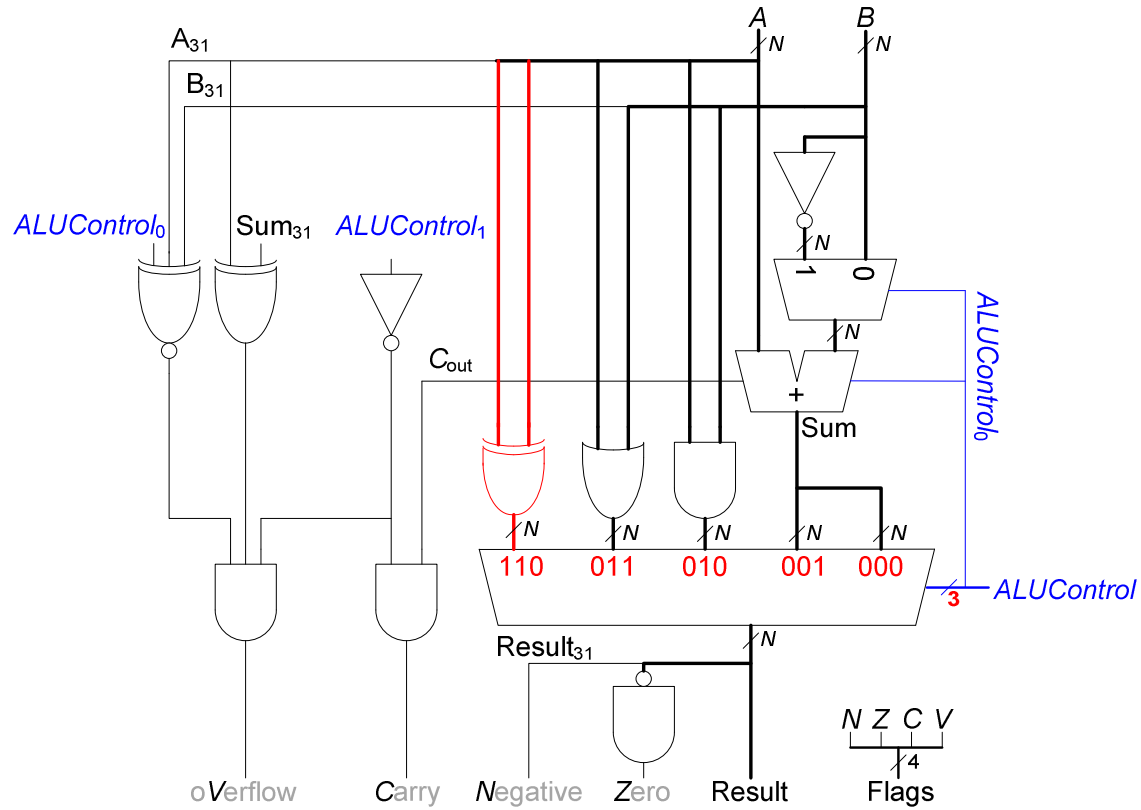
Control



ALU Decoder truth table

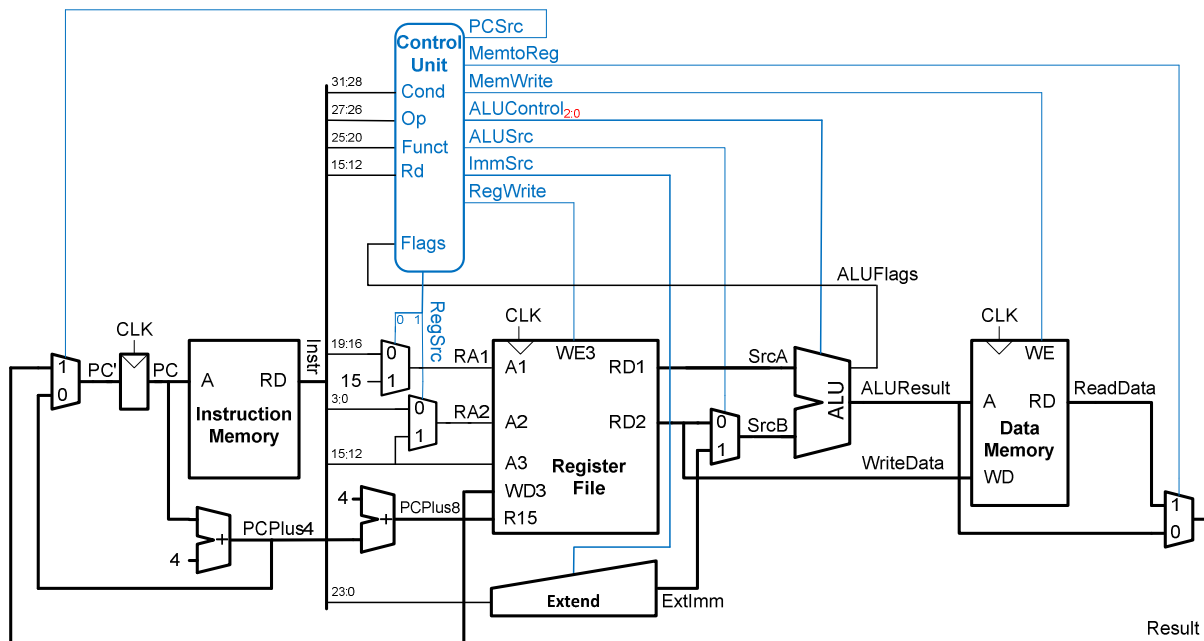
$ALUOp$	$Funct_{4:1} (cmd)$	$Funct_0 (S)$	Notes	$ALUControl_{2:0}$	$FlagW_{1:0}$	$NoWrite$
0	X	X	Not DP	000	00	0
1	0100	0	ADD	000	00	0
		1			11	0
	0010	0	SUB	001	00	0
		1			11	0
	0000	0	AND	010	00	0
		1			10	0
	1100	0	ORR	011	00	0
		1			10	0
	1001	1	TEQ	110	00	1

ALU

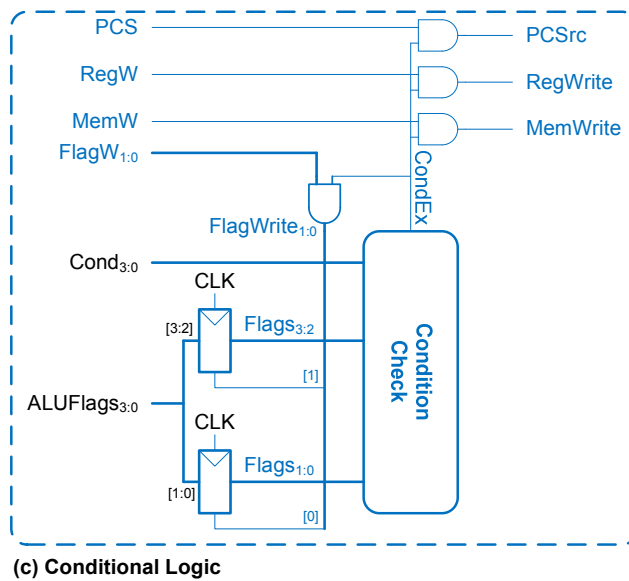
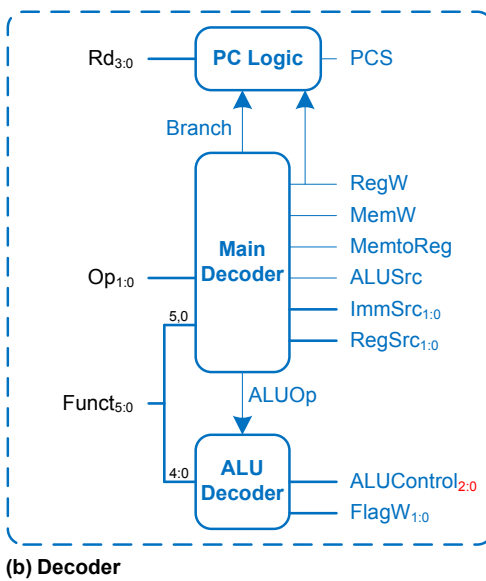
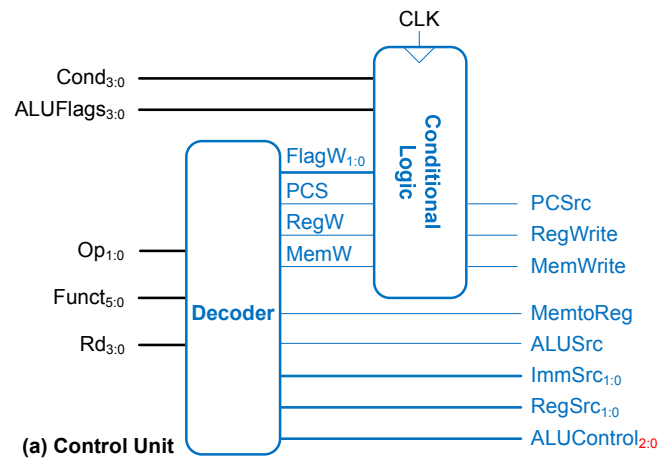


(d) RSB

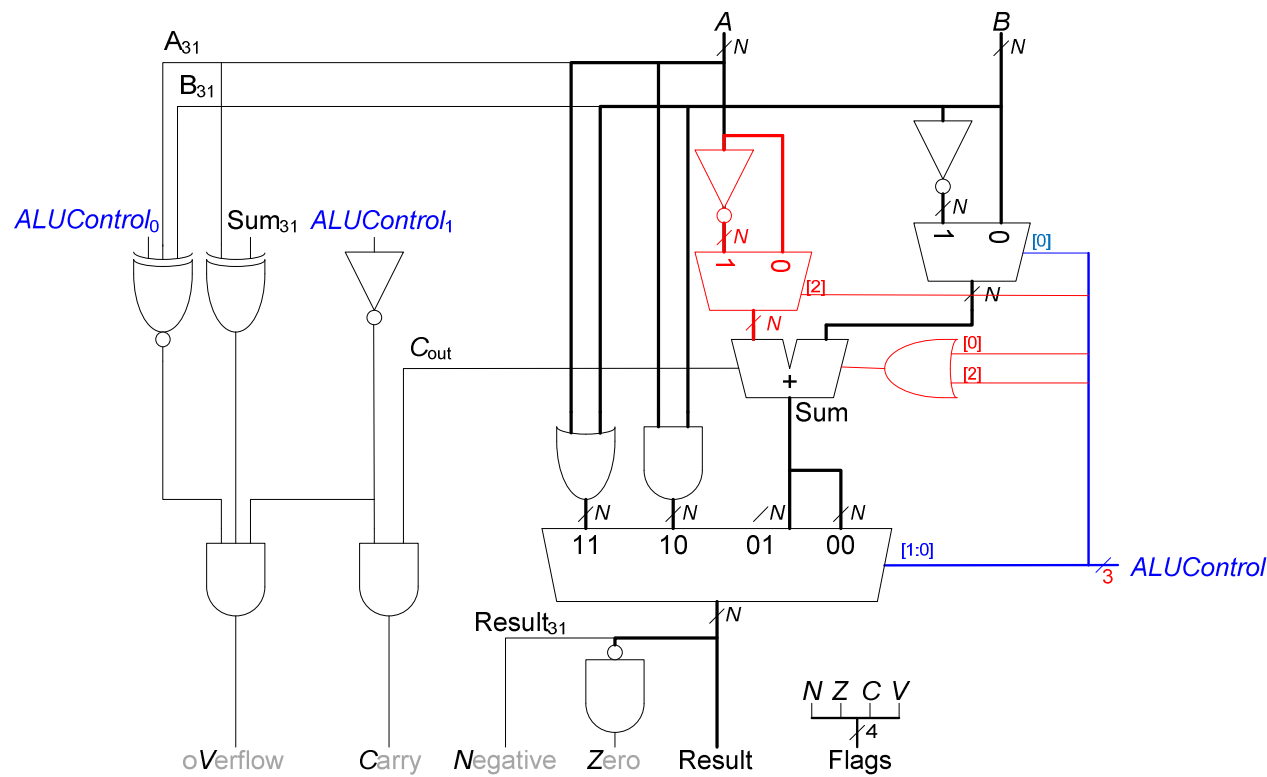
Datapath



Control



ALU



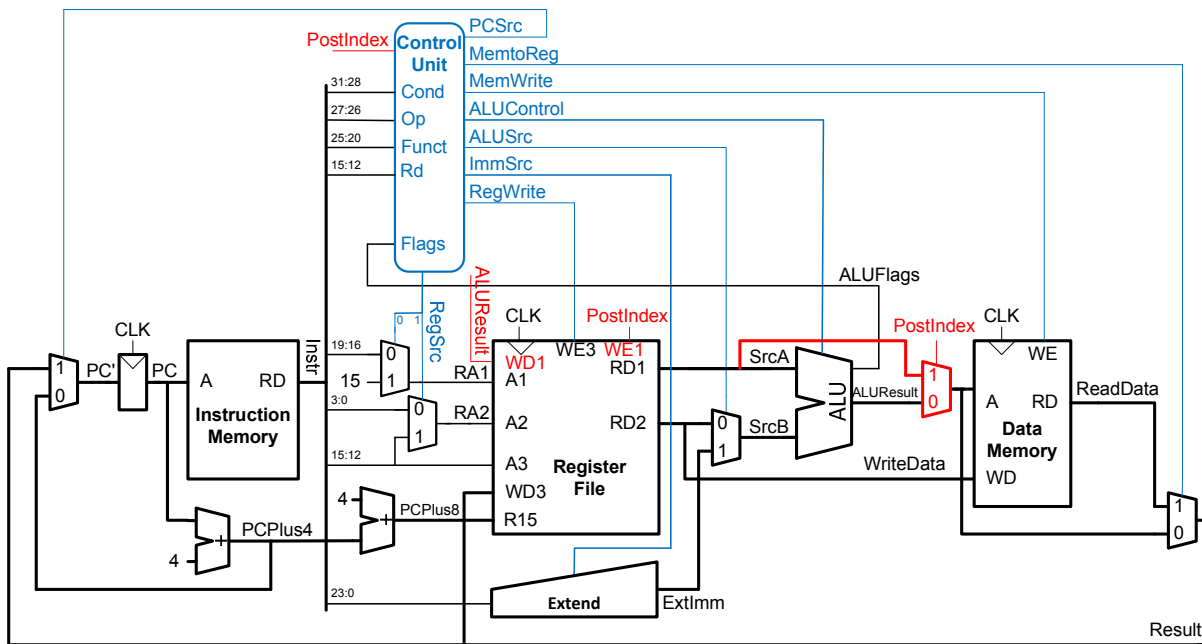
ALU Decoder truth table

<i>ALUOp</i>	<i>Funct</i> _{4:1} (<i>cmd</i>)	<i>Funct</i> ₀ (<i>S</i>)	Notes	<i>ALUControl</i> _{2:0}	<i>FlagW</i> _{1:0}
0	X	X	Not DP	000	00
1	0100	0	ADD	000	00
		1			11
	0010	0	SUB	001	00
		1			11
	0000	0	AND	010	00
		1			10
	1100	0	ORR	011	00
		1			10
	0011	0	RSB	100	00
		1			11

Exercise 7.5

It is not possible to implement this instruction without either modifying the register file or making the instruction take at least two cycles to execute. We modify the register file and datapath as shown below.

- Add WE1 and WD1 signals to the register file.
- WE1 connects to the PostIndex signal (from control unit)
- WD1 connects to ALUResult, which is the sum of $R_n + R_m$ (or $R_n + \text{Src2}$, more generally).
- Add multiplexer before Data Memory Address to choose between $(R_n + \text{Src2})$ and R_n .
With post-indexing, the Data Memory Address input connects to R_n .



We modified the Main Decoder truth table as shown below.

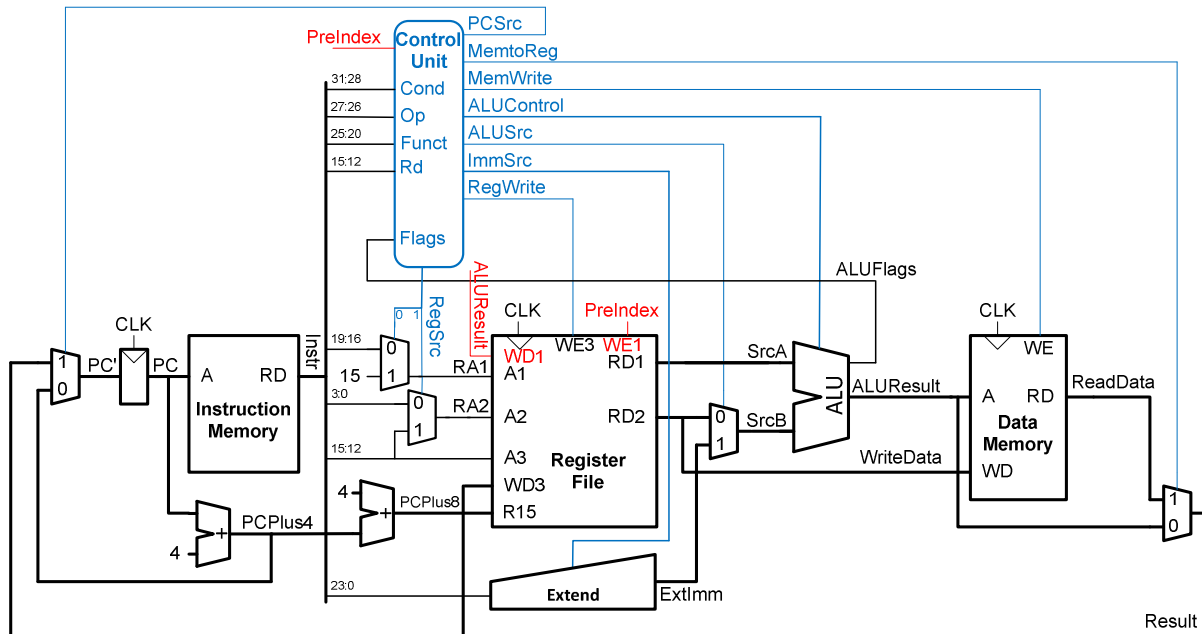
Op	Funct _{5:0}	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp	PostIndex
00	0XXXXX	DP Reg	0	0	0	0	XX	1	00	1	0
00	1XXXXX	DP Imm	0	0	0	1	00	1	X0	1	0
01	X00000	STR	0	X	1	1	01	0	10	0	0
01	011001	LDR (offset indexing, immediate	0	1	0	1	01	1	X0	0	0

		offset)									
01	111001	LDR (offset indexing, register offset)	0	1	0	0	01	1	00	0	0
01	001001	LDR (post- indexing, immediate offset)	0	1	0	1	01	1	X0	0	1
01	101001	LDR (post- indexing, register offset)	0	1	0	0	01	1	00	0	1

Exercise 7.6

It is not possible to implement this instruction without either modifying the register file or making the instruction take at least two cycles to execute. We modify the register file and datapath as shown below.

- Add WE1 and WD1 signals to the register file.
- WE1 connects to the PreIndex signal (from control unit)
- WD1 connects to ALUResult, which is the sum of $R_n + R_m$ (or $R_n + \text{Src2}$, more generally).



We modified the Main Decoder truth table as shown below.

Op	Funct _{5:0}	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp	PreIndex
00	0XXXXX	DP Reg	0	0	0	0	XX	1	00	1	0
00	1XXXXX	DP Imm	0	0	0	1	00	1	X0	1	0
01	X00000	STR	0	X	1	1	01	0	10	0	0
01	011001	LDR (offset indexing, immediate offset)	0	1	0	1	01	1	X0	0	0
01	111001	LDR (offset indexing, register offset)	0	1	0	0	01	1	00	0	0
01	011011	LDR (pre- indexing, immediate offset)	0	1	0	1	01	1	X0	0	1

01	111011	LDR (pre-indexing, register offset)	0	1	0	0	01	1	00	0	1
----	--------	--	---	---	---	---	----	---	----	---	---

Exercise 7.7

She should work on the memory. $t_{mem} = (200/2) \text{ ps} = 100 \text{ ps}$

From Equation 7.3, the new cycle time is:

$$T_{cl} = 40 + 2(100) + 70 + 100 + 120 + 2(25) + 60 = \mathbf{640 \text{ ps}}$$

Exercise 7.8

From Equation 7.3, the new cycle time is:

$$T_{cl} = 40 + 2(200) + 70 + 100 + \mathbf{100} + 2(25) + 60 = \mathbf{820 \text{ ps}}$$

From Equation 7.1, Execution time is:

$$T_1 = (100 \times 10^9 \text{ instruction}) (1 \text{ cycle/instruction}) (820 \times 10^{-12} \text{ s/cycle}) = \mathbf{82 \text{ seconds.}}$$

Exercise 7.9

SystemVerilog

```
// ex7.9 solutions
//
// single-cycle ARM processor
// additional instructions: TST, LSL, CMN, ADC

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 == 20 & WriteData == 2) begin
            $display("Simulation succeeded");
            $stop;
        end else begin
            $display("Simulation failed");
            $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
    arm arm(clk, reset, PC, Instr, MemWrite, DataAdr,
            WriteData, ReadData);
    imem imem(PC, Instr);
    dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule

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

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

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

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

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

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

    initial
        $readmemh("ex7.9_memfile.dat",RAM);

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

```

```

endmodule

module arm(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 [3:0] ALUFlags;
    logic RegWrite,
          ALUSrc, MemtoReg, PCSrc;
    logic [1:0] RegSrc, ImmSrc;
    logic [2:0] ALUControl; // ADC
    logic carry; // ADC
    logic Shift; // LSL

    controller c(clk, reset, Instr[31:12], ALUFlags,
                 RegSrc, RegWrite, ImmSrc,
                 ALUSrc, ALUControl,
                 MemWrite, MemtoReg, PCSrc,
                 carry, // ADC
                 Shift); // LSL
    datapath dp(clk, reset,
                RegSrc, RegWrite, ImmSrc,
                ALUSrc, ALUControl,
                MemtoReg, PCSrc,
                ALUFlags, PC, Instr,
                ALUResult, WriteData, ReadData,
                carry, // ADC
                Shift); // LSL
endmodule

module controller(input logic clk, reset,
                  input logic [31:12] Instr,
                  input logic [3:0] ALUFlags,
                  output logic [1:0] RegSrc,
                  output logic RegWrite,
                  output logic [1:0] ImmSrc,
                  output logic ALUSrc,
                  output logic [2:0] ALUControl, // ADC
                  output logic MemWrite, MemtoReg,
                  output logic PCSrc,
                  output logic carry, // ADC
                  output logic Shift); // LSL

    logic [1:0] FlagW;
    logic PCS, RegW, MemW;
    logic NoWrite; // TST, CMN

    decoder dec(Instr[27:26], Instr[25:20], Instr[15:12],
                FlagW, PCS, RegW, MemW,
                MemtoReg, ALUSrc, ImmSrc, RegSrc, ALUControl,
                NoWrite, // TST, CMN

```



```

        Shift);    // LSL
    condlogic cl(clk, reset, Instr[31:28], ALUFlags,
        FlagW, PCS, RegW, MemW,
        PCSrc, RegWrite, MemWrite,
        carry,    // ADC
        NoWrite); // TST, CMN
endmodule

module decoder(input  logic [1:0] Op,
               input  logic [5:0] Funct,
               input  logic [3:0] Rd,
               output logic [1:0] FlagW,
               output logic      PCS, RegW, MemW,
               output logic      MemtoReg, ALUSrc,
               output logic [1:0] ImmSrc, RegSrc,
               output logic [2:0] ALUControl, // ADC
               output logic      NoWrite,    // TST, CMN
               output logic      Shift);    // LSL
    logic [9:0] controls;
    logic      Branch, ALUOp;

    // Main Decoder

    always_comb
        case(Op)
            // Data processing immediate
            2'b00: if (Funct[5]) controls = 10'b0000101001;
                // Data processing register
                else controls = 10'b0000001001;
            // LDR
            2'b01: if (Funct[0]) controls = 10'b0001111000;
                // STR
                else controls = 10'b1001110100;
            // B
            2'b10: controls = 10'b0110100010;
                // Unimplemented
            default: controls = 10'bx;
        endcase

    assign {RegSrc, ImmSrc, ALUSrc, MemtoReg,
           RegW, MemW, Branch, ALUOp} = controls;

    // ALU Decoder
    always_comb
        if (ALUOp) begin // which DP Instr?
            case(Funct[4:1])
                4'b0100: begin // ADD
                    ALUControl = 3'b000;
                    NoWrite = 1'b0;
                    Shift = 1'b0;
                end
                4'b0010: begin // SUB
                    ALUControl = 3'b001;
                    NoWrite = 1'b0;
                end
            endcase
        end

```

```

        Shift = 1'b0;
    end
    4'b0000: begin                                // AND
        ALUControl = 3'b010;
        NoWrite = 1'b0;
        Shift = 1'b0;
    end
    4'b1100: begin                                // OR
        ALUControl = 3'b011;
        NoWrite = 1'b0;
        Shift = 1'b0;
    end
    4'b1000: begin                                // TST
        ALUControl = 3'b010;
        NoWrite = 1'b1;
        Shift = 1'b0;
    end
    4'b1101: begin                                // LSL
        ALUControl = 3'b000;
        NoWrite = 1'b0;
        Shift = 1'b1;
    end
    4'b1011: begin                                // CMN
        ALUControl = 3'b000;
        NoWrite = 1'b1;
        Shift = 1'b0;
    end
    4'b0101: begin                                // ADC
        ALUControl = 3'b100;
        NoWrite = 1'b0;
        Shift = 1'b0;
    end
    default: begin                                // unimplemented
        ALUControl = 3'bx;
        NoWrite = 1'bx;
        Shift = 1'bx;
    end
endcase

// update flags if S bit is set
// (C & V only updated for arith instructions)
FlagW[1] = Funct[0]; // FlagW[1] = S-bit
// FlagW[0] = S-bit & (ADD | SUB)
FlagW[0] = Funct[0] &
    (ALUControl[1:0] == 2'b00 | ALUControl[1:0] == 2'b01);

end else begin
    ALUControl = 3'b000; // add for non-DP instructions
    FlagW = 2'b00; // don't update Flags
    NoWrite = 1'b0;
    Shift = 1'b0;
end
end

```

```

// PC Logic
assign PCS = ((Rd == 4'b1111) & RegW) | Branch;
endmodule

module condlogic(input logic clk, reset,
                 input logic [3:0] Cond,
                 input logic [3:0] ALUFlags,
                 input logic [1:0] FlagW,
                 input logic PCS, RegW, MemW,
                 output logic PCSrc, RegWrite, MemWrite,
                 output logic carry, // ADC
                 input logic NoWrite); // TST, CMN

logic [1:0] FlagWrite;
logic [3:0] Flags;
logic CondEx;

flopnr #(2)flagreg1(clk, reset, FlagWrite[1],
                   ALUFlags[3:2], Flags[3:2]);
flopnr #(2)flagreg0(clk, reset, FlagWrite[0],
                   ALUFlags[1:0], Flags[1:0]);

// write controls are conditional
condcheck cc(Cond, Flags, CondEx);
assign FlagWrite = FlagW & {2{CondEx}};
assign RegWrite = RegW & CondEx & ~NoWrite; // TST, CMN
assign MemWrite = MemW & CondEx;
assign PCSrc = PCS & CondEx;

assign carry = Flags[1]; // ADC
endmodule

module condcheck(input logic [3:0] Cond,
                 input logic [3:0] Flags,
                 output logic CondEx);

logic neg, zero, carry, overflow, ge;

assign {neg, zero, carry, overflow} = Flags;
assign ge = (neg == overflow);

always_comb
case(Cond)
  4'b0000: CondEx = zero; // EQ
  4'b0001: CondEx = ~zero; // NE
  4'b0010: CondEx = carry; // CS
  4'b0011: CondEx = ~carry; // CC
  4'b0100: CondEx = neg; // MI
  4'b0101: CondEx = ~neg; // PL
  4'b0110: CondEx = overflow; // VS
  4'b0111: CondEx = ~overflow; // VC
  4'b1000: CondEx = carry & ~zero; // HI
  4'b1001: CondEx = ~(carry & ~zero); // LS

```

```

        4'b1010: CondEx = ge;           // GE
        4'b1011: CondEx = ~ge;        // LT
        4'b1100: CondEx = ~zero & ge;  // GT
        4'b1101: CondEx = ~(~zero & ge); // LE
        4'b1110: CondEx = 1'b1;       // Always
        default: CondEx = 1'bx;       // undefined
    endcase
endmodule

module datapath(input logic clk, reset,
                input logic [1:0] RegSrc,
                input logic RegWrite,
                input logic [1:0] ImmSrc,
                input logic ALUSrc,
                input logic [2:0] ALUControl, // ADC
                input logic MemtoReg,
                input logic PCSrc,
                output logic [3:0] ALUFlags,
                output logic [31:0] PC,
                input logic [31:0] Instr,
                output logic [31:0] ALUResultOut, // LSL
                output logic [31:0] WriteData,
                input logic [31:0] ReadData,
                input logic carry, // ADC
                input logic Shift); // LSL

    logic [31:0] PCNext, PCPlus4, PCPlus8;
    logic [31:0] ExtImm, SrcA, SrcB, Result;
    logic [3:0] RA1, RA2;
    logic [31:0] srcBshifted, ALUResult; // LSL

    // next PC logic
    mux2 #(32) pcmux(PCPlus4, Result, PCSrc, PCNext);
    flopr #(32) pcreg(clk, reset, PCNext, PC);
    adder #(32) pcadd1(PC, 32'b100, PCPlus4);
    adder #(32) pcadd2(PCPlus4, 32'b100, PCPlus8);

    // register file logic
    mux2 #(4) ralmux(Instr[19:16], 4'b1111, RegSrc[0], RA1);
    mux2 #(4) ra2mux(Instr[3:0], Instr[15:12], RegSrc[1], RA2);
    regfile rf(clk, RegWrite, RA1, RA2,
               Instr[15:12], Result, PCPlus8,
               SrcA, WriteData);
    mux2 #(32) resmux(ALUResultOut, ReadData, MemtoReg, Result);
    extend ext(Instr[23:0], ImmSrc, ExtImm);

    // ALU logic
    shifter sh(WriteData, Instr[11:7], Instr[6:5], srcBshifted); // LSL
    mux2 #(32) srcbmux(srcBshifted, ExtImm, ALUSrc, SrcB); // LSL
    alu alu(SrcA, SrcB, ALUControl,
            ALUResult, ALUFlags,
            carry); // ADC
    mux2 #(32) alureultmux(ALUResult, SrcB, Shift, ALUResultOut); // LSL

```

```

endmodule

module regfile(input  logic      clk,
               input  logic      we3,
               input  logic [3:0] ra1, ra2, wa3,
               input  logic [31:0] wd3, r15,
               output logic [31:0] rd1, rd2);

    logic [31:0] rf[14:0];

    // three ported register file
    // read two ports combinationaly
    // write third port on rising edge of clock
    // register 15 reads PC+8 instead

    always_ff @(posedge clk)
        if (we3) rf[wa3] <= wd3;

    assign rd1 = (ra1 == 4'b1111) ? r15 : rf[ra1];
    assign rd2 = (ra2 == 4'b1111) ? r15 : rf[ra2];
endmodule

module extend(input  logic [23:0] Instr,
              input  logic [1:0] ImmSrc,
              output logic [31:0] ExtImm);

    always_comb
        case(ImmSrc)
            // 8-bit unsigned immediate
            2'b00: ExtImm = {24'b0, Instr[7:0]};
            // 12-bit unsigned immediate
            2'b01: ExtImm = {20'b0, Instr[11:0]};
            // 24-bit two's complement shifted branch
            2'b10: ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00};
            default: ExtImm = 32'bx; // undefined
        endcase
endmodule

module adder #(parameter WIDTH=8)
    (input  logic [WIDTH-1:0] a, b,
     output logic [WIDTH-1:0] y);

    assign y = a + b;
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 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 alu(input  logic [31:0] a, b,
           input  logic [2:0]  ALUControl,           // ADC
           output logic [31:0] Result,
           output logic [3:0]  ALUFlags,
           input  logic          carry);             // ADC

    logic          neg, zero, carryout, overflow;
    logic [31:0]   condinvb;
    logic [32:0]   sum;
    logic          carryin;                          // ADC

    assign carryin = ALUControl[2] ? carry : ALUControl[0]; // ADC
    assign condinvb = ALUControl[0] ? ~b : b;
    assign sum = a + condinvb + carryin;                // ADC

    always_comb
        casex (ALUControl[1:0])
            2'b0?: Result = sum;
            2'b10: Result = a & b;
            2'b11: Result = a | b;
        endcase

    assign neg      = Result[31];
    assign zero     = (Result == 32'b0);
    assign carryout = (ALUControl[1] == 1'b0) & sum[32];
    assign overflow = (ALUControl[1] == 1'b0) &
        ~(a[31] ^ b[31] ^ ALUControl[0]) &
        (a[31] ^ sum[31]);
    assign ALUFlags = {neg, zero, carryout, overflow};
endmodule

// shifter needed for LSL
module shifter(input  logic [31:0] a,
               input  logic [4:0] shamt,

```

```

        input  logic [ 1:0] shtype,
        output logic [31:0] y);

always_comb
    case (shtype)
        2'b00: y = a << shamt;
        default: y = a;
    endcase
endmodule

```

VHDL

```

library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity testbench is
end;

architecture test of testbench is
    component top
        port (clk, reset:          in  STD_LOGIC;
              WriteData, DataAdr:  out STD_LOGIC_VECTOR(31 downto 0);
              MemWrite:           out STD_LOGIC);
    end component;
    signal WriteData, DataAdr:      STD_LOGIC_VECTOR(31 downto 0);
    signal clk, reset,  MemWrite:  STD_LOGIC;
begin

    -- instantiate device to be tested
    dut: top port map (clk, reset, WriteData, DataAdr, MemWrite);

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

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

    -- check that 0x80000001 gets written to address 20
    -- at end of program
    process (clk) begin
        if (clk'event and clk = '0' and MemWrite = '1') then
            if (to_integer(DataAdr) = 20 and
                to_integer(WriteData) = 2) then
                report "NO ERRORS: Simulation succeeded" severity failure;
            else

```

```

        report "Simulation failed" severity failure;
    end if;
end if;
end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity top is -- top-level design for testing
    port(clk, reset:          in    STD_LOGIC;
          WriteData, DataAdr:  buffer STD_LOGIC_VECTOR(31 downto 0);
          MemWrite:           buffer STD_LOGIC);
end;

architecture test of top is
    component arm
        port(clk, reset:          in    STD_LOGIC;
              PC:                out STD_LOGIC_VECTOR(31 downto 0);
              Instr:             in    STD_LOGIC_VECTOR(31 downto 0);
              MemWrite:          out STD_LOGIC;
              ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
              ReadData:          in    STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component imem
        port(a: in    STD_LOGIC_VECTOR(31 downto 0);
              rd: out STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component dmem
        port(clk, we: in STD_LOGIC;
              a, wd:  in STD_LOGIC_VECTOR(31 downto 0);
              rd:     out STD_LOGIC_VECTOR(31 downto 0));
    end component;
    signal PC, Instr,
            ReadData: STD_LOGIC_VECTOR(31 downto 0);
begin
    -- instantiate processor and memories
    i_arm: arm port map(clk, reset, PC, Instr, MemWrite, DataAdr,
                       WriteData, ReadData);
    i_imem: imem port map(PC, Instr);
    i_dmem: dmem port map(clk, MemWrite, DataAdr,
                       WriteData, ReadData);
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity dmem is -- data memory
    port(clk, we: in STD_LOGIC;
          a, wd:  in STD_LOGIC_VECTOR(31 downto 0);
          rd:     out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of dmem is
begin

```



```

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

library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity imem is -- instruction memory
  port(a:  in  STD_LOGIC_VECTOR(31 downto 0);
       rd: out STD_LOGIC_VECTOR(31 downto 0));
end;
architecture behave of imem is -- instruction memory
begin
  process is
    file mem_file: TEXT;
    variable L: line;
    variable ch: character;
    variable i, index, result: integer;
    type ramtype is array (63 downto 0) of
      STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin
    -- initialize memory from file
    for i in 0 to 63 loop -- set all contents low
      mem(i) := (others => '0');
    end loop;
    index := 0;
    FILE_OPEN(mem_file, "ex7.9_memfile.dat", READ_MODE);
    while not endfile(mem_file) loop
      readline(mem_file, L);
      result := 0;
      for i in 1 to 8 loop
        read(L, ch);
        if '0' <= ch and ch <= '9' then
          result := character'pos(ch) - character'pos('0');
        elsif 'a' <= ch and ch <= 'f' then
          result := character'pos(ch) - character'pos('a')+10;
        elsif 'A' <= ch and ch <= 'F' then
          result := character'pos(ch) - character'pos('A')+10;
        else report "Format error on line " & integer'image(index)
          severity error;
        end if;
      end loop;
    end while;
  end process;
end;

```

```

        end if;
        mem(index)(35-i*4 downto 32-i*4) :=
            to_std_logic_vector(result,4);
        end loop;
        index := index + 1;
    end loop;

    -- read memory
    loop
        rd <= mem(to_integer(a(7 downto 2)));
        wait on a;
    end loop;
end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity arm is -- single cycle processor
    port(clk, reset:      in  STD_LOGIC;
          PC:             out STD_LOGIC_VECTOR(31 downto 0);
          Instr:          in  STD_LOGIC_VECTOR(31 downto 0);
          MemWrite:       out STD_LOGIC;
          ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
          ReadData:       in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of arm is
    component controller
        port(clk, reset:      in  STD_LOGIC;
              Instr:          in  STD_LOGIC_VECTOR(31 downto 12);
              ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
              RegSrc:         out STD_LOGIC_VECTOR(1 downto 0);
              RegWrite:       out STD_LOGIC;
              ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
              ALUSrc:         out STD_LOGIC;
              ALUControl:     out STD_LOGIC_VECTOR(2 downto 0);      -- ADC
              MemWrite:       out STD_LOGIC;
              MemtoReg:       out STD_LOGIC;
              PCSrc:          out STD_LOGIC;
              carry:          out STD_LOGIC;      -- ADC
              Shift:          out STD_LOGIC);      -- LSL
    end component;
    component datapath
        port(clk, reset:      in  STD_LOGIC;
              RegSrc:         in  STD_LOGIC_VECTOR(1 downto 0);
              RegWrite:       in  STD_LOGIC;
              ImmSrc:         in  STD_LOGIC_VECTOR(1 downto 0);
              ALUSrc:         in  STD_LOGIC;
              ALUControl:     in  STD_LOGIC_VECTOR(2 downto 0);      -- ADC
              MemtoReg:       in  STD_LOGIC;
              PCSrc:          in  STD_LOGIC;
              ALUFlags:       out STD_LOGIC_VECTOR(3 downto 0);
              PC:             buffer STD_LOGIC_VECTOR(31 downto 0);
              Instr:          in  STD_LOGIC_VECTOR(31 downto 0);
              ALUResultOut:   buffer STD_LOGIC_VECTOR(31 downto 0);      -- LSL
        );
    end component;

```

```

        WriteData:      buffer STD_LOGIC_VECTOR(31 downto 0);
        ReadData:       in  STD_LOGIC_VECTOR(31 downto 0);
        carry:          in  STD_LOGIC;                      -- ADC
        Shift:          in  STD_LOGIC;                      -- LSL
    end component;
    signal ALUFlags: STD_LOGIC_VECTOR(3 downto 0);
    signal RegWrite, ALUSrc, MemtoReg, PCSrc: STD_LOGIC;
    signal RegSrc, ImmSrc: STD_LOGIC_VECTOR(1 downto 0);
    signal ALUControl: STD_LOGIC_VECTOR(2 downto 0);          -- ADC
    signal carry: STD_LOGIC;                                  -- ADC
    signal Shift: STD_LOGIC;                                  -- LSL
begin
    cont: controller port map(clk, reset, Instr(31 downto 12),
                             ALUFlags, RegSrc, RegWrite, ImmSrc,
                             ALUSrc, ALUControl, MemWrite,
                             MemtoReg, PCSrc,
                             carry, -- ADC
                             Shift); -- LSL
    dp: datapath port map(clk, reset, RegSrc, RegWrite, ImmSrc,
                         ALUSrc, ALUControl, MemtoReg, PCSrc,
                         ALUFlags, PC, Instr, ALUResult,
                         WriteData, ReadData,
                         carry, -- ADC
                         Shift); -- LSL
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- single cycle control decoder
    port(clk, reset:      in  STD_LOGIC;
          Instr:          in  STD_LOGIC_VECTOR(31 downto 12);
          ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
          RegSrc:         out STD_LOGIC_VECTOR(1 downto 0);
          RegWrite:       out STD_LOGIC;
          ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
          ALUSrc:         out STD_LOGIC;
          ALUControl:     out STD_LOGIC_VECTOR(2 downto 0); -- ADC
          MemWrite:       out STD_LOGIC;
          MemtoReg:       out STD_LOGIC;
          PCSrc:          out STD_LOGIC;
          carry:          out STD_LOGIC; -- ADC
          Shift:          out STD_LOGIC); -- LSL
end;

architecture struct of controller is
    component decoder
        port(Op:          in  STD_LOGIC_VECTOR(1 downto 0);
             Funct:       in  STD_LOGIC_VECTOR(5 downto 0);
             Rd:          in  STD_LOGIC_VECTOR(3 downto 0);
             FlagW:       out STD_LOGIC_VECTOR(1 downto 0);
             PCS, RegW, MemW: out STD_LOGIC;
             MemtoReg, ALUSrc: out STD_LOGIC;
             ImmSrc, RegSrc: out STD_LOGIC_VECTOR(1 downto 0);
             ALUControl:   out STD_LOGIC_VECTOR(2 downto 0); -- ADC
             NoWrite:      out STD_LOGIC;                    -- TST, CMN
    end component;

```

```

        Shift:                out STD_LOGIC);                -- LSL
end component;
component condlogic
  port(clk, reset:            in  STD_LOGIC;
        Cond:                 in  STD_LOGIC_VECTOR(3 downto 0);
        ALUFlags:             in  STD_LOGIC_VECTOR(3 downto 0);
        FlagW:                in  STD_LOGIC_VECTOR(1 downto 0);
        PCS, RegW, MemW:      in  STD_LOGIC;
        PCSrc, RegWrite:      out STD_LOGIC;
        MemWrite:             out STD_LOGIC;
        carry:                out STD_LOGIC; -- ADC
        NoWrite:              in  STD_LOGIC); -- TST, CMN
end component;
signal FlagW: STD_LOGIC_VECTOR(1 downto 0);
signal PCS, RegW, MemW: STD_LOGIC;
signal NoWrite: STD_LOGIC; -- TST, CMN
begin
  dec: decoder port map(Instr(27 downto 26), Instr(25 downto 20),
                        Instr(15 downto 12), FlagW, PCS,
                        RegW, MemW, MemtoReg, ALUSrc, ImmSrc,
                        RegSrc, ALUControl,
                        NoWrite, -- TST, CMN
                        Shift); -- LSL
  cl: condlogic port map(clk, reset, Instr(31 downto 28),
                        ALUFlags, FlagW, PCS, RegW, MemW,
                        PCSrc, RegWrite, MemWrite,
                        carry, -- ADC
                        NoWrite); -- TST, CMN
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity decoder is -- main control decoder
  port(Op:                in  STD_LOGIC_VECTOR(1 downto 0);
        Funct:            in  STD_LOGIC_VECTOR(5 downto 0);
        Rd:               in  STD_LOGIC_VECTOR(3 downto 0);
        FlagW:            out STD_LOGIC_VECTOR(1 downto 0);
        PCS, RegW, MemW:  out STD_LOGIC;
        MemtoReg, ALUSrc: out STD_LOGIC;
        ImmSrc, RegSrc:   out STD_LOGIC_VECTOR(1 downto 0);
        ALUControl:       out STD_LOGIC_VECTOR(2 downto 0); -- ADC
        NoWrite:          out STD_LOGIC; -- TST, CMN
        Shift:            out STD_LOGIC); -- LSL
end;

architecture behave of decoder is
  signal controls:        STD_LOGIC_VECTOR(9 downto 0);
  signal ALUOp, Branch: STD_LOGIC;
  signal op2:             STD_LOGIC_VECTOR(3 downto 0);
begin
  op2 <= (Op, Funct(5), Funct(0));
  process(all) begin -- Main Decoder
    case? (op2) is
      when "000-" => controls <= "00000001001";
      when "001-" => controls <= "0000101001";
    end case;
  end process;
end behave;

```

```

    when "01-0" => controls <= "1001110100";
    when "01-1" => controls <= "0001111000";
    when "10--" => controls <= "0110100010";
    when others => controls <= "-----";
end case?;
end process;

(RegSrc, ImmSrc, ALUSrc, MemtoReg, RegW, MemW,
 Branch, ALUOp) <= controls;

process(all) begin -- ALU Decoder
    if (ALUOp) then
        case Funct(4 downto 1) is
            when "0100" => ALUControl <= "000"; -- ADD
                           NoWrite <= '0';
                           Shift <= '0';
            when "0010" => ALUControl <= "001"; -- SUB
                           NoWrite <= '0';
                           Shift <= '0';
            when "0000" => ALUControl <= "010"; -- AND
                           NoWrite <= '0';
                           Shift <= '0';
            when "1100" => ALUControl <= "011"; -- ORR
                           NoWrite <= '0';
                           Shift <= '0';
            when "1000" => ALUControl <= "010"; -- TST
                           NoWrite <= '1';
                           Shift <= '0';
            when "1101" => ALUControl <= "000"; -- LSL
                           NoWrite <= '0';
                           Shift <= '1';
            when "1011" => ALUControl <= "000"; -- CMN
                           NoWrite <= '1';
                           Shift <= '0';
            when "0101" => ALUControl <= "100"; -- ADC
                           NoWrite <= '0';
                           Shift <= '0';
            when others => ALUControl <= "---"; -- unimplemented
                           NoWrite <= '-';
                           Shift <= '-';

            end case;
            FlagW(1) <= Funct(0);
            FlagW(0) <= Funct(0) and (not ALUControl(1));
        else
            ALUControl <= "000";
            NoWrite <= '0';
            Shift <= '0';
            FlagW <= "00";
        end if;
    end process;

    PCS <= ((and Rd) and RegW) or Branch;
end;
```

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condlogic is -- Conditional logic
  port(clk, reset:      in  STD_LOGIC;
        Cond:          in  STD_LOGIC_VECTOR(3 downto 0);
        ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
        FlagW:          in  STD_LOGIC_VECTOR(1 downto 0);
        PCS, RegW, MemW: in  STD_LOGIC;
        PCSrc, RegWrite: out STD_LOGIC;
        MemWrite:        out STD_LOGIC;
        carry:           out STD_LOGIC; -- ADC
        NoWrite:         in  STD_LOGIC); -- TST, CMN
end;

architecture behave of condlogic is
  component condcheck
    port(Cond:          in  STD_LOGIC_VECTOR(3 downto 0);
          Flags:        in  STD_LOGIC_VECTOR(3 downto 0);
          CondEx:       out STD_LOGIC);
  end component;
  component flopenr generic(width: integer);
    port(clk, reset, en: in  STD_LOGIC;
          d:           in  STD_LOGIC_VECTOR(width-1 downto 0);
          q:           out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal FlagWrite: STD_LOGIC_VECTOR(1 downto 0);
  signal Flags:     STD_LOGIC_VECTOR(3 downto 0);
  signal CondEx:    STD_LOGIC;
begin
  flagreg1: flopenr generic map(2)
    port map(clk, reset, FlagWrite(1),
              ALUFlags(3 downto 2), Flags(3 downto 2));
  flagreg0: flopenr generic map(2)
    port map(clk, reset, FlagWrite(0),
              ALUFlags(1 downto 0), Flags(1 downto 0));
  cc: condcheck port map(Cond, Flags, CondEx);

  FlagWrite <= FlagW and (CondEx, CondEx);
  RegWrite  <= RegW  and CondEx and (not NoWrite); -- TST, CMN
  MemWrite  <= MemW  and CondEx;
  PCSrc     <= PCS   and CondEx;

  carry <= Flags(1); -- ADC
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condcheck is
  port(Cond:          in  STD_LOGIC_VECTOR(3 downto 0);
        Flags:        in  STD_LOGIC_VECTOR(3 downto 0);
        CondEx:       out STD_LOGIC);
end;

architecture behave of condcheck is
  signal neg, zero, carry, overflow, ge: STD_LOGIC;
begin

```

```

(neg, zero, carry, overflow) <= Flags;
ge <= (neg xnor overflow);

process(all) begin -- Condition checking
  case Cond is
    when "0000" => CondEx <= zero;
    when "0001" => CondEx <= not zero;
    when "0010" => CondEx <= carry;
    when "0011" => CondEx <= not carry;
    when "0100" => CondEx <= neg;
    when "0101" => CondEx <= not neg;
    when "0110" => CondEx <= overflow;
    when "0111" => CondEx <= not overflow;
    when "1000" => CondEx <= carry and (not zero);
    when "1001" => CondEx <= not(carry and (not zero));
    when "1010" => CondEx <= ge;
    when "1011" => CondEx <= not ge;
    when "1100" => CondEx <= (not zero) and ge;
    when "1101" => CondEx <= not ((not zero) and ge);
    when "1110" => CondEx <= '1';
    when others => CondEx <= '-';
  end case;
end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity datapath is
  port(clk, reset:      in  STD_LOGIC;
        RegSrc:         in  STD_LOGIC_VECTOR(1 downto 0);
        RegWrite:       in  STD_LOGIC;
        ImmSrc:         in  STD_LOGIC_VECTOR(1 downto 0);
        ALUSrc:         in  STD_LOGIC;
        ALUControl:     in  STD_LOGIC_VECTOR(2 downto 0);      -- ADC
        MemtoReg:       in  STD_LOGIC;
        PCSrc:          in  STD_LOGIC;
        ALUFlags:       out STD_LOGIC_VECTOR(3 downto 0);
        PC:             buffer STD_LOGIC_VECTOR(31 downto 0);
        Instr:          in  STD_LOGIC_VECTOR(31 downto 0);
        ALUResultOut:   buffer STD_LOGIC_VECTOR(31 downto 0);  -- LSL
        WriteData:      buffer STD_LOGIC_VECTOR(31 downto 0);
        ReadData:       in  STD_LOGIC_VECTOR(31 downto 0);
        carry:          in  STD_LOGIC;                        -- ADC
        Shift:          in  STD_LOGIC);                       -- LSL
end;

architecture struct of datapath is
  component alu
    port(a, b:          in  STD_LOGIC_VECTOR(31 downto 0);
          ALUControl: in  STD_LOGIC_VECTOR(2 downto 0);  -- ADC
          Result:      buffer STD_LOGIC_VECTOR(31 downto 0);
          ALUFlags:    out  STD_LOGIC_VECTOR(3 downto 0);
          carry:       in  STD_LOGIC);                  -- ADC
  end component;
  component regfile

```

```

    port (clk:          in  STD_LOGIC;
          we3:          in  STD_LOGIC;
          ral, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
          wd3, r15:     in  STD_LOGIC_VECTOR(31 downto 0);
          rd1, rd2:     out STD_LOGIC_VECTOR(31 downto 0));
end component;
component adder
    port (a, b: in  STD_LOGIC_VECTOR(31 downto 0);
          y:   out STD_LOGIC_VECTOR(31 downto 0));
end component;
component extend
    port (Instr: in  STD_LOGIC_VECTOR(23 downto 0);
          ImmSrc: in STD_LOGIC_VECTOR(1 downto 0);
          ExtImm: out STD_LOGIC_VECTOR(31 downto 0));
end component;
component flopr generic(width: integer);
    port (clk, reset: in  STD_LOGIC;
          d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
          q:          out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
component mux2 generic(width: integer);
    port (d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
          s:      in  STD_LOGIC;
          y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
component shifter -- LSL
    port (a:          in  STD_LOGIC_VECTOR(31 downto 0);
          shamt:      in  STD_LOGIC_VECTOR(4 downto 0);
          shtype:     in  STD_LOGIC_VECTOR(1 downto 0);
          y:          out STD_LOGIC_VECTOR(31 downto 0));
end component;

signal PCNext, PCPlus4, PCPlus8: STD_LOGIC_VECTOR(31 downto 0);
signal ExtImm, Result:          STD_LOGIC_VECTOR(31 downto 0);
signal SrcA, SrcB:              STD_LOGIC_VECTOR(31 downto 0);
signal RA1, RA2:                STD_LOGIC_VECTOR(3 downto 0);
signal srcBshifted, ALUResult:  STD_LOGIC_VECTOR(31 downto 0); -- LSL
begin
    -- next PC logic
    pcmux: mux2 generic map(32)
        port map(PCPlus4, Result, PCSrc, PCNext);
    pcreg: flopr generic map(32) port map(clk, reset, PCNext, PC);
    pcadd1: adder port map(PC, X"00000004", PCPlus4);
    pcadd2: adder port map(PCPlus4, X"00000004", PCPlus8);

    -- register file logic
    ralmux: mux2 generic map (4)
        port map(Instr(19 downto 16), "1111", RegSrc(0), RA1);
    ra2mux: mux2 generic map (4) port map(Instr(3 downto 0),
        Instr(15 downto 12), RegSrc(1), RA2);
    rf: regfile port map(clk, RegWrite, RA1, RA2,
        Instr(15 downto 12), Result,
        PCPlus8, SrcA, WriteData);
    resmux: mux2 generic map(32)

```



```

    port map(ALUResult, ReadData, MemtoReg, Result);
    ext: extend port map(Instr(23 downto 0), ImmSrc, ExtImm);

    -- ALU logic
    sh: shifter port map(WriteData, Instr(11 downto 7), Instr(6 downto 5),
srcBshifted); -- LSL
    srcbmux: mux2 generic map(32)
    port map(srcBshifted, ExtImm, ALUSrc, SrcB); -- LSL
    i_alu: alu port map(SrcA, SrcB, ALUControl, ALUResult, ALUFlags,
    carry); -- ADC
    aluresultmux: mux2 generic map(32)
    port map(ALUResult, SrcB, Shift, ALUResultOut); -- LSL
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity regfile is -- three-port register file
    port(clk:          in  STD_LOGIC;
         we3:          in  STD_LOGIC;
         ra1, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
         wd3, r15:     in  STD_LOGIC_VECTOR(31 downto 0);
         rd1, rd2:     out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
    type ramtype is array (31 downto 0) of
        STD_LOGIC_VECTOR(31 downto 0);
    signal mem: ramtype;
begin
    process(clk) begin
        if rising_edge(clk) then
            if we3 = '1' then mem(to_integer(wa3)) <= wd3;
            end if;
        end if;
    end process;
    process(all) begin
        if (to_integer(ra1) = 15) then rd1 <= r15;
        else rd1 <= mem(to_integer(ra1));
        end if;
        if (to_integer(ra2) = 15) then rd2 <= r15;
        else rd2 <= mem(to_integer(ra2));
        end if;
    end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity adder is -- adder
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
         y:   out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of adder is
begin

```



```

        if reset then q <= (others => '0');
        elsif rising_edge(clk) then
            q <= d;
        end if;
    end process;
end;

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

architecture behave of mux2 is
begin
    y <= d1 when s else d0;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity alu is
    port(a, b:      in      STD_LOGIC_VECTOR(31 downto 0);
         ALUControl: in      STD_LOGIC_VECTOR(2 downto 0);           -- ADC
         Result:    buffer STD_LOGIC_VECTOR(31 downto 0);
         ALUFlags:  out      STD_LOGIC_VECTOR(3 downto 0);
         carry:     in      STD_LOGIC);                             -- ADC
end;

architecture behave of alu is
    signal condinvb:          STD_LOGIC_VECTOR(31 downto 0);
    signal sum:               STD_LOGIC_VECTOR(32 downto 0);
    signal neg, zero, carryout, overflow: STD_LOGIC;
    signal carryin:           STD_LOGIC;                             -- ADC
begin
    carryin <= carry when ALUControl(2) else ALUControl(0);         -- ADC
    condinvb <= not b when ALUControl(0) else b;
    sum <= ('0', a) + ('0', condinvb) + carryin;                     -- ADC

    process(all) begin
        case? ALUControl(1 downto 0) is
            when "0-" => result <= sum(31 downto 0);
            when "10" => result <= a and b;
            when "11" => result <= a or b;
            when others => result <= (others => '-');
        end case?;
    end process;

    neg      <= Result(31);
    zero     <= '1' when (Result = 0) else '0';
    carryout <= (not ALUControl(1)) and sum(32);
    overflow <= (not ALUControl(1)) and

```

```

        (not (a(31) xor b(31) xor ALUControl(0))) and
        (a(31) xor sum(31));
    ALUFlags    <= (neg, zero, carryout, overflow);
end;

-- shifter needed for LSL
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity shifter is
    port(a:      in  STD_LOGIC_VECTOR(31 downto 0);
          shamt:  in  STD_LOGIC_VECTOR(4  downto 0);
          shtype: in  STD_LOGIC_VECTOR(1  downto 0);
          y:      out STD_LOGIC_VECTOR(31 downto 0));
end;
architecture behave of shifter is
begin
    process (all) begin
        case shtype is
            when "00" => y <= TO_STDLOGICVECTOR(TO_BITVECTOR(a) sll
TO_INTEGER(shamt));
            when others => y <= a;
        end case;
    end process;
end;

```

Test ARM assembly code:

; If successful, it should write the value 2 to address 20

MAIN

```

    SUB R3, PC, PC      ; R3 = 0
    ADD R3, R3, #1      ; R3 = 0x1
    LSL R3, R3, #30     ; R3 = 0x80000000
    ADD R4, R3, #1      ; R4 = 0x80000001
    CMN R3, R4          ; set flags according to R3+R4: NZCV=0011
    ADC R3, R3, #5      ; R3 = 0x80000006
    TST R3, R4          ; set NZ flags according to R3&R4: NZCV=1011
    LSL R3, R3, #1      ; R3 = 0x0000000c
    LSL R4, R4, #1      ; R4 = 0x00000002
    STRVC R4, [R3, #4]  ; mem[16]<=0x2 if V=0:
                        ; shouldn't happen
    STRVS R4, [R3, #8]  ; mem[20]<=0x2 if V=1: should happen

; E04F300F  SUB      R3,PC,PC
; E2833001  ADD      R3,R3,#0x00000001
; E1A03F83  LSL      R3,R3,#31
; E2834001  ADD      R4,R3,#0x00000001
; E1730004  CMN      R3,R4
; E2A33005  ADC      R3,R3,#0x00000005
; E1130004  TST      R3,R4
; E1A03083  LSL      R3,R3,#1
; E1A04084  LSL      R4,R4,#1

```

```
; 75834004    STRVC        R4, [R3, #0x0004]
; 65834008    STRVS        R4, [R3, #0x0008]
```

ex7.9_memfile.dat

```
E04F300F
E2833001
E1A03F83
E2834001
E1730004
E2A33005
E1130004
E1A03083
E1A04084
75834004
65834008
```

Exercise 7.10

SystemVerilog

```
// ex7.10 solutions
//
// single-cycle ARM processor
// additional instructions: EOR, LSR, TEQ, RSB

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 === 12 & WriteData === 32'h7a) begin
```

```

        $display("Simulation succeeded");
        $stop;
    end else if (DataAdr != 16) begin
        $display("Simulation failed");
        $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
    arm arm(clk, reset, PC, Instr, MemWrite, DataAdr,
           WriteData, ReadData);
    imem imem(PC, Instr);
    dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule

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

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

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

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

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

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

    initial
        $readmemh("ex7.10_memfile.dat", RAM);

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

module arm(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 [3:0] ALUFlags;

```

```

    logic      RegWrite,
               ALUSrc, MemtoReg, PCSrc;
    logic [1:0] RegSrc, ImmSrc;
    logic [2:0] ALUControl; // EOR, RSB
    logic      Shift; // LSR

    controller c(clk, reset, Instr[31:12], ALUFlags,
                 RegSrc, RegWrite, ImmSrc,
                 ALUSrc, ALUControl,
                 MemWrite, MemtoReg, PCSrc,
                 Shift); // LSR
    datapath dp(clk, reset,
                RegSrc, RegWrite, ImmSrc,
                ALUSrc, ALUControl,
                MemtoReg, PCSrc,
                ALUFlags, PC, Instr,
                ALUResult, WriteData, ReadData,
                Shift); // LSR
endmodule

module controller(input  logic      clk, reset,
                  input  logic [31:12] Instr,
                  input  logic [3:0]  ALUFlags,
                  output logic [1:0]  RegSrc,
                  output logic      RegWrite,
                  output logic [1:0]  ImmSrc,
                  output logic      ALUSrc,
                  output logic [2:0]  ALUControl, // EOR, RSB
                  output logic      MemWrite, MemtoReg,
                  output logic      PCSrc,
                  output logic      Shift); // LSR

    logic [1:0] FlagW;
    logic      PCS, RegW, MemW;
    logic      NoWrite; // TEQ

    decoder dec(Instr[27:26], Instr[25:20], Instr[15:12],
                FlagW, PCS, RegW, MemW,
                MemtoReg, ALUSrc, ImmSrc, RegSrc, ALUControl,
                NoWrite, // TEQ
                Shift); // LSR
    condlogic cl(clk, reset, Instr[31:28], ALUFlags,
                 FlagW, PCS, RegW, MemW,
                 PCSrc, RegWrite, MemWrite,
                 NoWrite); // TEQ
endmodule

module decoder(input  logic [1:0] Op,
               input  logic [5:0] Funct,
               input  logic [3:0] Rd,
               output logic [1:0] FlagW,
               output logic      PCS, RegW, MemW,
               output logic      MemtoReg, ALUSrc,
               output logic [1:0] ImmSrc, RegSrc,

```

```

        output logic [2:0] ALUControl, // EOR, RSB
        output logic      NoWrite,    // TEQ
        output logic      Shift);    // LSR
logic [9:0] controls;
logic      Branch, ALUOp;

// Main Decoder

always_comb
    case(Op)
        // Data processing immediate
        2'b00: if (Funct[5]) controls = 10'b0000101001;
                // Data processing register
                else controls = 10'b0000001001;
                // LDR
        2'b01: if (Funct[0]) controls = 10'b0001111000;
                // STR
                else controls = 10'b1001110100;
                // B
        2'b10: controls = 10'b0110100010;
                // Unimplemented
        default: controls = 10'bx;
    endcase

assign {RegSrc, ImmSrc, ALUSrc, MemtoReg,
        RegW, MemW, Branch, ALUOp} = controls;

// ALU Decoder
always_comb
    if (ALUOp) begin // which DP Instr?
        case(Funct[4:1])
            4'b0100: begin // ADD
                ALUControl = 3'b000;
                NoWrite = 1'b0;
                Shift = 1'b0;
            end
            4'b0010: begin // SUB
                ALUControl = 3'b001;
                NoWrite = 1'b0;
                Shift = 1'b0;
            end
            4'b0000: begin // AND
                ALUControl = 3'b010;
                NoWrite = 1'b0;
                Shift = 1'b0;
            end
            4'b1100: begin // OR
                ALUControl = 3'b011;
                NoWrite = 1'b0;
                Shift = 1'b0;
            end
            4'b0001: begin // EOR
                ALUControl = 3'b110;
                NoWrite = 1'b0;

```



```

        Shift = 1'b0;
    end
    4'b1001: begin                                // TEQ
        ALUControl = 3'b110;
        NoWrite = 1'b1;
        Shift = 1'b0;
    end
    4'b1101: begin                                // LSR
        ALUControl = 3'b000;
        NoWrite = 1'b0;
        Shift = 1'b1;
    end
    4'b0011: begin                                // RSB
        ALUControl = 3'b100;
        NoWrite = 1'b0;
        Shift = 1'b0;
    end
    default: begin                                // unimplemented
        ALUControl = 3'bx;
        NoWrite = 1'bx;
        Shift = 1'bx;
    end
endcase

// update flags if S bit is set
// (C & V only updated for arith instructions)
FlagW[1] = Funct[0]; // FlagW[1] = S-bit
// FlagW[0] = S-bit & (ADD | SUB)
FlagW[0] = Funct[0] &
    (ALUControl[1:0] == 2'b00 | ALUControl[1:0] == 2'b01);

end else begin
    ALUControl = 3'b000; // add for non-DP instructions
    FlagW = 2'b00; // don't update Flags
end

// PC Logic
assign PCS = ((Rd == 4'b1111) & RegW) | Branch;
endmodule

module condlogic(input logic clk, reset,
    input logic [3:0] Cond,
    input logic [3:0] ALUFlags,
    input logic [1:0] FlagW,
    input logic PCS, RegW, MemW,
    output logic PCSrc, RegWrite, MemWrite,
    input logic NoWrite); // TEQ

    logic [1:0] FlagWrite;
    logic [3:0] Flags;
    logic CondEx;

```

```

flopenr #(2)flagreg1(clk, reset, FlagWrite[1],
                    ALUFlags[3:2], Flags[3:2]);
flopenr #(2)flagreg0(clk, reset, FlagWrite[0],
                    ALUFlags[1:0], Flags[1:0]);

// write controls are conditional
condcheck cc(Cond, Flags, CondEx);
assign FlagWrite = FlagW & {2{CondEx}};
assign RegWrite  = RegW  & CondEx & ~NoWrite; // TEQ
assign MemWrite  = MemW  & CondEx;
assign PCSrc     = PCS   & CondEx;

endmodule

module condcheck(input  logic [3:0] Cond,
                 input  logic [3:0] Flags,
                 output logic      CondEx);

logic neg, zero, carry, overflow, ge;

assign {neg, zero, carry, overflow} = Flags;
assign ge = (neg == overflow);

always_comb
    case(Cond)
        4'b0000: CondEx = zero;           // EQ
        4'b0001: CondEx = ~zero;          // NE
        4'b0010: CondEx = carry;          // CS
        4'b0011: CondEx = ~carry;         // CC
        4'b0100: CondEx = neg;            // MI
        4'b0101: CondEx = ~neg;           // PL
        4'b0110: CondEx = overflow;       // VS
        4'b0111: CondEx = ~overflow;      // VC
        4'b1000: CondEx = carry & ~zero;   // HI
        4'b1001: CondEx = ~(carry & ~zero); // LS
        4'b1010: CondEx = ge;             // GE
        4'b1011: CondEx = ~ge;            // LT
        4'b1100: CondEx = ~zero & ge;      // GT
        4'b1101: CondEx = ~(~zero & ge);  // LE
        4'b1110: CondEx = 1'b1;           // Always
        default: CondEx = 1'bx;           // undefined
    endcase
endmodule

module datapath(input  logic      clk, reset,
                input  logic [1:0] RegSrc,
                input  logic      RegWrite,
                input  logic [1:0] ImmSrc,
                input  logic      ALUSrc,
                input  logic [2:0] ALUControl, // EOR, RSB
                input  logic      MemtoReg,
                input  logic      PCSrc,
                output logic [3:0] ALUFlags,
                output logic [31:0] PC,

```

```

        input  logic [31:0] Instr,
        output logic [31:0] ALUResultOut, WriteData, // LSR
        input  logic [31:0] ReadData,
        input  logic      Shift); // LSR

logic [31:0] PCNext, PCPlus4, PCPlus8;
logic [31:0] ExtImm, SrcA, SrcB, Result;
logic [3:0]  RA1, RA2;
logic [31:0] srcBshifted, ALUResult; // LSR

// next PC logic
mux2 #(32) pcmux(PCPlus4, Result, PCSrc, PCNext);
flop #(32) pcreg(clk, reset, PCNext, PC);
adder #(32) pcadd1(PC, 32'b100, PCPlus4);
adder #(32) pcadd2(PCPlus4, 32'b100, PCPlus8);

// register file logic
mux2 #(4)  ralmux(Instr[19:16], 4'b1111, RegSrc[0], RA1);
mux2 #(4)  ra2mux(Instr[3:0], Instr[15:12], RegSrc[1], RA2);
regfile    rf(clk, RegWrite, RA1, RA2,
              Instr[15:12], Result, PCPlus8,
              SrcA, WriteData);
mux2 #(32) resmux(ALUResultOut, ReadData, MemtoReg, Result);
extend     ext(Instr[23:0], ImmSrc, ExtImm);

// ALU logic
shifter     sh(WriteData, Instr[11:7], Instr[6:5], srcBshifted); // LSR
mux2 #(32)  srcbmux(srcBshifted, ExtImm, ALUSrc, SrcB);           // LSR
alu         alu(SrcA, SrcB, ALUControl,
               ALUResult, ALUFlags);
mux2 #(32)  aluresultmux(ALUResult, SrcB, Shift, ALUResultOut); // LSR

endmodule

module regfile(input  logic      clk,
               input  logic      we3,
               input  logic [3:0] ra1, ra2, wa3,
               input  logic [31:0] wd3, r15,
               output logic [31:0] rd1, rd2);

logic [31:0] rf[14:0];

// three ported register file
// read two ports combinatorially
// write third port on rising edge of clock
// register 15 reads PC+8 instead

always_ff @(posedge clk)
    if (we3) rf[wa3] <= wd3;

assign rd1 = (ra1 == 4'b1111) ? r15 : rf[ra1];
assign rd2 = (ra2 == 4'b1111) ? r15 : rf[ra2];
endmodule

```

```

module extend(input  logic [23:0] Instr,
              input  logic [1:0]  ImmSrc,
              output logic [31:0] ExtImm);

    always_comb
        case(ImmSrc)
            // 8-bit unsigned immediate
            2'b00: ExtImm = {24'b0, Instr[7:0]};
            // 12-bit unsigned immediate
            2'b01: ExtImm = {20'b0, Instr[11:0]};
            // 24-bit two's complement shifted branch
            2'b10: ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00};
            default: ExtImm = 32'bx; // undefined
        endcase
    endmodule

module adder #(parameter WIDTH=8)
    (input  logic [WIDTH-1:0] a, b,
     output logic [WIDTH-1:0] y);

    assign y = a + b;
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 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 alu(input  logic [31:0] a, b,
           input  logic [2:0]  ALUControl,           // EOR, RSB
           output logic [31:0] Result,

```

```

        output logic [3:0] ALUFlags);

logic      neg, zero, carry, overflow;
logic [31:0] condinvb;
logic [31:0] condinva;                                // RSB
logic [32:0] sum;
logic      carryin;                                    // RSB

assign carryin = ALUControl[2] | ALUControl[0];        // RSB
assign condinvb = ALUControl[0] ? ~b : b;
assign condinva = ALUControl[2] ? ~a : a;              // RSB
assign sum = condinva + condinvb + carryin;            // RSB

always_comb
    casex (ALUControl)
        3'b00?:    Result = sum;
        3'b010:    Result = a & b;
        3'b011:    Result = a | b;
        3'b110:    Result = a ^ b;
        default:    Result = 32'bx;
    endcase

assign neg      = Result[31];
assign zero     = (Result == 32'b0);
assign carry    = (ALUControl[1] == 1'b0) & sum[32];
assign overflow = (ALUControl[1] == 1'b0) &
    ~(a[31] ^ b[31] ^ ALUControl[0]) &
    (a[31] ^ sum[31]);
assign ALUFlags = {neg, zero, carry, overflow};
endmodule

// shifter needed for LSR
module shifter(input  logic [31:0] a,
               input  logic [ 4:0] shamt,
               input  logic [ 1:0] shtype,
               output logic [31:0] y);

    always_comb
        case (shtype)
            2'b01: y = a >> shamt;
            default: y = a;
        endcase
endmodule

```

VHDL

```

library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity testbench is
end;

architecture test of testbench is
    component top

```

```

    port(clk, reset:          in  STD_LOGIC;
          WriteData, DataAdr: out STD_LOGIC_VECTOR(31 downto 0);
          MemWrite:           out STD_LOGIC);
end component;
signal WriteData, DataAdr:    STD_LOGIC_VECTOR(31 downto 0);
signal clk, reset,  MemWrite: STD_LOGIC;
begin

    -- instantiate device to be tested
    dut: top port map(clk, reset, WriteData, DataAdr, MemWrite);

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

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

    -- check that 122 gets written to address 12
    -- at end of program
    process (clk) begin
        if (clk'event and clk = '0' and MemWrite = '1') then
            if (to_integer(DataAdr) = 12 and
                to_integer(WriteData) = 122) then
                report "NO ERRORS: Simulation succeeded" severity failure;
            else
                report "Simulation failed" severity failure;
            end if;
        end if;
    end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity top is -- top-level design for testing
    port(clk, reset:          in  STD_LOGIC;
          WriteData, DataAdr: buffer STD_LOGIC_VECTOR(31 downto 0);
          MemWrite:           buffer STD_LOGIC);
end;

architecture test of top is
    component arm
        port(clk, reset:          in  STD_LOGIC;
              PC:                out STD_LOGIC_VECTOR(31 downto 0);
              Instr:              in  STD_LOGIC_VECTOR(31 downto 0);

```

```

        MemWrite:          out STD_LOGIC;
        ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
        ReadData:          in  STD_LOGIC_VECTOR(31 downto 0));
end component;
component imem
    port(a: in  STD_LOGIC_VECTOR(31 downto 0);
         rd: out STD_LOGIC_VECTOR(31 downto 0));
end component;
component dmem
    port(clk, we: in STD_LOGIC;
         a, wd:   in STD_LOGIC_VECTOR(31 downto 0);
         rd:      out STD_LOGIC_VECTOR(31 downto 0));
end component;
signal PC, Instr,
        ReadData: STD_LOGIC_VECTOR(31 downto 0);
begin
    -- instantiate processor and memories
    i_arm: arm port map(clk, reset, PC, Instr, MemWrite, DataAdr,
                       WriteData, ReadData);
    i_imem: imem port map(PC, Instr);
    i_dmem: dmem port map(clk, MemWrite, DataAdr,
                       WriteData, ReadData);
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity dmem is -- data memory
    port(clk, we: in STD_LOGIC;
         a, wd:   in STD_LOGIC_VECTOR(31 downto 0);
         rd:      out STD_LOGIC_VECTOR(31 downto 0));
end;

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

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

```

```

use IEEE.NUMERIC_STD_UNSIGNED.all;
entity imem is -- instruction memory
  port(a: in STD_LOGIC_VECTOR(31 downto 0);
        rd: out STD_LOGIC_VECTOR(31 downto 0));
end;
architecture behave of imem is -- instruction memory
begin
  process is
    file mem_file: TEXT;
    variable L: line;
    variable ch: character;
    variable i, index, result: integer;
    type ramtype is array (63 downto 0) of
      STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin
    -- initialize memory from file
    for i in 0 to 63 loop -- set all contents low
      mem(i) := (others => '0');
    end loop;
    index := 0;
    FILE_OPEN(mem_file, "ex7.10_memfile.dat", READ_MODE);
    while not endfile(mem_file) loop
      readline(mem_file, L);
      result := 0;
      for i in 1 to 8 loop
        read(L, ch);
        if '0' <= ch and ch <= '9' then
          result := character'pos(ch) - character'pos('0');
        elsif 'a' <= ch and ch <= 'f' then
          result := character'pos(ch) - character'pos('a')+10;
        elsif 'A' <= ch and ch <= 'F' then
          result := character'pos(ch) - character'pos('A')+10;
        else report "Format error on line " & integer'image(index)
              severity error;
        end if;
        mem(index)(35-i*4 downto 32-i*4) :=
          to_std_logic_vector(result,4);
      end loop;
      index := index + 1;
    end loop;

    -- read memory
    loop
      rd <= mem(to_integer(a(7 downto 2)));
      wait on a;
    end loop;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity arm is -- single cycle processor
  port(clk, reset: in STD_LOGIC;
        PC: out STD_LOGIC_VECTOR(31 downto 0);

```



```

    Instr:          in  STD_LOGIC_VECTOR(31 downto 0);
    MemWrite:       out STD_LOGIC;
    ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
    ReadData:       in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of arm is
    component controller
        port(clk, reset:          in  STD_LOGIC;
             Instr:              in  STD_LOGIC_VECTOR(31 downto 12);
             ALUFlags:           in  STD_LOGIC_VECTOR(3 downto 0);
             RegSrc:             out STD_LOGIC_VECTOR(1 downto 0);
             RegWrite:           out STD_LOGIC;
             ImmSrc:             out STD_LOGIC_VECTOR(1 downto 0);
             ALUSrc:             out STD_LOGIC;
             ALUControl:         out STD_LOGIC_VECTOR(2 downto 0); -- EOR, RSB
             MemWrite:           out STD_LOGIC;
             MemtoReg:           out STD_LOGIC;
             PCSrc:              out STD_LOGIC;
             Shift:              out STD_LOGIC);
    end component;
    component datapath
        port(clk, reset:          in  STD_LOGIC;
             RegSrc:             in  STD_LOGIC_VECTOR(1 downto 0);
             RegWrite:           in  STD_LOGIC;
             ImmSrc:             in  STD_LOGIC_VECTOR(1 downto 0);
             ALUSrc:             in  STD_LOGIC;
             ALUControl:         in  STD_LOGIC_VECTOR(2 downto 0); -- EOR, RSB
             MemtoReg:           in  STD_LOGIC;
             PCSrc:              in  STD_LOGIC;
             ALUFlags:           out STD_LOGIC_VECTOR(3 downto 0);
             PC:                 buffer STD_LOGIC_VECTOR(31 downto 0);
             Instr:              in  STD_LOGIC_VECTOR(31 downto 0);
             ALUResultOut:       buffer STD_LOGIC_VECTOR(31 downto 0); -- LSR
             WriteData:          buffer STD_LOGIC_VECTOR(31 downto 0);
             ReadData:           in  STD_LOGIC_VECTOR(31 downto 0);
             Shift:              in  STD_LOGIC); -- LSR
    end component;
    signal ALUFlags: STD_LOGIC_VECTOR(3 downto 0);
    signal RegWrite, ALUSrc, MemtoReg, PCSrc: STD_LOGIC;
    signal RegSrc, ImmSrc: STD_LOGIC_VECTOR(1 downto 0);
    signal ALUControl: STD_LOGIC_VECTOR(2 downto 0); -- EOR, RSB
    signal Shift: STD_LOGIC; -- LSR
begin
    cont: controller port map(clk, reset, Instr(31 downto 12),
                             ALUFlags, RegSrc, RegWrite, ImmSrc,
                             ALUSrc, ALUControl, MemWrite,
                             MemtoReg, PCSrc,
                             Shift); -- LSR
    dp: datapath port map(clk, reset, RegSrc, RegWrite, ImmSrc,
                          ALUSrc, ALUControl, MemtoReg, PCSrc,
                          ALUFlags, PC, Instr, ALUResult,
                          WriteData, ReadData,
                          Shift); -- LSR

```



```

        PCSrc, RegWrite, MemWrite,
        NoWrite); -- TEQ
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity decoder is -- main control decoder
    port(Op:                in  STD_LOGIC_VECTOR(1 downto 0);
          Funct:            in  STD_LOGIC_VECTOR(5 downto 0);
          Rd:               in  STD_LOGIC_VECTOR(3 downto 0);
          FlagW:            out STD_LOGIC_VECTOR(1 downto 0);
          PCS, RegW, MemW:  out STD_LOGIC;
          MemtoReg, ALUSrc: out STD_LOGIC;
          ImmSrc, RegSrc:   out STD_LOGIC_VECTOR(1 downto 0);
          ALUControl:       out STD_LOGIC_VECTOR(2 downto 0); -- EOR, RSB
          NoWrite:          out STD_LOGIC;                    -- TEQ
          Shift:            out STD_LOGIC);                    -- LSR
end;

architecture behave of decoder is
    signal controls:        STD_LOGIC_VECTOR(9 downto 0);
    signal ALUOp, Branch: STD_LOGIC;
    signal op2:              STD_LOGIC_VECTOR(3 downto 0);
begin
    op2 <= (Op, Funct(5), Funct(0));
    process(all) begin -- Main Decoder
        case? (op2) is
            when "000-" => controls <= "00000001001";
            when "001-" => controls <= "00001010101";
            when "01-0" => controls <= "1001110100";
            when "01-1" => controls <= "0001111000";
            when "10--" => controls <= "0110100010";
            when others => controls <= "-----";
        end case?;
    end process;

    (RegSrc, ImmSrc, ALUSrc, MemtoReg, RegW, MemW,
     Branch, ALUOp) <= controls;

    process(all) begin -- ALU Decoder
        if (ALUOp) then
            case Funct(4 downto 1) is
                when "0100" => ALUControl <= "000"; -- ADD
                                NoWrite <= '0';
                                Shift <= '0';
                when "0010" => ALUControl <= "001"; -- SUB
                                NoWrite <= '0';
                                Shift <= '0';
                when "0000" => ALUControl <= "010"; -- AND
                                NoWrite <= '0';
                                Shift <= '0';
                when "1100" => ALUControl <= "011"; -- ORR
                                NoWrite <= '0';
                                Shift <= '0';
                when "0001" => ALUControl <= "110"; -- EOR
            end case;
        end if;
    end process;
end behave;

```

```

        NoWrite <= '0';
        Shift <= '0';
    when "1001" => ALUControl <= "110"; -- TEQ
        NoWrite <= '1';
        Shift <= '0';
    when "1101" => ALUControl <= "000"; -- LSR
        NoWrite <= '0';
        Shift <= '1';
    when "1011" => ALUControl <= "100"; -- RSB
        NoWrite <= '0';
        Shift <= '0';
    when others => ALUControl <= "---"; -- unimplemented
        NoWrite <= '-';
        Shift <= '-';

    end case;
    FlagW(1) <= Funct(0);
    FlagW(0) <= Funct(0) and (not ALUControl(1));
else
    ALUControl <= "000";
    NoWrite <= '0';
    Shift <= '0';
    FlagW <= "00";
end if;
end process;

PCS <= ((and Rd) and RegW) or Branch;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condlogic is -- Conditional logic
    port(clk, reset:      in  STD_LOGIC;
        Cond:            in  STD_LOGIC_VECTOR(3 downto 0);
        ALUFlags:        in  STD_LOGIC_VECTOR(3 downto 0);
        FlagW:           in  STD_LOGIC_VECTOR(1 downto 0);
        PCS, RegW, MemW: in  STD_LOGIC;
        PCSrc, RegWrite: out STD_LOGIC;
        MemWrite:        out STD_LOGIC;
        NoWrite:         in  STD_LOGIC); -- TEQ
end;

architecture behave of condlogic is
    component condcheck
        port(Cond:      in  STD_LOGIC_VECTOR(3 downto 0);
            Flags:      in  STD_LOGIC_VECTOR(3 downto 0);
            CondEx:     out STD_LOGIC);
    end component;
    component flopenr generic(width: integer);
        port(clk, reset, en: in  STD_LOGIC;
            d:      in  STD_LOGIC_VECTOR(width-1 downto 0);
            q:      out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    signal FlagWrite: STD_LOGIC_VECTOR(1 downto 0);
    signal Flags:     STD_LOGIC_VECTOR(3 downto 0);
    signal CondEx:    STD_LOGIC;

```

```

begin
    flagreg1: flopenr generic map(2)
        port map(clk, reset, FlagWrite(1),
            ALUFlags(3 downto 2), Flags(3 downto 2));
    flagreg0: flopenr generic map(2)
        port map(clk, reset, FlagWrite(0),
            ALUFlags(1 downto 0), Flags(1 downto 0));
    cc: condcheck port map(Cond, Flags, CondEx);

    FlagWrite <= FlagW and (CondEx, CondEx);
    RegWrite  <= RegW  and CondEx and (not NoWrite); -- TEQ
    MemWrite  <= MemW  and CondEx;
    PCSrc     <= PCS   and CondEx;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condcheck is
    port(Cond:          in  STD_LOGIC_VECTOR(3 downto 0);
         Flags:         in  STD_LOGIC_VECTOR(3 downto 0);
         CondEx:        out STD_LOGIC);
end;

architecture behave of condcheck is
    signal neg, zero, carry, overflow, ge: STD_LOGIC;
begin
    (neg, zero, carry, overflow) <= Flags;
    ge <= (neg xnor overflow);

    process(all) begin -- Condition checking
        case Cond is
            when "0000" => CondEx <= zero;
            when "0001" => CondEx <= not zero;
            when "0010" => CondEx <= carry;
            when "0011" => CondEx <= not carry;
            when "0100" => CondEx <= neg;
            when "0101" => CondEx <= not neg;
            when "0110" => CondEx <= overflow;
            when "0111" => CondEx <= not overflow;
            when "1000" => CondEx <= carry and (not zero);
            when "1001" => CondEx <= not(carry and (not zero));
            when "1010" => CondEx <= ge;
            when "1011" => CondEx <= not ge;
            when "1100" => CondEx <= (not zero) and ge;
            when "1101" => CondEx <= not ((not zero) and ge);
            when "1110" => CondEx <= '1';
            when others => CondEx <= '-';
        end case;
    end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity datapath is
    port(clk, reset:      in  STD_LOGIC;
         RegSrc:         in  STD_LOGIC_VECTOR(1 downto 0);

```

```

    RegWrite:      in   STD_LOGIC;
    ImmSrc:        in   STD_LOGIC_VECTOR(1 downto 0);
    ALUSrc:        in   STD_LOGIC;
    ALUControl:    in   STD_LOGIC_VECTOR(2 downto 0);  -- EOR, RSB
    MentoReg:      in   STD_LOGIC;
    PCSrc:         in   STD_LOGIC;
    ALUFlags:      out  STD_LOGIC_VECTOR(3 downto 0);
    PC:            buffer STD_LOGIC_VECTOR(31 downto 0);
    Instr:         in   STD_LOGIC_VECTOR(31 downto 0);
    ALUResultOut:  out  STD_LOGIC_VECTOR(31 downto 0);  -- LSR
    WriteData:     buffer STD_LOGIC_VECTOR(31 downto 0);
    ReadData:      in   STD_LOGIC_VECTOR(31 downto 0);
    Shift:         in   STD_LOGIC);  -- LSR
end;
```

architecture struct of datapath is

```

    component alu
        port(a, b:      in   STD_LOGIC_VECTOR(31 downto 0);
              ALUControl: in   STD_LOGIC_VECTOR(2 downto 0);  -- EOR, RSB
              Result:    buffer STD_LOGIC_VECTOR(31 downto 0);
              ALUFlags:   out  STD_LOGIC_VECTOR(3 downto 0));
    end component;
    component regfile
        port(clk:      in   STD_LOGIC;
              we3:      in   STD_LOGIC;
              ral, ra2, wa3: in   STD_LOGIC_VECTOR(3 downto 0);
              wd3, r15:  in   STD_LOGIC_VECTOR(31 downto 0);
              rd1, rd2:  out  STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component adder
        port(a, b: in   STD_LOGIC_VECTOR(31 downto 0);
              y:   out  STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component extend
        port(Instr: in   STD_LOGIC_VECTOR(23 downto 0);
              ImmSrc: in   STD_LOGIC_VECTOR(1 downto 0);
              ExtImm: out  STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component flopr generic(width: integer);
        port(clk, reset: in   STD_LOGIC;
              d:         in   STD_LOGIC_VECTOR(width-1 downto 0);
              q:         out  STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component mux2 generic(width: integer);
        port(d0, d1: in   STD_LOGIC_VECTOR(width-1 downto 0);
              s:     in   STD_LOGIC;
              y:     out  STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component shifter -- LSL
        port(a:      in   STD_LOGIC_VECTOR(31 downto 0);
              shamt:  in   STD_LOGIC_VECTOR(4  downto 0);
              shtype: in   STD_LOGIC_VECTOR(1  downto 0);
              y:      out  STD_LOGIC_VECTOR(31 downto 0));
    end component;
```

```

signal PCNext, PCPlus4, PCPlus8: STD_LOGIC_VECTOR(31 downto 0);
signal ExtImm, Result:          STD_LOGIC_VECTOR(31 downto 0);
signal SrcA, SrcB:              STD_LOGIC_VECTOR(31 downto 0);
signal RA1, RA2:                STD_LOGIC_VECTOR(3 downto 0);
signal srcBshifted, ALUResult:  STD_LOGIC_VECTOR(31 downto 0); -- LSR
begin
  -- next PC logic
  pcmux: mux2 generic map(32)
    port map(PCPlus4, Result, PCSrc, PCNext);
  pcreg: flopr generic map(32) port map(clk, reset, PCNext, PC);
  pcadd1: adder port map(PC, X"00000004", PCPlus4);
  pcadd2: adder port map(PCPlus4, X"00000004", PCPlus8);

  -- register file logic
  ralmux: mux2 generic map (4)
    port map(Instr(19 downto 16), "1111", RegSrc(0), RA1);
  ra2mux: mux2 generic map (4) port map(Instr(3 downto 0),
    Instr(15 downto 12), RegSrc(1), RA2);
  rf: regfile port map(clk, RegWrite, RA1, RA2,
    Instr(15 downto 12), Result,
    PCPlus8, SrcA, WriteData);
  resmux: mux2 generic map(32)
    port map(ALUResultOut, ReadData, MemtoReg, Result); -- LSR
  ext: extend port map(Instr(23 downto 0), ImmSrc, ExtImm);

  -- ALU logic
  sh: shifter port map(WriteData, Instr(11 downto 7), Instr(6 downto 5),
srcBshifted); -- LSR
  srcbmux: mux2 generic map(32)
    port map(srcBshifted, ExtImm, ALUSrc, SrcB); -- LSR
  i_alu: alu port map(SrcA, SrcB, ALUControl, ALUResult, ALUFlags);
  aluresultmux: mux2 generic map(32)
    port map(ALUResult, SrcB, Shift, ALUResultOut); -- LSR

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity regfile is -- three-port register file
  port(clk:          in  STD_LOGIC;
        we3:         in  STD_LOGIC;
        ra1, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
        wd3, r15:    in  STD_LOGIC_VECTOR(31 downto 0);
        rd1, rd2:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
  type ramtype is array (31 downto 0) of
    STD_LOGIC_VECTOR(31 downto 0);
  signal mem: ramtype;
begin
  process(clk) begin
    if rising_edge(clk) then
      if we3 = '1' then mem(to_integer(wa3)) <= wd3;

```

```

        end if;
    end if;
end process;
process(all) begin
    if (to_integer(ra1) = 15) then rd1 <= r15;
    else rd1 <= mem(to_integer(ra1));
    end if;
    if (to_integer(ra2) = 15) then rd2 <= r15;
    else rd2 <= mem(to_integer(ra2));
    end if;
end process;
end;
```

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity adder is -- adder
port(a, b: in STD_LOGIC_VECTOR(31 downto 0);
y: out STD_LOGIC_VECTOR(31 downto 0));
end;

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

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity extend is
port(Instr: in STD_LOGIC_VECTOR(23 downto 0);
ImmSrc: in STD_LOGIC_VECTOR(1 downto 0);
ExtImm: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of extend is
begin
process(all) begin
case ImmSrc is
when "00" => ExtImm <= (X"000000", Instr(7 downto 0));
when "01" => ExtImm <= (X"00000", Instr(11 downto 0));
when "10" => ExtImm <= (Instr(23), Instr(23), Instr(23),
Instr(23), Instr(23), Instr(23), Instr(23), Instr(23 downto 0), "00");
when others => ExtImm <= X"-----";
end case;
end process;
end;

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

architecture asynchronous of flopenr is


```

begin
  process(clk, reset) begin
    if reset then q <= (others => '0');
    elsif rising_edge(clk) then
      if en then
        q <= d;
      end if;
    end if;
  end process;
end;

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

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

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

architecture behave of mux2 is
begin
  y <= d1 when s else d0;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity alu is
  port(a, b:          in      STD_LOGIC_VECTOR(31 downto 0);
        ALUControl: in      STD_LOGIC_VECTOR(2 downto 0);      -- EOR, RSB
        Result:      buffer STD_LOGIC_VECTOR(31 downto 0);
        ALUFlags:    out     STD_LOGIC_VECTOR(3 downto 0));
end;

architecture behave of alu is
  signal neg, zero, carry, overflow: STD_LOGIC;

```

```

    signal condinvb: STD_LOGIC_VECTOR(31 downto 0);
    signal condinva: STD_LOGIC_VECTOR(31 downto 0);           -- RSB
    signal sum:      STD_LOGIC_VECTOR(32 downto 0);
    signal carryin:  STD_LOGIC;                               -- RSB
begin
    carryin  <= ALUControl(2) or ALUControl(0);               -- RSB
    condinvb <= not b when ALUControl(0) else b;
    condinva <= not a when ALUControl(2) else a;               -- RSB
    sum <= ('0', condinva) + ('0', condinvb) + carryin;       -- RSB

    process(all) begin
        case ALUControl is
            when "000" => result <= sum(31 downto 0);
            when "001" => result <= sum(31 downto 0);
            when "010" => result <= a and b;
            when "011" => result <= a or b;
            when "110" => result <= a xor b;
            when others => result <= (others => '-');
        end case;
    end process;

    neg      <= Result(31);
    zero     <= '1' when (Result = 0) else '0';
    carry    <= (not ALUControl(1)) and sum(32);
    overflow <= (not ALUControl(1)) and
                (not (a(31) xor b(31) xor ALUControl(0))) and
                (a(31) xor sum(31));
    ALUFlags <= (neg, zero, carry, overflow);
end;

-- shifter needed for LSR
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity shifter is
    port(a:      in  STD_LOGIC_VECTOR(31 downto 0);
          shamt:  in  STD_LOGIC_VECTOR(4  downto 0);
          shtype: in  STD_LOGIC_VECTOR(1  downto 0);
          y:      out STD_LOGIC_VECTOR(31 downto 0));
end;
architecture behave of shifter is
begin
    process (all) begin
        case shtype is
            when "01" => y <= TO_STDLOGICVECTOR(TO_BITVECTOR(a) srl
TO_INTEGER(shamt));
            when others => y <= a;
        end case;
    end process;
end;

```

Test assembly code

```

; If successful, it should write the value 0x7A to address 12
MAIN

```

```

SUB R3, PC, PC           ; R3 = 0
ADD R4, R3, #0x7A        ; R4 = 0x7A
ADD R5, R3, #0x6C        ; R5 = 0x6C
EOR R6, R4, R5           ; R6 = R4 ^ R5 = 0x16
LSR R6, R6, #2           ; R6 = R6 >> 2 = 5
TEQ R4, R6               ; set flags according to 0x7a ^ 5: NZCV = 0000
STREQ R4, [R6, #3]       ; mem[8]<=0x7A if Z=1: shouldn't happen
TEQ R6, R6               ; set flags according to 5 ^ 5: NZCV = 0100
STREQ R4, [R6, #7]       ; mem[12]<=0x7A if Z=1: should happen

; E04F300F  SUB      R3,PC,PC
; E283407A  ADD      R4,R3,#0x0000007A
; E283506C  ADD      R5,R3,#0x0000006C
; E0246005  EOR      R6,R4,R5
; E1A06126  MOV      R6,R6,LSR #2
; E1340006  TEQ      R4,R6
; 05864003  STREQ    R4,[R6,#0x0003]
; E1360006  TEQ      R6,R6
; 05864007  STREQ    R4,[R6,#0x0007]

```

ex7.9_memfile.dat

```

E04F300F
E283407A
E283506C
E0246005
E1A06126
E1340006
05864003
E1360006
05864007

```

Exercise 7.11

- (a) STR: it stores the value in the register specified by bits 3:0 (Rm) instead of bits 15:12 (Rd).
- (b) LDR, STR: the memory always reads the value at the address specified by the PC, instead of a data memory address.
- (c) All instructions. PC+4 is never written to the PC register.

Exercise 7.12

- (a) ADD, SUB, AND, ORR with register Src2: the second source is the value in the register specified by bits 15:12 (Rm) instead of bits 3:0 (Rd).
- (b) All instructions. PC is not connected to the memory address port, so instructions are never fetched from memory.
- (c) All instructions. PC is updated with bogus values.

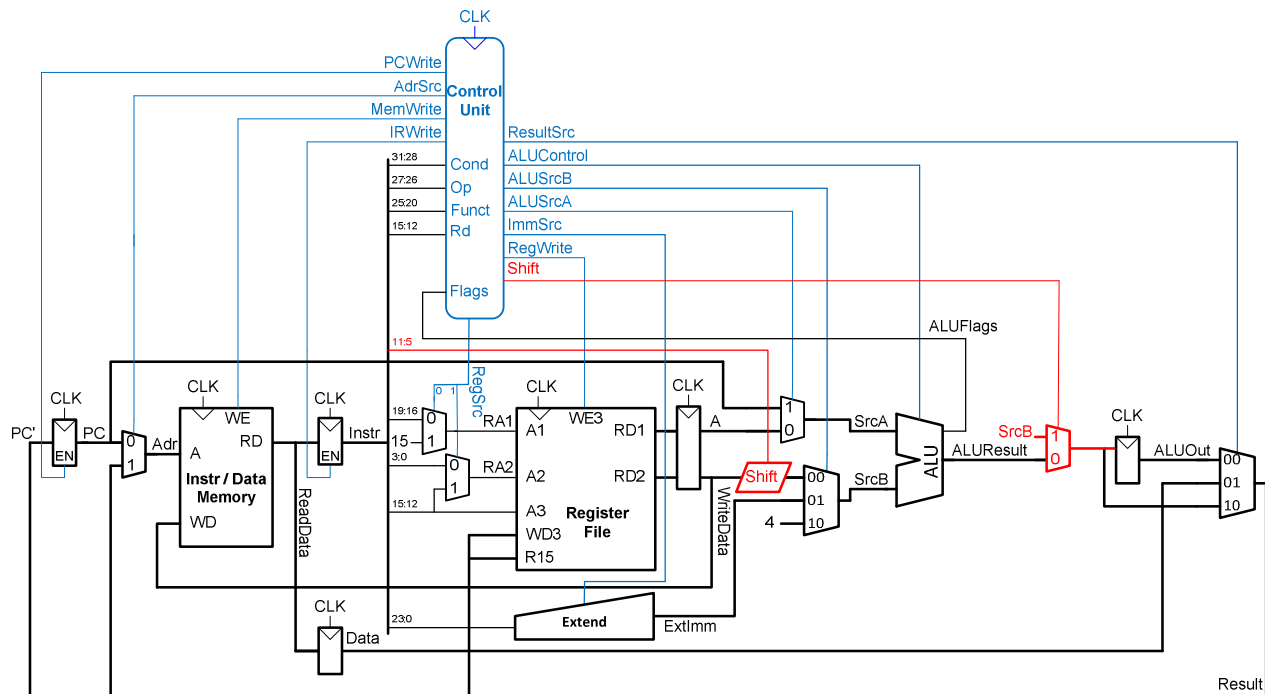
Exercise 7.13

(a) ASR

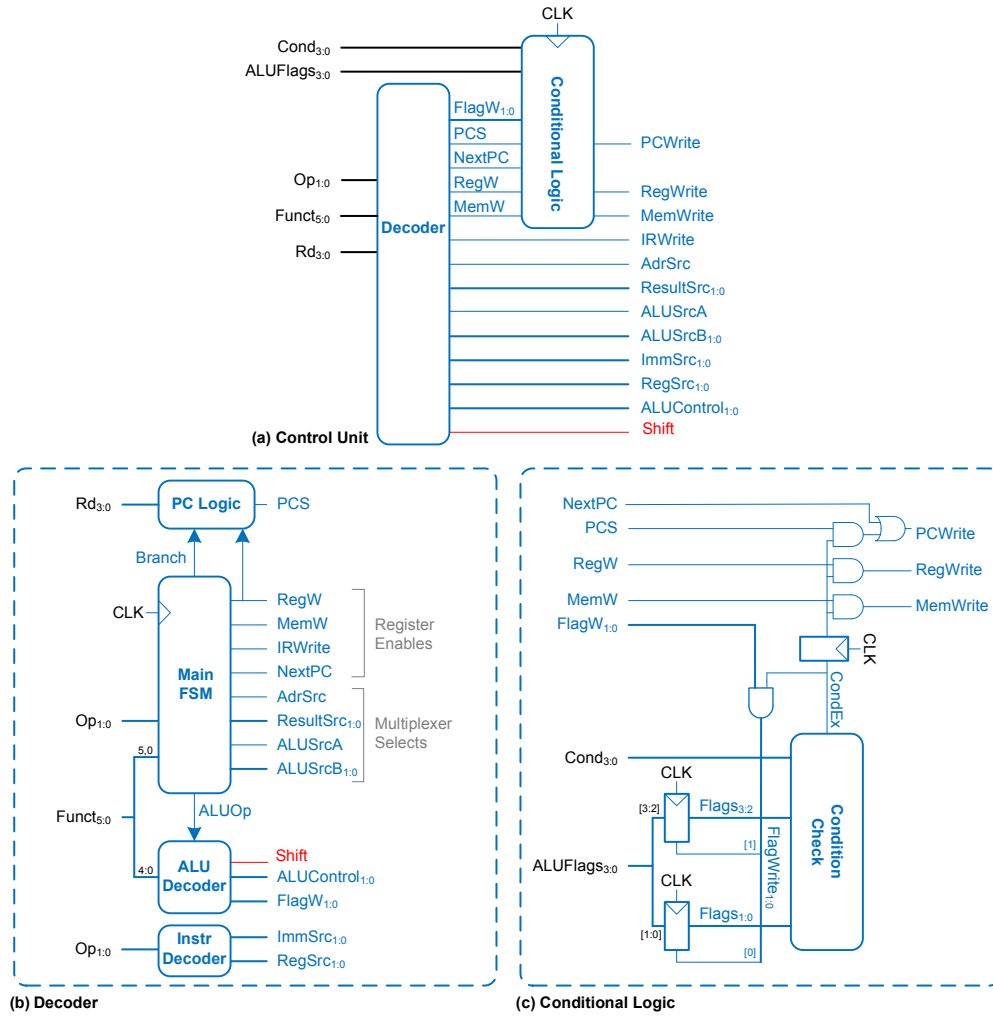
ALU Decoder truth table

<i>ALUOp</i>	<i>Funct_{4:1} (cmd)</i>	<i>Funct₀ (S)</i>	Notes	<i>ALUControl_{1:0}</i>	<i>FlagW_{1:0}</i>	<i>Shift</i>
0	X	X	Not DP	00	00	0
1	0100	0	ADD	00	00	0
		1			11	0
	0010	0	SUB	01	00	0
		1			11	0
	0000	0	AND	10	00	0
		1			10	0
	1100	0	ORR	11	00	0
		1			10	0
	1101	0	ASR	XX	00	1
		1	ASR	XX	10	1

Datapath



Control

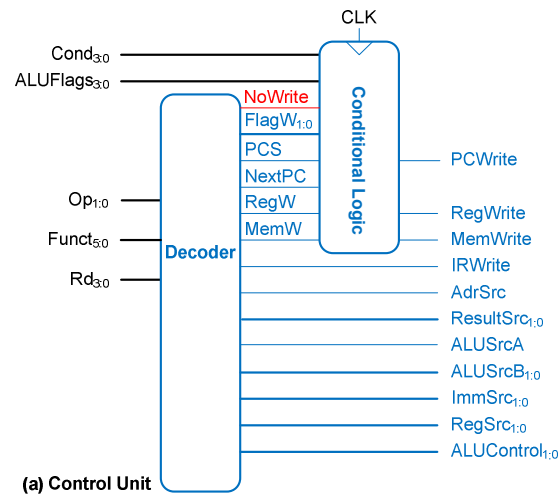


(b) TST

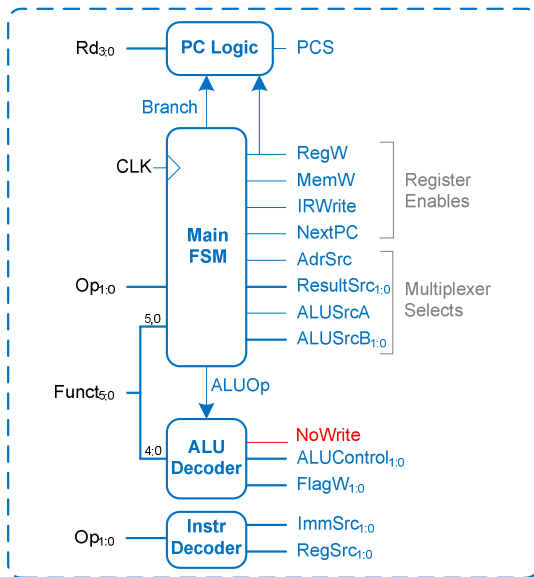
ALU Decoder truth table

$ALUOp$	$Funct_{4:1} (cmd)$	$Funct_0 (S)$	Notes	$ALUControl_{1:0}$	$FlagW_{1:0}$	$NoWrite$
0	X	X	Not DP	00	00	0
1	0100	0	ADD	00	00	0
		1			11	0
	0010	0	SUB	01	00	0
		1			11	0
	0000	0	AND	10	00	0
		1			10	0
	1100	0	ORR	11	00	0
		1			10	0
	1000	1	TST	10	10	1

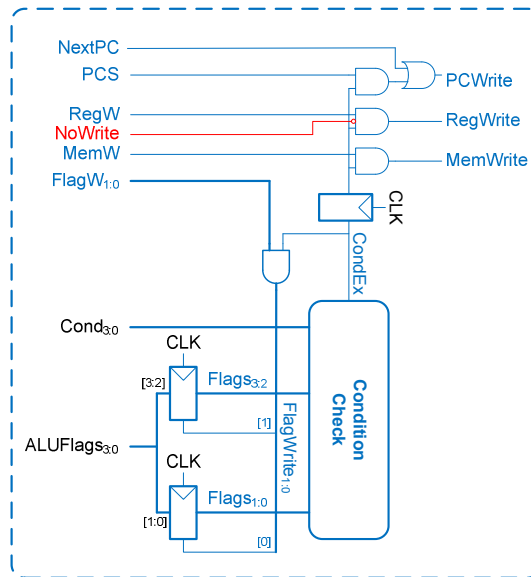
Control



(a) Control Unit



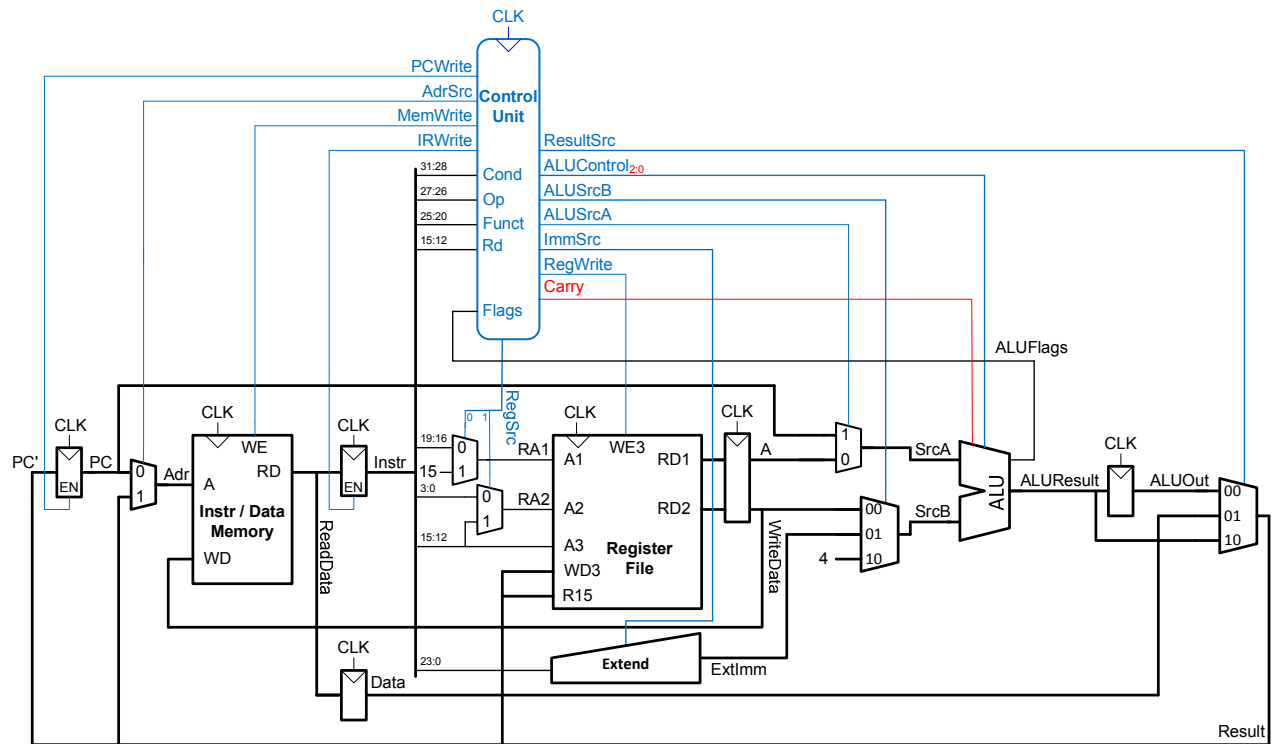
(b) Decoder



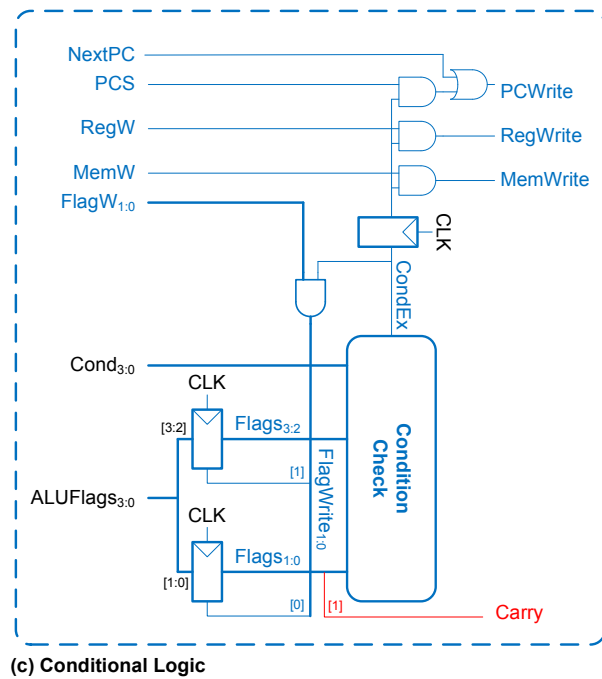
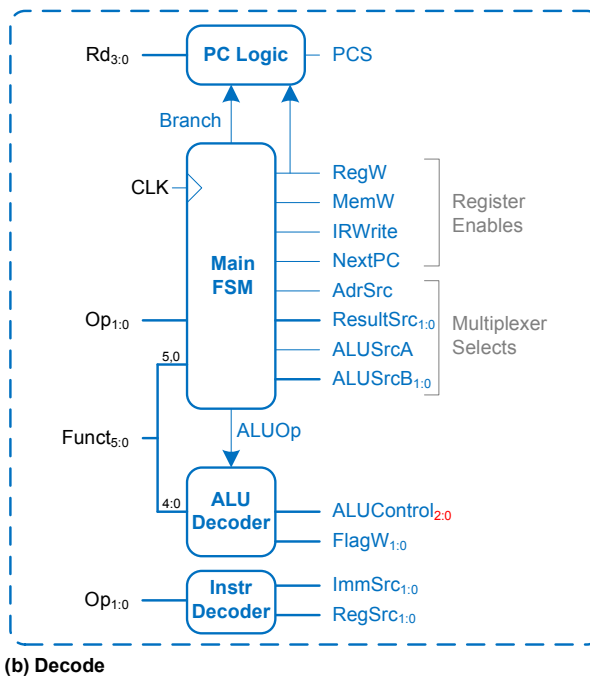
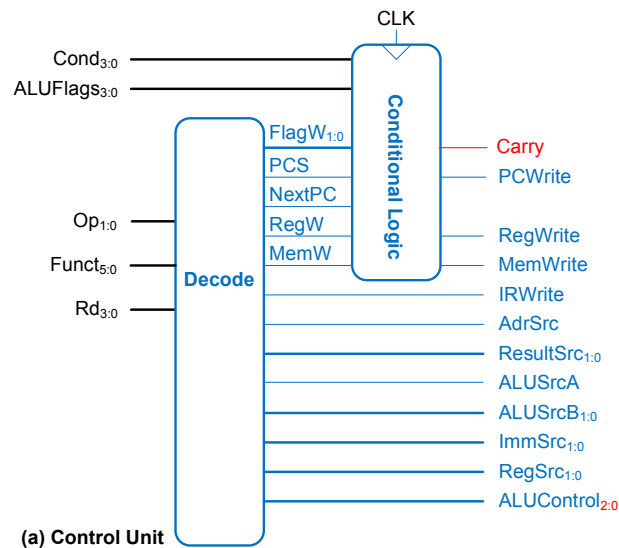
(c) Conditional Logic

(c) SBC

ALU



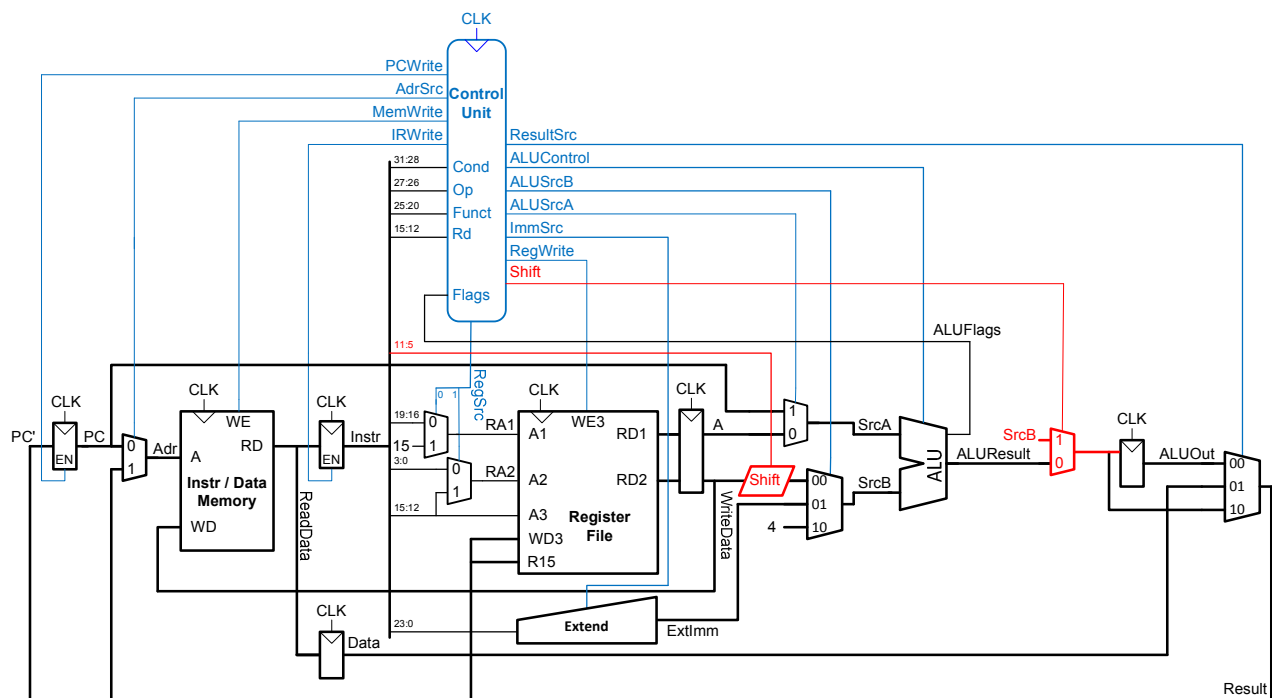
Control



(d) ROR

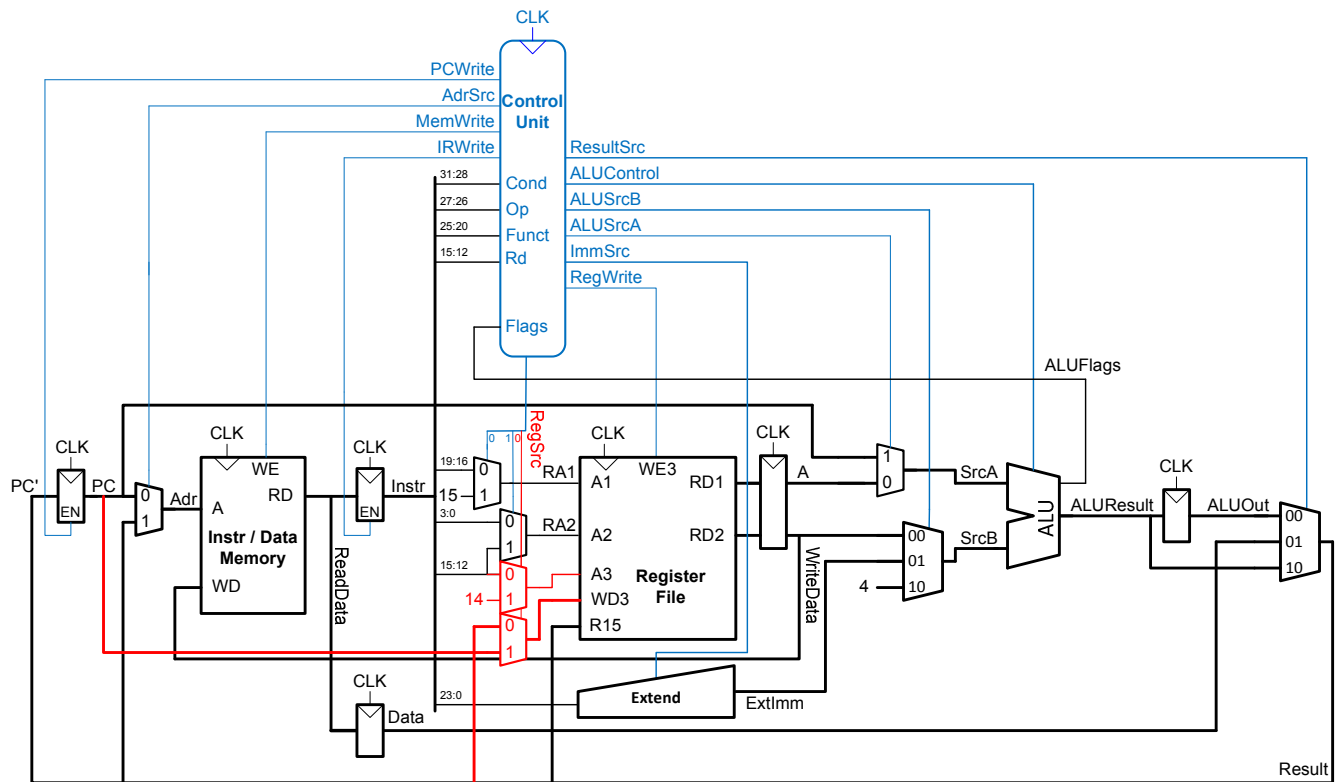
ALU Decoder truth table

<i>ALUOp</i>	<i>Funct</i> _{4:1} (<i>cmd</i>)	<i>Funct</i> ₀ (<i>S</i>)	Notes	<i>ALUControl</i> _{1:0}	<i>FlagW</i> _{1:0}	<i>Shift</i>
0	X	X	Not DP	00	00	0
1	0100	0	ADD	00	00	0
		1			11	0
	0010	0	SUB	01	00	0
		1			11	0
	0000	0	AND	10	00	0
		1			10	0
	1100	0	ORR	11	00	0
		1			10	0
	1101	0	ROR	XX	00	1
		1		XX	10	1

Datapath**Exercise 7.14**

(a) BL

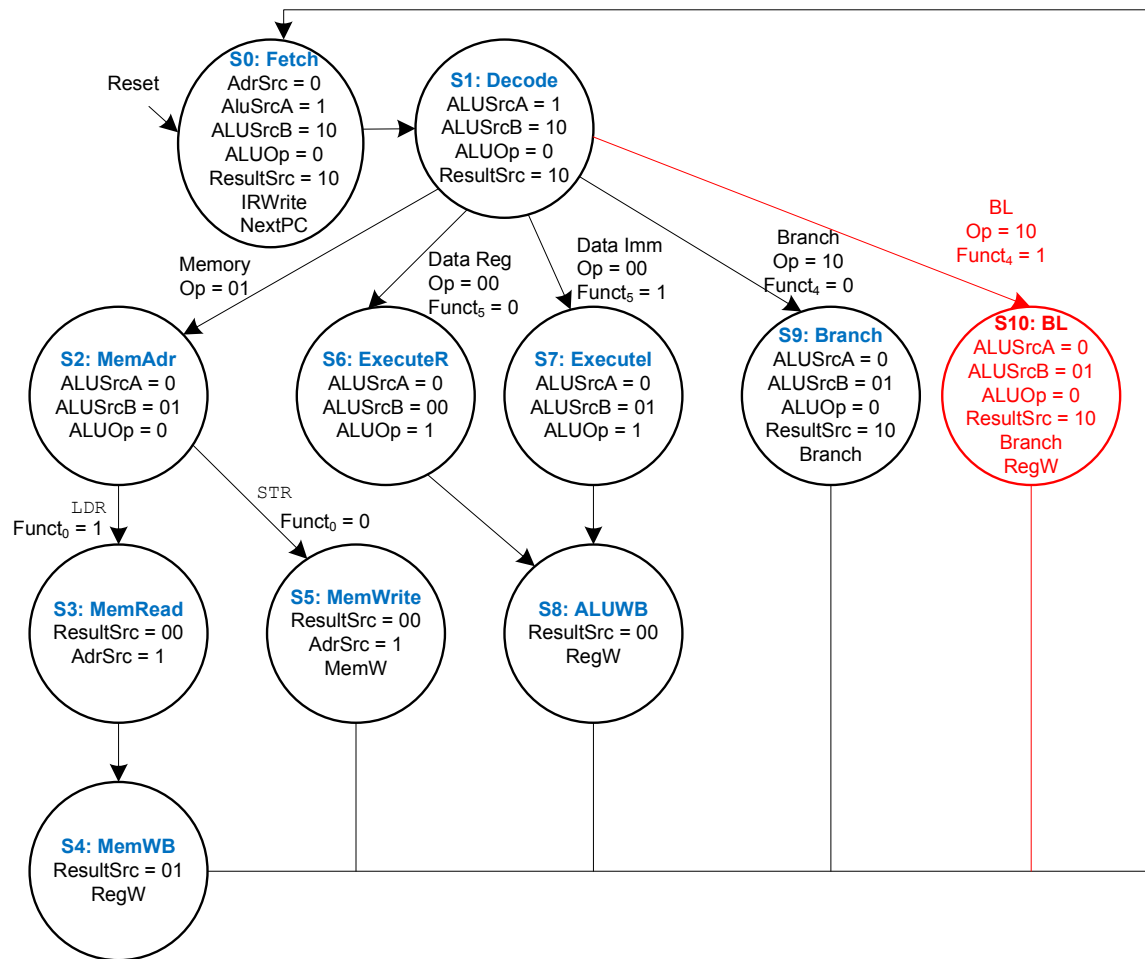
Datapath



Instr Decoder logic for RegSrc_{1:0} and ImmSrc_{1:0}

Instruction	Op	Funct ₅	Funct ₄	Funct ₀	RegSrc _{2:0}	ImmSrc _{1:0}
LDR	01	X	X	1	X0	01
STR	01	X	X	0	10	01
DP imm	00	1	X	X	X0	00
DP reg	00	0	X	X	00	00
B	10	X	0	X	X1	10
BL	10	X	1	X	X1	10

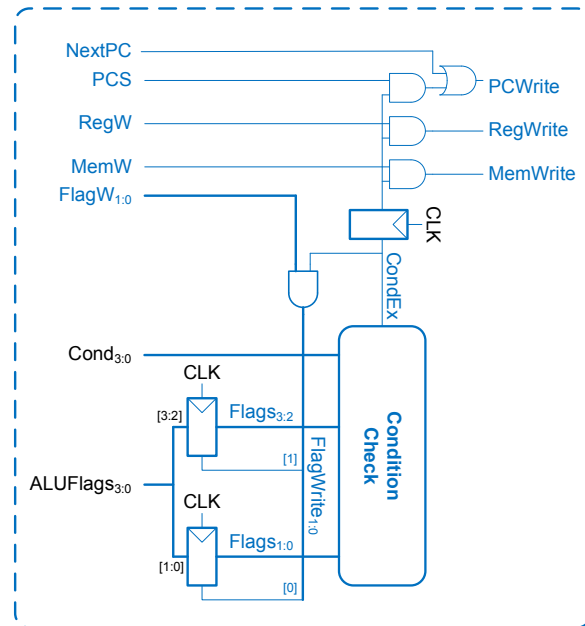
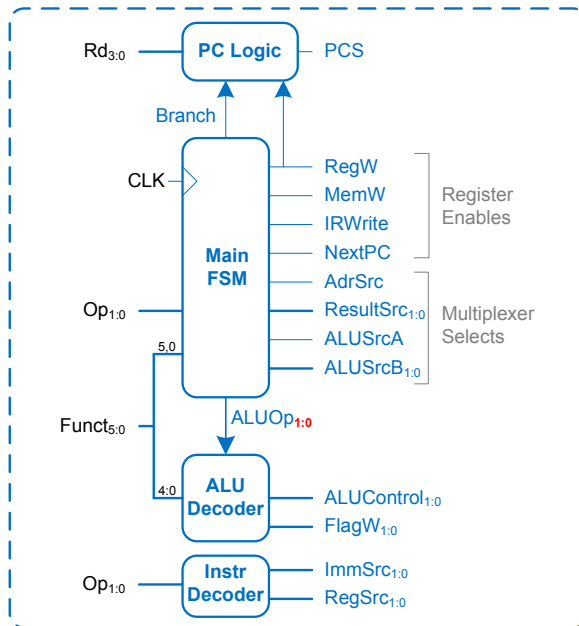
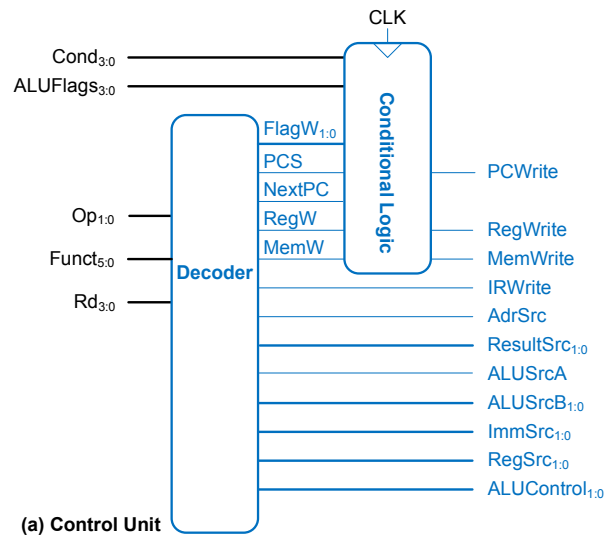
FSM



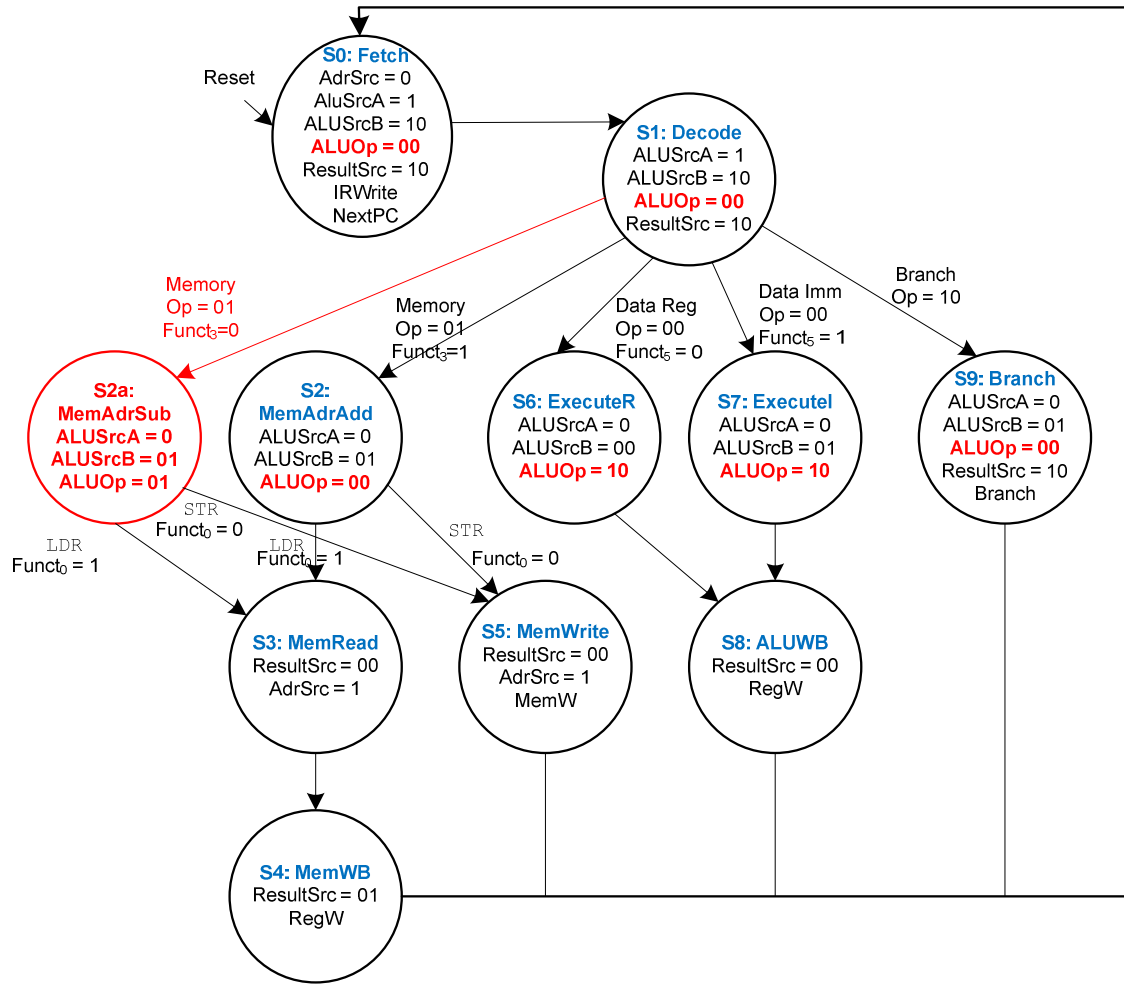
State	Datapath μ Op
Fetch	Instr \leftarrow Mem[PC]; PC \leftarrow PC+4
Decode	ALUOut \leftarrow PC+4
MemAdr	ALUOut \leftarrow Rn + Imm
MemRead	Data \leftarrow Mem[ALUOut]
MemWB	Rd \leftarrow Data
MemWrite	Mem[ALUOut] \leftarrow Rd
ExecuteR	ALUOut \leftarrow Rn op Rm
Executel	ALUOut \leftarrow Rn op Imm
ALUWB	Rd \leftarrow ALUOut
Branch	PC \leftarrow R15 + offset
BL	PC \leftarrow R15 + offset, LR \leftarrow PC+4

(b) LDR (with addition or subtraction of imm12)

Control



FSM



State	Datapath μ Op
Fetch	Instr \leftarrow Mem[PC]; PC \leftarrow PC+4
Decode	ALUOut \leftarrow PC+4
MemAdrAdd	ALUOut \leftarrow Rn + Imm
MemAdrSub	ALUOut \leftarrow Rn - Imm
MemRead	Data \leftarrow Mem[ALUOut]
MemWB	Rd \leftarrow Data
MemWrite	Mem[ALUOut] \leftarrow Rd
ExecuteR	ALUOut \leftarrow Rn op Rm
Executel	ALUOut \leftarrow Rn op Imm
ALUWB	Rd \leftarrow ALUOut
Branch	PC \leftarrow R15 + offset

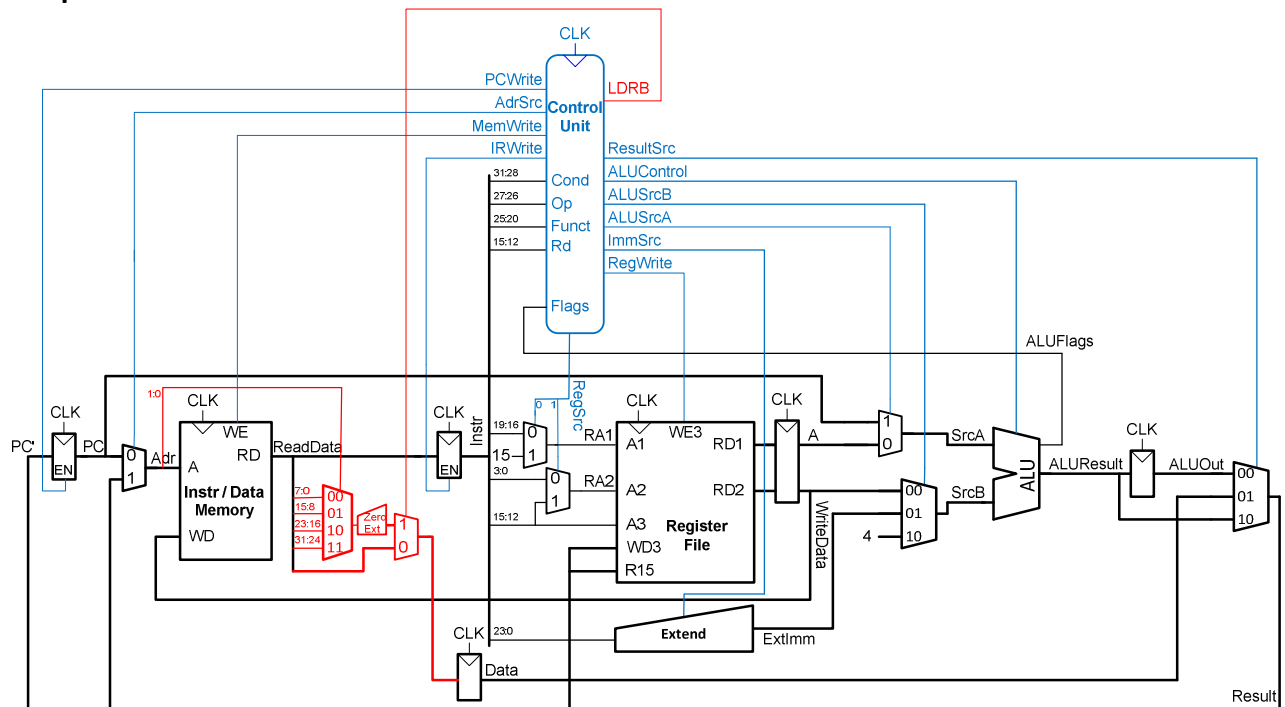
ALU Decoder truth table

$ALUOp_{1:0}$	$Funct_{4:1} (cmd)$	$Funct_0 (S)$	Notes	$ALUControl_{1:0}$	$FlagW_{1:0}$
00	X	X	Not DP: add	00	00
01	X	X	Not DP: sub	01	00
10	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00

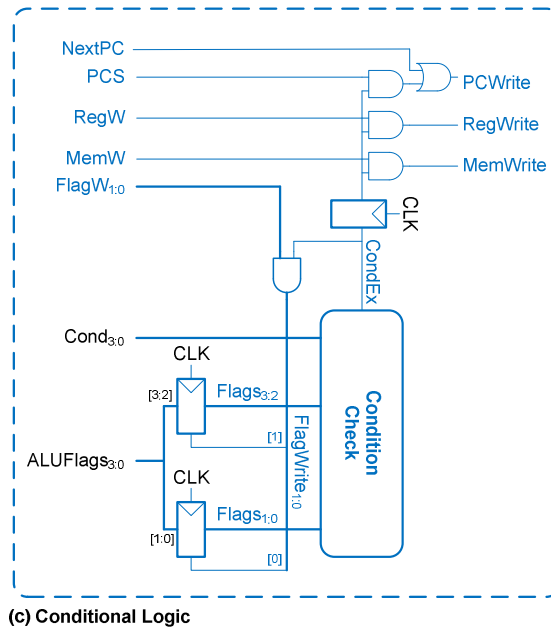
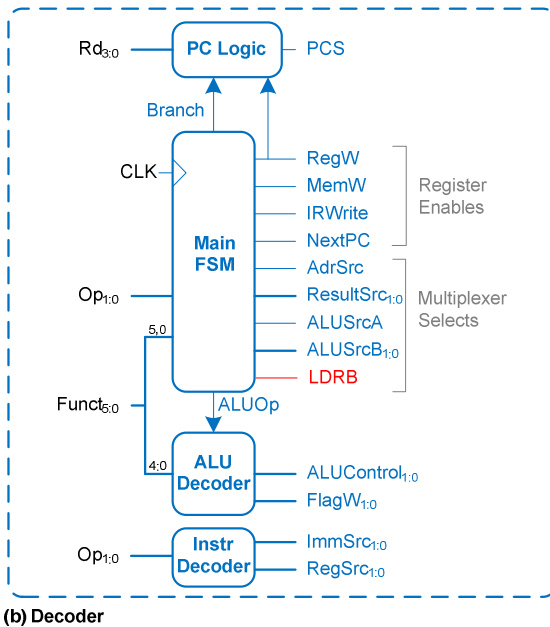
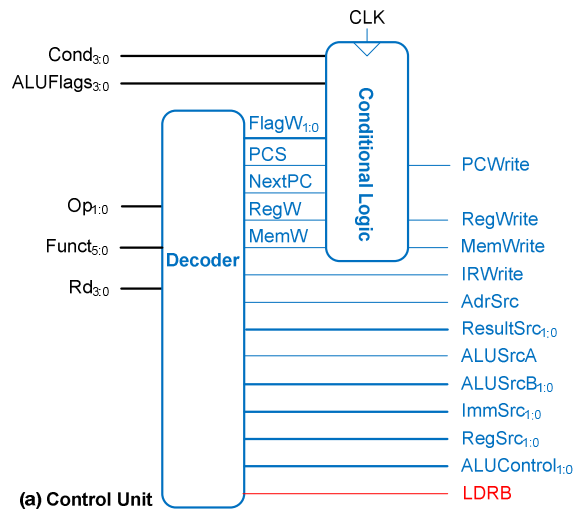
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10

(c) LDRB (with positive immediate offset only)

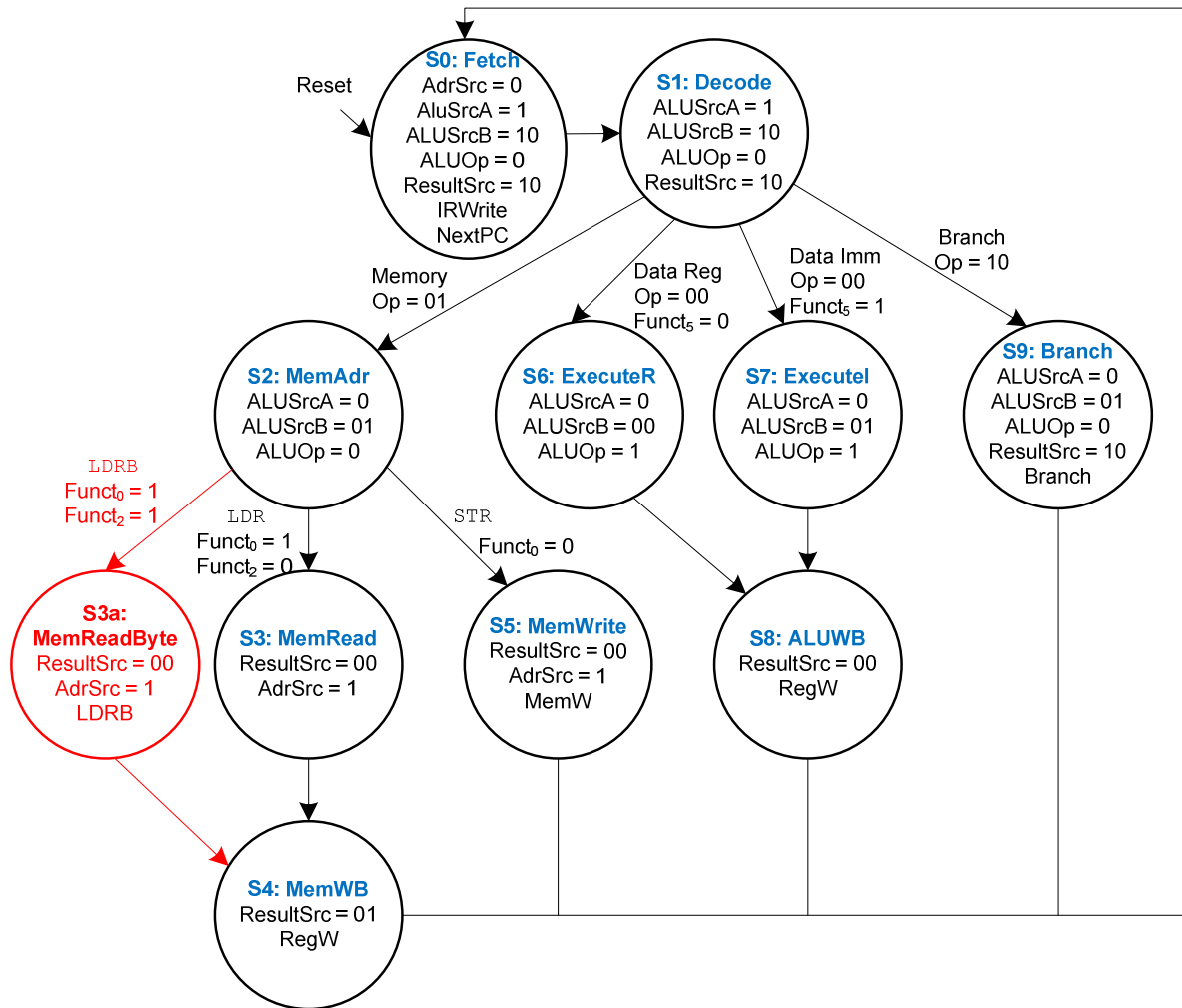
Datapath



Control



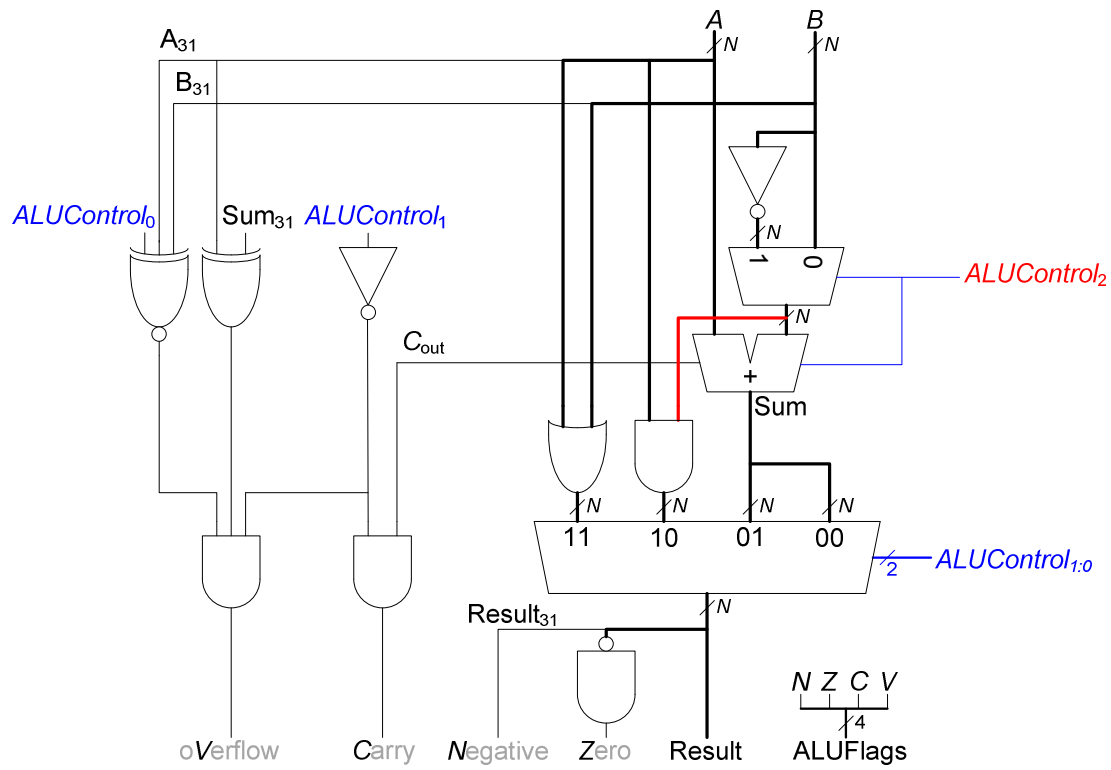
FSM



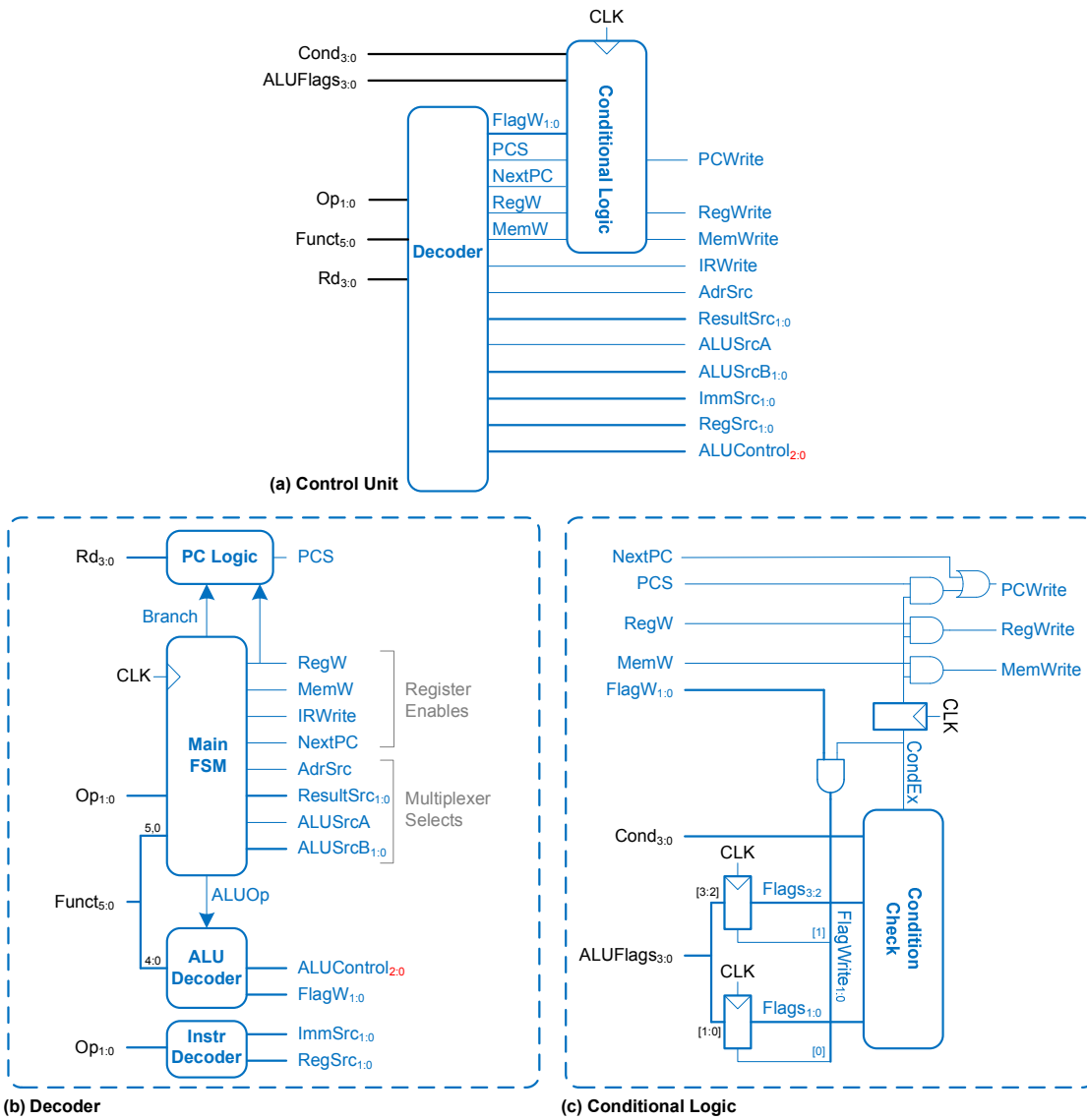
State	Datapath μ Op
Fetch	$\text{Instr} \leftarrow \text{Mem}[\text{PC}]; \text{PC} \leftarrow \text{PC} + 4$
Decode	$\text{ALUOut} \leftarrow \text{PC} + 4$
MemAdr	$\text{ALUOut} \leftarrow \text{Rn} + \text{Imm}$
MemRead	$\text{Data} \leftarrow \text{Mem}[\text{ALUOut}]$
MemReadByte	$\text{Data} \leftarrow \text{Mem}[\text{ALUOut}]_{7:0}$
MemWB	$\text{Rd} \leftarrow \text{Data}$
MemWrite	$\text{Mem}[\text{ALUOut}] \leftarrow \text{Rd}$
ExecuteR	$\text{ALUOut} \leftarrow \text{Rn} \text{ op } \text{Rm}$
Executel	$\text{ALUOut} \leftarrow \text{Rn} \text{ op } \text{Imm}$
ALUWB	$\text{Rd} \leftarrow \text{ALUOut}$
Branch	$\text{PC} \leftarrow \text{R15} + \text{offset}$

(d) BIC

ALU



Control Unit

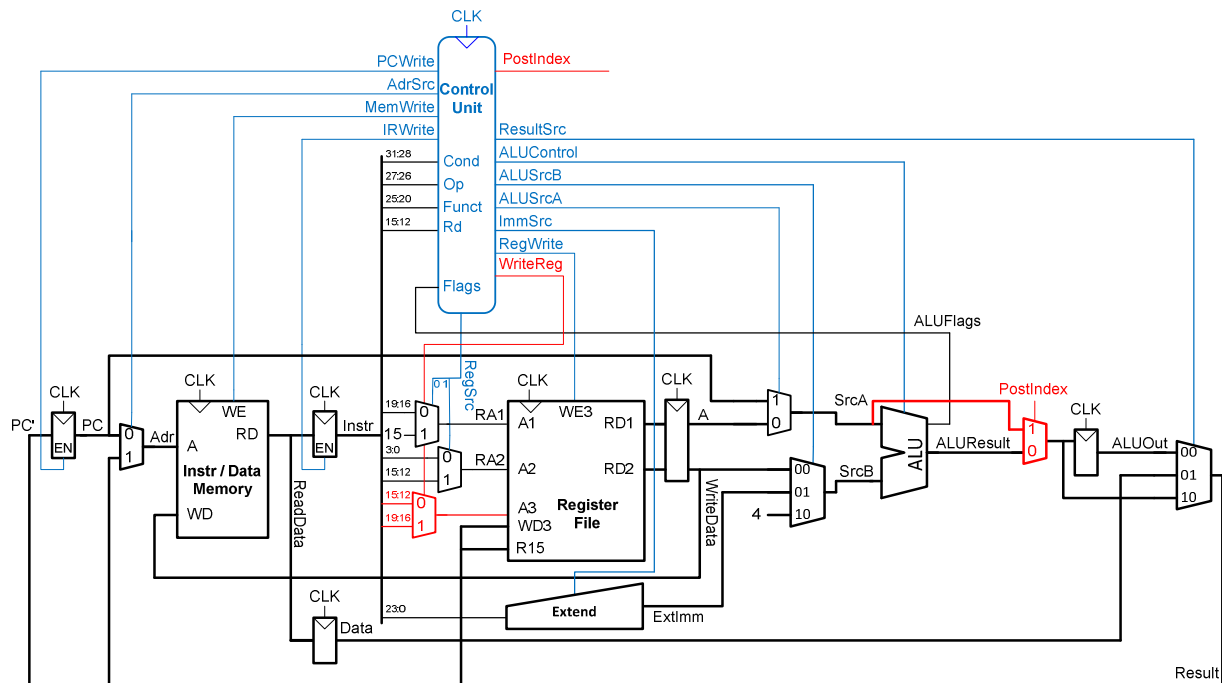


ALU Decoder truth table

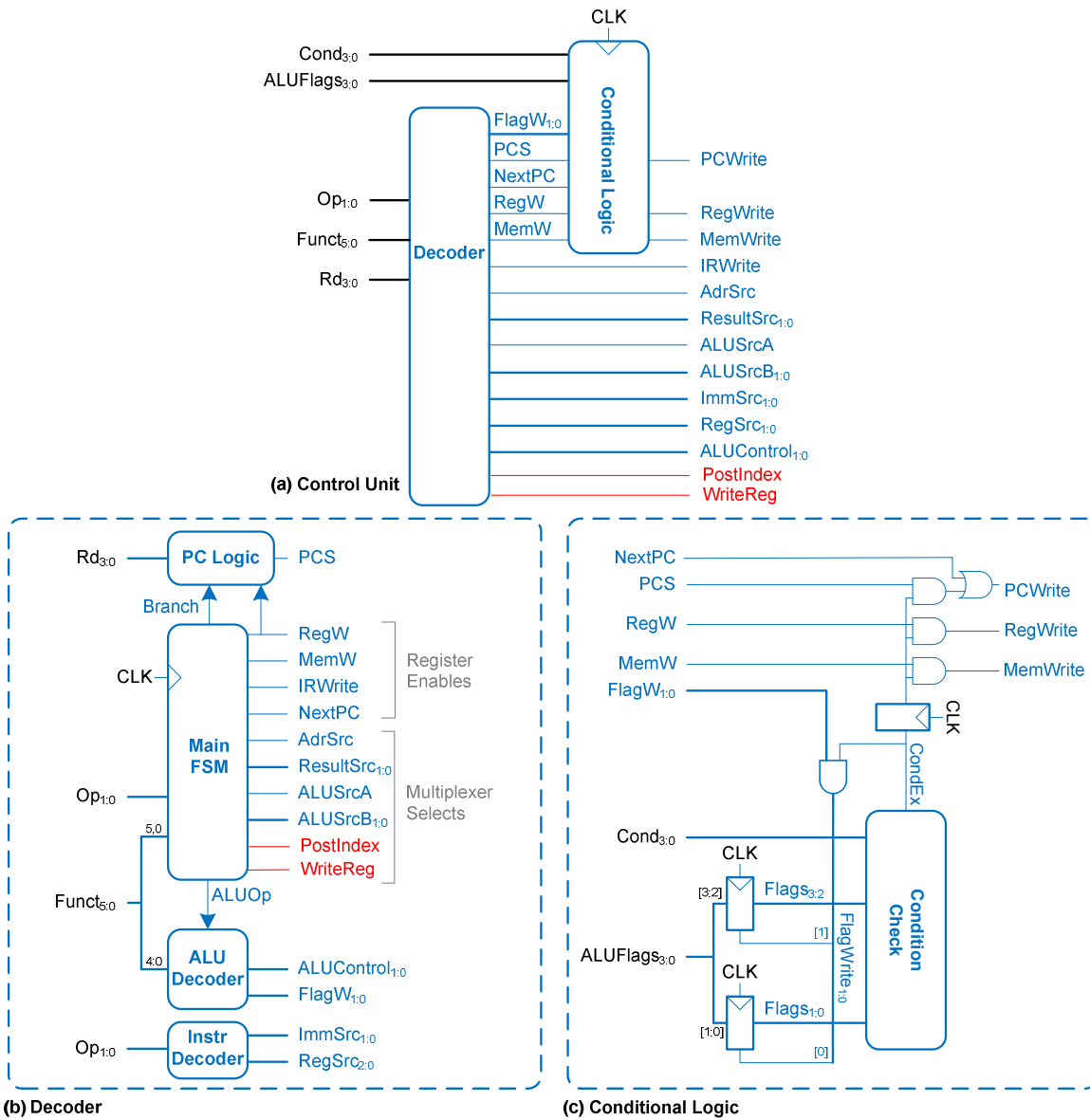
<i>ALUOp</i>	<i>Funct</i> _{4:1} (<i>cmd</i>)	<i>Funct</i> ₀ (<i>S</i>)	Notes	<i>ALUControl</i> _{2:0}	<i>FlagW</i> _{1:0}
0	X	X	Not DP	000	00
1	0100	0	ADD	000	00
		1			11
	0010	0	SUB	101	00
		1			11
	0000	0	AND	010	00
		1			10
	1100	0	ORR	011	00
		1			10
	1110	0	BIC	110	00
		1		110	10

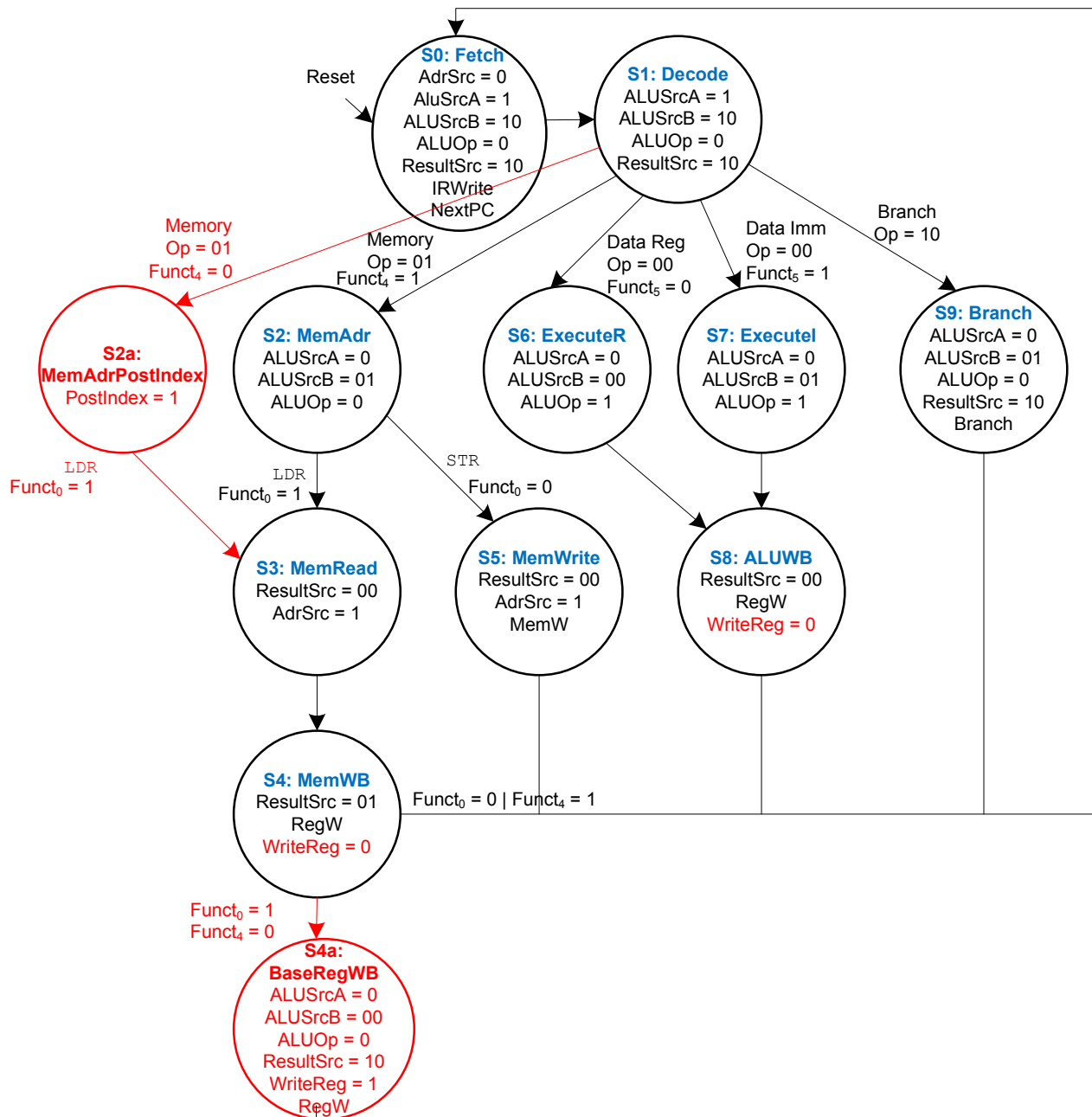
Exercise 7.15

Yes, it is possible to add this instruction without modifying the register file. First we show the modifications to the datapath.



Because two different registers will be written (first Rd with the loaded value, then Rn with $Rn + Src2$), the select signal for the A3 multiplexer (WriteReg) must be an output of the FSM. Here are the control unit schematic and the Main FSM state transition diagram.





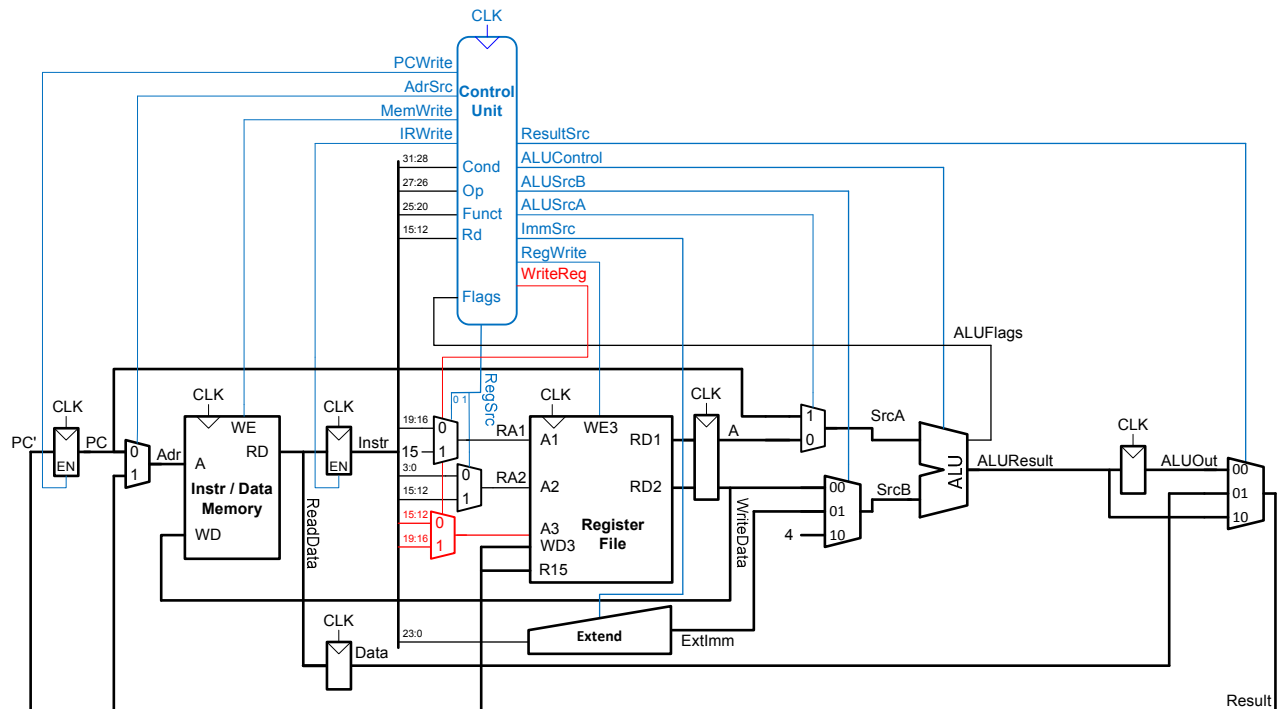
State	Datapath μ Op
Fetch	Instr \leftarrow Mem[PC]; PC \leftarrow PC+4
Decode	ALUOut \leftarrow PC+4
MemAdr	ALUOut \leftarrow Rn + Imm
MemAdrPostIndex	ALUOut \leftarrow Rn
MemRead	Data \leftarrow Mem[ALUOut]
MemWB	Rd \leftarrow Data
BaseRegWB	Rn \leftarrow Rn + Rm
MemWrite	Mem[ALUOut] \leftarrow Rd
ExecuteR	ALUOut \leftarrow Rn op Rm
Executel	ALUOut \leftarrow Rn op Imm
ALUWB	Rd \leftarrow ALUOut
Branch	PC \leftarrow R15 + offset

Now we modify the **Instr Decoder** logic for RegSrc_{1:0} and ImmSrc_{1:0} (similar to Table 7.6 in the text).

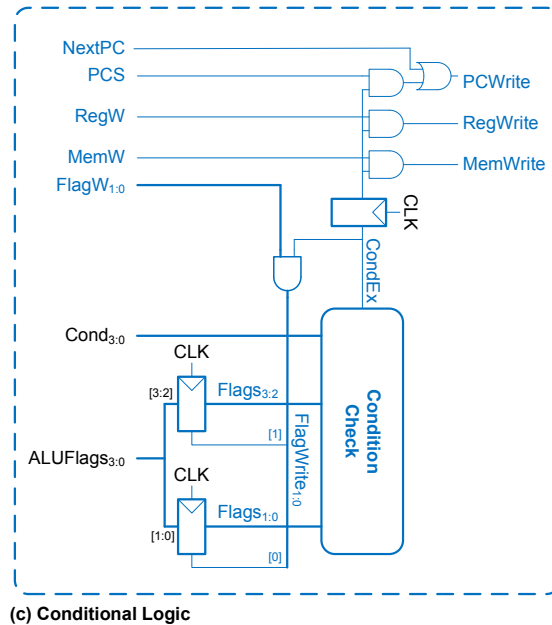
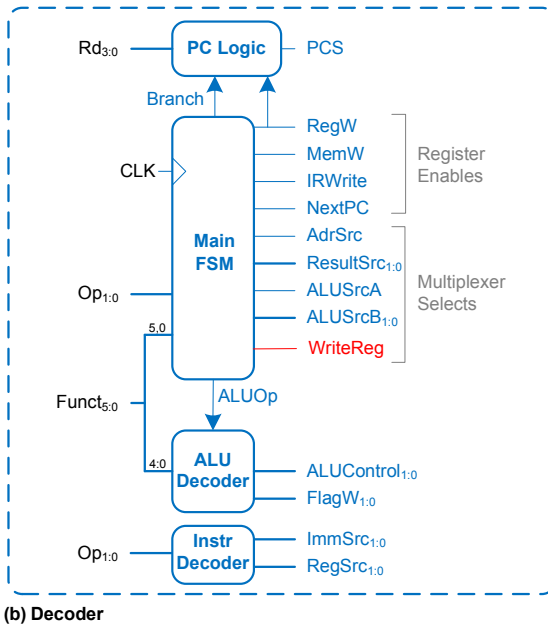
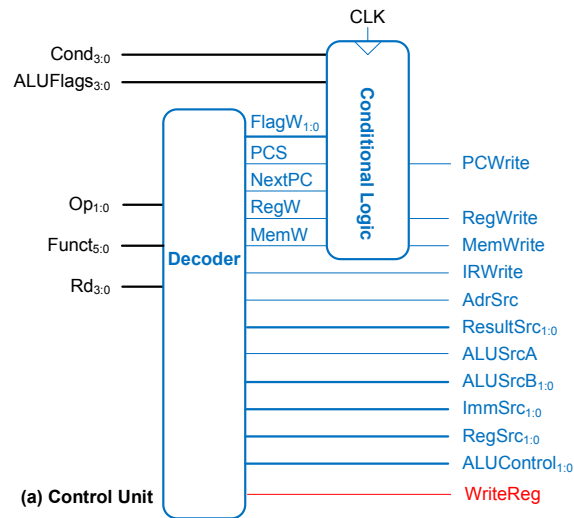
Instruction	Op	Funct _{5:0}	RegSrc _{1:0}	ImmSrc _{1:0}
LDR (offset indexing, imm offset)	01	011001	X0	01
LDR (post-indexing, reg offset)	01	1010X1	00	XX
STR	01	XXXXXX	10	01
DP imm	00	1XXXXX	X0	00
DP reg	00	0XXXXX	00	00
B	10	XXXXXX	X1	10

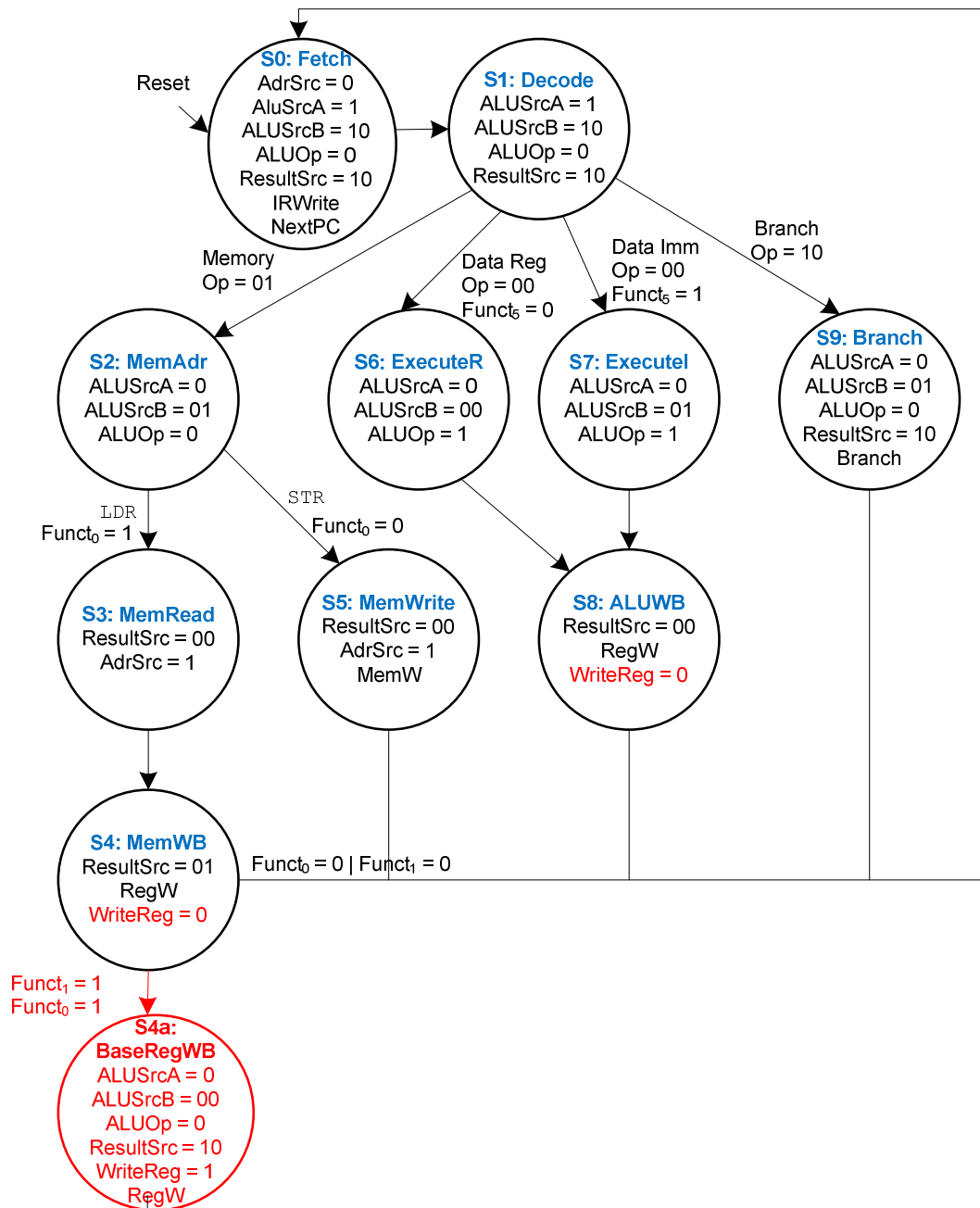
Exercise 7.16

Yes, it is possible to add this instruction without modifying the register file. First we show the modifications to the datapath.



Because two different registers will be written (first Rd with the loaded value, then Rn with Rn + Src2), the select signal for the A3 multiplexer (WriteReg) must be an output of the FSM. Here are the control unit schematic and the Main FSM state transition diagram.





State	Datapath μOp
Fetch	Instr ← Mem[PC]; PC ← PC+4
Decode	ALUOut ← PC+4
MemAdr	ALUOut ← Rn + Imm
MemRead	Data ← Mem[ALUOut]
MemWB	Rd ← Data
BaseRegWB	Rn ← Rn + Rm
MemWrite	Mem[ALUOut] ← Rd
ExecuteR	ALUOut ← Rn op Rm
Executel	ALUOut ← Rn op Imm
ALUWB	Rd ← ALUOut
Branch	PC ← R15 + offset

Now we modify the **Instr Decoder** logic for RegSrc_{1:0} and ImmSrc_{1:0} (similar to Table 7.6 in the text).

Instruction	Op	Funct _{5:0}	RegSrc _{1:0}	ImmSrc _{1:0}
LDR (offset indexing, imm offset)	01	011001	X0	01
LDR (pre-indexing, reg offset)	01	111011	00	XX
STR	01	XXXXXX	10	01
DP imm	00	1XXXXX	X0	00
DP reg	00	0XXXXX	00	00
B	10	XXXXXX	X1	10

Exercise 7.17

From Equation 7.4, $T_{c2} = t_{pcq} + 2t_{mux} + \max[t_{ALU} + t_{mux}, t_{mem}] + t_{setup}$

She should choose to decrease the delay of the memory.

$$t_{mem} = (200/2) \text{ ps} = 100 \text{ ps}$$

With this new memory delay, the ALU is on the critical path instead of the memory.

$$\begin{aligned} T_{c2} &= [40 + 2(25) + \max[120 + 25, 100] + 50] \text{ ps} \\ &= [40 + 2(25) + 145 + 50] \text{ ps} \\ &= \mathbf{285 \text{ ps}} \end{aligned}$$

Exercise 7.18

The ALU is not on the critical path, so decreasing its delay does not affect performance. Thus, the results are the same as Example 7.6

$$T_{c2} = \mathbf{340 \text{ ps}}$$

$$T_2 = (100 \times 10^9 \text{ instructions})(4.12 \text{ cycles/instruction}) (340 \times 10^{-12} \text{ s/cycle}) = \mathbf{140 \text{ seconds}}$$

Exercise 7.19

She should choose the memory. The new delay should be 145 ps. Making it less than that does not improve performance.

$$t_{mem} = \mathbf{15 \text{ ps}}$$

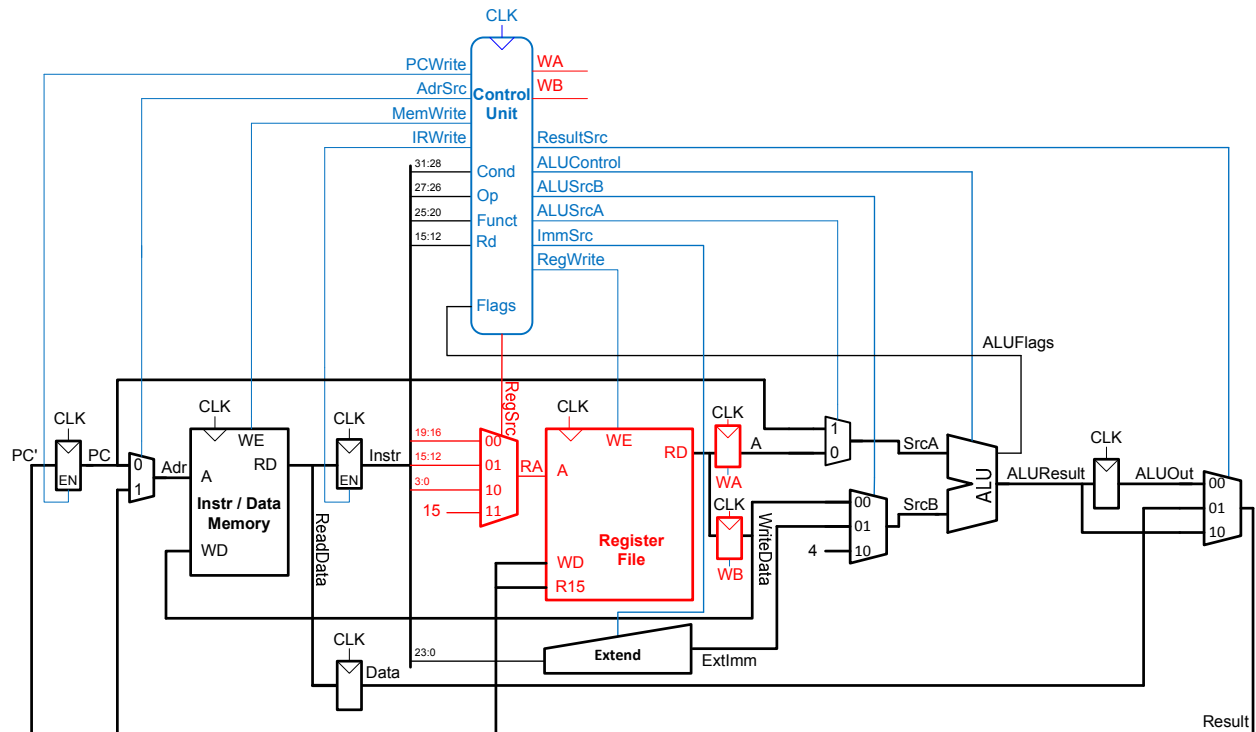
With this new memory delay, the ALU is on the critical path instead of the memory.

$$\begin{aligned} T_{c2} &= [40 + 2(25) + \max[120 + 25, 145] + 50] \text{ ps} \\ &= [40 + 2(25) + 145 + 50] \text{ ps} \end{aligned}$$

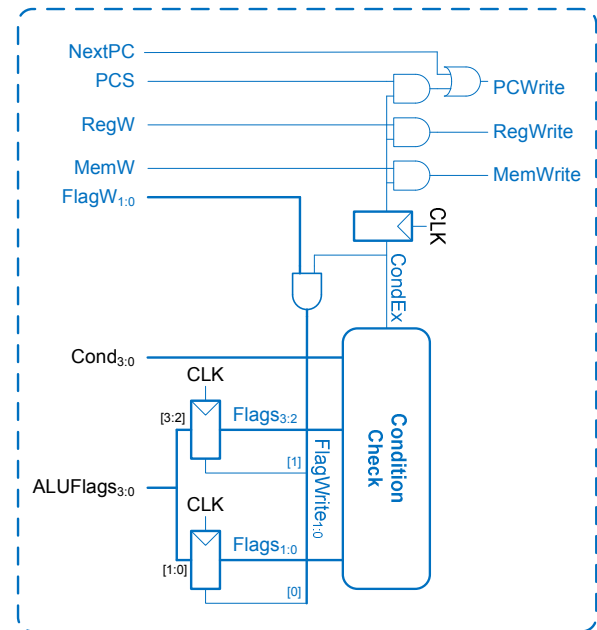
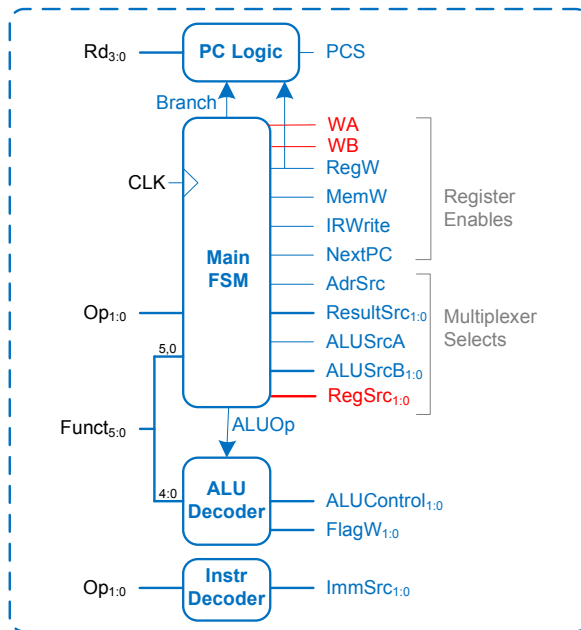
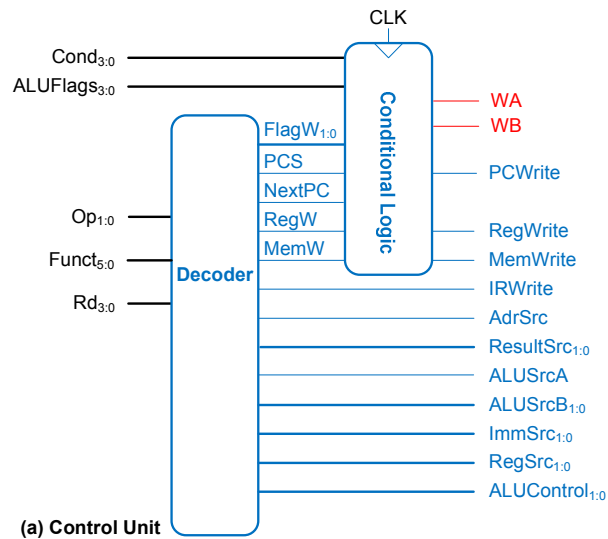
= 285 ps

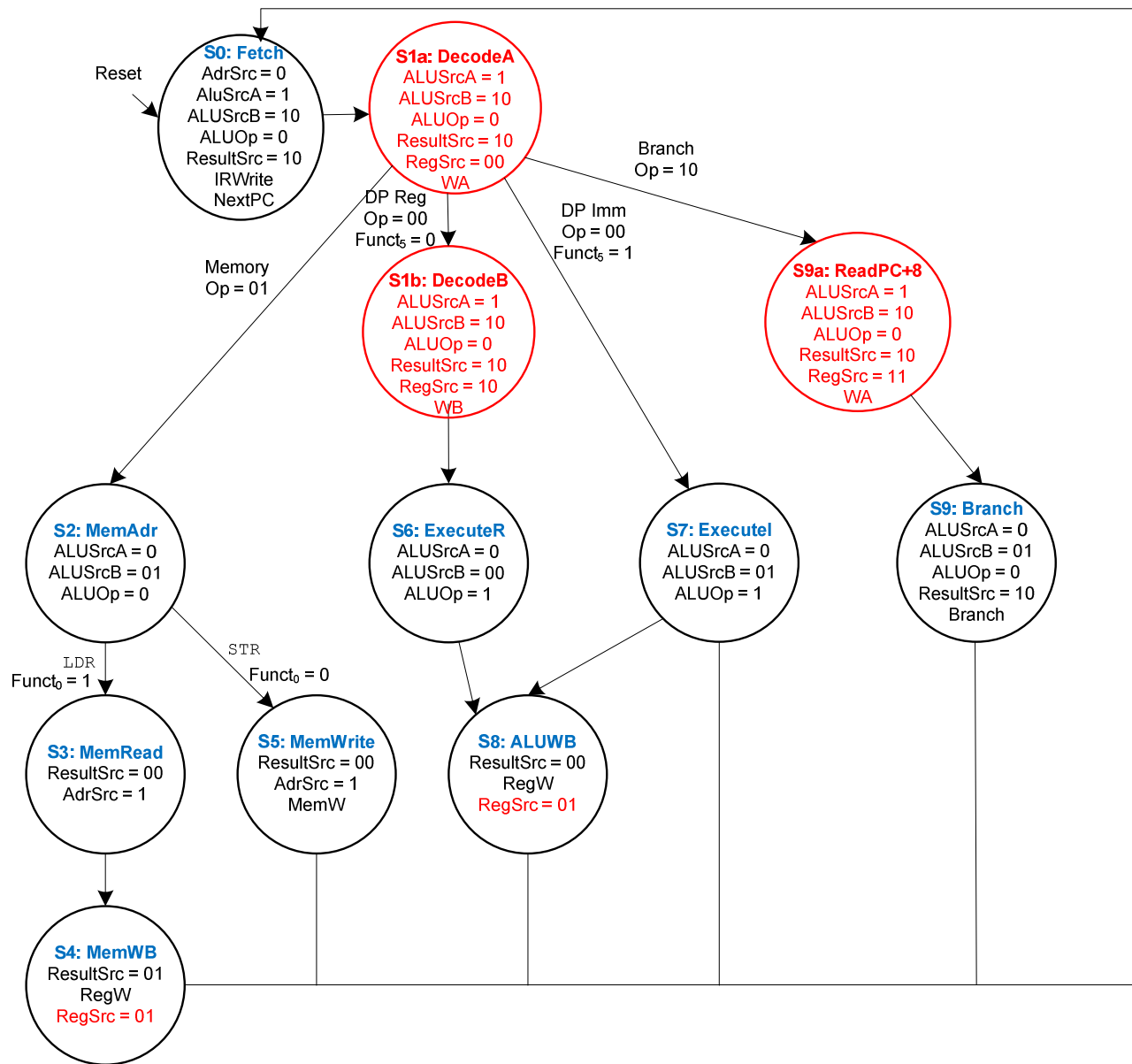
Exercise 7.20

Datapath



Control





State	Datapath μ Op
Fetch	Instr \leftarrow Mem[PC]; PC \leftarrow PC+4
DecodeA	Read SrcA (Rn) from RF ALUOut \leftarrow PC+4
DecodeB	Read SrcB (Rm) from RF ALUOut \leftarrow PC+4
MemAdr	ALUOut \leftarrow Rn + Imm
MemRead	Data \leftarrow Mem[ALUOut]
MemWB	Rd \leftarrow Data
MemWrite	Mem[ALUOut] \leftarrow Rd
ExecuteR	ALUOut \leftarrow Rn op Rm
ExecuteI	ALUOut \leftarrow Rn op Imm
ALUWB	Rd \leftarrow ALUOut
ReadPC+8	Read R15 (PC+8) into A register
Branch	PC \leftarrow R15 + offset

Exercise 7.21

Yes, Alyssa should switch to the slower but lower power register file for her multicycle processor design.

Doubling the delay of the register file does not put it on the critical path. The setup time constraint affected by the register file delay (i.e., between the instruction register and the A and B registers) is:

$$\begin{aligned} T_c &= t_{pcq} + t_{mux} + t_{RFread} + t_{setup} \\ &= (40 + 25 + 200 + 50) \text{ ps} = \mathbf{315 \text{ ps}} \end{aligned}$$

This is still less than the 340 ps of the critical path (see Example 7.6), so increasing the delay of the register file does not affect the cycle time.

Exercise 7.22

The CPI is not affected by changes in component delays, so it is the same as was calculated in Example 7.5: CPI = **4.12**.

Exercise 7.23

The program executes 2 data-processing instructions before the loop. It executes the entire loop 5 times and then executes the `CMP` and `BEQ` only on the sixth iteration, for a total of: 2 DP instructions + 5 (2 DP + 2 Branch) + (1 DP + 1 B) = 13 DP + 11 B. Each data-processing instruction takes 4 cycles and each branch instruction takes 3 cycles, so the total number of cycles required to execute the program is:

$$13(4) + 11(3) = \mathbf{85 \text{ cycles}}$$

Exercise 7.24

The program executes 3 data-processing instructions before the loop. It executes the entire loop 10 times and then executes the `CMP` and `BEQ` only on the eleventh iteration, for a total of: 3 DP instructions + 10 (3 DP + 2 Branch) + (1 DP + 1 B) = 34 DP + 21 B. Each data-processing instruction takes 4 cycles and each branch instruction takes 3 cycles, so the total number of cycles required to execute the program is:

$$34(4) + 21(3) = \mathbf{199 \text{ cycles}}$$

Exercise 7.25

SystemVerilog

```
// ARM 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 === 7) 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] WriteData, Adr,
           output logic        MemWrite);

    logic [31:0] ReadData;

    // instantiate processor and shared memory
    arm arm(clk, reset, MemWrite, Adr,
            WriteData, ReadData);
    mem mem(clk, MemWrite, Adr, WriteData, ReadData);
endmodule
```



```

endmodule

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

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

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

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

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

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

    logic [31:0] Instr;
    logic [3:0] ALUFlags;
    logic PCWrite, RegWrite, IRWrite;
    logic AdrSrc, ALUSrcA;
    logic [1:0] RegSrc, ALUSrcB, ImmSrc, ALUControl, ResultSrc;

    controller c(clk, reset, Instr[31:12], ALUFlags,
                 PCWrite, MemWrite, RegWrite, IRWrite,
                 AdrSrc, RegSrc, ALUSrcA, ALUSrcB, ResultSrc,
                 ImmSrc, ALUControl);
    datapath dp(clk, reset, Adr, WriteData, ReadData, Instr, ALUFlags,
                PCWrite, RegWrite, IRWrite,
                AdrSrc, RegSrc, ALUSrcA, ALUSrcB, ResultSrc,
                ImmSrc, ALUControl);
endmodule

module controller(input logic clk,
                  input logic reset,
                  input logic [31:12] Instr,
                  input logic [3:0] ALUFlags,
                  output logic PCWrite,
                  output logic MemWrite,
                  output logic RegWrite,
                  output logic IRWrite,
                  output logic AdrSrc,
                  output logic [1:0] RegSrc,
                  output logic ALUSrcA,
                  output logic [1:0] ALUSrcB,
                  output logic [1:0] ResultSrc,
                  output logic [1:0] ImmSrc,
                  output logic [1:0] ALUControl);

```

```

logic [1:0] FlagW;
logic      PCS, NextPC, RegW, MemW;

decoder dec(clk, reset, Instr[27:26], Instr[25:20], Instr[15:12],
            FlagW, PCS, NextPC, RegW, MemW,
            IRWrite, AdrSrc, ResultSrc,
            ALUSrcA, ALUSrcB, ImmSrc, RegSrc, ALUControl);
condlogic cl(clk, reset, Instr[31:28], ALUFlags,
            FlagW, PCS, NextPC, RegW, MemW,
            PCWrite, RegWrite, MemWrite);
endmodule

module decoder(input logic      clk, reset,
               input logic [1:0] Op,
               input logic [5:0] Funct,
               input logic [3:0] Rd,
               output logic [1:0] FlagW,
               output logic      PCS, NextPC, RegW, MemW,
               output logic      IRWrite, AdrSrc,
               output logic [1:0] ResultSrc,
               output logic      ALUSrcA,
               output logic [1:0] ALUSrcB, ImmSrc, RegSrc, ALUControl);

logic      Branch, ALUOp;

// Main FSM
mainfsm fsm(clk, reset, Op, Funct,
            IRWrite, AdrSrc,
            ALUSrcA, ALUSrcB, ResultSrc,
            NextPC, RegW, MemW, Branch, ALUOp);

always_comb
  if (ALUOp) begin // which Data-processing Instr?
    case(Funct[4:1])
      4'b0100: ALUControl = 2'b00; // ADD
      4'b0010: ALUControl = 2'b01; // SUB
      4'b0000: ALUControl = 2'b10; // AND
      4'b1100: ALUControl = 2'b11; // ORR
      default: ALUControl = 2'bx; // unimplemented
    endcase
    FlagW[1] = Funct[0]; // update N & Z flags if S bit is set
    FlagW[0] = Funct[0] & (ALUControl == 2'b00 | ALUControl ==
2'b01);
  end else begin
    ALUControl = 2'b00; // add for non data-processing instructions
    FlagW      = 2'b00; // don't update Flags
  end

// PC Logic
assign PCS = ((Rd == 4'b1111) & RegW) | Branch;

// Instr Decoder
assign ImmSrc = Op;
assign RegSrc[0] = (Op == 2'b10); // read PC on Branch

```

```

    assign RegSrc[1] = (Op == 2'b01); // read Rd on STR
endmodule

module mainfsm(input  logic      clk,
               input  logic      reset,
               input  logic [1:0] Op,
               input  logic [5:0] Funct,
               output logic      IRWrite,
               output logic      AdrSrc, ALUSrcA,
               output logic [1:0] ALUSrcB, ResultSrc,
               output logic      NextPC, RegW, MemW, Branch, ALUOp);

    typedef enum logic [3:0] {FETCH, DECODE, MEMADR, MEMRD, MEMWB,
                              MEMWR, EXECUTER, EXECUTEI, ALUWB, BRANCH,
                              UNKNOWN}
    statetype;

    statetype state, nextstate;
    logic [11:0] 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: case(Op)
                2'b00:
                    if (Funct[5]) nextstate = EXECUTEI;
                    else          nextstate = EXECUTER;
                2'b01:            nextstate = MEMADR;
                2'b10:            nextstate = BRANCH;
                default:          nextstate = UNKNOWN;
            endcase
            EXECUTER:              nextstate = ALUWB;
            EXECUTEI:              nextstate = ALUWB;
            MEMADR:
                if (Funct[0])      nextstate = MEMRD;
                else               nextstate = MEMWR;
            MEMRD:                 nextstate = MEMWB;
            default:               nextstate = FETCH;
        endcase

    // state-dependent output logic
    always_comb
        case(state)
            FETCH:    controls = 12'b10001_010_1100;
            DECODE:   controls = 12'b00000_010_1100;
            EXECUTER: controls = 12'b00000_000_0001;
            EXECUTEI: controls = 12'b00000_000_0011;
            ALUWB:    controls = 12'b00010_000_0000;
        endcase

```

```

MEMADR:    controls = 12'b00000_000_0010;
MEMWR:     controls = 12'b00100_100_0000;
MEMRD:     controls = 12'b00000_100_0000;
MEMWB:     controls = 12'b00010_001_0000;
BRANCH:    controls = 12'b01000_010_0010;
default:   controls = 12'bxxxxx_xxx_xxxx;
endcase

assign {NextPC, Branch, MemW, RegW, IRWrite,
      AdrSrc, ResultSrc,
      ALUSrcA, ALUSrcB, ALUOp} = controls;
endmodule

module condlogic(input  logic      clk, reset,
                 input  logic [3:0] Cond,
                 input  logic [3:0] ALUFlags,
                 input  logic [1:0] FlagW,
                 input  logic      PCS, NextPC, RegW, MemW,
                 output logic      PCWrite, RegWrite, MemWrite);

  logic [1:0] FlagWrite;
  logic [3:0] Flags;
  logic      CondEx, CondExDelayed;

  flopenr #(2)flagreg1(clk, reset, FlagWrite[1], ALUFlags[3:2],
Flags[3:2]);
  flopenr #(2)flagreg0(clk, reset, FlagWrite[0], ALUFlags[1:0],
Flags[1:0]);

  // write controls are conditional
  condcheck cc(Cond, Flags, CondEx);
  flopr #(1)condreg(clk, reset, CondEx, CondExDelayed);
  assign FlagWrite = FlagW & {2{CondEx}};
  assign RegWrite  = RegW  & CondExDelayed;
  assign MemWrite  = MemW  & CondExDelayed;
  assign PCWrite   = (PCS  & CondExDelayed) | NextPC;
endmodule

module condcheck(input  logic [3:0] Cond,
                 input  logic [3:0] Flags,
                 output logic      CondEx);

  logic neg, zero, carry, overflow, ge;

  assign {neg, zero, carry, overflow} = Flags;
  assign ge = (neg == overflow);

  always_comb
  case(Cond)
    4'b0000: CondEx = zero;           // EQ
    4'b0001: CondEx = ~zero;         // NE
    4'b0010: CondEx = carry;         // CS
    4'b0011: CondEx = ~carry;        // CC
    4'b0100: CondEx = neg;           // MI

```

```

4'b0101: CondEx = ~neg;                // PL
4'b0110: CondEx = overflow;            // VS
4'b0111: CondEx = ~overflow;           // VC
4'b1000: CondEx = carry & ~zero;       // HI
4'b1001: CondEx = ~(carry & ~zero);    // LS
4'b1010: CondEx = ge;                  // GE
4'b1011: CondEx = ~ge;                 // LT
4'b1100: CondEx = ~zero & ge;          // GT
4'b1101: CondEx = ~(~zero & ge);       // LE
4'b1110: CondEx = 1'b1;               // Always
default: CondEx = 1'bx;                // undefined
endcase
endmodule

module datapath(input  logic      clk, reset,
                output logic [31:0] Adr, WriteData,
                input  logic [31:0] ReadData,
                output logic [31:0] Instr,
                output logic [3:0]  ALUFlags,
                input  logic      PCWrite, RegWrite,
                input  logic      IRWrite,
                input  logic      AdrSrc,
                input  logic [1:0] RegSrc,
                input  logic      ALUSrcA,
                input  logic [1:0] ALUSrcB, ResultSrc,
                input  logic [1:0] ImmSrc, ALUControl);

logic [31:0] PCNext, PC;
logic [31:0] ExtImm, SrcA, SrcB, Result;
logic [31:0] Data, RD1, RD2, A, ALUResult, ALUOut;
logic [3:0]  RA1, RA2;

// next PC logic
flopnr #(32) pcreg(clk, reset, PCWrite, Result, PC);

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

// register file logic
mux2 #(4) ralmux(Instr[19:16], 4'b1111, RegSrc[0], RA1);
mux2 #(4) ra2mux(Instr[3:0], Instr[15:12], RegSrc[1], RA2);
regfile rf(clk, RegWrite, RA1, RA2,
            Instr[15:12], Result, Result,
            RD1, RD2);
flopnr #(32) srcareg(clk, reset, RD1, A);
flopnr #(32) wdreg(clk, reset, RD2, WriteData);
extend      ext(Instr[23:0], ImmSrc, ExtImm);

// ALU logic
mux2 #(32) srcamux(A, PC, ALUSrcA, SrcA);
mux3 #(32) srcbmux(WriteData, ExtImm, 32'd4, ALUSrcB, SrcB);
alu       alu(SrcA, SrcB, ALUControl, ALUResult, ALUFlags);

```

```

    flopr #(32) aluoutreg(clk, reset, ALUResult, ALUOut);
    mux3 #(32) resmux(ALUOut, Data, ALUResult, ResultSrc, Result);
endmodule

```

```

module regfile(input logic clk,
               input logic we3,
               input logic [3:0] ra1, ra2, wa3,
               input logic [31:0] wd3, r15,
               output logic [31:0] rd1, rd2);

```

```

    logic [31:0] rf[14:0];

```

```

    // three ported register file
    // read two ports combinationaly
    // write third port on rising edge of clock
    // register 15 reads PC+8 instead

```

```

    always_ff @(posedge clk)
        if (we3) rf[wa3] <= wd3;

```

```

    assign rd1 = (ra1 == 4'b1111) ? r15 : rf[ra1];
    assign rd2 = (ra2 == 4'b1111) ? r15 : rf[ra2];
endmodule

```

```

module extend(input logic [23:0] Instr,
              input logic [1:0] ImmSrc,
              output logic [31:0] ExtImm);

```

```

    always_comb
        case(ImmSrc)
            // 8-bit unsigned immediate
            2'b00: ExtImm = {24'b0, Instr[7:0]};
            // 12-bit unsigned immediate
            2'b01: ExtImm = {20'b0, Instr[11:0]};
            // 24-bit two's complement shifted branch
            2'b10: ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00};
            default: ExtImm = 32'bx; // undefined
        endcase
endmodule

```

```

module adder #(parameter WIDTH=8)
    (input logic [WIDTH-1:0] a, b,
     output logic [WIDTH-1:0] y);

```

```

    assign y = a + b;
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 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 alu(input  logic [31:0] a, b,
           input  logic [1:0]  ALUControl,
           output logic [31:0] Result,
           output logic [3:0]  ALUFlags);

    logic          neg, zero, carry, overflow;
    logic [31:0]   condinvb;
    logic [32:0]   sum;

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

    always_comb
        casex (ALUControl[1:0])
            2'b0?: Result = sum;
            2'b10: Result = a & b;
            2'b11: Result = a | b;
        endcase

    assign neg      = Result[31];
    assign zero     = (Result == 32'b0);
    assign carry    = (ALUControl[1] == 1'b0) & sum[32];
    assign overflow = (ALUControl[1] == 1'b0) & ~(a[31] ^ b[31] ^
                                                ALUControl[0]) & (a[31] ^ sum[31]);
    assign ALUFlags = {neg, zero, carry, overflow};

```

```
endmodule
```

VHDL

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

architecture test of testbench is
  component top
    port (clk, reset:          in  STD_LOGIC;
          WriteData, Adr:      out STD_LOGIC_VECTOR(31 downto 0);
          MemWrite:           out STD_LOGIC);
  end component;
  signal WriteData, DataAdr:    STD_LOGIC_VECTOR(31 downto 0);
  signal clk, reset,  MemWrite: STD_LOGIC;
begin

  -- instantiate device to be tested
  dut: top port map (clk, reset, WriteData, DataAdr, MemWrite);

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

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

  -- check that 7 gets written to address 84
  -- at end of program
  process (clk) begin
    if (clk'event and clk = '0' and MemWrite = '1') then
      if (to_integer(DataAdr) = 100 and
          to_integer(WriteData) = 7) then
        report "NO ERRORS: Simulation succeeded" severity failure;
      elsif (DataAdr /= 96) then
        report "Simulation failed" severity failure;
      end if;
    end if;
  end process;
end;

library IEEE;
```



```

use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity top is -- top-level design for testing
  port(clk, reset:      in      STD_LOGIC;
        WriteData, Adr:  buffer STD_LOGIC_VECTOR(31 downto 0);
        MemWrite:       buffer STD_LOGIC);
end;

```

```

architecture test of top is
  component arm
    port(clk, reset:      in      STD_LOGIC;
          MemWrite:      out     STD_LOGIC;
          Adr, WriteData: out     STD_LOGIC_VECTOR(31 downto 0);
          ReadData:      in      STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component mem
    port(clk, we:  in  STD_LOGIC;
          a, wd:   in  STD_LOGIC_VECTOR(31 downto 0);
          rd:      out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  signal ReadData: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- instantiate processor and memories
  i_arm: arm port map(clk, reset, MemWrite, Adr,
                     WriteData, ReadData);
  i_mem: mem port map(clk, MemWrite, Adr,
                     WriteData, ReadData);
end;

```

```

library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity mem is -- memory
  port(clk, we:  in  STD_LOGIC;
        a, wd:   in  STD_LOGIC_VECTOR(31 downto 0);
        rd:      out STD_LOGIC_VECTOR(31 downto 0));
end;

```

```

architecture behave of mem is -- instruction and data memory
begin
  process is
    file mem_file: TEXT;
    variable L: line;
    variable ch: character;
    variable i, index, result: integer;

    type ramtype is array (63 downto 0) of
      STD_LOGIC_VECTOR(31 downto 0);
    variable ram: ramtype;
  begin
    -- initialize memory from file
    for i in 0 to 63 loop -- set all contents low
      ram(i) := (others => '0');
    end loop;
    index := 0;
  end process;
end;

```

```

FILE_OPEN(mem_file, "memfile.dat", READ_MODE);
while not endfile(mem_file) loop
  readline(mem_file, L);
  result := 0;
  for i in 1 to 8 loop
    read(L, ch);
    if '0' <= ch and ch <= '9' then
      result := character'pos(ch) - character'pos('0');
    elsif 'a' <= ch and ch <= 'f' then
      result := character'pos(ch) - character'pos('a')+10;
    elsif 'A' <= ch and ch <= 'F' then
      result := character'pos(ch) - character'pos('A')+10;
    else report "Format error on line " & integer'image(index)
      severity error;
    end if;
    ram(index)(35-i*4 downto 32-i*4) :=
      to_std_logic_vector(result,4);
  end loop;
  index := index + 1;
end loop;

-- read or write memory
loop
  if clk'event and clk = '1' then
    if (we = '1') then
      ram(to_integer(a(7 downto 2))) := wd;
    end if;
  end if;
  rd <= ram(to_integer(a(7 downto 2)));
  wait on clk, a;
end loop;
end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity arm is -- multicycle processor
  port(clk, reset:      in  STD_LOGIC;
        MemWrite:       out STD_LOGIC;
        Adr, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
        ReadData:       in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of arm is
  component controller
    port(clk, reset:      in  STD_LOGIC;
          Instr:          in  STD_LOGIC_VECTOR(31 downto 12);
          ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
          PCWrite:        out STD_LOGIC;
          MemWrite:       out STD_LOGIC;
          RegWrite:       out STD_LOGIC;
          IRWrite:        out STD_LOGIC;
          AdrSrc:         out STD_LOGIC);
  end component;

```

```

        RegSrc:          out STD_LOGIC_VECTOR(1 downto 0);
        ALUSrcA:         out STD_LOGIC;
        ALUSrcB:         out STD_LOGIC_VECTOR(1 downto 0);
        ResultSrc:       out STD_LOGIC_VECTOR(1 downto 0);
        ImmSrc:          out STD_LOGIC_VECTOR(1 downto 0);
        ALUControl:      out STD_LOGIC_VECTOR(1 downto 0));
end component;
component datapath
  port (clk, reset:      in  STD_LOGIC;
        Adr:            out STD_LOGIC_VECTOR(31 downto 0);
        WriteData:      out STD_LOGIC_VECTOR(31 downto 0);
        ReadData:       in  STD_LOGIC_VECTOR(31 downto 0);
        Instr:          out STD_LOGIC_VECTOR(31 downto 0);
        ALUFlags:       out STD_LOGIC_VECTOR(3 downto 0);
        PCWrite:        in  STD_LOGIC;
        RegWrite:       in  STD_LOGIC;
        IRWrite:        in  STD_LOGIC;
        AdrSrc:         in  STD_LOGIC;
        RegSrc:         in  STD_LOGIC_VECTOR(1 downto 0);
        ALUSrcA:        in  STD_LOGIC;
        ALUSrcB:        in  STD_LOGIC_VECTOR(1 downto 0);
        ResultSrc:      in  STD_LOGIC_VECTOR(1 downto 0);
        ImmSrc:         in  STD_LOGIC_VECTOR(1 downto 0);
        ALUControl:     in  STD_LOGIC_VECTOR(1 downto 0));
end component;
signal Instr: STD_LOGIC_VECTOR(31 downto 0);
signal ALUFlags: STD_LOGIC_VECTOR(3 downto 0);
signal PCWrite, RegWrite, IRWrite: STD_LOGIC;
signal AdrSrc, ALUSrcA: STD_LOGIC;
signal RegSrc, ALUSrcB: STD_LOGIC_VECTOR(1 downto 0);
signal ImmSrc, ALUControl, ResultSrc: STD_LOGIC_VECTOR(1 downto 0);

begin
  cont: controller port map(clk, reset, Instr(31 downto 12),
                           ALUFlags, PCWrite, MemWrite, RegWrite,
                           IRWrite, AdrSrc, RegSrc, ALUSrcA,
                           ALUSrcB, ResultSrc, ImmSrc, ALUControl);

  dp: datapath port map(clk, reset, Adr, WriteData, ReadData,
                       Instr, ALUFlags,
                       PCWrite, RegWrite, IRWrite,
                       AdrSrc, RegSrc, ALUSrcA, ALUSrcB, ResultSrc,
                       ImmSrc, ALUControl);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- single cycle control decoder
  port (clk, reset:      in  STD_LOGIC;
        Instr:          in  STD_LOGIC_VECTOR(31 downto 12);
        ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
        PCWrite:        out STD_LOGIC;
        MemWrite:       out STD_LOGIC;
        RegWrite:       out STD_LOGIC;

```

```

        IRWrite:          out STD_LOGIC;
        AdrSrc:           out STD_LOGIC;
        RegSrc:           out STD_LOGIC_VECTOR(1 downto 0);
        ALUSrcA:          out STD_LOGIC;
        ALUSrcB:          out STD_LOGIC_VECTOR(1 downto 0);
        ResultSrc:        out STD_LOGIC_VECTOR(1 downto 0);
        ImmSrc:           out STD_LOGIC_VECTOR(1 downto 0);
        ALUControl:       out STD_LOGIC_VECTOR(1 downto 0));
end;
architecture struct of controller is
    component decoder
        port (clk, reset:      in  STD_LOGIC;
              Op:              in  STD_LOGIC_VECTOR(1 downto 0);
              Funct:           in  STD_LOGIC_VECTOR(5 downto 0);
              Rd:              in  STD_LOGIC_VECTOR(3 downto 0);
              FlagW:           out STD_LOGIC_VECTOR(1 downto 0);
              PCS, NextPC:     out STD_LOGIC;
              RegW, MemW:      out STD_LOGIC;
              IRWrite, AdrSrc: out STD_LOGIC;
              ResultSrc:       out STD_LOGIC_VECTOR(1 downto 0);
              ALUSrcA:         out STD_LOGIC;
              ALUSrcB, ImmSrc: out STD_LOGIC_VECTOR(1 downto 0);
              RegSrc:          out STD_LOGIC_VECTOR(1 downto 0);
              ALUControl:      out STD_LOGIC_VECTOR(1 downto 0));
    end component;
    component condlogic
        port (clk, reset:      in  STD_LOGIC;
              Cond:            in  STD_LOGIC_VECTOR(3 downto 0);
              ALUFlags:        in  STD_LOGIC_VECTOR(3 downto 0);
              FlagW:           in  STD_LOGIC_VECTOR(1 downto 0);
              PCS, NextPC:     in  STD_LOGIC;
              RegW, MemW:      in  STD_LOGIC;
              PCWrite, RegWrite: out STD_LOGIC;
              MemWrite:        out STD_LOGIC);
    end component;
    signal FlagW: STD_LOGIC_VECTOR(1 downto 0);
    signal PCS, NextPC, RegW, MemW: STD_LOGIC;
begin
    dec: decoder port map (clk, reset, Instr(27 downto 26), Instr(25 downto
20),
                        Instr(15 downto 12), FlagW, PCS,
                        NextPC, RegW, MemW,
                        IRWrite, AdrSrc, ResultSrc,
                        ALUSrcA, ALUSrcB, ImmSrc, RegSrc, ALUControl);
    cl: condlogic port map (clk, reset, Instr(31 downto 28),
                        ALUFlags, FlagW, PCS, NextPC, RegW, MemW,
                        PCWrite, RegWrite, MemWrite);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity decoder is -- main control decoder
    port (clk, reset:      in  STD_LOGIC;
          Op:              in  STD_LOGIC_VECTOR(1 downto 0);
          Funct:           in  STD_LOGIC_VECTOR(5 downto 0);

```

```

    Rd:                in  STD_LOGIC_VECTOR(3 downto 0);
    FlagW:              out STD_LOGIC_VECTOR(1 downto 0);
    PCS, NextPC:        out STD_LOGIC;
    RegW, MemW:          out STD_LOGIC;
    IRWrite, AdrSrc:     out STD_LOGIC;
    ResultSrc:           out STD_LOGIC_VECTOR(1 downto 0);
    ALUSrcA:             out STD_LOGIC;
    ALUSrcB, ImmSrc:     out STD_LOGIC_VECTOR(1 downto 0);
    RegSrc:              out STD_LOGIC_VECTOR(1 downto 0);
    ALUControl:          out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture behave of decoder is
    component mainfsm
        port (clk, reset:      in  STD_LOGIC;
              Op:              in  STD_LOGIC_VECTOR(1 downto 0);
              Funct:           in  STD_LOGIC_VECTOR(5 downto 0);
              IRWrite:         out STD_LOGIC;
              AdrSrc, ALUSrcA: out STD_LOGIC;
              ALUSrcB:         out STD_LOGIC_VECTOR(1 downto 0);
              ResultSrc:       out STD_LOGIC_VECTOR(1 downto 0);
              NextPC, RegW:     out STD_LOGIC;
              MemW, Branch:     out STD_LOGIC;
              ALUOp:           out STD_LOGIC);
    end component;
    signal Branch, ALUOp: STD_LOGIC;
begin
    -- Main FSM
    fsm: mainfsm port map (clk, reset, Op, Funct,
                          IRWrite, AdrSrc,
                          ALUSrcA, ALUSrcB, ResultSrc,
                          NextPC, RegW, MemW, Branch, ALUOp);

    process (all) begin -- ALU Decoder
        if (ALUOp) then
            case Funct(4 downto 1) is
                when "0100" => ALUControl <= "00"; -- ADD
                when "0010" => ALUControl <= "01"; -- SUB
                when "0000" => ALUControl <= "10"; -- AND
                when "1100" => ALUControl <= "11"; -- ORR
                when others => ALUControl <= "--"; -- unimplemented
            end case;
            FlagW(1) <= Funct(0);
            FlagW(0) <= Funct(0) and (not ALUControl(1));
        else
            ALUControl <= "00";
            FlagW <= "00";
        end if;
    end process;

    -- PC Logic
    PCS <= ((and Rd) and RegW) or Branch;

    -- Instr Decoder

```

```

    ImmSrc <= Op;
    RegSrc(0) <= '1' when (Op = 2B"10") else '0';
    RegSrc(1) <= '1' when (Op = 2B"01") else '0';
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mainfsm is
    port(clk, reset:      in  STD_LOGIC;
          Op:             in  STD_LOGIC_VECTOR(1 downto 0);
          Funct:          in  STD_LOGIC_VECTOR(5 downto 0);
          IRWrite:        out STD_LOGIC;
          AdrSrc, ALUSrcA: out STD_LOGIC;
          ALUSrcB:        out STD_LOGIC_VECTOR(1 downto 0);
          ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
          NextPC, RegW:    out STD_LOGIC;
          MemW, Branch:    out STD_LOGIC;
          ALUOp:          out STD_LOGIC);
end;

architecture synth of mainfsm is
    type statetype is (FETCH, DECODE, MEMADR, MEMRD, MEMWB, MEMWR,
                      EXECUTER, EXECUTEI, ALUWB, BR, UNKNOWN);
    signal state, nextstate: statetype;
    signal controls: STD_LOGIC_VECTOR(11 downto 0);
begin
    --state register
    process(clk, reset) begin
        if reset then state <= FETCH;
        elsif rising_edge(clk) then
            state <= nextstate;
        end if;
    end process;

    -- next state logic
    process(all) begin
        case state is
            when FETCH =>          nextstate <= DECODE;
            when DECODE =>
                case Op is
                    when "00" =>      nextstate <= ExecuteI when (Funct(5) = '1')
                                     else EXECUTER;
                    when "01" =>      nextstate <= MEMADR;
                    when "10" =>      nextstate <= BR;
                    when others =>    nextstate <= UNKNOWN;
                end case;
            when EXECUTER =>        nextstate <= ALUWB;
            when EXECUTEI =>        nextstate <= ALUWB;
            when MEMADR  =>        nextstate <= MEMRD when (Funct(0) = '1')
                                     else MEMWR;
            when MEMRD   =>        nextstate <= MEMWB;
            when others  =>        nextstate <= FETCH;
        end case;
    end process;
end synth;

```

```

-- state-dependent output logic
process(all) begin
  case state is
    when FETCH =>   controls <= 12B"100010101100";
    when DECODE =>  controls <= 12B"000000101100";
    when EXECUTER => controls <= 12B"000000000001";
    when EXECUTEI => controls <= 12B"000000000011";
    when ALUWB =>   controls <= 12B"000100000000";
    when MEMADR =>  controls <= 12B"000000000010";
    when MEMWR =>   controls <= 12B"001001000000";
    when MEMRD =>   controls <= 12B"000001000000";
    when MEMWB =>   controls <= 12B"000100010000";
    when BR =>      controls <= 12B"010000100010";
    when others =>  controls <= "XXXXXXXXXXXX";
  end case;
end process;

(NextPC, Branch, MemW, RegW, IRWrite,
 AdrSrc, ResultSrc,
 ALUSrcA, ALUSrcB, ALUOp) <= controls;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condlogic is -- Conditional logic
  port(clk, reset:      in  STD_LOGIC;
        Cond:           in  STD_LOGIC_VECTOR(3 downto 0);
        ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
        FlagW:          in  STD_LOGIC_VECTOR(1 downto 0);
        PCS, NextPC:    in  STD_LOGIC;
        RegW, MemW:     in  STD_LOGIC;
        PCWrite, RegWrite: out STD_LOGIC;
        MemWrite:       out STD_LOGIC);
end;

architecture behave of condlogic is
  component condcheck
    port(Cond:           in  STD_LOGIC_VECTOR(3 downto 0);
          Flags:         in  STD_LOGIC_VECTOR(3 downto 0);
          CondEx:        out STD_LOGIC);
  end component;
  component flopenr generic(width: integer);
    port(clk, reset, en: in  STD_LOGIC;
          d:           in  STD_LOGIC_VECTOR(width-1 downto 0);
          q:           out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component flopr generic(width: integer);
    port(clk, reset: in  STD_LOGIC;
          d:         in  STD_LOGIC_VECTOR(width-1 downto 0);
          q:         out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;

  signal FlagWrite: STD_LOGIC_VECTOR(1 downto 0);
  signal Flags:     STD_LOGIC_VECTOR(3 downto 0);

```

```

    signal CondEx:          STD_LOGIC_VECTOR(0 downto 0);
    signal CondExDelayed: STD_LOGIC_VECTOR(0 downto 0);
begin
    flagreg1: flopenr generic map(2)
        port map(clk, reset, FlagWrite(1),
            ALUFlags(3 downto 2), Flags(3 downto 2));
    flagreg0: flopenr generic map(2)
        port map(clk, reset, FlagWrite(0),
            ALUFlags(1 downto 0), Flags(1 downto 0));
    cc: condcheck port map(Cond, Flags, CondEx(0));
    condreg: flopr generic map(1)
        port map(clk, reset, CondEx, CondExDelayed);

    FlagWrite <= FlagW and (CondEx(0), CondEx(0));
    RegWrite  <= RegW  and CondExDelayed(0);
    MemWrite  <= MemW  and CondExDelayed(0);
    PCWrite   <= (PCS  and CondExDelayed(0)) or NextPC;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condcheck is
    port(Cond:          in  STD_LOGIC_VECTOR(3 downto 0);
          Flags:        in  STD_LOGIC_VECTOR(3 downto 0);
          CondEx:        out STD_LOGIC);
end;

architecture behave of condcheck is
    signal neg, zero, carry, overflow, ge: STD_LOGIC;
begin
    (neg, zero, carry, overflow) <= Flags;
    ge <= (neg xnor overflow);

    process(all) begin -- Condition checking
        case Cond is
            when "0000" => CondEx <= zero;
            when "0001" => CondEx <= not zero;
            when "0010" => CondEx <= carry;
            when "0011" => CondEx <= not carry;
            when "0100" => CondEx <= neg;
            when "0101" => CondEx <= not neg;
            when "0110" => CondEx <= overflow;
            when "0111" => CondEx <= not overflow;
            when "1000" => CondEx <= carry and (not zero);
            when "1001" => CondEx <= not(carry and (not zero));
            when "1010" => CondEx <= ge;
            when "1011" => CondEx <= not ge;
            when "1100" => CondEx <= (not zero) and ge;
            when "1101" => CondEx <= not ((not zero) and ge);
            when "1110" => CondEx <= '1';
            when others => CondEx <= '-';
        end case;
    end process;
end;

```



```

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity datapath is
  port (clk, reset:      in  STD_LOGIC;
        Adr:            out STD_LOGIC_VECTOR(31 downto 0);
        WriteData:      out STD_LOGIC_VECTOR(31 downto 0);
        ReadData:       in  STD_LOGIC_VECTOR(31 downto 0);
        Instr:          out STD_LOGIC_VECTOR(31 downto 0);
        ALUFlags:       out STD_LOGIC_VECTOR(3 downto 0);
        PCWrite:        in  STD_LOGIC;
        RegWrite:       in  STD_LOGIC;
        IRWrite:        in  STD_LOGIC;
        AdrSrc:         in  STD_LOGIC;
        RegSrc:         in  STD_LOGIC_VECTOR(1 downto 0);
        ALUSrcA:        in  STD_LOGIC;
        ALUSrcB:        in  STD_LOGIC_VECTOR(1 downto 0);
        ResultSrc:      in  STD_LOGIC_VECTOR(1 downto 0);
        ImmSrc:         in  STD_LOGIC_VECTOR(1 downto 0);
        ALUControl:     in  STD_LOGIC_VECTOR(1 downto 0));
end;

architecture struct of datapath is
  component alu
    port (a, b:          in  STD_LOGIC_VECTOR(31 downto 0);
          ALUControl: in  STD_LOGIC_VECTOR(1 downto 0);
          Result:       buffer STD_LOGIC_VECTOR(31 downto 0);
          ALUFlags:     out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  component regfile
    port (clk:          in  STD_LOGIC;
          we3:          in  STD_LOGIC;
          ra1, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
          wd3, r15:     in  STD_LOGIC_VECTOR(31 downto 0);
          rd1, rd2:     out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component adder
    port (a, b: in  STD_LOGIC_VECTOR(31 downto 0);
          y:   out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component extend
    port (Instr: in  STD_LOGIC_VECTOR(23 downto 0);
          ImmSrc: in  STD_LOGIC_VECTOR(1 downto 0);
          ExtImm: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component flopenr generic (width: integer);
    port (clk, reset, en: in  STD_LOGIC;
          d:             in  STD_LOGIC_VECTOR(width-1 downto 0);
          q:             out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component flopr generic (width: integer);
    port (clk, reset: in  STD_LOGIC;
          d:         in  STD_LOGIC_VECTOR(width-1 downto 0);
          q:         out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;

```

```

component mux2 generic(width: integer);
  port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
        s:      in  STD_LOGIC;
        y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
component mux3 generic(width: integer);
  port(d0, d1, d2: in  STD_LOGIC_VECTOR(width-1 downto 0);
        s:      in  STD_LOGIC_VECTOR(1 downto 0);
        y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
signal PCNext, PC: STD_LOGIC_VECTOR(31 downto 0);
signal ExtImm, SrcA, SrcB: STD_LOGIC_VECTOR(31 downto 0);
signal Result: STD_LOGIC_VECTOR(31 downto 0);
signal Data, RD1, RD2, A: STD_LOGIC_VECTOR(31 downto 0);
signal ALUResult, ALUOut: STD_LOGIC_VECTOR(31 downto 0);
signal RA1, RA2: STD_LOGIC_VECTOR(3 downto 0);
begin
  -- next PC logic
  pcreg: flopenr generic map(32)
    port map(clk, reset, PCWrite, Result, PC);

  -- memory logic
  adrmux: mux2 generic map(32)
    port map(PC, ALUOut, AdrSrc, Adr);
  ir: flopenr generic map(32)
    port map(clk, reset, IRWrite, ReadData, Instr);
  datareg: flopr generic map(32)
    port map(clk, reset, ReadData, Data);

  -- register file logic
  ralmux: mux2 generic map (4)
    port map(Instr(19 downto 16), "1111", RegSrc(0), RA1);
  ra2mux: mux2 generic map (4) port map(Instr(3 downto 0),
    Instr(15 downto 12), RegSrc(1), RA2);
  rf: regfile port map(clk, RegWrite, RA1, RA2,
    Instr(15 downto 12), Result, Result,
    RD1, RD2);
  srcareg: flopr generic map(32)
    port map(clk, reset, RD1, A);
  wdreg: flopr generic map(32)
    port map(clk, reset, RD2, WriteData);
  ext: extend port map(Instr(23 downto 0), ImmSrc, ExtImm);

  -- ALU logic
  srcamux: mux2 generic map(32)
    port map(A, PC, ALUSrcA, SrcA);
  srcbmux: mux3 generic map(32)
    port map(WriteData, ExtImm, 32D"4", ALUSrcB, SrcB);
  i_alu: alu port map(SrcA, SrcB, ALUControl, ALUResult, ALUFlags);
  aluoutreg: flopr generic map(32)
    port map(clk, reset, ALUResult, ALUOut);
  resmux: mux3 generic map(32)
    port map(ALUOut, Data, ALUResult, ResultSrc, Result);
end;

```

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity regfile is -- three-port register file
    port (clk:          in  STD_LOGIC;
          we3:          in  STD_LOGIC;
          ra1, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
          wd3, r15:     in  STD_LOGIC_VECTOR(31 downto 0);
          rd1, rd2:     out STD_LOGIC_VECTOR(31 downto 0));
end;

```

```

architecture behave of regfile is
    type ramtype is array (31 downto 0) of
        STD_LOGIC_VECTOR(31 downto 0);
    signal mem: ramtype;
begin
    process(clk) begin
        if rising_edge(clk) then
            if we3 = '1' then mem(to_integer(wa3)) <= wd3;
            end if;
        end if;
    end process;
    process(all) begin
        if (to_integer(ra1) = 15) then rd1 <= r15;
        else rd1 <= mem(to_integer(ra1));
        end if;
        if (to_integer(ra2) = 15) then rd2 <= r15;
        else rd2 <= mem(to_integer(ra2));
        end if;
    end process;
end;

```

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity adder is -- adder
    port (a, b: in  STD_LOGIC_VECTOR(31 downto 0);
          y:   out STD_LOGIC_VECTOR(31 downto 0));
end;

```

```

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

```

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity extend is
    port (Instr: in  STD_LOGIC_VECTOR(23 downto 0);
          ImmSrc: in  STD_LOGIC_VECTOR(1 downto 0);
          ExtImm: out STD_LOGIC_VECTOR(31 downto 0));
end;

```

```

architecture behave of extend is
begin
    process(all) begin
        case ImmSrc is

```

```

        when "00"    => ExtImm <= (X"000000", Instr(7 downto 0));
        when "01"    => ExtImm <= (X"00000", Instr(11 downto 0));
        when "10"    => ExtImm <= (Instr(23), Instr(23), Instr(23),
            Instr(23), Instr(23), Instr(23), Instr(23 downto 0), "00");
        when others => ExtImm <= X"-----";
    end case;
end process;
end;

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

architecture behave of mux2 is
begin
    y <= d1 when s else d0;
end;

```

```

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

```

```

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

```

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity alu is
    port(a, b:           in  STD_LOGIC_VECTOR(31 downto 0);
         ALUControl: in  STD_LOGIC_VECTOR(1 downto 0);
         Result:        buffer STD_LOGIC_VECTOR(31 downto 0);
         ALUFlags:      out STD_LOGIC_VECTOR(3 downto 0));
end;

```

```

architecture behave of alu is
    signal condinvb: STD_LOGIC_VECTOR(31 downto 0);
    signal sum:      STD_LOGIC_VECTOR(32 downto 0);
    signal neg, zero, carry, overflow: STD_LOGIC;
begin
    condinvb <= not b when ALUControl(0) else b;
    sum <= ('0', a) + ('0', condinvb) + ALUControl(0);

    process(all) begin
        case? ALUControl(1 downto 0) is
            when "0-"    => result <= sum(31 downto 0);
            when "10"    => result <= a and b;
            when "11"    => result <= a or b;
            when others => result <= (others => '-');
        end case?;
    end process;
end;

```

```

neg      <= Result(31);
zero     <= '1' when (Result = 0) else '0';
carry    <= (not ALUControl(1)) and sum(32);
overflow <= (not ALUControl(1)) and
            (not (a(31) xor b(31) xor ALUControl(0))) and
            (a(31) xor sum(31));
ALUFlags <= (neg, zero, carry, overflow);
end;

```

Test ARM assembly code

// If successful, it should write the value 7 to address 100

```

MAIN  SUB R0, R15, R15          ; R0 = 0
      ADD R2, R0, #5           ; R2 = 5
      ADD R3, R0, #12          ; R3 = 12
      SUB R7, R3, #9           ; R7 = 3
      ORR R4, R7, R2           ; R4 = 3 OR 5 = 7
      AND R5, R3, R4           ; R5 = 12 AND 7 = 4
      ADD R5, R5, R4           ; R5 = 4 + 7 = 11
      SUBS R8, R5, R7           ; R8 <= 11 - 3 = 8, set Flags
      BEQ END                  ; shouldn't be taken
      SUBS R8, R3, R4           ; R8 = 12 - 7 = 5
      BGE AROUND               ; should be taken
      ADD R5, R0, #0           ; should be skipped
AROUND
      SUBS R8, R7, R2           ; R8 = 3 - 5 = -2, set Flags
      ADDLT R7, R5, #1         ; R7 = 11 + 1 = 12
      SUB R7, R7, R2           ; R7 = 12 - 5 = 7
      STR R7, [R3, #84]        ; mem[12+84] = 7
      LDR R2, [R0, #96]        ; R2 = mem[96] = 7
      ADD R15, R15, R0         ; PC <- PC + 8 (skips next)
      ADD R2, R0, #14          ; shouldn't happen
      B END                    ; always taken
      ADD R2, R0, #13          ; shouldn't happen
      ADD R2, R0, #10          ; shouldn't happen
END    STR R2, [R0, #100]      ; mem[100] = 7

```

memfile.dat

```

E04F000F
E2802005
E280300C
E2437009
E1874002
E0035004
E0855004
E0558007
0A00000C
E0538004
AA000000
E2805000
E0578002
B2857001

```

```

E0477002
E5837054
E5902060
E08FF000
E280200E
EA000001
E280200D
E280200A
E5802064

```

Exercise 7.26

SystemVerilog

```

// Added instructions:
// BL, LDR, LDRB, BIC

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 == 208 & WriteData == 57) begin
                    $display("Simulation succeeded");
                    $stop;
                end else if (DataAdr != 200) begin
                    $display("Simulation failed");
                    $stop;
                end
            end
        end
endmodule

```

```

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

    logic [31:0] ReadData;

    // instantiate processor and shared memory
    arm arm(clk, reset, MemWrite, Adr,
            WriteData, ReadData);
    mem mem(clk, MemWrite, Adr, WriteData, ReadData);
endmodule

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

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

    initial
        $readmemh("ex7.26_memfile.dat",RAM);

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

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

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

    logic [31:0] Instr;
    logic [3:0] ALUFlags;
    logic PCWrite, RegWrite, IRWrite;
    logic AdrSrc, ALUSrcA;
    logic [1:0] ALUSrcB, ImmSrc, ResultSrc;
    logic [2:0] ALUControl; // BIC
    logic [2:0] RegSrc; // BL
    logic LDRB; // LDRB

    controller c(clk, reset, Instr[31:12], ALUFlags,
                 PCWrite, MemWrite, RegWrite, IRWrite,
                 AdrSrc, RegSrc, ALUSrcA, ALUSrcB, ResultSrc,
                 ImmSrc, ALUControl, LDRB);
    datapath dp(clk, reset, Adr, WriteData, ReadData, Instr, ALUFlags,
                PCWrite, RegWrite, IRWrite,
                AdrSrc, RegSrc, ALUSrcA, ALUSrcB, ResultSrc,
                ImmSrc, ALUControl, LDRB);
endmodule

module controller(input logic clk,

```



```

        input  logic      reset,
        input  logic [31:12] Instr,
        input  logic [3:0]  ALUFlags,
        output logic      PCWrite,
        output logic      MemWrite,
        output logic      RegWrite,
        output logic      IRWrite,
        output logic      AdrSrc,
        output logic [2:0]  RegSrc,          // BL
        output logic      ALUSrcA,
        output logic [1:0]  ALUSrcB,
        output logic [1:0]  ResultSrc,
        output logic [1:0]  ImmSrc,
        output logic [2:0]  ALUControl,    // BIC
        output logic      LDRB);          // LDRB
logic [1:0] FlagW;
logic      PCS, NextPC, RegW, MemW;

decoder dec(clk, reset, Instr[27:26], Instr[25:20], Instr[15:12],
            FlagW, PCS, NextPC, RegW, MemW,
            IRWrite, AdrSrc, ResultSrc,
            ALUSrcA, ALUSrcB, ImmSrc, RegSrc, ALUControl,
            LDRB); // LDRB
condlogic cl(clk, reset, Instr[31:28], ALUFlags,
            FlagW, PCS, NextPC, RegW, MemW,
            PCWrite, RegWrite, MemWrite);
endmodule

module decoder(input  logic      clk, reset,
               input  logic [1:0] Op,
               input  logic [5:0] Funct,
               input  logic [3:0] Rd,
               output logic [1:0] FlagW,
               output logic      PCS, NextPC, RegW, MemW,
               output logic      IRWrite, AdrSrc,
               output logic [1:0] ResultSrc,
               output logic      ALUSrcA,
               output logic [1:0] ALUSrcB, ImmSrc,
               output logic [2:0] RegSrc,          // BL
               output logic [2:0] ALUControl,      // BIC
               output logic      LDRB);          // LDRB

logic      Branch;
logic [1:0] ALUOp; // LDR (with +- imm12)

// Main FSM
mainfsm fsm(clk, reset, Op, Funct,
            IRWrite, AdrSrc,
            ALUSrcA, ALUSrcB, ResultSrc,
            NextPC, RegW, MemW, Branch, ALUOp,
            LDRB); // LDRB

always_comb
    case (ALUOp)

```

```

2'b00:                                // not DP: add
begin
    ALUControl = 3'b000; // add
    FlagW = 2'b00;      // don't update Flags
end
2'b01:                                // not DP: subtract
begin
    ALUControl = 3'b101; // subtract
    FlagW = 2'b00;      // don't update Flags
end

2'b10:                                // which Data-processing Instr?
begin
    case(Funct[4:1])
        4'b0100: ALUControl = 3'b000; // ADD
        4'b0010: ALUControl = 3'b101; // SUB
        4'b0000: ALUControl = 3'b010; // AND
        4'b1100: ALUControl = 3'b011; // ORR
        4'b1110: ALUControl = 3'b110; // BIC
        default: ALUControl = 3'bx;   // unimplemented
    endcase
    FlagW[1]      = Funct[0]; // update N & Z flags if S bit is set
    FlagW[0]      = Funct[0] &
                    (ALUControl == 3'b000 | ALUControl == 3'b101);
end
default:
begin
    ALUControl = 3'bx;
    FlagW = 2'bx;
end
endcase

// PC Logic
assign PCS = ((Rd == 4'b1111) & RegW) | Branch;

// Instr Decoder
assign ImmSrc = Op;
assign RegSrc[0] = (Op == 2'b10); // read PC on Branch
assign RegSrc[1] = (Op == 2'b01); // read Rd on STR
// write PC+4 to LR on BL
assign RegSrc[2] = ((Op == 2'b10) & (Funct[4]==1));
endmodule

module mainfsm(input  logic      clk,
               input  logic      reset,
               input  logic [1:0] Op,
               input  logic [5:0] Funct,
               output logic      IRWrite,
               output logic      AdrSrc, ALUSrcA,
               output logic [1:0] ALUSrcB, ResultSrc,
               output logic      NextPC, RegW, MemW, Branch,
               output logic [1:0] ALUOp,    // LDR (with +- imm12)
               output logic      LDRB);    // LDRB

```

```

typedef enum logic [3:0] {FETCH, DECODE, MEMRD, MEMWB,
                          MEMWR, EXECUTER, EXECUTEI, ALUWB, BRANCH,
                          BL, // BL
                          MEMADRADD, MEMADRSUB, // LDR +- imm12
                          MEMREADBYTE, // LDRB
                          UNKNOWN}

statetype;

statetype state, nextstate;
logic [13:0] controls; // LDRB, LDR +- imm12

// 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: case(Op)
            2'b00:
                if (Funct[5]) nextstate = EXECUTEI;
                else nextstate = EXECUTER;
            2'b01:
                if (Funct[3]) nextstate = MEMADRADD; //LDR +- imm12
                else nextstate = MEMADRSUB;
            2'b10:
                if (Funct[4]) nextstate = BL; //BL
                else nextstate = BRANCH;
            default:
                nextstate = UNKNOWN;
        endcase
        EXECUTER: nextstate = ALUWB;
        EXECUTEI: nextstate = ALUWB;
        MEMADRADD: nextstate = MEMREADBYTE; //LDR +- imm12
        if (Funct[0]&Funct[2])
        else if (Funct[0]&~Funct[2]) nextstate = MEMRD;
        else nextstate = MEMWR;
        MEMADRSUB: nextstate = MEMREADBYTE; //LDR +- imm12
        if (Funct[0]&Funct[2])
        else if (Funct[0]&~Funct[2]) nextstate = MEMRD;
        else nextstate = MEMWR;
        MEMRD: nextstate = MEMWB;
        MEMREADBYTE: nextstate = MEMWB;
        default: nextstate = FETCH;
    endcase

// state-dependent output logic
always_comb
    case(state)
        FETCH: controls = 14'b10001_010_11000_0;
        DECODE: controls = 14'b00000_010_11000_0;
        EXECUTER: controls = 14'b00000_000_00010_0;
    endcase

```

```

EXECUTEI:      controls = 14'b00000_000_00110_0;
ALUWB:         controls = 14'b00010_000_00000_0;
MEMWR:         controls = 14'b00100_100_00000_0;
MEMRD:         controls = 14'b00000_100_00000_0;
MEMWB:         controls = 14'b00010_001_00000_0;
BRANCH:        controls = 14'b01000_010_00100_0;
BL:            controls = 14'b01010_010_00100_0; // BL
MEMADRADD:      controls = 14'b00000_000_00100_0; // LDR +- imm12
MEMADRSUB:      controls = 14'b00000_000_00101_0; // LDR +- imm12
MEMREADBYTE:    controls = 14'b00000_100_00000_1; // LDRB
default:        controls = 14'bxxxxx_xxx_xxxxx_x;
endcase

assign {NextPC, Branch, MemW, RegW, IRWrite,
      AdrSrc, ResultSrc,
      ALUSrcA, ALUSrcB, ALUOp,
      LDRB} = controls;
endmodule

module condlogic(input  logic      clk, reset,
                 input  logic [3:0] Cond,
                 input  logic [3:0] ALUFlags,
                 input  logic [1:0] FlagW,
                 input  logic      PCS, NextPC, RegW, MemW,
                 output logic      PCWrite, RegWrite, MemWrite);

  logic [1:0] FlagWrite;
  logic [3:0] Flags;
  logic      CondEx, CondExDelayed;

  flopenr #(2)flagreg1(clk, reset, FlagWrite[1], ALUFlags[3:2],
Flags[3:2]);
  flopenr #(2)flagreg0(clk, reset, FlagWrite[0], ALUFlags[1:0],
Flags[1:0]);

  // write controls are conditional
  condcheck cc(Cond, Flags, CondEx);
  flopr #(1)condreg(clk, reset, CondEx, CondExDelayed);
  assign FlagWrite = FlagW & {2{CondEx}};
  assign RegWrite  = RegW  & CondExDelayed;
  assign MemWrite  = MemW  & CondExDelayed;
  assign PCWrite   = (PCS  & CondExDelayed) | NextPC;
endmodule

module condcheck(input  logic [3:0] Cond,
                 input  logic [3:0] Flags,
                 output logic      CondEx);

  logic neg, zero, carry, overflow, ge;

  assign {neg, zero, carry, overflow} = Flags;
  assign ge = (neg == overflow);

  always_comb

```

```

case(Cond)
  4'b0000: CondEx = zero;           // EQ
  4'b0001: CondEx = ~zero;          // NE
  4'b0010: CondEx = carry;           // CS
  4'b0011: CondEx = ~carry;          // CC
  4'b0100: CondEx = neg;             // MI
  4'b0101: CondEx = ~neg;            // PL
  4'b0110: CondEx = overflow;        // VS
  4'b0111: CondEx = ~overflow;       // VC
  4'b1000: CondEx = carry & ~zero;    // HI
  4'b1001: CondEx = ~(carry & ~zero); // LS
  4'b1010: CondEx = ge;              // GE
  4'b1011: CondEx = ~ge;            // LT
  4'b1100: CondEx = ~zero & ge;      // GT
  4'b1101: CondEx = ~(~zero & ge);  // LE
  4'b1110: CondEx = 1'b1;           // Always
  default: CondEx = 1'bx;           // undefined
endcase
endmodule

module datapath(input  logic      clk, reset,
                output logic [31:0] Adr, WriteData,
                input  logic [31:0] ReadData,
                output logic [31:0] Instr,
                output logic [3:0]  ALUFlags,
                input  logic      PCWrite, RegWrite,
                input  logic      IRWrite,
                input  logic      AdrSrc,
                input  logic [2:0] RegSrc,      // BL
                input  logic      ALUSrcA,
                input  logic [1:0] ALUSrcB, ResultSrc,
                input  logic [1:0] ImmSrc,
                input  logic [2:0] ALUControl, // BIC
                input  logic      LDRB);      // LDRB

  logic [31:0] PCNext, PC;
  logic [31:0] ExtImm, SrcA, SrcB, Result;
  logic [31:0] Data, RD1, RD2, A, ALUResult, ALUOut;
  logic [3:0]  RA1, RA2;
  logic [3:0]  RA3;           // BL
  logic [31:0] WD3;           // BL
  logic [7:0]  DataByte;      // LDRB
  logic [31:0] MemData, DataByteExt; // LDRB

  // next PC logic
  flopenr #(32) pcreg(clk, reset, PCWrite, Result, PC);

  // memory logic
  mux2      #(32)  adrmux(PC, ALUOut, AdrSrc, Adr);
  flopenr #(32) ir(clk, reset, IRWrite, ReadData, Instr);
  // LDRB
  mux4      #(8)  ldrbmux(ReadData[7:0], ReadData[15:8], ReadData[23:16],
                        ReadData[31:24], Adr[1:0], DataByte);
  zeroextend ze(DataByte, DataByteExt);

```

```

mux2    #(32)    datamux(ReadData, DataByteExt, LDRB, MemData);
flopr   #(32)    datareg(clk, reset, MemData, Data);

// register file logic
mux2 #(4)    ralmux(Instr[19:16], 4'b1111, RegSrc[0], RA1);
mux2 #(4)    ra2mux(Instr[3:0], Instr[15:12], RegSrc[1], RA2);
mux2 #(4)    ra3mux(Instr[15:12], 4'b1110, RegSrc[2], RA3); // BL
mux2 #(32)    rwd3mux(Result, PC, RegSrc[2], WD3);           // BL

regfile    rf(clk, RegWrite, RA1, RA2,
               RA3, WD3, Result,
               RD1, RD2); // BL
flopr #(32) srcareg(clk, reset, RD1, A);
flopr #(32) wdreg(clk, reset, RD2, WriteData);
extend     ext(Instr[23:0], ImmSrc, ExtImm);

// ALU logic
mux2 #(32)    srcamux(A, PC, ALUSrcA, SrcA);
mux3 #(32)    srcbmux(WriteData, ExtImm, 32'd4, ALUSrcB, SrcB);
alu           alu(SrcA, SrcB, ALUControl, ALUResult, ALUFlags);
flopr #(32)    aluoutreg(clk, reset, ALUResult, ALUOut);
mux3 #(32)    resmux(ALUOut, Data, ALUResult, ResultSrc, Result);
endmodule

module regfile(input  logic      clk,
               input  logic      we3,
               input  logic [3:0] ra1, ra2, wa3,
               input  logic [31:0] wd3, r15,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[14:0];

  // three ported register file
  // read two ports combinationaly
  // write third port on rising edge of clock
  // register 15 reads PC+8 instead

  always_ff @(posedge clk)
    if (we3) rf[wa3] <= wd3;

  assign rd1 = (ra1 == 4'b1111) ? r15 : rf[ra1];
  assign rd2 = (ra2 == 4'b1111) ? r15 : rf[ra2];
endmodule

module extend(input  logic [23:0] Instr,
              input  logic [1:0] ImmSrc,
              output logic [31:0] ExtImm);

  always_comb
    case(ImmSrc)
      // 8-bit unsigned immediate
      2'b00: ExtImm = {24'b0, Instr[7:0]};
      // 12-bit unsigned immediate
      2'b01: ExtImm = {20'b0, Instr[11:0]};
    endcase
endmodule

```

```

                // 24-bit two's complement shifted branch
2'b10:  ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00};
default: ExtImm = 32'bx; // undefined
endcase
endmodule

module adder #(parameter WIDTH=8)
    (input  logic [WIDTH-1:0] a, b,
     output logic [WIDTH-1:0] y);

    assign y = a + b;
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 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 alu(input  logic [31:0] a, b,
           input  logic [2:0]  ALUControl, // BIC
           output logic [31:0] Result,
           output logic [3:0]  ALUFlags);

```

```

logic      neg, zero, carry, overflow;
logic [31:0] condinvb;
logic [32:0] sum;

assign condinvb = ALUControl[2] ? ~b : b;
assign sum = a + condinvb + ALUControl[2]; // BIC

always_comb
  casex (ALUControl[1:0])
    2'b0?: Result = sum;
    2'b10: Result = a & condinvb;           // BIC
    2'b11: Result = a | b;
  endcase

assign neg      = Result[31];
assign zero     = (Result == 32'b0);
assign carry    = (ALUControl[1] == 1'b0) & sum[32];
assign overflow = (ALUControl[1] == 1'b0) & ~(a[31] ^ b[31] ^
        ALUControl[0]) & (a[31] ^ sum[31]);
assign ALUFlags = {neg, zero, carry, overflow};
endmodule

// zeroextend needed for LDRB
module zeroextend (input  [7:0] a,
                  output [31:0] y);

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

// mux4 needed for LDRB
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 = d0;
    endcase
endmodule

```

VHDL

```

library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity testbench is
end;

architecture test of testbench is
  component top

```



```

    port(clk, reset:          in  STD_LOGIC;
          WriteData, Addr:    out STD_LOGIC_VECTOR(31 downto 0);
          MemWrite:           out STD_LOGIC);
end component;
signal WriteData, DataAdr:    STD_LOGIC_VECTOR(31 downto 0);
signal clk, reset,  MemWrite: STD_LOGIC;
begin

    -- instantiate device to be tested
    dut: top port map(clk, reset, WriteData, DataAdr, MemWrite);

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

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

    -- check that 7 gets written to address 84
    -- at end of program
    process (clk) begin
        if (clk'event and clk = '0' and MemWrite = '1') then
            if (to_integer(DataAdr) = 208 and
                to_integer(WriteData) = 57) then
                report "NO ERRORS: Simulation succeeded" severity failure;
            elsif (DataAdr /= 200) then
                report "Simulation failed" severity failure;
            end if;
        end if;
    end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity top is -- top-level design for testing
    port(clk, reset:          in  STD_LOGIC;
          WriteData, Addr:    buffer STD_LOGIC_VECTOR(31 downto 0);
          MemWrite:           buffer STD_LOGIC);
end;

architecture test of top is
    component arm
        port(clk, reset:          in  STD_LOGIC;
              MemWrite:           out STD_LOGIC);
    end component;

```

```

        Adr, WriteData:    out STD_LOGIC_VECTOR(31 downto 0);
        ReadData:         in  STD_LOGIC_VECTOR(31 downto 0));
end component;
component mem
    port(clk, we:  in  STD_LOGIC;
          a, wd:   in  STD_LOGIC_VECTOR(31 downto 0);
          rd:      out STD_LOGIC_VECTOR(31 downto 0));
end component;
signal ReadData: STD_LOGIC_VECTOR(31 downto 0);
begin
    -- instantiate processor and memories
    i_arm: arm port map(clk, reset, MemWrite, Adr,
                       WriteData, ReadData);
    i_mem: mem port map(clk, MemWrite, Adr,
                       WriteData, ReadData);
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity mem is -- memory
    port(clk, we:  in STD_LOGIC;
          a, wd:   in STD_LOGIC_VECTOR(31 downto 0);
          rd:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of mem is -- instruction and data memory
begin
    process is
        file mem_file: TEXT;
        variable L: line;
        variable ch: character;
        variable i, index, result: integer;

        type ramtype is array (63 downto 0) of
            STD_LOGIC_VECTOR(31 downto 0);
        variable ram: ramtype;
    begin
        -- initialize memory from file
        for i in 0 to 63 loop -- set all contents low
            ram(i) := (others => '0');
        end loop;
        index := 0;
        FILE_OPEN(mem_file, "ex7.26_memfile.dat", READ_MODE);
        while not endfile(mem_file) loop
            readline(mem_file, L);
            result := 0;
            for i in 1 to 8 loop
                read(L, ch);
                if '0' <= ch and ch <= '9' then
                    result := character'pos(ch) - character'pos('0');
                elsif 'a' <= ch and ch <= 'f' then
                    result := character'pos(ch) - character'pos('a')+10;
                elsif 'A' <= ch and ch <= 'F' then

```

```

        result := character'pos(ch) - character'pos('A')+10;
    else report "Format error on line " & integer'image(index)
        severity error;
    end if;
    ram(index)(35-i*4 downto 32-i*4) :=
        to_std_logic_vector(result,4);
    end loop;
    index := index + 1;
end loop;

-- read or write memory
loop
    if clk'event and clk = '1' then
        if (we = '1') then
            ram(to_integer(a(7 downto 2))) := wd;
        end if;
    end if;
    rd <= ram(to_integer(a(7 downto 2)));
    wait on clk, a;
end loop;
end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity arm is -- multicycle processor
    port(clk, reset:      in  STD_LOGIC;
          MemWrite:       out STD_LOGIC;
          Adr, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
          ReadData:       in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of arm is
    component controller
        port(clk, reset:      in  STD_LOGIC;
              Instr:          in  STD_LOGIC_VECTOR(31 downto 12);
              ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
              PCWrite:        out STD_LOGIC;
              MemWrite:       out STD_LOGIC;
              RegWrite:       out STD_LOGIC;
              IRWrite:        out STD_LOGIC;
              AdrSrc:         out STD_LOGIC;
              RegSrc:         out STD_LOGIC_VECTOR(1 downto 0);
              ALUSrcA:        out STD_LOGIC;
              ALUSrcB:        out STD_LOGIC_VECTOR(1 downto 0);
              ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
              ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
              ALUControl:     out STD_LOGIC_VECTOR(2 downto 0); -- BIC
              LDRB:           out STD_LOGIC); -- LDRB
    end component;
    component datapath
        port(clk, reset:      in  STD_LOGIC;
              Adr:            out STD_LOGIC_VECTOR(31 downto 0);

```

```

        WriteData:      out STD_LOGIC_VECTOR(31 downto 0);
        ReadData:       in  STD_LOGIC_VECTOR(31 downto 0);
        Instr:          out STD_LOGIC_VECTOR(31 downto 0);
        ALUFlags:       out STD_LOGIC_VECTOR(3 downto 0);
        PCWrite:        in  STD_LOGIC;
        RegWrite:       in  STD_LOGIC;
        IRWrite:        in  STD_LOGIC;
        AdrSrc:         in  STD_LOGIC;
        RegSrc:         in  STD_LOGIC_VECTOR(1 downto 0);
        ALUSrcA:        in  STD_LOGIC;
        ALUSrcB:        in  STD_LOGIC_VECTOR(1 downto 0);
        ResultSrc:      in  STD_LOGIC_VECTOR(1 downto 0);
        ImmSrc:         in  STD_LOGIC_VECTOR(1 downto 0);
        ALUControl:     in  STD_LOGIC_VECTOR(2 downto 0); -- BIC
        LDRB:           in  STD_LOGIC); -- LDRB
    end component;
    signal Instr: STD_LOGIC_VECTOR(31 downto 0);
    signal ALUFlags: STD_LOGIC_VECTOR(3 downto 0);
    signal PCWrite, RegWrite, IRWrite: STD_LOGIC;
    signal AdrSrc, ALUSrcA: STD_LOGIC;
    signal RegSrc, ALUSrcB: STD_LOGIC_VECTOR(1 downto 0);
    signal ImmSrc, ResultSrc: STD_LOGIC_VECTOR(1 downto 0);
    signal ALUControl: STD_LOGIC_VECTOR(2 downto 0); -- BIC
    signal LDRB: STD_LOGIC; -- LDRB
begin
    cont: controller port map(clk, reset, Instr(31 downto 12),
                             ALUFlags, PCWrite, MemWrite, RegWrite,
                             IRWrite, AdrSrc, RegSrc, ALUSrcA,
                             ALUSrcB, ResultSrc, ImmSrc, ALUControl,
                             LDRB); -- LDRB

    dp: datapath port map(clk, reset, Adr, WriteData, ReadData,
                          Instr, ALUFlags,
                          PCWrite, RegWrite, IRWrite,
                          AdrSrc, RegSrc, ALUSrcA, ALUSrcB, ResultSrc,
                          ImmSrc, ALUControl,
                          LDRB); -- LDRB
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- single cycle control decoder
    port(clk, reset:      in  STD_LOGIC;
          Instr:          in  STD_LOGIC_VECTOR(31 downto 12);
          ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
          PCWrite:        out STD_LOGIC;
          MemWrite:       out STD_LOGIC;
          RegWrite:       out STD_LOGIC;
          IRWrite:        out STD_LOGIC;
          AdrSrc:         out STD_LOGIC;
          RegSrc:         out STD_LOGIC_VECTOR(1 downto 0);
          ALUSrcA:        out STD_LOGIC;
          ALUSrcB:        out STD_LOGIC_VECTOR(1 downto 0);
          ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
    );
end entity;

```

```

        ImmSrc:          out STD_LOGIC_VECTOR(1 downto 0);
        ALUControl:      out STD_LOGIC_VECTOR(2 downto 0);  -- BIC
        LDRB:            out STD_LOGIC); -- LDRB
end;
architecture struct of controller is
    component decoder
        port(clk, reset:      in  STD_LOGIC;
             Op:              in  STD_LOGIC_VECTOR(1 downto 0);
             Funct:          in  STD_LOGIC_VECTOR(5 downto 0);
             Rd:             in  STD_LOGIC_VECTOR(3 downto 0);
             FlagW:          out STD_LOGIC_VECTOR(1 downto 0);
             PCS, NextPC:    out STD_LOGIC;
             RegW, MemW:     out STD_LOGIC;
             IRWrite, AdrSrc: out STD_LOGIC;
             ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
             ALUSrcA:        out STD_LOGIC;
             ALUSrcB, ImmSrc: out STD_LOGIC_VECTOR(1 downto 0);
             RegSrc:         out STD_LOGIC_VECTOR(1 downto 0);
             ALUControl:     out STD_LOGIC_VECTOR(2 downto 0);  -- BIC
             LDRB:           out STD_LOGIC); -- LDRB
    end component;
    component condlogic
        port(clk, reset:      in  STD_LOGIC;
             Cond:           in  STD_LOGIC_VECTOR(3 downto 0);
             ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
             FlagW:         in  STD_LOGIC_VECTOR(1 downto 0);
             PCS, NextPC:    in  STD_LOGIC;
             RegW, MemW:     in  STD_LOGIC;
             PCWrite, RegWrite: out STD_LOGIC;
             MemWrite:       out STD_LOGIC);
    end component;
    signal FlagW: STD_LOGIC_VECTOR(1 downto 0);
    signal PCS, NextPC, RegW, MemW: STD_LOGIC;
begin
    dec: decoder port map(clk, reset, Instr(27 downto 26), Instr(25 downto
20),
                        Instr(15 downto 12), FlagW, PCS,
                        NextPC, RegW, MemW,
                        IRWrite, AdrSrc, ResultSrc,
                        ALUSrcA, ALUSrcB, ImmSrc, RegSrc, ALUControl,
                        LDRB); -- LDRB
    cl: condlogic port map(clk, reset, Instr(31 downto 28),
                        ALUFlags, FlagW, PCS, NextPC, RegW, MemW,
                        PCWrite, RegWrite, MemWrite);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity decoder is -- main control decoder
    port(clk, reset:      in  STD_LOGIC;
         Op:              in  STD_LOGIC_VECTOR(1 downto 0);
         Funct:          in  STD_LOGIC_VECTOR(5 downto 0);
         Rd:             in  STD_LOGIC_VECTOR(3 downto 0);
         FlagW:          out STD_LOGIC_VECTOR(1 downto 0);
         PCS, NextPC:    out STD_LOGIC;

```

```

    RegW, MemW:      out STD_LOGIC;
    IRWrite, AdrSrc: out STD_LOGIC;
    ResultSrc:       out STD_LOGIC_VECTOR(1 downto 0);
    ALUSrcA:         out STD_LOGIC;
    ALUSrcB, ImmSrc: out STD_LOGIC_VECTOR(1 downto 0);
    RegSrc:          out STD_LOGIC_VECTOR(1 downto 0);
    ALUControl:      out STD_LOGIC_VECTOR(2 downto 0); -- BIC
    LDRB:            out STD_LOGIC); -- LDRB
end;

architecture behave of decoder is
    component mainfsm
        port (clk, reset:      in  STD_LOGIC;
              Op:              in  STD_LOGIC_VECTOR(1 downto 0);
              Funct:           in  STD_LOGIC_VECTOR(5 downto 0);
              IRWrite:         out STD_LOGIC;
              AdrSrc, ALUSrcA: out STD_LOGIC;
              ALUSrcB:         out STD_LOGIC_VECTOR(1 downto 0);
              ResultSrc:       out STD_LOGIC_VECTOR(1 downto 0);
              NextPC, RegW:    out STD_LOGIC;
              MemW, Branch:    out STD_LOGIC;
              ALUOp:           out STD_LOGIC_VECTOR(1 downto 0); -- LDR +-
        );
    end component;
    signal Branch: STD_LOGIC;
    signal ALUOp: STD_LOGIC_VECTOR(1 downto 0); -- LDR +- imm12
begin
    -- Main FSM
    fsm: mainfsm port map (clk, reset, Op, Funct,
                          IRWrite, AdrSrc,
                          ALUSrcA, ALUSrcB, ResultSrc,
                          NextPC, RegW, MemW, Branch, ALUOp,
                          LDRB); -- LDRB

    process(all) begin -- ALU Decoder
        if (ALUOp = "10") then
            case Funct(4 downto 1) is
                when "0100" => ALUControl <= "000"; -- ADD
                when "0010" => ALUControl <= "101"; -- SUB
                when "0000" => ALUControl <= "010"; -- AND
                when "1100" => ALUControl <= "011"; -- ORR
                when "1110" => ALUControl <= "110"; -- BIC
                when others => ALUControl <= "---"; -- unimplemented
            end case;
            FlagW(1) <= Funct(0);
            FlagW(0) <= Funct(0) and (not ALUControl(1));
        elsif (ALUOp = "01") then
            ALUControl <= "101";
            FlagW <= "00";
        else
            ALUControl <= "000";
            FlagW <= "00";
        end if;
    end process;
end behave;

```

```

end process;

-- PC Logic
PCS <= ((and Rd) and RegW) or Branch;

-- Instr Decoder
ImmSrc <= Op;
RegSrc(0) <= '1' when (Op = 2B"10") else '0';
RegSrc(1) <= '1' when (Op = 2B"01") else '0';
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mainfsm is
  port(clk, reset:      in  STD_LOGIC;
        Op:            in  STD_LOGIC_VECTOR(1 downto 0);
        Funct:         in  STD_LOGIC_VECTOR(5 downto 0);
        IRWrite:       out STD_LOGIC;
        AdrSrc, ALUSrcA: out STD_LOGIC;
        ALUSrcB:       out STD_LOGIC_VECTOR(1 downto 0);
        ResultSrc:     out STD_LOGIC_VECTOR(1 downto 0);
        NextPC, RegW:  out STD_LOGIC;
        MemW, Branch:  out STD_LOGIC;
        ALUOp:         out STD_LOGIC_VECTOR(1 downto 0); -- LDR +- imm12
        LDRB:          out STD_LOGIC); -- LDRB
end;

architecture synth of mainfsm is
  type statetype is (FETCH, DECODE, MEMADRADD, MEMADRSUB, MEMRD,
                    MEMRDBYTE, -- LDRB
                    MEMWB, MEMWR,
                    EXECUTER, EXECUTEI, ALUWB, BR,
                    BL, -- BL
                    UNKNOWN);
  signal state, nextstate: statetype;
  signal controls: STD_LOGIC_VECTOR(13 downto 0);
begin
  --state register
  process(clk, reset) begin
    if reset then state <= FETCH;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(all) begin
    case state is
      when FETCH =>      nextstate <= DECODE;
      when DECODE =>
        case Op is
          when "00" =>    nextstate <= ExecuteI when (Funct(5) = '1')
                           else EXECUTER;
          when "01" =>    nextstate <= MEMADRADD when (Funct(3) = '1')

```

```

                                else MEMADRSUB; -- LDR +- imm
when "10" => nextstate <= BL when (Funct(4) = '1')
                                else BR;
when others => nextstate <= UNKNOWN;
end case;
when EXECUTER => nextstate <= ALUWB;
when EXECUTEI => nextstate <= ALUWB;
when MEMADRADD => nextstate <= MEMRDBYTE when ((Funct(0) = '1')
and (Funct(2)='1')) -- LDRB
                                else MEMRD when ((Funct(0) = '1') and
(Funct(2)='0'))
                                else MEMWR;
when MEMADRSUB => nextstate <= MEMRD when (Funct(0) = '1')
                                else MEMWR;
when MEMRD => nextstate <= MEMWB;
when MEMRDBYTE => nextstate <= MEMWB; -- LDRB
when others => nextstate <= FETCH;
end case;
end process;

-- state-dependent output logic
process(all) begin
  case state is
    when FETCH => controls <= 14B"10001010110000";
    when DECODE => controls <= 14B"00000010110000";
    when EXECUTER => controls <= 14B"00000000000100";
    when EXECUTEI => controls <= 14B"00000000001100";
    when ALUWB => controls <= 14B"00010000000000";
    when MEMADRADD => controls <= 14B"00000000001000";
    when MEMADRSUB => controls <= 14B"00000000001010";
    when MEMWR => controls <= 14B"00100100000000";
    when MEMRD => controls <= 14B"00000100000000";
    when MEMRDBYTE => controls <= 14B"00000100000001";
    when MEMWB => controls <= 14B"00010001000000";
    when BR => controls <= 14B"01000010001000";
    when BL => controls <= 14B"01010010001000";
    when others => controls <= "XXXXXXXXXXXX";
  end case;
end process;

(NextPC, Branch, MemW, RegW, IRWrite,
AdrSrc, ResultSrc,
ALUSrcA, ALUSrcB, ALUOp, LDRB) <= controls;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condlogic is -- Conditional logic
  port(clk, reset: in STD_LOGIC;
        Cond: in STD_LOGIC_VECTOR(3 downto 0);
        ALUFlags: in STD_LOGIC_VECTOR(3 downto 0);
        FlagW: in STD_LOGIC_VECTOR(1 downto 0);
        PCS, NextPC: in STD_LOGIC;
        RegW, MemW: in STD_LOGIC;
        PCWrite, RegWrite: out STD_LOGIC;

```



```

        MemWrite:          out STD_LOGIC);
end;

architecture behave of condlogic is
    component condcheck
        port(Cond:          in  STD_LOGIC_VECTOR(3 downto 0);
              Flags:        in  STD_LOGIC_VECTOR(3 downto 0);
              CondEx:       out STD_LOGIC);
    end component;
    component flopenr generic(width: integer);
        port(clk, reset, en: in  STD_LOGIC;
              d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
              q:          out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component flopr generic(width: integer);
        port(clk, reset: in  STD_LOGIC;
              d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
              q:          out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;

    signal FlagWrite:      STD_LOGIC_VECTOR(1 downto 0);
    signal Flags:          STD_LOGIC_VECTOR(3 downto 0);
    signal CondEx:         STD_LOGIC_VECTOR(0 downto 0);
    signal CondExDelayed: STD_LOGIC_VECTOR(0 downto 0);
begin
    flagreg1: flopenr generic map(2)
        port map(clk, reset, FlagWrite(1),
                  ALUFlags(3 downto 2), Flags(3 downto 2));
    flagreg0: flopenr generic map(2)
        port map(clk, reset, FlagWrite(0),
                  ALUFlags(1 downto 0), Flags(1 downto 0));
    cc: condcheck port map(Cond, Flags, CondEx(0));
    condreg: flopr generic map(1)
        port map(clk, reset, CondEx, CondExDelayed);

    FlagWrite <= FlagW and (CondEx(0), CondEx(0));
    RegWrite  <= RegW  and CondExDelayed(0);
    MemWrite  <= MemW  and CondExDelayed(0);
    PCWrite   <= (PCS  and CondExDelayed(0)) or NextPC;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condcheck is
    port(Cond:          in  STD_LOGIC_VECTOR(3 downto 0);
          Flags:        in  STD_LOGIC_VECTOR(3 downto 0);
          CondEx:       out STD_LOGIC);
end;

architecture behave of condcheck is
    signal neg, zero, carry, overflow, ge: STD_LOGIC;
begin
    (neg, zero, carry, overflow) <= Flags;
    ge <= (neg xnor overflow);

```

```

process(all) begin -- Condition checking
  case Cond is
    when "0000" => CondEx <= zero;
    when "0001" => CondEx <= not zero;
    when "0010" => CondEx <= carry;
    when "0011" => CondEx <= not carry;
    when "0100" => CondEx <= neg;
    when "0101" => CondEx <= not neg;
    when "0110" => CondEx <= overflow;
    when "0111" => CondEx <= not overflow;
    when "1000" => CondEx <= carry and (not zero);
    when "1001" => CondEx <= not(carry and (not zero));
    when "1010" => CondEx <= ge;
    when "1011" => CondEx <= not ge;
    when "1100" => CondEx <= (not zero) and ge;
    when "1101" => CondEx <= not ((not zero) and ge);
    when "1110" => CondEx <= '1';
    when others => CondEx <= '-';
  end case;
end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity datapath is
  port(clk, reset:          in  STD_LOGIC;
        Adr:                out STD_LOGIC_VECTOR(31 downto 0);
        WriteData:          out STD_LOGIC_VECTOR(31 downto 0);
        ReadData:           in  STD_LOGIC_VECTOR(31 downto 0);
        Instr:              out STD_LOGIC_VECTOR(31 downto 0);
        ALUFlags:           out STD_LOGIC_VECTOR(3 downto 0);
        PCWrite:            in  STD_LOGIC;
        RegWrite:           in  STD_LOGIC;
        IRWrite:            in  STD_LOGIC;
        AdrSrc:             in  STD_LOGIC;
        RegSrc:             in  STD_LOGIC_VECTOR(1 downto 0);
        ALUSrcA:            in  STD_LOGIC;
        ALUSrcB:            in  STD_LOGIC_VECTOR(1 downto 0);
        ResultSrc:          in  STD_LOGIC_VECTOR(1 downto 0);
        ImmSrc:             in  STD_LOGIC_VECTOR(1 downto 0);
        ALUControl:         in  STD_LOGIC_VECTOR(2 downto 0); -- BIC
        LDRB:               in  STD_LOGIC); -- LDRB
end;

architecture struct of datapath is
  component alu
    port(a, b:          in  STD_LOGIC_VECTOR(31 downto 0);
          ALUControl: in  STD_LOGIC_VECTOR(2 downto 0); -- BIC
          Result:       buffer STD_LOGIC_VECTOR(31 downto 0);
          ALUFlags:     out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  component regfile
    port(clk:          in  STD_LOGIC;
          we3:         in  STD_LOGIC;

```

```

        ral, ra2, wa3: in  STD_LOGIC_VECTOR(31 downto 0);
        wd3, r15:      in  STD_LOGIC_VECTOR(31 downto 0);
        rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
end component;
component adder
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
         y:  out STD_LOGIC_VECTOR(31 downto 0));
end component;
component extend
    port(Instr: in  STD_LOGIC_VECTOR(23 downto 0);
         ImmSrc: in STD_LOGIC_VECTOR(1 downto 0);
         ExtImm: out STD_LOGIC_VECTOR(31 downto 0));
end component;
component flopenr generic(width: integer);
    port(clk, reset, en: in  STD_LOGIC;
         d:      in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:      out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
component flopr generic(width: integer);
    port(clk, reset: in  STD_LOGIC;
         d:      in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:      out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
component mux2 generic(width: integer);
    port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
component mux3 generic(width: integer);
    port(d0, d1, d2: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:      in  STD_LOGIC_VECTOR(1 downto 0);
         y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
component mux4 generic(width: integer);
    port(d0, d1, d2, d3: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:      in  STD_LOGIC_VECTOR(1 downto 0);
         y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
component zeroextend
    port(a: in  STD_LOGIC_VECTOR(7 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
end component;
signal PCNext, PC:          STD_LOGIC_VECTOR(31 downto 0);
signal ExtImm, SrcA, SrcB:  STD_LOGIC_VECTOR(31 downto 0);
signal Result:              STD_LOGIC_VECTOR(31 downto 0);
signal Data, RD1, RD2, A:   STD_LOGIC_VECTOR(31 downto 0);
signal ALUResult, ALUOut:   STD_LOGIC_VECTOR(31 downto 0);
signal RA1, RA2:            STD_LOGIC_VECTOR(3 downto 0);
signal DataByte:            STD_LOGIC_VECTOR(7 downto 0); -- LDRB
signal MemData, DataByteExt: STD_LOGIC_VECTOR(31 downto 0); -- LDRB
signal WA3:                  STD_LOGIC_VECTOR(3 downto 0); -- BL
signal WD3:                  STD_LOGIC_VECTOR(31 downto 0); -- BL

```

```
begin
```

```

-- next PC logic
pcreg: flopenr generic map(32)
  port map(clk, reset, PCWrite, Result, PC);

-- memory logic
adrmux: mux2 generic map(32)
  port map(PC, ALUOut, AdrSrc, Adr);
ir: flopenr generic map(32)
  port map(clk, reset, IRWrite, ReadData, Instr);
ldrbmux: mux4 generic map(8)
  port map(Instr(31 downto 24), Instr(23 downto 16), Instr(15 downto 8),
Instr(7 downto 0),
    Adr(1 downto 0), DataByte);
ze: zeroextend port map(DataByte, DataByteExt);
datamux: mux2 generic map(32)
  port map(ReadData, DataByteExt, LDRB, MemData);
datareg: flopr generic map(32)
  port map(clk, reset, MemData, Data);

-- register file logic
ralmux: mux2 generic map(4)
  port map(Instr(19 downto 16), "1111", RegSrc(0), RA1);
ra2mux: mux2 generic map(4) port map(Instr(3 downto 0),
    Instr(15 downto 12), RegSrc(1), RA2);
wa3mux: mux2 generic map(4) port map(Instr(15 downto 12), -- BL
    "1110", RegSrc(0), WA3);
wd3mux: mux2 generic map(32) port map(Result, -- BL
    PC, RegSrc(0), WD3);
rf: regfile port map(clk, RegWrite, RA1, RA2,
    WA3, WD3, -- BL
    Result, RD1, RD2);
srcareg: flopr generic map(32)
  port map(clk, reset, RD1, A);
wdreg: flopr generic map(32)
  port map(clk, reset, RD2, WriteData);
ext: extend port map(Instr(23 downto 0), ImmSrc, ExtImm);

-- ALU logic
srcamux: mux2 generic map(32)
  port map(A, PC, ALUSrcA, SrcA);
srcbmux: mux3 generic map(32)
  port map(WriteData, ExtImm, 32D"4", ALUSrcB, SrcB);
i_alu: alu port map(SrcA, SrcB, ALUControl, ALUResult, ALUFlags);
aluoutreg: flopr generic map(32)
  port map(clk, reset, ALUResult, ALUOut);
resmux: mux3 generic map(32)
  port map(ALUOut, Data, ALUResult, ResultSrc, Result);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity regfile is -- three-port register file
  port(clk:          in  STD_LOGIC;
        we3:         in  STD_LOGIC;

```

```

        ral, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
        wd3, r15:      in  STD_LOGIC_VECTOR(31 downto 0);
        rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
    type ramtype is array (31 downto 0) of
        STD_LOGIC_VECTOR(31 downto 0);
    signal mem: ramtype;
begin
    process(clk) begin
        if rising_edge(clk) then
            if we3 = '1' then mem(to_integer(wa3)) <= wd3;
            end if;
        end if;
    end process;
    process(all) begin
        if (to_integer(ral) = 15) then rd1 <= r15;
        else rd1 <= mem(to_integer(ral));
        end if;
        if (to_integer(ra2) = 15) then rd2 <= r15;
        else rd2 <= mem(to_integer(ra2));
        end if;
    end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity adder is -- adder
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
         y:   out STD_LOGIC_VECTOR(31 downto 0));
end;

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

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity extend is
    port(Instr: in  STD_LOGIC_VECTOR(23 downto 0);
         ImmSrc: in  STD_LOGIC_VECTOR(1 downto 0);
         ExtImm: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of extend is
begin
    process(all) begin
        case ImmSrc is
            when "00"    => ExtImm <= (X"000000", Instr(7 downto 0));
            when "01"    => ExtImm <= (X"00000", Instr(11 downto 0));
            when "10"    => ExtImm <= (Instr(23), Instr(23), Instr(23),
                Instr(23), Instr(23), Instr(23), Instr(23 downto 0), "00");
            when others => ExtImm <= X"-----";
        end case;
    end process;
end;

```

```

        end case;
    end process;
end;

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

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

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

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

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

architecture behave of mux2 is
begin
    y <= d1 when s else d0;
end;

```

```

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

```

```

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

```

```

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

```

```

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

```

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity alu is
  port(a, b:           in  STD_LOGIC_VECTOR(31 downto 0);
        ALUControl: in  STD_LOGIC_VECTOR(2 downto 0); -- BIC
        Result:       buffer STD_LOGIC_VECTOR(31 downto 0);
        ALUFlags:     out STD_LOGIC_VECTOR(3 downto 0));
end;

```

```

architecture behave of alu is
    signal condinvb: STD_LOGIC_VECTOR(31 downto 0);
    signal sum:      STD_LOGIC_VECTOR(32 downto 0);
    signal neg, zero, carry, overflow: STD_LOGIC;
begin
    condinvb <= not b when ALUControl(2) else b; -- BIC
    sum <= ('0', a) + ('0', condinvb) + ALUControl(2); -- BIC

    process(all) begin
        case? ALUControl(1 downto 0) is
            when "0-" => result <= sum(31 downto 0);
            when "10"  => result <= a and condinvb; -- BIC
            when "11"  => result <= a or b;
            when others => result <= (others => '-');
        end case?;
    end process;

    neg      <= Result(31);
    zero     <= '1' when (Result = 0) else '0';
    carry    <= (not ALUControl(1)) and sum(32);
    overflow <= (not ALUControl(1)) and
                (not (a(31) xor b(31) xor ALUControl(0))) and
                (a(31) xor sum(31));
    ALUFlags <= (neg, zero, carry, overflow);
end;

-- zeroextend needed for LDRB
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity zeroextend is -- zero-extension unit
    port(a: in  STD_LOGIC_VECTOR(7 downto 0);
          y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of zeroextend is
begin
    y <= ("00000000000000000000000000", a);
end;

```

Test ARM assembly

```

MAIN
    BL  TEST                ; call TEST
    SUB R3, PC, PC          ; R3 = 0
    ADD R4, R3, #0xC7       ; R4 = 0xC7
    ADD R5, R3, #0xDF       ; R5 = 0xDF
    ADD R5, R5, #0xFF       ; R5 = 0x1DE
    STR R5, [R4, #1]        ; mem[0xC8] <= 0x1DE
    LDRB R6, [R3, #0xC9]    ; R6 <= mem[0xC9]7:0 = 1
    LDRB R7, [R3, #0xC8]    ; R7 <= mem[0xC8]7:0 = 0xDE
    LDR  R8, [R7, #-0x16]   ; R8 <= mem[0xC8] = 0x1DE
    ADD  R3, R3, #57        ; R3 = 0x39
    STR  R3, [R4, #9]       ; mem[0xD0] <= 0x39
TEST
    ADD  PC, LR, #0         ; PC = LR (return to point of call)

```



```

; 0x00 EB000009  BL      TEST
; 0x04 E04F300F  SUB      R3,PC,PC
; 0x08 E28340C7  ADD      R4,R3,#0xC7
; 0x0c E28350DF  ADD      R5,R3,#0xDF
; 0x20 E28550FF  ADD      R5,R5,#0xFF
; 0x24 E5845001  STR      R5,[R4,#0x1]
; 0x28 E5D360C9  LDRB     R6,[R3,#0xC9]
; 0x2c E5D370C8  LDRB     R7,[R3,#0xC8]
; 0x30 E5178016  LDR      R8,[R7,#-0x16]
; 0x34 E2833039  ADD      R3,R3,#0x39
; 0x38 E5843009  STR      R3,[R4,#0x9]
; 0x3c E28EF000  ADD      PC,R14,#0

```

ex7.26_memfile.dat

```

EB000009
E04F300F
E28340C7
E28350DF
E28550FF
E5845001
E5D360C9
E5D370C8
E5178016
E2833039
E5843009
E28EF000

```

Exercise 7.27

SystemVerilog

```

// Multi-cycle implementation of a subset of ARMv4
// Added instructions:
//   ASR, TST, SBC, ROR

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 == 88 & WriteData == 32'h2ffffffe) begin
            $display("Simulation succeeded");
            $stop;
        end else begin
            $display("Simulation failed");
            $stop;
        end
    end
end
endmodule

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

    logic [31:0] ReadData;

    // instantiate processor and shared memory
    arm arm(clk, reset, MemWrite, Adr,
           WriteData, ReadData);
    mem mem(clk, MemWrite, Adr, WriteData, ReadData);
endmodule

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

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

    initial
        $readmemh("ex7.27_memfile.dat",RAM);

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

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

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

```

```

logic [31:0] Instr;
logic [3:0] ALUFlags;
logic      PCWrite, RegWrite, IRWrite;
logic      AdrSrc, ALUSrcA;
logic [1:0] RegSrc, ALUSrcB, ImmSrc, ResultSrc;
logic [2:0] ALUControl; // SBC
logic      carry;      // SBC
logic      Shift;      // ASR, ROR

controller c(clk, reset, Instr[31:12], ALUFlags,
             PCWrite, MemWrite, RegWrite, IRWrite,
             AdrSrc, RegSrc, ALUSrcA, ALUSrcB, ResultSrc,
             ImmSrc, ALUControl, carry, Shift);
datapath dp(clk, reset, Adr, WriteData, ReadData, Instr, ALUFlags,
            PCWrite, RegWrite, IRWrite,
            AdrSrc, RegSrc, ALUSrcA, ALUSrcB, ResultSrc,
            ImmSrc, ALUControl, carry, Shift);
endmodule

module controller(input logic      clk,
                  input logic      reset,
                  input logic [31:12] Instr,
                  input logic [3:0] ALUFlags,
                  output logic      PCWrite,
                  output logic      MemWrite,
                  output logic      RegWrite,
                  output logic      IRWrite,
                  output logic      AdrSrc,
                  output logic [1:0] RegSrc,
                  output logic      ALUSrcA,
                  output logic [1:0] ALUSrcB,
                  output logic [1:0] ResultSrc,
                  output logic [1:0] ImmSrc,
                  output logic [2:0] ALUControl, // SBC
                  output logic      carry,      // SBC
                  output logic      Shift      // ASR, ROR
);

logic [1:0] FlagW;
logic      PCS, NextPC, RegW, MemW;
logic      NoWrite; // TST

decode dec(clk, reset, Instr[27:26], Instr[25:20], Instr[15:12],
          FlagW, PCS, NextPC, RegW, MemW,
          IRWrite, AdrSrc, ResultSrc,
          ALUSrcA, ALUSrcB, ImmSrc, RegSrc, ALUControl,
          NoWrite, // TST
          Shift); // ASR, ROR
condlogic cl(clk, reset, Instr[31:28], ALUFlags,
            FlagW, PCS, NextPC, RegW, MemW,
            PCWrite, RegWrite, MemWrite,
            carry, // SBC
            NoWrite); // TST
endmodule

```

```

module decode(input  logic      clk, reset,
              input  logic [1:0] Op,
              input  logic [5:0] Funct,
              input  logic [3:0] Rd,
              output logic [1:0] FlagW,
              output logic      PCS, NextPC, RegW, MemW,
              output logic      IRWrite, AdrSrc,
              output logic [1:0] ResultSrc,
              output logic      ALUSrcA,
              output logic [1:0] ALUSrcB, ImmSrc, RegSrc,
              output logic [2:0] ALUControl, // SBC
              output logic      NoWrite,    // TST
              output logic      Shift);    // ASR, ROR

logic      Branch, ALUOp;

// Main FSM
mainfsm fsm(clk, reset, Op, Funct,
            IRWrite, AdrSrc,
            ALUSrcA, ALUSrcB, ResultSrc,
            NextPC, RegW, MemW, Branch, ALUOp);

always_comb
  if (ALUOp) begin // which Data-processing Instr?
    case(Funct[4:1])
      4'b0100: begin ALUControl = 3'b000; // ADD
                    Shift = 1'b0;
                    NoWrite = 1'b0;
                  end
      4'b0010: begin ALUControl = 3'b001; // SUB
                    Shift = 1'b0;
                    NoWrite = 1'b0;
                  end
      4'b0000: begin ALUControl = 3'b010; // AND
                    Shift = 1'b0;
                    NoWrite = 1'b0;
                  end
      4'b1100: begin ALUControl = 3'b011; // ORR
                    Shift = 1'b0;
                    NoWrite = 1'b0;
                  end
      4'b1101: begin ALUControl = 3'b000; // ASR, ROR
                    Shift = 1'b1;
                    NoWrite = 1'b0;
                  end
      4'b1000: begin ALUControl = 3'b010; // TST
                    Shift = 1'b0;
                    NoWrite = 1'b1;
                  end
      4'b0110: begin ALUControl = 3'b101; // SBC
                    Shift = 1'b0;
                    NoWrite = 1'b0;
                  end
    end
  end

```



```

        else                nextstate = EXECUTER;
        2'b01:              nextstate = MEMADR;
        2'b10:              nextstate = BRANCH;
        default:            nextstate = UNKNOWN;
    endcase
EXECUTER:                  nextstate = ALUWB;
EXECUTEI:                  nextstate = ALUWB;
MEMADR:
    if (Func[0])            nextstate = MEMRD;
    else                    nextstate = MEMWR;
MEMRD:                    nextstate = MEMWB;
default:                  nextstate = FETCH;
endcase

// state-dependent output logic
always_comb
    case(state)
        FETCH:              controls = 12'b10001_010_1100;
        DECODE:             controls = 12'b00000_010_1100;
        EXECUTER:           controls = 12'b00000_000_0001;
        EXECUTEI:           controls = 12'b00000_000_0011;
        ALUWB:              controls = 12'b00010_000_0000;
        MEMADR:             controls = 12'b00000_000_0010;
        MEMWR:              controls = 12'b00100_100_0000;
        MEMRD:              controls = 12'b00000_100_0000;
        MEMWB:              controls = 12'b00010_001_0000;
        BRANCH:             controls = 12'b01000_010_0010;
        default:            controls = 12'bxxxxx_xxx_xxxx;
    endcase

    assign {NextPC, Branch, MemW, RegW, IRWrite,
            AdrSrc, ResultSrc,
            ALUSrcA, ALUSrcB, ALUOp} = controls;
endmodule

module condlogic(input  logic      clk, reset,
                 input  logic [3:0] Cond,
                 input  logic [3:0] ALUFlags,
                 input  logic [1:0] FlagW,
                 input  logic      PCS, NextPC, RegW, MemW,
                 output logic      PCWrite, RegWrite, MemWrite,
                 output logic      carry,      // SBC
                 input  logic      NoWrite); // TST

    logic [1:0] FlagWrite;
    logic [3:0] Flags;
    logic      CondEx, CondExDelayed;
    logic      NoWriteDelayed; // TST

    flopenr #(2)flagreg1(clk, reset, FlagWrite[1], ALUFlags[3:2],
Flags[3:2]);
    flopenr #(2)flagreg0(clk, reset, FlagWrite[0], ALUFlags[1:0],
Flags[1:0]);

```

```

// write controls are conditional
condcheck cc(Cond, Flags, CondEx);
flopr #(1)nowritereg(clk, reset, NoWrite, NoWriteDelayed);
flopr #(1)condreg(clk, reset, CondEx, CondExDelayed);
assign FlagWrite = FlagW & {2{CondEx}};
assign RegWrite  = RegW  & CondExDelayed & ~NoWriteDelayed; // TST
assign MemWrite  = MemW  & CondExDelayed;
assign PCWrite   = (PCS  & CondExDelayed) | NextPC;

assign carry     = Flags[1]; // SBC
endmodule

module condcheck(input  logic [3:0] Cond,
                 input  logic [3:0] Flags,
                 output logic      CondEx);

logic neg, zero, carry, overflow, ge;

assign {neg, zero, carry, overflow} = Flags;
assign ge = (neg == overflow);

always_comb
case(Cond)
  4'b0000: CondEx = zero;           // EQ
  4'b0001: CondEx = ~zero;          // NE
  4'b0010: CondEx = carry;          // CS
  4'b0011: CondEx = ~carry;         // CC
  4'b0100: CondEx = neg;            // MI
  4'b0101: CondEx = ~neg;           // PL
  4'b0110: CondEx = overflow;       // VS
  4'b0111: CondEx = ~overflow;      // VC
  4'b1000: CondEx = carry & ~zero;   // HI
  4'b1001: CondEx = ~(carry & ~zero); // LS
  4'b1010: CondEx = ge;             // GE
  4'b1011: CondEx = ~ge;            // LT
  4'b1100: CondEx = ~zero & ge;      // GT
  4'b1101: CondEx = ~(~zero & ge);  // LE
  4'b1110: CondEx = 1'b1;           // Always
  default: CondEx = 1'bx;           // undefined
endcase
endmodule

module datapath(input  logic      clk, reset,
                output logic [31:0] Adr, WriteData,
                input  logic [31:0] ReadData,
                output logic [31:0] Instr,
                output logic [3:0]  ALUFlags,
                input  logic      PCWrite, RegWrite,
                input  logic      IRWrite,
                input  logic      AdrSrc,
                input  logic [1:0] RegSrc,
                input  logic      ALUSrcA,
                input  logic [1:0] ALUSrcB, ResultSrc,
                input  logic [1:0] ImmSrc,

```

```

        input  logic [2:0]  ALUControl,  // SBC
        input  logic        carry,       // SBC
        input  logic        Shift);      // ASR, ROR

logic [31:0] PCNext, PC;
logic [31:0] ExtImm, SrcA, SrcB, Result;
logic [31:0] Data, RD1, RD2, A, ALUResult, ALUOut;
logic [3:0]  RA1, RA2;
logic [31:0] srcBshifted, ALUResultOut; // ASR, ROR

// next PC logic
flopnr #(32) pcreg(clk, reset, PCWrite, Result, PC);

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

// register file logic
mux2 #(4)   ralmux(Instr[19:16], 4'b1111, RegSrc[0], RA1);
mux2 #(4)   ra2mux(Instr[3:0], Instr[15:12], RegSrc[1], RA2);
regfile     rf(clk, RegWrite, RA1, RA2,
               Instr[15:12], Result, Result,
               RD1, RD2);
flopr #(32) srcareg(clk, reset, RD1, A);
flopr #(32) wdreg(clk, reset, RD2, WriteData);
extend      ext(Instr[23:0], ImmSrc, ExtImm);

// ALU logic
mux2 #(32)  srcamux(A, PC, ALUSrcA, SrcA);
// ASR, ROR
mux3 #(32)  srcbmux(srcBshifted, ExtImm, 32'd4, ALUSrcB, SrcB);
shifter     sh(WriteData, Instr[11:7], Instr[6:5], srcBshifted);
alu         alu(SrcA, SrcB, ALUControl, ALUResult, ALUFlags, carry);
mux2 #(32)  aluresultmux(ALUResult, SrcB, Shift, ALUResultOut);
flopr #(32) aluoutreg(clk, reset, ALUResultOut, ALUOut);
mux3 #(32)  resmux(ALUOut, Data, ALUResultOut, ResultSrc, Result);
endmodule

module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [3:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3, r15,
               output logic [31:0] rd1, rd2);

logic [31:0] rf[14:0];

// three ported register file
// read two ports combinationaly
// write third port on rising edge of clock
// register 15 reads PC+8 instead

always_ff @(posedge clk)

```



```

    if (we3) rf[wa3] <= wd3;

    assign rd1 = (ra1 == 4'b1111) ? r15 : rf[ra1];
    assign rd2 = (ra2 == 4'b1111) ? r15 : rf[ra2];
endmodule

module extend(input  logic [23:0] Instr,
              input  logic [1:0] ImmSrc,
              output logic [31:0] ExtImm);

    always_comb
        case(ImmSrc)
            // 8-bit unsigned immediate
            2'b00: ExtImm = {24'b0, Instr[7:0]};
            // 12-bit unsigned immediate
            2'b01: ExtImm = {20'b0, Instr[11:0]};
            // 24-bit two's complement shifted branch
            2'b10: ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00};
            default: ExtImm = 32'bx; // undefined
        endcase
endmodule

module adder #(parameter WIDTH=8)
    (input  logic [WIDTH-1:0] a, b,
     output logic [WIDTH-1:0] y);

    assign y = a + b;
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 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 alu(input  logic [31:0] a, b,
           input  logic [2:0]  ALUControl, // SBC
           output logic [31:0] Result,
           output logic [3:0]  ALUFlags,
           input  logic        carry);    // SBC

    logic        neg, zero, carryout, overflow;
    logic [31:0] condinvb;
    logic [32:0] sum;
    logic        carryin;                // SBC                // SBC

    assign carryin = ALUControl[2] ? carry : ALUControl[0]; // SBC

    assign condinvb = ALUControl[0] ? ~b : b;
    assign sum = a + condinvb + carryin;    // SBC

    always_comb
        casex (ALUControl[1:0])
            2'b0?: Result = sum;
            2'b10: Result = a & b;
            2'b11: Result = a | b;
        endcase

    assign neg      = Result[31];
    assign zero     = (Result == 32'b0);
    assign carryout = (ALUControl[1] == 1'b0) & sum[32];
    assign overflow = (ALUControl[1] == 1'b0) & ~(a[31] ^ b[31] ^
        ALUControl[0]) & (a[31] ^ sum[31]);
    assign ALUFlags = {neg, zero, carryout, overflow};
endmodule

// shifter needed for ASR, ROR
module shifter(input  logic signed [31:0] a,
              input  logic      [ 4:0] shamt,
              input  logic      [ 1:0] shtype,
              output logic signed [31:0] y);

    always_comb
        case (shtype)
            2'b10: y = a >>> shamt;
            2'b11: y = (a >> shamt) | (a << (32-shamt));
        endcase

```

```

        default: y = a;
    endcase
endmodule

```

VHDL

```

library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity testbench is
end;

architecture test of testbench is
    component top
        port (clk, reset:          in  STD_LOGIC;
              WriteData, Addr:     out STD_LOGIC_VECTOR(31 downto 0);
              MemWrite:           out STD_LOGIC);
    end component;
    signal WriteData, DataAdr:     STD_LOGIC_VECTOR(31 downto 0);
    signal clk, reset,  MemWrite:  STD_LOGIC;
begin

    -- instantiate device to be tested
    dut: top port map (clk, reset, WriteData, DataAdr, MemWrite);

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

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

    -- check that 7 gets written to address 84
    -- at end of program
    process (clk) begin
        if (clk'event and clk = '0' and MemWrite = '1') then
            if (to_integer(DataAdr) = 88 and
                to_integer(WriteData) = 32X"2FFFFFFE") then
                report "NO ERRORS: Simulation succeeded" severity failure;
            else
                report "Simulation failed" severity failure;
            end if;
        end if;
    end process;
end;

```

```

library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity top is -- top-level design for testing
    port(clk, reset:      in      STD_LOGIC;
          WriteData, Adr:  buffer STD_LOGIC_VECTOR(31 downto 0);
          MemWrite:        buffer STD_LOGIC);
end;

```

```

architecture test of top is
    component arm
        port(clk, reset:      in      STD_LOGIC;
              MemWrite:       out     STD_LOGIC;
              Adr, WriteData:  out     STD_LOGIC_VECTOR(31 downto 0);
              ReadData:       in      STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component mem
        port(clk, we:  in  STD_LOGIC;
              a, wd:   in  STD_LOGIC_VECTOR(31 downto 0);
              rd:      out STD_LOGIC_VECTOR(31 downto 0));
    end component;
    signal ReadData: STD_LOGIC_VECTOR(31 downto 0);
begin
    -- instantiate processor and memories
    i_arm: arm port map(clk, reset, MemWrite, Adr,
                       WriteData, ReadData);
    i_mem: mem port map(clk, MemWrite, Adr,
                       WriteData, ReadData);
end;

```

```

library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity mem is -- memory
    port(clk, we:  in STD_LOGIC;
          a, wd:   in STD_LOGIC_VECTOR(31 downto 0);
          rd:      out STD_LOGIC_VECTOR(31 downto 0));
end;

```

```

architecture behave of mem is -- instruction and data memory
begin
    process is
        file mem_file: TEXT;
        variable L: line;
        variable ch: character;
        variable i, index, result: integer;

        type ramtype is array (63 downto 0) of
            STD_LOGIC_VECTOR(31 downto 0);
        variable ram: ramtype;
    begin
        -- initialize memory from file
        for i in 0 to 63 loop -- set all contents low
            ram(i) := (others => '0');
        end loop;
    end process;
end;

```

```

index := 0;
FILE_OPEN(mem_file, "ex7.27_memfile.dat", READ_MODE);
while not endfile(mem_file) loop
  readline(mem_file, L);
  result := 0;
  for i in 1 to 8 loop
    read(L, ch);
    if '0' <= ch and ch <= '9' then
      result := character'pos(ch) - character'pos('0');
    elsif 'a' <= ch and ch <= 'f' then
      result := character'pos(ch) - character'pos('a')+10;
    elsif 'A' <= ch and ch <= 'F' then
      result := character'pos(ch) - character'pos('A')+10;
    else report "Format error on line " & integer'image(index)
      severity error;
    end if;
    ram(index)(35-i*4 downto 32-i*4) :=
      to_std_logic_vector(result,4);
  end loop;
  index := index + 1;
end loop;

-- read or write memory
loop
  if clk'event and clk = '1' then
    if (we = '1') then
      ram(to_integer(a(7 downto 2))) := wd;
    end if;
  end if;
  rd <= ram(to_integer(a(7 downto 2)));
  wait on clk, a;
end loop;
end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity arm is -- multicycle processor
  port(clk, reset:      in  STD_LOGIC;
        MemWrite:       out STD_LOGIC;
        Adr, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
        ReadData:       in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of arm is
  component controller
    port(clk, reset:      in  STD_LOGIC;
          Instr:         in  STD_LOGIC_VECTOR(31 downto 12);
          ALUFlags:      in  STD_LOGIC_VECTOR(3 downto 0);
          PCWrite:       out STD_LOGIC;
          MemWrite:      out STD_LOGIC;
          RegWrite:      out STD_LOGIC;
          IRWrite:       out STD_LOGIC);
  end component;

```



```

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- single cycle control decoder
  port(clk, reset:      in  STD_LOGIC;
        Instr:          in  STD_LOGIC_VECTOR(31 downto 12);
        ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
        PCWrite:        out STD_LOGIC;
        MemWrite:       out STD_LOGIC;
        RegWrite:       out STD_LOGIC;
        IRWrite:        out STD_LOGIC;
        AdrSrc:         out STD_LOGIC;
        RegSrc:         out STD_LOGIC_VECTOR(1 downto 0);
        ALUSrcA:        out STD_LOGIC;
        ALUSrcB:        out STD_LOGIC_VECTOR(1 downto 0);
        ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
        ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
        ALUControl:     out STD_LOGIC_VECTOR(2 downto 0); -- SBC
        carry, Shift:   out STD_LOGIC); -- SBC, ASR, ROR
end;

architecture struct of controller is
  component decoder
    port(clk, reset:      in  STD_LOGIC;
          Op:            in  STD_LOGIC_VECTOR(1 downto 0);
          Funct:         in  STD_LOGIC_VECTOR(5 downto 0);
          Rd:            in  STD_LOGIC_VECTOR(3 downto 0);
          FlagW:         out STD_LOGIC_VECTOR(1 downto 0);
          PCS, NextPC:   out STD_LOGIC;
          RegW, MemW:    out STD_LOGIC;
          IRWrite, AdrSrc: out STD_LOGIC;
          ResultSrc:     out STD_LOGIC_VECTOR(1 downto 0);
          ALUSrcA:       out STD_LOGIC;
          ALUSrcB, ImmSrc: out STD_LOGIC_VECTOR(1 downto 0);
          RegSrc:        out STD_LOGIC_VECTOR(1 downto 0);
          ALUControl:    out STD_LOGIC_VECTOR(2 downto 0); -- SBC
          NoWrite:       out STD_LOGIC; -- TST
          Shift:         out STD_LOGIC); -- ASR, ROR
  end component;
  component condlogic
    port(clk, reset:      in  STD_LOGIC;
          Cond:          in  STD_LOGIC_VECTOR(3 downto 0);
          ALUFlags:      in  STD_LOGIC_VECTOR(3 downto 0);
          FlagW:         in  STD_LOGIC_VECTOR(1 downto 0);
          PCS, NextPC:   in  STD_LOGIC;
          RegW, MemW:    in  STD_LOGIC;
          PCWrite, RegWrite: out STD_LOGIC;
          MemWrite:      out STD_LOGIC;
          carry:         out STD_LOGIC; -- SBC
          NoWrite:       in  STD_LOGIC); -- TST
  end component;
  signal FlagW: STD_LOGIC_VECTOR(1 downto 0);
  signal PCS, NextPC, RegW, MemW: STD_LOGIC;
  signal NoWrite: STD_LOGIC; -- TST
begin
  dec: decoder port map(clk, reset, Instr(27 downto 26), Instr(25 downto

```

```

20),
    Instr(15 downto 12), FlagW, PCS,
    NextPC, RegW, MemW,
    IRWrite, AdrSrc, ResultSrc,
    ALUSrcA, ALUSrcB, ImmSrc, RegSrc, ALUControl,
    NoWrite,    -- TST
    Shift);    -- ASR, ROR
cl: condlogic port map(clk, reset, Instr(31 downto 28),
    ALUFlags, FlagW, PCS, NextPC, RegW, MemW,
    PCWrite, RegWrite, MemWrite,
    carry,    -- SBC
    NoWrite); -- TST
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity decoder is -- main control decoder
    port(clk, reset:      in  STD_LOGIC;
          Op:             in  STD_LOGIC_VECTOR(1 downto 0);
          Funct:          in  STD_LOGIC_VECTOR(5 downto 0);
          Rd:             in  STD_LOGIC_VECTOR(3 downto 0);
          FlagW:          out STD_LOGIC_VECTOR(1 downto 0);
          PCS, NextPC:    out STD_LOGIC;
          RegW, MemW:     out STD_LOGIC;
          IRWrite, AdrSrc: out STD_LOGIC;
          ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
          ALUSrcA:        out STD_LOGIC;
          ALUSrcB, ImmSrc: out STD_LOGIC_VECTOR(1 downto 0);
          RegSrc:         out STD_LOGIC_VECTOR(1 downto 0);
          ALUControl:     out STD_LOGIC_VECTOR(2 downto 0); -- SBC
          NoWrite:        out STD_LOGIC; -- TST
          Shift:          out STD_LOGIC); -- ASR, ROR
end;

architecture behave of decoder is
    component mainfsm
        port(clk, reset:      in  STD_LOGIC;
              Op:             in  STD_LOGIC_VECTOR(1 downto 0);
              Funct:          in  STD_LOGIC_VECTOR(5 downto 0);
              IRWrite:        out STD_LOGIC;
              AdrSrc, ALUSrcA: out STD_LOGIC;
              ALUSrcB:        out STD_LOGIC_VECTOR(1 downto 0);
              ResultSrc:      out STD_LOGIC_VECTOR(1 downto 0);
              NextPC, RegW:    out STD_LOGIC;
              MemW, Branch:    out STD_LOGIC;
              ALUOp:          out STD_LOGIC);
    end component;
    signal Branch, ALUOp: STD_LOGIC;
begin
    -- Main FSM
    fsm: mainfsm port map(clk, reset, Op, Funct,
        IRWrite, AdrSrc,
        ALUSrcA, ALUSrcB, ResultSrc,
        NextPC, RegW, MemW, Branch, ALUOp);
end behave;

```



```

process(all) begin -- ALU Decoder
  if (ALUOp) then
    case Funct(4 downto 1) is
      when "0100" => ALUControl <= "000"; -- ADD
                     NoWrite <= '0';
                     Shift <= '0';
      when "0010" => ALUControl <= "001"; -- SUB
                     NoWrite <= '0';
                     Shift <= '0';
      when "0000" => ALUControl <= "010"; -- AND
                     NoWrite <= '0';
                     Shift <= '0';
      when "1100" => ALUControl <= "011"; -- ORR
                     NoWrite <= '0';
                     Shift <= '0';
      when "1101" => ALUControl <= "010"; -- ASR, ROR
                     NoWrite <= '0';
                     Shift <= '1';
      when "1000" => ALUControl <= "010"; -- TST
                     NoWrite <= '1';
                     Shift <= '0';
      when "0110" => ALUControl <= "101"; -- SBC
                     NoWrite <= '0';
                     Shift <= '0';
      when others => ALUControl <= "---"; -- unimplemented
                     NoWrite <= '-';
                     Shift <= '-';

    end case;
    FlagW(1) <= Funct(0);
    FlagW(0) <= Funct(0) and (not ALUControl(1));
  else
    ALUControl <= "000";
    FlagW <= "00";
    Shift <= '0';
    NoWrite <= '0';
  end if;
end process;

-- PC Logic
PCS <= ((and Rd) and RegW) or Branch;

-- Instr Decoder
ImmSrc <= Op;
RegSrc(0) <= '1' when (Op = 2B"10") else '0';
RegSrc(1) <= '1' when (Op = 2B"01") else '0';
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mainfsm is
  port(clk, reset:      in  STD_LOGIC;
        Op:            in  STD_LOGIC_VECTOR(1 downto 0);
        Funct:         in  STD_LOGIC_VECTOR(5 downto 0);
        IRWrite:       out STD_LOGIC;

```

```

    AdrSrc, ALUSrcA: out STD_LOGIC;
    ALUSrcB:         out STD_LOGIC_VECTOR(1 downto 0);
    ResultSrc:       out STD_LOGIC_VECTOR(1 downto 0);
    NextPC, RegW:    out STD_LOGIC;
    MemW, Branch:    out STD_LOGIC;
    ALUOp:           out STD_LOGIC);
end;

architecture synth of mainfsm is
    type statetype is (FETCH, DECODE, MEMADR, MEMRD, MEMWB, MEMWR,
                      EXECUTER, EXECUTEI, ALUWB, BR, UNKNOWN);
    signal state, nextstate: statetype;
    signal controls: STD_LOGIC_VECTOR(11 downto 0);
begin
    --state register
    process(clk, reset) begin
        if reset then state <= FETCH;
        elsif rising_edge(clk) then
            state <= nextstate;
        end if;
    end process;

    -- next state logic
    process(all) begin
        case state is
            when FETCH =>          nextstate <= DECODE;
            when DECODE =>
                case Op is
                    when "00" =>    nextstate <= ExecuteI when (Func(5) = '1')
                                   else EXECUTER;
                    when "01" =>    nextstate <= MEMADR;
                    when "10" =>    nextstate <= BR;
                    when others =>  nextstate <= UNKNOWN;
                end case;
            when EXECUTER =>        nextstate <= ALUWB;
            when EXECUTEI =>        nextstate <= ALUWB;
            when MEMADR  =>        nextstate <= MEMRD when (Func(0) = '1')
                                   else MEMWR;
            when MEMRD   =>        nextstate <= MEMWB;
            when others  =>        nextstate <= FETCH;
        end case;
    end process;

    -- state-dependent output logic
    process(all) begin
        case state is
            when FETCH =>          controls <= 12B"100010101100";
            when DECODE =>         controls <= 12B"000000101100";
            when EXECUTER =>        controls <= 12B"000000000001";
            when EXECUTEI =>        controls <= 12B"000000000011";
            when ALUWB  =>          controls <= 12B"000100000000";
            when MEMADR =>          controls <= 12B"000000000010";
            when MEMWR  =>          controls <= 12B"001001000000";
            when MEMRD  =>          controls <= 12B"000001000000";
        end case;
    end process;
end;

```

```

        when MEMWB =>      controls <= 12B"000100010000";
        when BR =>         controls <= 12B"010000100010";
        when others =>     controls <= "XXXXXXXXXXXX";
    end case;
end process;

(NextPC, Branch, MemW, RegW, IRWrite,
 AdrSrc, ResultSrc,
 ALUSrcA, ALUSrcB, ALUOp) <= controls;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condlogic is -- Conditional logic
    port (clk, reset:      in  STD_LOGIC;
          Cond:            in  STD_LOGIC_VECTOR(3 downto 0);
          ALUFlags:        in  STD_LOGIC_VECTOR(3 downto 0);
          FlagW:           in  STD_LOGIC_VECTOR(1 downto 0);
          PCS, NextPC:     in  STD_LOGIC;
          RegW, MemW:      in  STD_LOGIC;
          PCWrite, RegWrite: out STD_LOGIC;
          MemWrite:        out STD_LOGIC;
          carry:           out STD_LOGIC; -- SBC
          NoWrite:         in  STD_LOGIC); -- TST
end;

architecture behave of condlogic is
    component condcheck
        port (Cond:      in  STD_LOGIC_VECTOR(3 downto 0);
              Flags:     in  STD_LOGIC_VECTOR(3 downto 0);
              CondEx:    out STD_LOGIC);
    end component;
    component flopenr generic(width: integer);
        port (clk, reset, en: in  STD_LOGIC;
              d:      in  STD_LOGIC_VECTOR(width-1 downto 0);
              q:      out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component flopr generic(width: integer);
        port (clk, reset: in  STD_LOGIC;
              d:      in  STD_LOGIC_VECTOR(width-1 downto 0);
              q:      out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;

    signal FlagWrite:      STD_LOGIC_VECTOR(1 downto 0);
    signal Flags:          STD_LOGIC_VECTOR(3 downto 0);
    signal CondEx:         STD_LOGIC_VECTOR(0 downto 0);
    signal CondExDelayed:  STD_LOGIC_VECTOR(0 downto 0);
    signal NoWritevect:    STD_LOGIC_VECTOR(0 downto 0); -- TST
    signal NoWriteDelayed: STD_LOGIC_VECTOR(0 downto 0); -- TST
begin

    NoWritevect(0) <= NoWrite;
    flagreg1: flopenr generic map(2)
        port map (clk, reset, FlagWrite(1),
                  ALUFlags(3 downto 2), Flags(3 downto 2));

```

```

flagreg0: flopenr generic map(2)
  port map(clk, reset, FlagWrite(0),
           ALUFlags(1 downto 0), Flags(1 downto 0));
cc: condcheck port map(Cond, Flags, CondEx(0));
condreg: flopr generic map(1)
  port map(clk, reset, CondEx, CondExDelayed);
nowritereg: flopr generic map(1)
  port map(clk, reset, NoWritevect, NoWriteDelayed);

FlagWrite <= FlagW and (CondEx(0), CondEx(0));
RegWrite  <= RegW  and CondExDelayed(0) and (not NoWriteDelayed(0)); --
TST
MemWrite  <= MemW  and CondExDelayed(0);
PCWrite   <= (PCS  and CondExDelayed(0)) or NextPC;
carry <= Flags(1); -- SBC
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condcheck is
  port(Cond:          in  STD_LOGIC_VECTOR(3 downto 0);
       Flags:         in  STD_LOGIC_VECTOR(3 downto 0);
       CondEx:        out STD_LOGIC);
end;

architecture behave of condcheck is
  signal neg, zero, carry, overflow, ge: STD_LOGIC;
begin
  (neg, zero, carry, overflow) <= Flags;
  ge <= (neg xnor overflow);

  process(all) begin -- Condition checking
    case Cond is
      when "0000" => CondEx <= zero;
      when "0001" => CondEx <= not zero;
      when "0010" => CondEx <= carry;
      when "0011" => CondEx <= not carry;
      when "0100" => CondEx <= neg;
      when "0101" => CondEx <= not neg;
      when "0110" => CondEx <= overflow;
      when "0111" => CondEx <= not overflow;
      when "1000" => CondEx <= carry and (not zero);
      when "1001" => CondEx <= not(carry and (not zero));
      when "1010" => CondEx <= ge;
      when "1011" => CondEx <= not ge;
      when "1100" => CondEx <= (not zero) and ge;
      when "1101" => CondEx <= not ((not zero) and ge);
      when "1110" => CondEx <= '1';
      when others => CondEx <= '-';
    end case;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

```

entity datapath is

```

port (clk, reset:      in  STD_LOGIC;
      Adr:             out STD_LOGIC_VECTOR(31 downto 0);
      WriteData:       out STD_LOGIC_VECTOR(31 downto 0);
      ReadData:        in  STD_LOGIC_VECTOR(31 downto 0);
      Instr:           out STD_LOGIC_VECTOR(31 downto 0);
      ALUFlags:        out STD_LOGIC_VECTOR(3 downto 0);
      PCWrite:         in  STD_LOGIC;
      RegWrite:        in  STD_LOGIC;
      IRWrite:         in  STD_LOGIC;
      AdrSrc:          in  STD_LOGIC;
      RegSrc:          in  STD_LOGIC_VECTOR(1 downto 0);
      ALUSrcA:         in  STD_LOGIC;
      ALUSrcB:         in  STD_LOGIC_VECTOR(1 downto 0);
      ResultSrc:       in  STD_LOGIC_VECTOR(1 downto 0);
      ImmSrc:          in  STD_LOGIC_VECTOR(1 downto 0);
      ALUControl:      in  STD_LOGIC_VECTOR(2 downto 0); -- SBC
      carry, Shift:    in  STD_LOGIC); -- SBC, ASR, ROR

```

end;

architecture struct of datapath is

```

component alu
  port (a, b:          in  STD_LOGIC_VECTOR(31 downto 0);
        ALUControl: in  STD_LOGIC_VECTOR(2 downto 0); -- SBC
        Result:       buffer STD_LOGIC_VECTOR(31 downto 0);
        ALUFlags:     out STD_LOGIC_VECTOR(3 downto 0);
        carry:        in  STD_LOGIC); -- SBC
end component;
component regfile
  port (clk:          in  STD_LOGIC;
        we3:         in  STD_LOGIC;
        ra1, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
        wd3, r15:     in  STD_LOGIC_VECTOR(31 downto 0);
        rd1, rd2:     out STD_LOGIC_VECTOR(31 downto 0));
end component;
component adder
  port (a, b: in  STD_LOGIC_VECTOR(31 downto 0);
        y:   out STD_LOGIC_VECTOR(31 downto 0));
end component;
component extend
  port (Instr: in  STD_LOGIC_VECTOR(23 downto 0);
        ImmSrc: in  STD_LOGIC_VECTOR(1 downto 0);
        ExtImm: out STD_LOGIC_VECTOR(31 downto 0));
end component;
component flopenr generic (width: integer);
  port (clk, reset, en: in  STD_LOGIC;
        d:             in  STD_LOGIC_VECTOR(width-1 downto 0);
        q:             out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
component flopr generic (width: integer);
  port (clk, reset: in  STD_LOGIC;
        d:         in  STD_LOGIC_VECTOR(width-1 downto 0);
        q:         out STD_LOGIC_VECTOR(width-1 downto 0));

```

```

end component;
component mux2 generic(width: integer);
  port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
        s:      in  STD_LOGIC;
        y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
component mux3 generic(width: integer);
  port(d0, d1, d2: in  STD_LOGIC_VECTOR(width-1 downto 0);
        s:      in  STD_LOGIC_VECTOR(1 downto 0);
        y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
component shifter -- LSL
  port(a:      in  STD_LOGIC_VECTOR(31 downto 0);
        shamt:  in  STD_LOGIC_VECTOR(4  downto 0);
        shtype: in  STD_LOGIC_VECTOR(1  downto 0);
        y:      out STD_LOGIC_VECTOR(31 downto 0));
end component;

signal PCNext, PC: STD_LOGIC_VECTOR(31 downto 0);
signal ExtImm, SrcA, SrcB: STD_LOGIC_VECTOR(31 downto 0);
signal Result: STD_LOGIC_VECTOR(31 downto 0);
signal Data, RD1, RD2, A: STD_LOGIC_VECTOR(31 downto 0);
signal ALUResult, ALUOut: STD_LOGIC_VECTOR(31 downto 0);
signal RA1, RA2: STD_LOGIC_VECTOR(3 downto 0);
signal srcBshifted, ALUResultOut: STD_LOGIC_VECTOR(31 downto 0); -- ASR,
ROR
begin
  -- next PC logic
  pcreg: flopenr generic map(32)
    port map(clk, reset, PCWrite, Result, PC);

  -- memory logic
  adrmux: mux2 generic map(32)
    port map(PC, ALUOut, AdrSrc, Adr);
  ir: flopenr generic map(32)
    port map(clk, reset, IRWrite, ReadData, Instr);
  datareg: flopr generic map(32)
    port map(clk, reset, ReadData, Data);

  -- register file logic
  ralmux: mux2 generic map (4)
    port map(Instr(19 downto 16), "1111", RegSrc(0), RA1);
  ra2mux: mux2 generic map (4) port map(Instr(3 downto 0),
    Instr(15 downto 12), RegSrc(1), RA2);
  rf: regfile port map(clk, RegWrite, RA1, RA2,
    Instr(15 downto 12), Result, Result,
    RD1, RD2);
  srcareg: flopr generic map(32)
    port map(clk, reset, RD1, A);
  wdreg: flopr generic map(32)
    port map(clk, reset, RD2, WriteData);
  ext: extend port map(Instr(23 downto 0), ImmSrc, ExtImm);

  -- ALU logic

```

```

srcamux: mux2 generic map(32)
  port map(A, PC, ALUSrcA, SrcA);

-- ASR, ROR
srcbmux: mux3 generic map (32)
  port map(srcBshifted, ExtImm, 32X"00000004", ALUSrcB, SrcB);
sh: shifter port map(WriteData, Instr(11 downto 7), Instr(6 downto 5),
srcBshifted);
i_alu: alu port map(SrcA, SrcB, ALUControl, ALUResult, ALUFlags, carry);
alureultmux: mux2 generic map(32)
  port map(ALUResult, SrcB, Shift, ALUResultOut);
aluoutreg: flopr generic map (32)
  port map(clk, reset, ALUResultOut, ALUOut);
resmux: mux3 generic map(32)
  port map(ALUOut, Data, ALUResultOut, ResultSrc, Result);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity regfile is -- three-port register file
  port(clk:          in  STD_LOGIC;
        we3:         in  STD_LOGIC;
        ra1, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
        wd3, r15:     in  STD_LOGIC_VECTOR(31 downto 0);
        rd1, rd2:     out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
  type ramtype is array (31 downto 0) of
    STD_LOGIC_VECTOR(31 downto 0);
  signal mem: ramtype;
begin
  process(clk) begin
    if rising_edge(clk) then
      if we3 = '1' then mem(to_integer(wa3)) <= wd3;
      end if;
    end if;
  end process;
  process(all) begin
    if (to_integer(ra1) = 15) then rd1 <= r15;
    else rd1 <= mem(to_integer(ra1));
    end if;
    if (to_integer(ra2) = 15) then rd2 <= r15;
    else rd2 <= mem(to_integer(ra2));
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity adder is -- adder
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
        y:   out STD_LOGIC_VECTOR(31 downto 0));
end;

```

```

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

```

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity extend is
    port(Instr: in  STD_LOGIC_VECTOR(23 downto 0);
         ImmSrc: in  STD_LOGIC_VECTOR(1 downto 0);
         ExtImm: out STD_LOGIC_VECTOR(31 downto 0));
end;

```

```

architecture behave of extend is
begin
    process(all) begin
        case ImmSrc is
            when "00" => ExtImm <= (X"000000", Instr(7 downto 0));
            when "01" => ExtImm <= (X"00000", Instr(11 downto 0));
            when "10" => ExtImm <= (Instr(23), Instr(23), Instr(23),
                                     Instr(23), Instr(23), Instr(23), Instr(23 downto 0), "00");
            when others => ExtImm <= X"-----";
        end case;
    end process;
end;

```

```

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

```

```

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

```

```

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

```



```

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

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

architecture behave of mux2 is
begin
  y <= d1 when s else d0;
end;

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

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

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity alu is
  port(a, b:      in  STD_LOGIC_VECTOR(31 downto 0);
       ALUControl: in  STD_LOGIC_VECTOR(2 downto 0); -- SBC
       Result:    buffer STD_LOGIC_VECTOR(31 downto 0);
       ALUFlags:  out STD_LOGIC_VECTOR(3 downto 0);
       carry:     in  STD_LOGIC); -- SBC

```

```
end;
```

```
architecture behave of alu is
```

```
    signal condinvb: STD_LOGIC_VECTOR(31 downto 0);
    signal sum:      STD_LOGIC_VECTOR(32 downto 0);
    signal neg, zero, carryout, overflow: STD_LOGIC;
    signal carryin: STD_LOGIC; -- SBC
```

```
begin
```

```
    carryin <= carry when ALUControl(2) else ALUControl(0);
    condinvb <= not b when ALUControl(0) else b;
    sum <= ('0', a) + ('0', condinvb) + carryin;
```

```
    process(all) begin
```

```
        case? ALUControl(1 downto 0) is
            when "0-" => result <= sum(31 downto 0);
            when "10" => result <= a and b;
            when "11" => result <= a or b;
            when others => result <= (others => '-');
```

```
        end case?;
```

```
    end process;
```

```
    neg      <= Result(31);
    zero     <= '1' when (Result = 0) else '0';
    carryout <= (not ALUControl(1)) and sum(32);
    overflow <= (not ALUControl(1)) and
                (not (a(31) xor b(31) xor ALUControl(0))) and
                (a(31) xor sum(31));
```

```
    ALUFlags <= (neg, zero, carryout, overflow);
```

```
end;
```

```
-- shifter needed for ASR, ROR
```

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

```
use IEEE.NUMERIC_STD_UNSIGNED.all;
```

```
entity shifter is
```

```
    port(a:      in  STD_LOGIC_VECTOR(31 downto 0);
          shamt:  in  STD_LOGIC_VECTOR(4  downto 0);
          shtype: in  STD_LOGIC_VECTOR(1  downto 0);
          y:      out STD_LOGIC_VECTOR(31 downto 0));
```

```
end;
```

```
architecture behave of shifter is
```

```
begin
```

```
    process (all) begin
```

```
        case shtype is
```

```
            when "10" => y <= TO_STDLOGICVECTOR(TO_BITVECTOR(a) sra
TO_INTEGER(shamt));
            when "11" => y <= ( (TO_STDLOGICVECTOR(TO_BITVECTOR(a) srl
TO_INTEGER(shamt))) or (TO_STDLOGICVECTOR(TO_BITVECTOR(a) sll (32-
TO_INTEGER(shamt)))) );
```

```
            when others => y <= a;
```

```
        end case;
```

```
    end process;
```

```
end;
```

Test ARM assembly

// If successful, it should write the value 0x2FFFFFFE to address 0x58

```

MAIN
    SUB R3, PC, PC           ; R3 = 0
    SUB R4, R3, #30          ; R4 = -30 (0xFFFFFFFFE2)
    ASR R5, R4, #1           ; R5 = -15 (0xFFFFFFFFF1)
    TST R4, R5               ; set flags based on R4 & R5: NZCV=1000
    ADDMIS R6, R4, R5        ; R6 = -30 + (-15)=-45 (0xFFFFFD3) if N = 1
    (should happen)

    SBCS R7, R5, R6          ; also set flags: NZCV=1010
                             ; R7 = -15 - (-45) - 0 = 30 (0x1E)
                             ; also set flags: NZCV = 0010
    ADDS R3, R3, #25         ; R3 = 25, set flags: NZCV = 0000
    SBC R8, R7, R5           ; R8 = 30 - (-15) - 1 = 44 (0x2c)
    ROR R9, R4, #4           ; R9 = 0xFFFFFFFFE2 ROR 4 = 0x2FFFFFFE
    STR R9, [R8, #0x2c]      ; mem[0x30] <= 0x2FFFFFFE

;0x00  E04F300F  SUB      R3,PC,PC
;0x04  E243401E  SUB      R4,R3,#0x1E
;0x08  E1A050C4  ASR      R5,R4,#1
;0x0C  E1140005  TST      R4,R5
;0x10  40946005  ADDMIS   R6,R4,R5
;0x14  E0D57006  SBCS     R7,R5,R6
;0x18  E2933019  ADDS     R3,R3,#0x19
;0x1C  E0C78005  SBC      R8,R7,R5
;0x20  E1A09264  ROR      R9,R4,#4
;0x24  E588902C  STR      R9,[R8,#0x2C

```

ex7.27_memfile.dat

```

E04F300F
E243401E
E1A050C4
E1140005
40946005
E0D57006
E2933019
E0C78005
E1A09264
E588902C

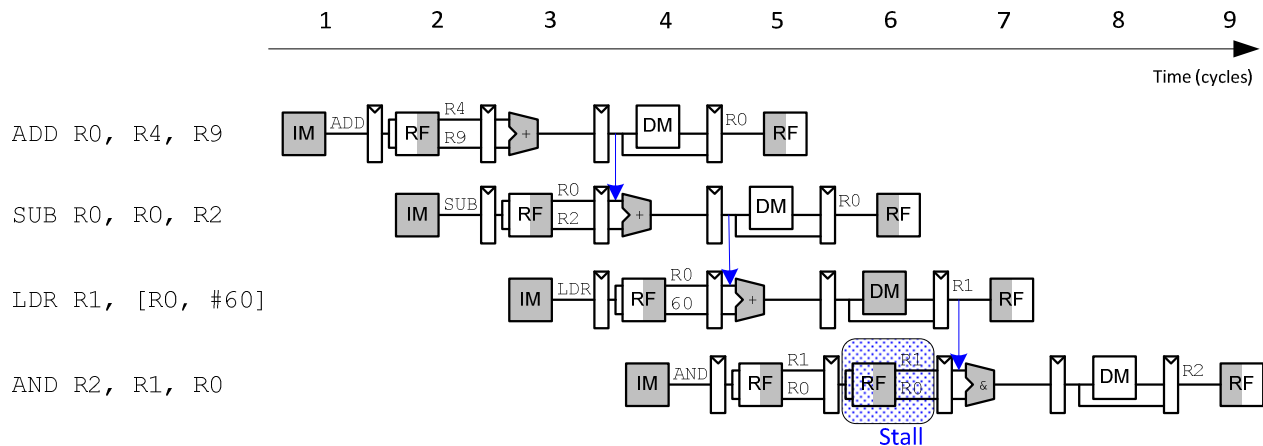
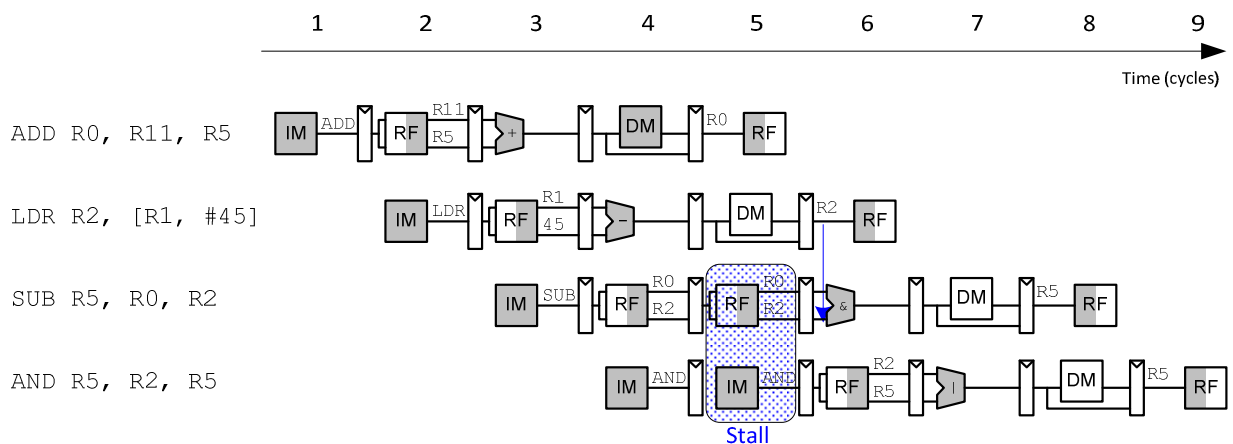
```

Exercise 7.28

In cycle 5, R1 is being both read and written. R1 is written by the MOV instruction and read (in the second half of the cycle) by the STR instruction.

Exercise 7.29

In cycle 5, R0 is being written (by ADD) and registers R2 and R5 are being read (by ORR).

Exercise 7.30**Exercise 7.31****Exercise 7.32**

34 cycles are required for the pipelined ARM processor to issue all of the instructions: 2 cycles for the first two MOV instructions, 6 cycles for each of the 5 loop iterations (4 for fetching instructions and 2 for the branch delay penalty), and 2 for the final CMP and BEQ that branches out of the loop.

The number of instructions fetched is $2 + 5 \cdot 4 + 2 = 24$ instructions. Thus, CPI is $34 \text{ c.c.} / 24 \text{ instr} = \mathbf{1.42}$.

Exercise 7.33

75 cycles are required for the pipelined ARM processor to issue all of the instructions: 3 cycles for the first three MOV instructions, 7 cycles for each of the 10 loop iterations (5 for fetching instructions and 2 for the branch delay penalty), and 2 for the final CMP and BEQ that branches out of the loop.

The number of instructions fetched is $3 + 10 \cdot 5 + 2 = 55$ instructions. Thus, CPI is $75 \text{ c.c.} / 55 \text{ instr} = \mathbf{1.36}$.

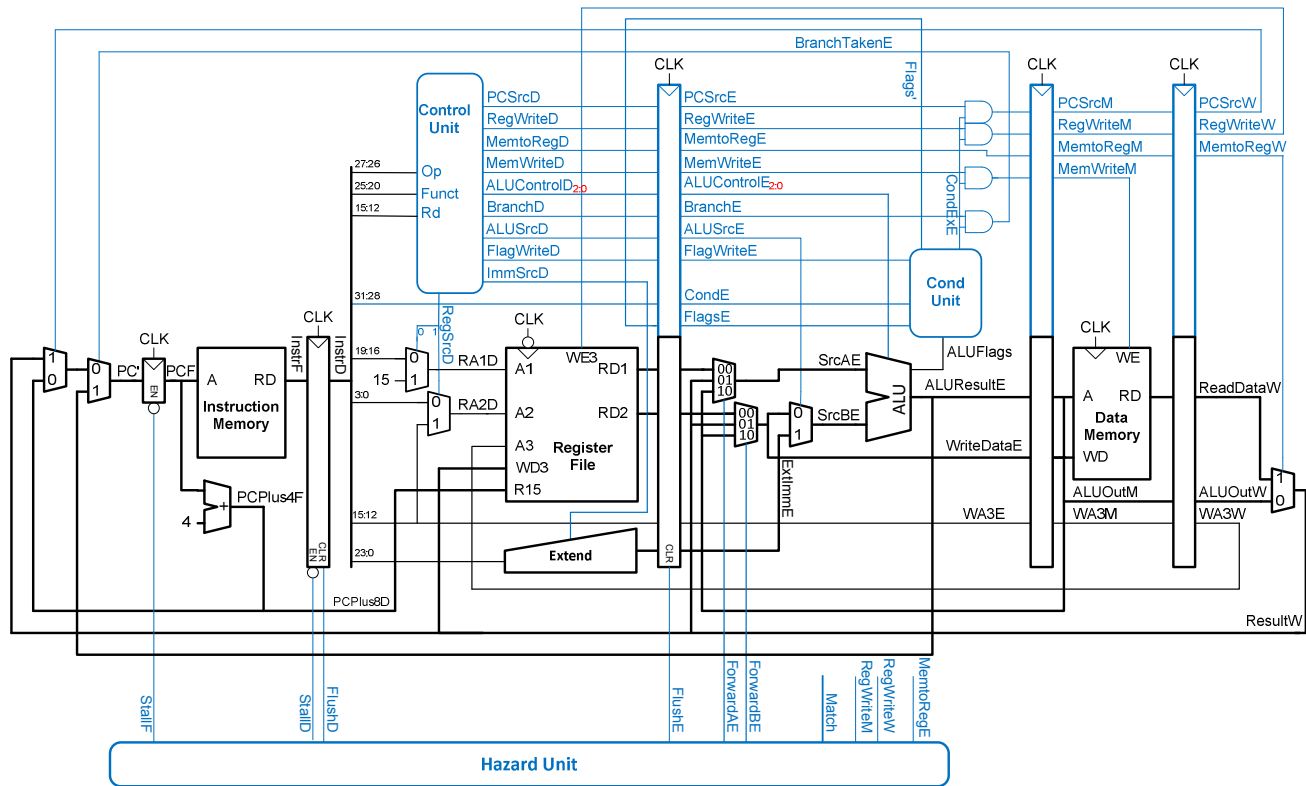
Exercise 7.34

Changes to the pipelined processor for the EOR instruction.

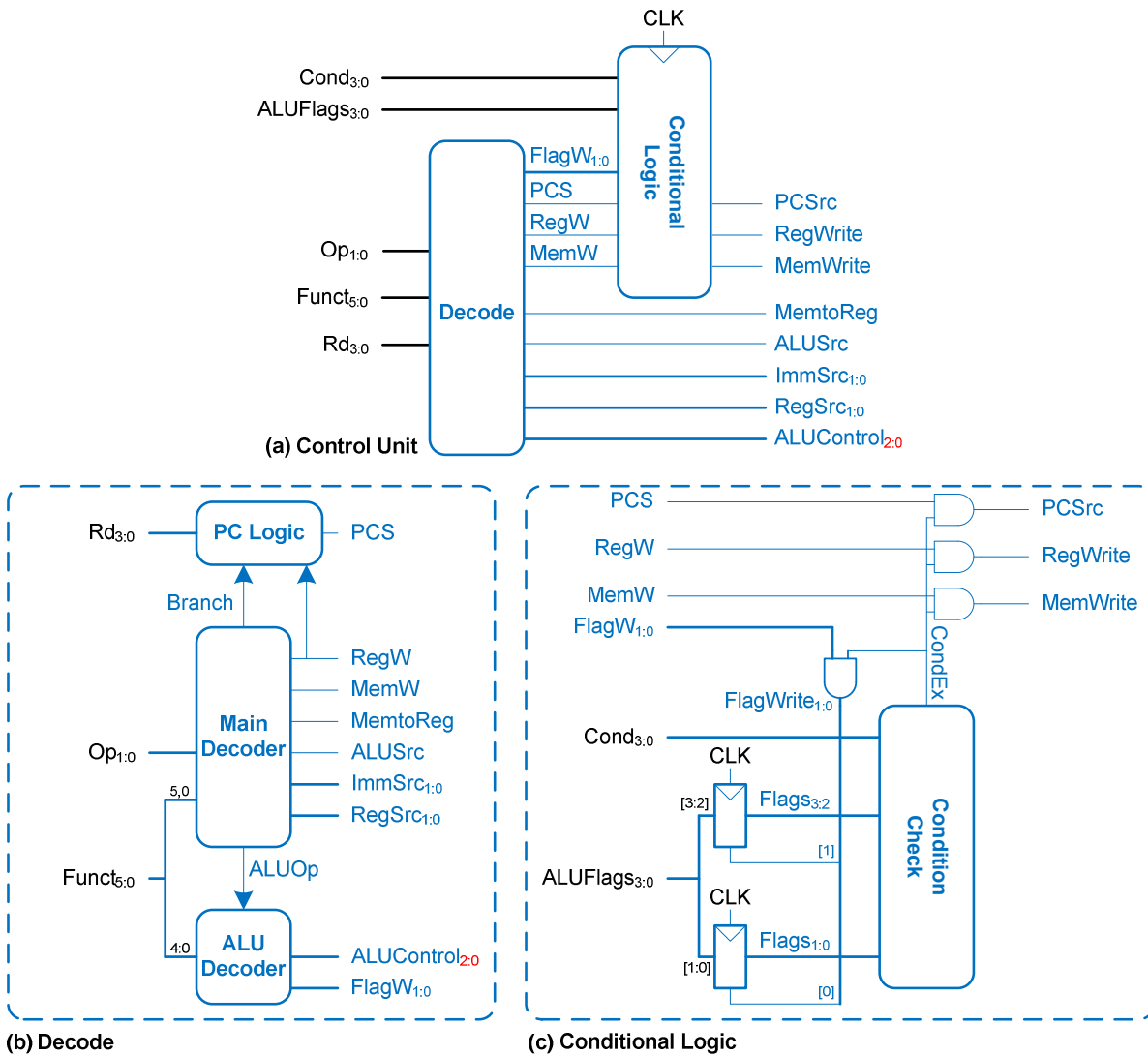
ALU Decoder truth table:

<i>ALUOp</i>	<i>Funct_{4:1} (cmd)</i>	<i>Funct₀ (S)</i>	Notes	<i>ALUControl_{2:0}</i>	<i>FlagW_{1:0}</i>
0	X	X	Not DP	000	00
1	0100	0	ADD	000	00
		1			11
	0010	0	SUB	001	00
		1			11
	0000	0	AND	010	00
		1			10
	1100	0	ORR	011	00
		1			10
	0001	0	EOR	100	00
		1			10

ALU



Control



Exercise 7.35

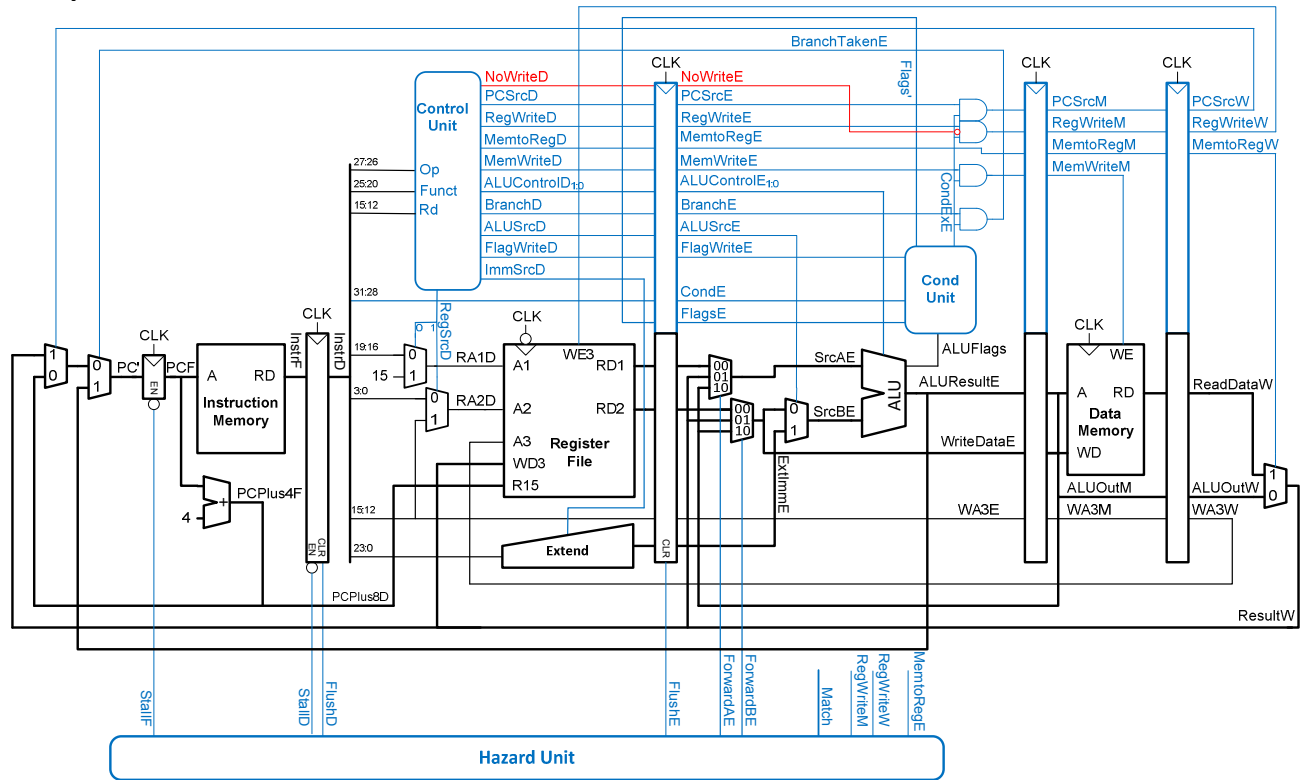
Changes to the pipelined processor for the CMN instruction.

ALU Decoder truth table

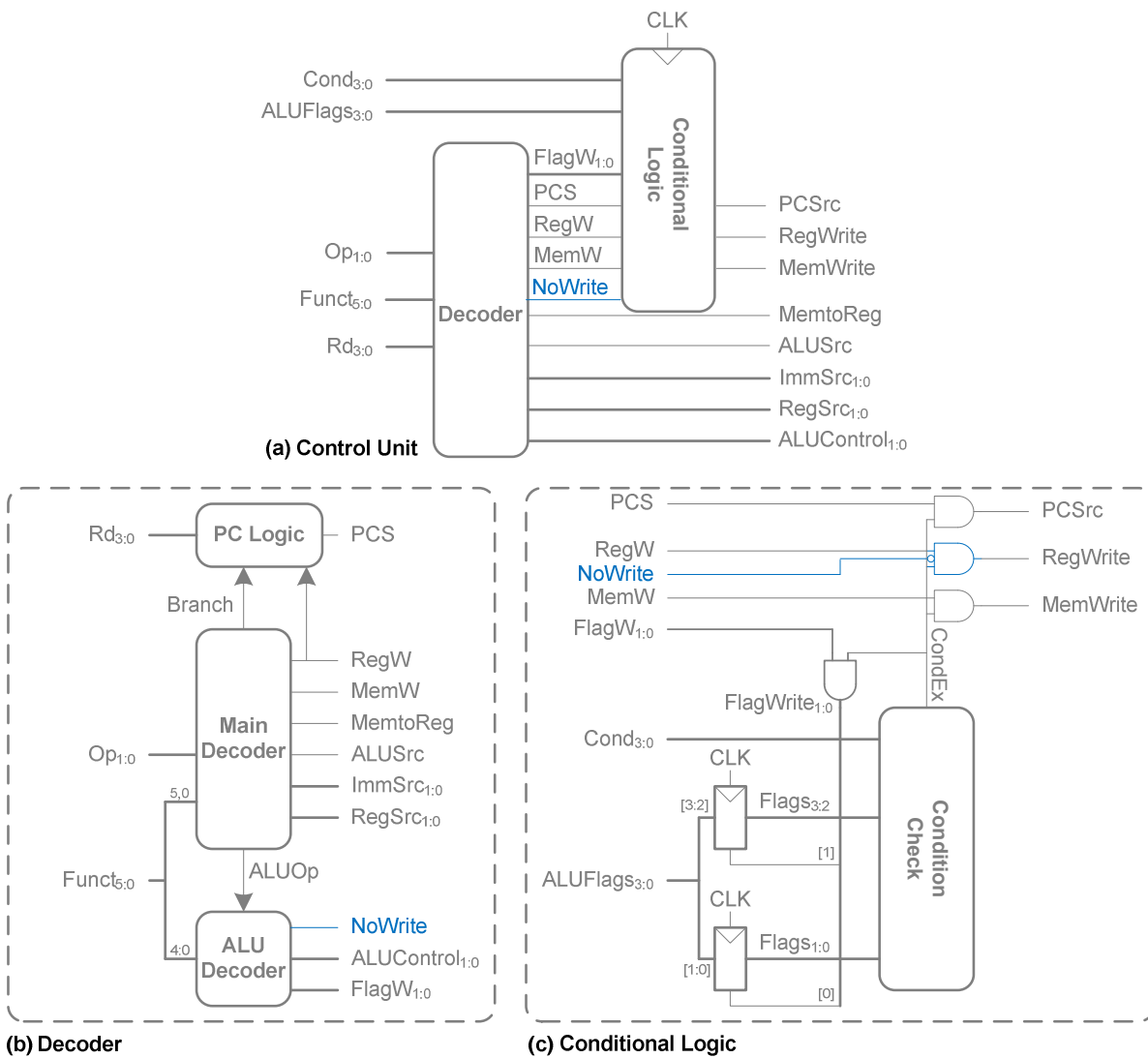
$ALUOp$	$Func_{4:1} (cmd)$	$Func_0 (S)$	Notes	$ALUControl_{1:0}$	$FlagW_{1:0}$	NoWrite
0	X	X	Not DP	00	00	0
1	0100	0	ADD	00	00	0
		1			11	0
	0010	0	SUB	01	00	0
		1			11	0
	0000	0	AND	10	00	0
		1			10	0

	1100	0	ORR	11	00	0
		1			10	0
	1011	1	CMN	00	11	1

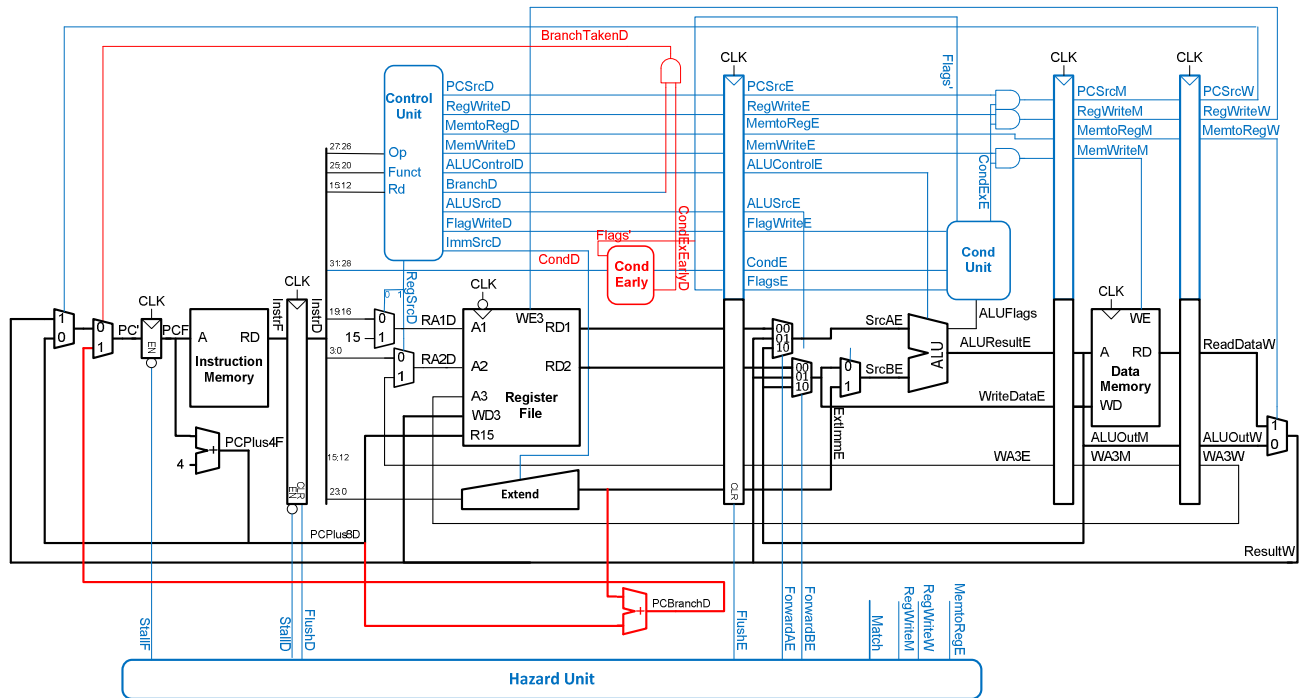
Datapath



Control



Exercise 7.36



Flushing hardware changes to:

```
FlushE = LDRstall
FlushD = PCWrPendingF + PCSrcW + BranchTakenD
```

Exercise 7.37

She should work on the **register file** because it is the unit that's in the critical path (Decode stage) causing the cycle time (T_{c3}) to be 300 ps. The next longest paths are 290 ps (for the Fetch stage and for the Memory stage). Reducing the register file read delay by 5 ps (to 95 ps) reduces the cycle time to 290 ps (see Equation 7.5). Reducing the delay any more would not improve performance any further. Thus, $t_{R\text{Fread}}$ should be reduced to **95 ps**, and the resulting cycle time, T_{c3} , is $2(t_{R\text{Fread}} + t_{\text{setup}}) = 2(95 + 50) \text{ ps} = \mathbf{290 \text{ ps}}$.

Exercise 7.38

No, the cycle time would not change if the **ALU were 20% faster**, because the ALU is not in the critical path.

No, the cycle time would not change if the **ALU were 20% slower**. With a 20% slowdown, the ALU's delay would be $(120 \text{ ps}) \cdot 1.2 = 144 \text{ ps}$. This would make the delay of the execute stage $(40 + 2(25) + 144 + 50) \text{ ps} = \mathbf{284 \text{ ps}}$ (see Equation 7.5). This still isn't longer than the current critical path (the Decode stage) which results in a cycle time of 300 ps.

Exercise 7.39

Suppose the ARM pipelined processor is divided into 10 stages of 400 ps each, including sequencing overhead. Assume the instruction mix of Example 7.7. Also assume that 50% of the loads are immediately followed by an instruction that uses the result, requiring six stalls, and that 30% of the branches are mispredicted. The target address of a branch instruction is not computed until the end of the second stage. Calculate the average CPI and execution time of computing 100 billion instructions from the SPECINT2000 benchmark for this 10-stage pipelined processor.

$$\text{CPI} = 0.25(1+0.5*6) + 0.1(1) + 0.13(1+0.3*1)+0.52(1) = 1.789 \approx \mathbf{1.8}$$

$$\text{Execution Time} = (100 \times 10^9 \text{ instructions})(1.789 \text{ cycles/instruction})(400 \times 10^{-12} \text{ s/cycle}) = 71.56 \text{ s} \approx \mathbf{72 \text{ s}}$$

Exercise 7.40

SystemVerilog

```
// ARM 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 === 7) begin
                    $display("Simulation succeeded");
                end
            end
        end
endmodule
```

```

        $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
    arm arm(clk, reset, PCF, InstrF, MemWriteM, DataAdrM,
            WriteDataM, ReadDataM);
    imem imem(PCF, InstrF);
    dmem dmem(clk, MemWriteM, DataAdrM, WriteDataM, ReadDataM);
endmodule

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

    logic [31:0] RAM[2097151:0];

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

    assign rd = RAM[a[22:2]]; // word aligned

    always_ff @(posedge clk)
        if (we) RAM[a[22:2]] <= wd;
endmodule

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

    logic [31:0] RAM[2097151:0];

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

    assign rd = RAM[a[22:2]]; // word aligned
endmodule

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

```

```

logic [1:0]  RegSrcD, ImmSrcD, ALUControlE;
logic      ALUSrcE, BranchTakenE, MemtoRegW, PCSrcW, RegWriteW;
logic [3:0] ALUFlagsE;
logic [31:0] InstrD;
logic      RegWriteM, MemtoRegE, PCWrPendingF;
logic [1:0] ForwardAE, ForwardBE;
logic      StallF, StallD, FlushD, FlushE;
logic      Match_1E_M, Match_1E_W, Match_2E_M, Match_2E_W,
Match_12D_E;

controller c(clk, reset, InstrD[31:12], ALUFlagsE,
             RegSrcD, ImmSrcD,
             ALUSrcE, BranchTakenE, ALUControlE,
             MemWriteM,
             MemtoRegW, PCSrcW, RegWriteW,
             RegWriteM, MemtoRegE, PCWrPendingF,
             FlushE);
datapath dp(clk, reset,
            RegSrcD, ImmSrcD,
            ALUSrcE, BranchTakenE, ALUControlE,
            MemtoRegW, PCSrcW, RegWriteW,
            PCF, InstrF, InstrD,
            ALUOutM, WriteDataM, ReadDataM,
            ALUFlagsE,
            Match_1E_M, Match_1E_W, Match_2E_M, Match_2E_W,
Match_12D_E,
            ForwardAE, ForwardBE, StallF, StallD, FlushD);
hazard h(clk, reset, Match_1E_M, Match_1E_W, Match_2E_M, Match_2E_W,
Match_12D_E,
         RegWriteM, RegWriteW, BranchTakenE, MemtoRegE,
         PCWrPendingF, PCSrcW,
         ForwardAE, ForwardBE,
         StallF, StallD, FlushD, FlushE);

endmodule

module controller(input  logic      clk, reset,
                 input  logic [31:12] InstrD,
                 input  logic [3:0]  ALUFlagsE,
                 output logic [1:0]   RegSrcD, ImmSrcD,
                 output logic         ALUSrcE, BranchTakenE,
                 output logic [1:0]   ALUControlE,
                 output logic         MemWriteM,
                 output logic         MemtoRegW, PCSrcW, RegWriteW,
                 // hazard interface
                 output logic         RegWriteM, MemtoRegE,
                 output logic         PCWrPendingF,
                 input  logic         FlushE);

logic [9:0] controlsD;
logic      CondExE, ALUOpD;
logic [1:0] ALUControlD;
logic      ALUSrcD;

```

```

logic      MemtoRegD, MemtoRegM;
logic      RegWriteD, RegWriteE, RegWriteGatedE;
logic      MemWriteD, MemWriteE, MemWriteGatedE;
logic      BranchD, BranchE;
logic [1:0] FlagWriteD, FlagWriteE;
logic      PCSrcD, PCSrcE, PCSrcM;
logic [3:0] FlagsE, FlagsNextE, ConDE;

// Decode stage

always_comb
    casex(InstrD[27:26])
        2'b00: if (InstrD[25]) controlsD = 10'b0000101001; // DP imm
                else                controlsD = 10'b0000001001; // DP reg
        2'b01: if (InstrD[20]) controlsD = 10'b0001111000; // LDR
                else                controlsD = 10'b1001110100; // STR
        2'b10:                controlsD = 10'b0110100010; // B
        default:                controlsD = 10'bx;          //
    unimplemented
    endcase

assign {RegSrcD, ImmSrcD, ALUSrcD, MemtoRegD,
        RegWriteD, MemWriteD, BranchD, ALUOpD} = controlsD;

always_comb
    if (ALUOpD) begin                // which Data-processing Instr?
        case(InstrD[24:21])
            4'b0100: ALUControlD = 2'b00; // ADD
            4'b0010: ALUControlD = 2'b01; // SUB
            4'b0000: ALUControlD = 2'b10; // AND
            4'b1100: ALUControlD = 2'b11; // ORR
            default: ALUControlD = 2'bx;  // unimplemented
        endcase
        FlagWriteD[1] = InstrD[20]; // update N and Z Flags if S bit is
set
        FlagWriteD[0] = InstrD[20] & (ALUControlD == 2'b00 | ALUControlD
== 2'b01);
        end else begin
            ALUControlD = 2'b00; // perform addition for non-
dataprocessing instr
            FlagWriteD = 2'b00; // don't update Flags
        end

        assign PCSrcD = (((InstrD[15:12] == 4'b1111) & RegWriteD) |
BranchD);

// Execute stage
floprrc #(7) flushedregsE(clk, reset, FlushE,
                        {FlagWriteD, BranchD, MemWriteD, RegWriteD,
PCSrcD, MemtoRegD},
                        {FlagWriteE, BranchE, MemWriteE, RegWriteE,
PCSrcE, MemtoRegE});
floprr #(3) regsE(clk, reset,
                  {ALUSrcD, ALUControlD},

```

```

        {ALUSrcE, ALUControlE});

floprr # (4) condregE (clk, reset, InstrD[31:28], CondeE);
floprr # (4) flagsreg (clk, reset, FlagsNextE, FlagsE);

// write and Branch controls are conditional
conditional Cond (CondeE, FlagsE, ALUFlagsE, FlagWriteE, CondExE,
FlagsNextE);
assign BranchTakenE    = BranchE & CondExE;
assign RegWriteGatedE  = RegWriteE & CondExE;
assign MemWriteGatedE  = MemWriteE & CondExE;
assign PCSrcGatedE     = PCSrcE & CondExE;

// Memory stage
floprr # (4) regsM (clk, reset,
                    {MemWriteGatedE, MemtoRegE, RegWriteGatedE,
PCSrcGatedE},
                    {MemWriteM, MemtoRegM, RegWriteM, PCSrcM});

// Writeback stage
floprr # (3) regsW (clk, reset,
                    {MemtoRegM, RegWriteM, PCSrcM},
                    {MemtoRegW, RegWriteW, PCSrcW});

// Hazard Prediction
assign PCWrPendingF = PCSrcD | PCSrcE | PCSrcM;

endmodule

module conditional (input  logic [3:0] Cond,
                   input  logic [3:0] Flags,
                   input  logic [3:0] ALUFlags,
                   input  logic [1:0] FlagsWrite,
                   output logic      CondEx,
                   output logic [3:0] FlagsNext);

    logic neg, zero, carry, overflow, ge;

    assign {neg, zero, carry, overflow} = Flags;
    assign ge = (neg == overflow);

    always_comb
        case (Cond)
            4'b0000: CondEx = zero;           // EQ
            4'b0001: CondEx = ~zero;         // NE
            4'b0010: CondEx = carry;         // CS
            4'b0011: CondEx = ~carry;        // CC
            4'b0100: CondEx = neg;           // MI
            4'b0101: CondEx = ~neg;          // PL
            4'b0110: CondEx = overflow;      // VS
            4'b0111: CondEx = ~overflow;     // VC
            4'b1000: CondEx = carry & ~zero; // HI
            4'b1001: CondEx = ~(carry & ~zero); // LS
            4'b1010: CondEx = ge;           // GE
        endcase

```



```

        4'b1011: CondEx = ~ge;                // LT
        4'b1100: CondEx = ~zero & ge;        // GT
        4'b1101: CondEx = ~(~zero & ge);     // LE
        4'b1110: CondEx = 1'b1;              // Always
        default: CondEx = 1'bx;              // undefined
    endcase

    assign FlagsNext[3:2] = (FlagsWrite[1] & CondEx) ? ALUFlags[3:2] :
Flags[3:2];
    assign FlagsNext[1:0] = (FlagsWrite[0] & CondEx) ? ALUFlags[1:0] :
Flags[1:0];
endmodule

module datapath(input logic clk, reset,
                input logic [1:0] RegSrcD, ImmSrcD,
                input logic ALUSrcE, BranchTakenE,
                input logic [1:0] ALUControlE,
                input logic MemtoRegW, PCSrcW, RegWriteW,
                output logic [31:0] PCF,
                input logic [31:0] InstrF,
                output logic [31:0] InstrD,
                output logic [31:0] ALUOutM, WriteDataM,
                input logic [31:0] ReadDataM,
                output logic [3:0] ALUFlagsE,
                // hazard logic
                output logic Match_1E_M, Match_1E_W, Match_2E_M,
Match_2E_W, Match_12D_E,
                input logic [1:0] ForwardAE, ForwardBE,
                input logic StallF, StallD, FlushD);

    logic [31:0] PCPlus4F, PCnext1F, PCnextF;
    logic [31:0] ExtImmD, rd1D, rd2D, PCPlus8D;
    logic [31:0] rd1E, rd2E, ExtImmE, SrcAE, SrcBE, WriteDataE, ALUResultE;
    logic [31:0] ReadDataW, ALUOutW, ResultW;
    logic [3:0] RA1D, RA2D, RA1E, RA2E, WA3E, WA3M, WA3W;
    logic Match_1D_E, Match_2D_E;

    // Fetch stage
    mux2 #(32) pcnextmux(PCPlus4F, ResultW, PCSrcW, PCnext1F);
    mux2 #(32) branchmux(PCnext1F, ALUResultE, BranchTakenE, PCnextF);
    flopenr #(32) pcreg(clk, reset, ~StallF, PCnextF, PCF);
    adder #(32) pcadd(PCF, 32'h4, PCPlus4F);

    // Decode Stage
    assign PCPlus8D = PCPlus4F; // skip register
    flopenrc #(32) instrreg(clk, reset, ~StallD, FlushD, InstrF, InstrD);
    mux2 #(4) ralmux(InstrD[19:16], 4'b1111, RegSrcD[0], RA1D);
    mux2 #(4) ra2mux(InstrD[3:0], InstrD[15:12], RegSrcD[1], RA2D);
    regfile rf(clk, RegWriteW, RA1D, RA2D,
                WA3W, ResultW, PCPlus8D,
                rd1D, rd2D);
    extend ext(InstrD[23:0], ImmSrcD, ExtImmD);

```

```

// Execute Stage
flop1 # (32) rd1reg(clk, reset, rd1D, rd1E);
flop2 # (32) rd2reg(clk, reset, rd2D, rd2E);
flop3 # (32) immreg(clk, reset, ExtImmD, ExtImmE);
flop4 # (4) wa3ereg(clk, reset, InstrD[15:12], WA3E);
flop5 # (4) ralreg(clk, reset, RA1D, RA1E);
flop6 # (4) ra2reg(clk, reset, RA2D, RA2E);
mux3 # (32) byp1mux(rd1E, ResultW, ALUOutM, ForwardAE, SrcAE);
mux3 # (32) byp2mux(rd2E, ResultW, ALUOutM, ForwardBE, WriteDataE);
mux2 # (32) srcbmux(WriteDataE, ExtImmE, ALUSrcE, SrcBE);
alu      alu(SrcAE, SrcBE, ALUControlE, ALUResultE, ALUFlagsE);

// Memory Stage
flop1 # (32) aluresreg(clk, reset, ALUResultE, ALUOutM);
flop2 # (32) wdreg(clk, reset, WriteDataE, WriteDataM);
flop3 # (4) wa3mreg(clk, reset, WA3E, WA3M);

// Writeback Stage
flop1 # (32) aluoutreg(clk, reset, ALUOutM, ALUOutW);
flop2 # (32) rdreg(clk, reset, ReadDataM, ReadDataW);
flop3 # (4) wa3wreg(clk, reset, WA3M, WA3W);
mux2 # (32) resmux(ALUOutW, ReadDataW, MemtoRegW, ResultW);

// hazard comparison
eqcmp # (4) m0(WA3M, RA1E, Match_1E_M);
eqcmp # (4) m1(WA3W, RA1E, Match_1E_W);
eqcmp # (4) m2(WA3M, RA2E, Match_2E_M);
eqcmp # (4) m3(WA3W, RA2E, Match_2E_W);
eqcmp # (4) m4a(WA3E, RA1D, Match_1D_E);
eqcmp # (4) m4b(WA3E, RA2D, Match_2D_E);
assign Match_12D_E = Match_1D_E | Match_2D_E;

endmodule

module hazard(input  logic      clk, reset,
              input  logic      Match_1E_M, Match_1E_W, Match_2E_M,
              Match_2E_W, Match_12D_E,
              input  logic      RegWriteM, RegWriteW,
              input  logic      BranchTakenE, MemtoRegE,
              input  logic      PCWrPendingF, PCSrcW,
              output logic [1:0] ForwardAE, ForwardBE,
              output logic      StallF, StallD,
              output logic      FlushD, FlushE);

    logic ldrStallD;

    // forwarding logic
    always_comb begin
        if (Match_1E_M & RegWriteM)      ForwardAE = 2'b10;
        else if (Match_1E_W & RegWriteW) ForwardAE = 2'b01;
        else                             ForwardAE = 2'b00;

        if (Match_2E_M & RegWriteM)      ForwardBE = 2'b10;
        else if (Match_2E_W & RegWriteW) ForwardBE = 2'b01;
    end

```

```

        else                                ForwardBE = 2'b00;
    end

    // stalls and flushes
    // Load RAW
    //   when an instruction reads a register loaded by the previous,
    //   stall in the decode stage until it is ready
    // Branch hazard
    //   When a branch is taken, flush the incorrectly fetched instrs
    //   from decode and execute stages
    // PC Write Hazard
    //   When the PC might be written, stall all following instructions
    //   by stalling the fetch and flushing the decode stage
    // when a stage stalls, stall all previous and flush next

    assign ldrStallD = Match_12D_E & MemtoRegE;

    assign StallD = ldrStallD;
    assign StallF = ldrStallD | PCWrPendingF;
    assign FlushE = ldrStallD | BranchTakenE;
    assign FlushD = PCWrPendingF | PCSrcW | BranchTakenE;

endmodule

module regfile(input  logic      clk,
               input  logic      we3,
               input  logic [3:0] ra1, ra2, wa3,
               input  logic [31:0] wd3, r15,
               output logic [31:0] rd1, rd2);

    logic [31:0] rf[14:0];

    // three ported register file
    // read two ports combinationaly
    // write third port on falling edge of clock (midcycle)
    //   so that writes can be read on same cycle
    // register 15 reads PC+8 instead

    always_ff @(negedge clk)
        if (we3) rf[wa3] <= wd3;

    assign rd1 = (ra1 == 4'b1111) ? r15 : rf[ra1];
    assign rd2 = (ra2 == 4'b1111) ? r15 : rf[ra2];
endmodule

module extend(input  logic [23:0] Instr,
              input  logic [1:0] ImmSrc,
              output logic [31:0] ExtImm);

    always_comb
        case(ImmSrc)
            2'b00: ExtImm = {24'b0, Instr[7:0]}; // 8-bit unsigned immediate
            2'b01: ExtImm = {20'b0, Instr[11:0]}; // 12-bit unsigned immediate
            2'b10: ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00}; // Branch
        endcase
endmodule

```

```

        default: ExtImm = 32'bx; // undefined
    endcase
endmodule

module alu(input  logic [31:0] a, b,
           input  logic [1:0] ALUControl,
           output logic [31:0] Result,
           output logic [3:0]  Flags);

    logic      neg, zero, carry, overflow;
    logic [31:0] condinvb;
    logic [32:0] sum;

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

    always_comb
        casex (ALUControl[1:0])
            2'b0?: Result = sum;
            2'b10: Result = a & b;
            2'b11: Result = a | b;
        endcase

    assign neg      = Result[31];
    assign zero     = (Result == 32'b0);
    assign carry    = (ALUControl[1] == 1'b0) & sum[32];
    assign overflow = (ALUControl[1] == 1'b0) & ~(a[31] ^ b[31] ^
ALUControl[0]) &
                                                                    (a[31] ^ sum[31]);

    assign Flags = {neg, zero, carry, overflow};
endmodule

module adder #(parameter WIDTH=8)
    (input  logic [WIDTH-1:0] a, b,
     output logic [WIDTH-1:0] y);

    assign y = a + b;
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 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 flopenrc #(parameter WIDTH = 8)
    (input  logic          clk, reset, en, 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 (en)
            if (clear) q <= 0;
            else      q <= d;
endmodule

module floprc #(parameter WIDTH = 8)
    (input  logic          clk, reset, 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 eqcmp #(parameter WIDTH = 8)
    (input  logic [WIDTH-1:0] a, b,
     output logic          y);

    assign y = (a == b);
endmodule

```

VHDL

```

library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity testbench is
end;

architecture test of testbench is
  component top
    port (clk, reset:          in  STD_LOGIC;
          WriteDataM, DataAdrM: out STD_LOGIC_VECTOR(31 downto 0);
          MemWriteM:          out STD_LOGIC);
  end component;
  signal WriteData, DataAdr:    STD_LOGIC_VECTOR(31 downto 0);
  signal clk, reset, MemWrite:  STD_LOGIC;
begin

  -- instantiate device to be tested
  dut: top port map(clk, reset, WriteData, DataAdr, MemWrite);

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

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

  -- check that 7 gets written to address 84
  -- at end of program
  process (clk) begin
    if (clk'event and clk = '0' and MemWrite = '1') then
      if (to_integer(DataAdr) = 100 and
          to_integer(WriteData) = 7) then
        report "NO ERRORS: Simulation succeeded" severity failure;
      elsif (DataAdr /= 96) then
        report "Simulation failed" severity failure;
      end if;
    end if;
  end process;
end;

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

```

```

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

architecture test of top is
    component arm
        port(clk, reset:          in      STD_LOGIC;
              PCF:                out    STD_LOGIC_VECTOR(31 downto 0);
              InstrF:             in      STD_LOGIC_VECTOR(31 downto 0);
              MemWriteM:          out    STD_LOGIC;
              ALUOutM, WriteDataM: out    STD_LOGIC_VECTOR(31 downto 0);
              ReadDataM:          in      STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component imem
        port(a: in  STD_LOGIC_VECTOR(31 downto 0);
              rd: out STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component dmem
        port(clk, we: in  STD_LOGIC;
              a, wd:  in  STD_LOGIC_VECTOR(31 downto 0);
              rd:     out STD_LOGIC_VECTOR(31 downto 0));
    end component;
    signal PCF, InstrF, ReadDataM: STD_LOGIC_VECTOR(31 downto 0);
begin
    -- instantiate processor and memories
    i_arm: arm port map(clk, reset, PCF, InstrF, MemWriteM, DataAdrM,
                       WriteDataM, ReadDataM);
    i_imem: imem port map(PCF, InstrF);
    i_dmem: dmem port map(clk, MemWriteM, DataAdrM, WriteDataM, ReadDataM);
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity imem is -- instruction memory
    port(a: in  STD_LOGIC_VECTOR(31 downto 0);
          rd: out STD_LOGIC_VECTOR(31 downto 0));
end;
architecture behave of imem is -- instruction memory
begin
    process is
        file mem_file: TEXT;
        variable L: line;
        variable ch: character;
        variable i, index, result: integer;
        type ramtype is array (63 downto 0) of
            STD_LOGIC_VECTOR(31 downto 0);
        variable mem: ramtype;
    begin
        -- initialize memory from file
        for i in 0 to 63 loop -- set all contents low
            mem(i) := (others => '0');
        end loop;
    end process;
end;

```

```

end loop;
index := 0;
FILE_OPEN(mem_file, "memfile.dat", READ_MODE);
while not endfile(mem_file) loop
  readline(mem_file, L);
  result := 0;
  for i in 1 to 8 loop
    read(L, ch);
    if '0' <= ch and ch <= '9' then
      result := character'pos(ch) - character'pos('0');
    elsif 'a' <= ch and ch <= 'f' then
      result := character'pos(ch) - character'pos('a')+10;
    elsif 'A' <= ch and ch <= 'F' then
      result := character'pos(ch) - character'pos('A')+10;
    else report "Format error on line " & integer'image(index)
      severity error;
    end if;
    mem(index)(35-i*4 downto 32-i*4) :=
      to_std_logic_vector(result,4);
  end loop;
  index := index + 1;
end loop;

-- read memory
loop
  rd <= mem(to_integer(a(7 downto 2)));
  wait on a;
end loop;
end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity dmem is -- data memory
  port(clk, we: in STD_LOGIC;
        a, wd: in STD_LOGIC_VECTOR(31 downto 0);
        rd: out STD_LOGIC_VECTOR(31 downto 0));
end;

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

```



```

        wait on clk, a;
    end loop;
end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity arm is -- pipelined processor
    port (clk, reset:      in  STD_LOGIC;
          PCF:             out STD_LOGIC_VECTOR(31 downto 0);
          InstrF:          in  STD_LOGIC_VECTOR(31 downto 0);
          MemWriteM:       out STD_LOGIC;
          ALUOutM, WriteDataM: out STD_LOGIC_VECTOR(31 downto 0);
          ReadDataM:       in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of arm is
    component controller
        port (clk, reset:      in  STD_LOGIC;
              InstrD:          in  STD_LOGIC_VECTOR(31 downto 12);
              ALUFlagsE:       in  STD_LOGIC_VECTOR(3 downto 0);
              RegSrcD, ImmSrcD: out STD_LOGIC_VECTOR(1 downto 0);
              ALUSrcE:         out STD_LOGIC;
              BranchTakenE:    out STD_LOGIC;
              ALUControlE:     out STD_LOGIC_VECTOR(1 downto 0);
              MemWriteM:       out STD_LOGIC;
              MemtoRegW:       out STD_LOGIC;
              PCSrcW:          out STD_LOGIC;
              RegWriteW:       out STD_LOGIC;
              -- hazard interface
              RegWriteM:       out STD_LOGIC;
              MemtoRegE:       out STD_LOGIC;
              PCWrPendingF:    out STD_LOGIC;
              FlushE:          in  STD_LOGIC);
    end component;
    component datapath
        port (clk, reset:      in  STD_LOGIC;
              RegSrcD, ImmSrcD: in  STD_LOGIC_VECTOR(1 downto 0);
              ALUSrcE:         in  STD_LOGIC;
              BranchTakenE:    in  STD_LOGIC;
              ALUControlE:     in  STD_LOGIC_VECTOR(1 downto 0);
              MemtoRegW:       in  STD_LOGIC;
              PCSrcW:          in  STD_LOGIC;
              RegWriteW:       in  STD_LOGIC;
              PCF:             out STD_LOGIC_VECTOR(31 downto 0);
              InstrF:          in  STD_LOGIC_VECTOR(31 downto 0);
              InstrD:          out STD_LOGIC_VECTOR(31 downto 0);
              ALUOutM:         out STD_LOGIC_VECTOR(31 downto 0);
              WriteDataM:      out STD_LOGIC_VECTOR(31 downto 0);
              ReadDataM:       in  STD_LOGIC_VECTOR(31 downto 0);
              ALUFlagsE:       out STD_LOGIC_VECTOR(3 downto 0);
              -- hazard logic
              Match_1E_M:      out STD_LOGIC;
              Match_1E_W:      out STD_LOGIC;
              Match_2E_M:      out STD_LOGIC);
    end component;

```

```

        Match_2E_W:      out STD_LOGIC;
        Match_12D_E:     out STD_LOGIC;
        ForwardAE:       in  STD_LOGIC_VECTOR(1 downto 0);
        ForwardBE:       in  STD_LOGIC_VECTOR(1 downto 0);
        StallF:          in  STD_LOGIC;
        StallD:          in  STD_LOGIC;
        FlushD:          in  STD_LOGIC);
end component;
component hazard
  port (clk, reset:      in  STD_LOGIC;
        Match_1E_M:     in  STD_LOGIC;
        Match_1E_W:     in  STD_LOGIC;
        Match_2E_M:     in  STD_LOGIC;
        Match_2E_W:     in  STD_LOGIC;
        Match_12D_E:    in  STD_LOGIC;
        RegWriteM:      in  STD_LOGIC;
        RegWriteW:      in  STD_LOGIC;
        BranchTakenE:   in  STD_LOGIC;
        MemtoRegE:      in  STD_LOGIC;
        PCWrPendingF:   in  STD_LOGIC;
        PCSrcW:         in  STD_LOGIC;
        ForwardAE:      out STD_LOGIC_VECTOR(1 downto 0);
        ForwardBE:      out STD_LOGIC_VECTOR(1 downto 0);
        StallF, StallD: out STD_LOGIC;
        FlushD, FlushE: out STD_LOGIC);
end component;
signal RegSrcD, ImmSrcD, ALUControlE: STD_LOGIC_VECTOR(1 downto 0);
signal ALUSrcE, BranchTakenE, MemtoRegW, PCSrcW, RegWriteW: STD_LOGIC;
signal ALUFlagsE: STD_LOGIC_VECTOR(3 downto 0);
signal InstrD: STD_LOGIC_VECTOR(31 downto 0);
signal RegWriteM, MemtoRegE, PCWrPendingF: STD_LOGIC;
signal ForwardAE, ForwardBE: STD_LOGIC_VECTOR(1 downto 0);
signal StallF, StallD, FlushD, FlushE: STD_LOGIC;
signal Match_1E_M, Match_1E_W, Match_2E_M, Match_2E_W, Match_12D_E:
STD_LOGIC;

begin
  c: controller port map(clk, reset, InstrD(31 downto 12), ALUFlagsE,
                        RegSrcD, ImmSrcD,
                        ALUSrcE, BranchTakenE, ALUControlE,
                        MemWriteM,
                        MemtoRegW, PCSrcW, RegWriteW,
                        RegWriteM, MemtoRegE, PCWrPendingF,
                        FlushE);

  dp: datapath port map(clk, reset,
                        RegSrcD, ImmSrcD,
                        ALUSrcE, BranchTakenE, ALUControlE,
                        MemtoRegW, PCSrcW, RegWriteW,
                        PCF, InstrF, InstrD,
                        ALUOutM, WriteDataM, ReadDataM,
                        ALUFlagsE,
                        Match_1E_M, Match_1E_W, Match_2E_M,
                        Match_2E_W, Match_12D_E,

```

```

        ForwardAE, ForwardBE, StallF, StallD, FlushD);
h: hazard port map(clk, reset, Match_1E_M, Match_1E_W,
        Match_2E_M, Match_2E_W, Match_12D_E,
        RegWriteM, RegWriteW, BranchTakenE, MemtoRegE,
        PCWrPendingF, PCSrcW,
        ForwardAE, ForwardBE,
        StallF, StallD, FlushD, FlushE);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- pipelined control decoder
    port(clk, reset:      in  STD_LOGIC;
          InstrD:         in  STD_LOGIC_VECTOR(31 downto 12);
          ALUFlagsE:      in  STD_LOGIC_VECTOR(3 downto 0);
          RegSrcD, ImmSrcD: out STD_LOGIC_VECTOR(1 downto 0);
          ALUSrcE:        out STD_LOGIC;
          BranchTakenE:   out STD_LOGIC;
          ALUControlE:    out STD_LOGIC_VECTOR(1 downto 0);
          MemWriteM:      out STD_LOGIC;
          MemtoRegW:      out STD_LOGIC;
          PCSrcW:         out STD_LOGIC;
          RegWriteW:      out STD_LOGIC;
          -- hazard interface
          RegWriteM:      out STD_LOGIC;
          MemtoRegE:      out STD_LOGIC;
          PCWrPendingF:   out STD_LOGIC;
          FlushE:         in  STD_LOGIC);
end;
architecture synth of controller is
    component flopr generic(width: integer);
        port(clk, reset:      in  STD_LOGIC;
              d:              in  STD_LOGIC_VECTOR(width-1 downto 0);
              q:              out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component floprc generic(width: integer);
        port(clk, reset, clear: in  STD_LOGIC;
              d:              in  STD_LOGIC_VECTOR(width-1 downto 0);
              q:              out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component conditional
        port(Cond:           in  STD_LOGIC_VECTOR(3 downto 0);
              Flags:         in  STD_LOGIC_VECTOR(3 downto 0);
              ALUFlags:      in  STD_LOGIC_VECTOR(3 downto 0);
              FlagsWrite:    in  STD_LOGIC_VECTOR(1 downto 0);
              CondEx:        out STD_LOGIC;
              FlagsNext:     out STD_LOGIC_VECTOR(3 downto 0));
    end component;
    signal controlsD: STD_LOGIC_VECTOR(9 downto 0);
    signal CondExE, ALUOpD: STD_LOGIC;
    signal ALUControlD: STD_LOGIC_VECTOR(1 downto 0);
    signal ALUSrcD: STD_LOGIC;
    signal MemtoRegD, MemtoRegM: STD_LOGIC;
    signal RegWrited, RegWriteE, RegWriteGatedE: STD_LOGIC;
    signal MemWrited, MemWriteE, MemWriteGatedE: STD_LOGIC;

```

```

signal BranchD, BranchE: STD_LOGIC;
signal FlagWriteD, FlagWriteE: STD_LOGIC_VECTOR(1 downto 0);
signal PCSrcD, PCSrcE, PCSrcM: STD_LOGIC;
signal FlagsE, FlagsNextE, Conde: STD_LOGIC_VECTOR(3 downto 0);
signal Funct: STD_LOGIC_VECTOR(5 downto 0);
signal Rd: STD_LOGIC_VECTOR(3 downto 0);
signal PCSrcGatedE: STD_LOGIC;
signal FlushedValsEnext, FlushedValse: STD_LOGIC_VECTOR(6 downto 0);
signal ValsEnext, Valse: STD_LOGIC_VECTOR(2 downto 0);
signal ValsMnext, ValsM: STD_LOGIC_VECTOR(3 downto 0);
signal ValsWnext, ValsW: STD_LOGIC_VECTOR(2 downto 0);
begin
  -- Decode stage
  -- Main Decoder
  process(all) begin
    case InstrD(27 downto 26) is
      when "00" => controlsD <= "0000101001" when InstrD(25) -- DP imm
                                     else "0000001001";         -- DP reg
      when "01" => controlsD <= "0001111000" when InstrD(20) -- LDR
                                     else "1001110100";         -- STR
      when "10" => controlsD <= "0110100010";                 -- B
      when others => controlsD <= "-----";                   --
    unimplemented
    end case;
  end process;

  (RegSrcD, ImmSrcD, ALUSrcD, MemtoRegD,
   RegWriteD, MemWriteD, BranchD, ALUOpD) <= controlsD;

  -- ALU Decoder
  Funct <= InstrD(25 downto 20);
  Rd <= InstrD(15 downto 12);
  process(all) begin
    if (ALUOpD) then
      case Funct(4 downto 1) is
        when "0100" => ALUControlD <= "00"; -- ADD
        when "0010" => ALUControlD <= "01"; -- SUB
        when "0000" => ALUControlD <= "10"; -- AND
        when "1100" => ALUControlD <= "11"; -- ORR
        when others => ALUControlD <= "--"; -- unimplemented
      end case;
      FlagWriteD(1) <= Funct(0);
      FlagWriteD(0) <= Funct(0) and (not ALUControlD(1));
    else
      ALUControlD <= "00";
      FlagWriteD <= "00";
    end if;
  end process;

  PCSrcD <= ((and Rd) and RegWriteD) or BranchD;

  -- Execute stage
  FlushedValsEnext <= (FlagWriteD, BranchD, MemWriteD, RegWriteD,
                      PCSrcD, MemtoRegD);

```

```

ValsEnext <= (ALUSrcD, ALUControlD);
flushedregsE: floprc generic map (7)
  port map(clk, reset, FlushE, FlushedValsEnext, FlushedValsE);
regsE: flopr generic map (3)
  port map(clk, reset, ValsEnext, ValsE);
condregE: flopr generic map (4)
  port map(clk, reset, InstrD(31 downto 28), CondE);
flagsreg: flopr generic map (4)
  port map(clk, reset, FlagsNextE, FlagsE);

(FlagWriteE, BranchE, MemWriteE, RegWriteE, PCSrcE, MemtoRegE) <=
FlushedValsE;
(ALUSrcE, ALUControlE) <= ValsE;

-- write and Branch controls are conditional
Cond: conditional port map(CondE, FlagsE, ALUFlagsE, FlagWriteE,
CondExE, FlagsNextE);
BranchTakenE    <= BranchE and CondExE;
RegWriteGatedE  <= RegWriteE and CondExE;
MemWriteGatedE  <= MemWriteE and CondExE;
PCSrcGatedE     <= PCSrcE and CondExE;

-- Memory stage
ValsMnext <= (MemWriteGatedE, MemtoRegE, RegWriteGatedE, PCSrcGatedE);
regsM: flopr generic map (4)
  port map(clk, reset, ValsMnext, ValsM);
(MemWriteM, MemtoRegM, RegWriteM, PCSrcM) <= ValsM;

-- Writeback stage
ValsWnext <= (MemtoRegM, RegWriteM, PCSrcM);
regsW: flopr generic map (3)
  port map(clk, reset, ValsWnext, ValsW);
(MemtoRegW, RegWriteW, PCSrcW) <= ValsW;

-- Hazard Prediction
PCWrPendingF <= PCSrcD or PCSrcE or PCSrcM;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity conditional is
  port(Cond:          in  STD_LOGIC_VECTOR(3 downto 0);
        Flags:        in  STD_LOGIC_VECTOR(3 downto 0);
        ALUFlags:     in  STD_LOGIC_VECTOR(3 downto 0);
        FlagsWrite:   in  STD_LOGIC_VECTOR(1 downto 0);
        CondEx:       out STD_LOGIC;
        FlagsNext:    out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture behave of conditional is
  signal neg, zero, carry, overflow, ge: STD_LOGIC;
begin
  (neg, zero, carry, overflow) <= Flags;
  ge <= (neg xnor overflow);

```

```

process(all) begin -- Condition checking
  case Cond is
    when "0000" => CondEx <= zero;
    when "0001" => CondEx <= not zero;
    when "0010" => CondEx <= carry;
    when "0011" => CondEx <= not carry;
    when "0100" => CondEx <= neg;
    when "0101" => CondEx <= not neg;
    when "0110" => CondEx <= overflow;
    when "0111" => CondEx <= not overflow;
    when "1000" => CondEx <= carry and (not zero);
    when "1001" => CondEx <= not(carry and (not zero));
    when "1010" => CondEx <= ge;
    when "1011" => CondEx <= not ge;
    when "1100" => CondEx <= (not zero) and ge;
    when "1101" => CondEx <= not ((not zero) and ge);
    when "1110" => CondEx <= '1';
    when others => CondEx <= '-';
  end case;
end process;

FlagsNext(3 downto 2) <= ALUFlags(3 downto 2) when (FlagsWrite(1) and
CondEx) else  Flags(3 downto 2);
FlagsNext(1 downto 0) <= ALUFlags(1 downto 0) when (FlagsWrite(0) and
CondEx) else  Flags(1 downto 0);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity datapath is
  port(clk, reset:      in  STD_LOGIC;
        RegSrcD, ImmSrcD: in  STD_LOGIC_VECTOR(1 downto 0);
        ALUSrcE:        in  STD_LOGIC;
        BranchTakenE:   in  STD_LOGIC;
        ALUControlE:    in  STD_LOGIC_VECTOR(1 downto 0);
        MemtoRegW:      in  STD_LOGIC;
        PCSrcW:         in  STD_LOGIC;
        RegWriteW:      in  STD_LOGIC;
        PCF:            out STD_LOGIC_VECTOR(31 downto 0);
        InstrF:         in  STD_LOGIC_VECTOR(31 downto 0);
        InstrD:         out STD_LOGIC_VECTOR(31 downto 0);
        ALUOutM:        out STD_LOGIC_VECTOR(31 downto 0);
        WriteDataM:     out STD_LOGIC_VECTOR(31 downto 0);
        ReadDataM:      in  STD_LOGIC_VECTOR(31 downto 0);
        ALUFlagsE:      out STD_LOGIC_VECTOR(3 downto 0);
        -- hazard logic
        Match_1E_M:     out STD_LOGIC;
        Match_1E_W:     out STD_LOGIC;
        Match_2E_M:     out STD_LOGIC;
        Match_2E_W:     out STD_LOGIC;
        Match_12D_E:    out STD_LOGIC;
        ForwardAE:      in  STD_LOGIC_VECTOR(1 downto 0);
        ForwardBE:      in  STD_LOGIC_VECTOR(1 downto 0);
        StallF:         in  STD_LOGIC;
        StallD:         in  STD_LOGIC;

```

```

        FlushD:                in STD_LOGIC);
end;
architecture struct of datapath is
    component alu
        port(a, b:              in STD_LOGIC_VECTOR(31 downto 0);
              ALUControl:       in STD_LOGIC_VECTOR(1 downto 0);
              Result:           buffer STD_LOGIC_VECTOR(31 downto 0);
              ALUFlags:         out STD_LOGIC_VECTOR(3 downto 0));
    end component;
    component regfile
        port(clk:               in STD_LOGIC;
              we3:              in STD_LOGIC;
              ra1, ra2, wa3:     in STD_LOGIC_VECTOR(3 downto 0);
              wd3, r15:         in STD_LOGIC_VECTOR(31 downto 0);
              rd1, rd2:         out STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component adder
        port(a, b: in STD_LOGIC_VECTOR(31 downto 0);
              y:   out STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component extend
        port(Instr: in STD_LOGIC_VECTOR(23 downto 0);
              ImmSrc: in STD_LOGIC_VECTOR(1 downto 0);
              ExtImm: out STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component flopr generic(width: integer);
        port(clk, reset: in STD_LOGIC;
              d:         in STD_LOGIC_VECTOR(width-1 downto 0);
              q:         out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component flopenrc generic(width: integer);
        port(clk, reset, en, clear: in STD_LOGIC;
              d:                   in STD_LOGIC_VECTOR(width-1 downto 0);
              q:                   out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component flopenr generic(width: integer);
        port(clk, reset, en: in STD_LOGIC;
              d:             in STD_LOGIC_VECTOR(width-1 downto 0);
              q:             out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component mux2 generic(width: integer);
        port(d0, d1: in STD_LOGIC_VECTOR(width-1 downto 0);
              s:     in STD_LOGIC;
              y:     out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component mux3 generic(width: integer);
        port(d0, d1, d2: in STD_LOGIC_VECTOR(width-1 downto 0);
              s:         in STD_LOGIC_VECTOR(1 downto 0);
              y:         out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component eqcmp generic(width: integer);
        port(a, b: in STD_LOGIC_VECTOR(width-1 downto 0);
              y:   out STD_LOGIC);
    end component;

```

```

signal PCPlus4F, PCnext1F, PCnextF: STD_LOGIC_VECTOR(31 downto 0);
signal ExtImmD, rd1D, rd2D, PCPlus8D: STD_LOGIC_VECTOR(31 downto 0);
signal rd1E, rd2E, ExtImmE, SrcAE: STD_LOGIC_VECTOR(31 downto 0);
signal SrcBE, WriteDataE, ALUResultE: STD_LOGIC_VECTOR(31 downto 0);
signal ReadDataW, ALUOutW, ResultW: STD_LOGIC_VECTOR(31 downto 0);
signal RA1D, RA2D, RA1E, RA2E: STD_LOGIC_VECTOR(3 downto 0);
signal WA3E, WA3M, WA3W: STD_LOGIC_VECTOR(3 downto 0);
signal Match_1D_E, Match_2D_E: STD_LOGIC;
signal notStallF: STD_LOGIC;
begin
  -- Fetch stage
  notStallF <= (not StallF);
  pcnextmux: mux2 generic map (32)
    port map(PCPlus4F, ResultW, PCSrcW, PCnext1F);
  branchmux: mux2 generic map (32)
    port map(PCnext1F, ALUResultE, BranchTakenE, PCnextF);
  pcreg: flopenr generic map (32)
    port map(clk, reset, notStallF, PCnextF, PCF);
  pcadd: adder generic map (32)
    port map(PCF, 32D"4", PCPlus4F);

  -- Decode Stage
  PCPlus8D <= PCPlus4F; -- skip register
  instrreg: flopenrc generic map (32)
    port map(clk, reset, (not StallD), FlushD, InstrF, InstrD);
  ralmux: mux2 generic map (4)
    port map(InstrD(19 downto 16), 4D"15", RegSrcD(0), RA1D);
  ra2mux: mux2 generic map (4)
    port map(InstrD(3 downto 0), InstrD(15 downto 12), RegSrcD(1), RA2D);
  rf: regfile
    port map(clk, RegWriteW, RA1D, RA2D,
             WA3W, ResultW, PCPlus8D,
             rd1D, rd2D);
  ext: extend
    port map(InstrD(23 downto 0), ImmSrcD, ExtImmD);

  -- Execute Stage
  rd1reg: flopr generic map (32)
    port map(clk, reset, rd1D, rd1E);
  rd2reg: flopr generic map (32)
    port map(clk, reset, rd2D, rd2E);
  immreg: flopr generic map (32)
    port map(clk, reset, ExtImmD, ExtImmE);
  wa3ereg: flopr generic map (4)
    port map(clk, reset, InstrD(15 downto 12), WA3E);
  ralreg: flopr generic map (4)
    port map(clk, reset, RA1D, RA1E);
  ra2reg: flopr generic map (4)
    port map(clk, reset, RA2D, RA2E);
  byp1mux: mux3 generic map (32)
    port map(rd1E, ResultW, ALUOutM, ForwardAE, SrcAE);
  byp2mux: mux3 generic map (32)
    port map(rd2E, ResultW, ALUOutM, ForwardBE, WriteDataE);

```



```

srcbmux: mux2 generic map (32)
  port map(WriteDataE, ExtImmE, ALUSrcE, SrcBE);
i_alu: alu
  port map(SrcAE, SrcBE, ALUControlE, ALUResultE, ALUFlagsE);

-- Memory Stage
aluresreg: flopr generic map (32)
  port map(clk, reset, ALUResultE, ALUOutM);
wdreg: flopr generic map (32)
  port map(clk, reset, WriteDataE, WriteDataM);
wa3mreg: flopr generic map (4)
  port map(clk, reset, WA3E, WA3M);

-- Writeback Stage
aluoutreg: flopr generic map (32)
  port map(clk, reset, ALUOutM, ALUOutW);
rdreg: flopr generic map (32)
  port map(clk, reset, ReadDataM, ReadDataW);
wa3wreg: flopr generic map (4)
  port map(clk, reset, WA3M, WA3W);
resmux: mux2 generic map (32)
  port map(ALUOutW, ReadDataW, MemtoRegW, ResultW);

-- hazard comparison
m0: eqcmp generic map (4)
  port map(WA3M, RA1E, Match_1E_M);
m1: eqcmp generic map (4)
  port map(WA3W, RA1E, Match_1E_W);
m2: eqcmp generic map (4)
  port map(WA3M, RA2E, Match_2E_M);
m3: eqcmp generic map (4)
  port map(WA3W, RA2E, Match_2E_W);
m4a: eqcmp generic map (4)
  port map(WA3E, RA1D, Match_1D_E);
m4b: eqcmp generic map (4)
  port map(WA3E, RA2D, Match_2D_E);
Match_12D_E <= Match_1D_E or Match_2D_E;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity hazard is
  port(clk, reset:          in  STD_LOGIC;
        Match_1E_M:        in  STD_LOGIC;
        Match_1E_W:        in  STD_LOGIC;
        Match_2E_M:        in  STD_LOGIC;
        Match_2E_W:        in  STD_LOGIC;
        Match_12D_E:       in  STD_LOGIC;
        RegWriteM:         in  STD_LOGIC;
        RegWriteW:         in  STD_LOGIC;
        BranchTakenE:      in  STD_LOGIC;
        MemtoRegE:         in  STD_LOGIC;
        PCWrPendingF:      in  STD_LOGIC;
        PCSrcW:            in  STD_LOGIC;
        ForwardAE:         out STD_LOGIC_VECTOR(1 downto 0);
  );
end entity;

```

```

        ForwardBE:          out STD_LOGIC_VECTOR(1 downto 0);
        StallF, StallD:     out STD_LOGIC;
        FlushD, FlushE:     out STD_LOGIC);
end;

architecture behave of hazard is
    signal ldrStallD: STD_LOGIC;
begin
    ForwardAE(1) <= '1' when (Match_1E_M and RegWriteM) else '0';
    ForwardAE(0) <= '1' when (Match_1E_W and RegWriteW and (not
ForwardAE(1))) else '0';

    ForwardBE(1) <= '1' when (Match_2E_M and RegWriteM) else '0';
    ForwardBE(0) <= '1' when (Match_2E_W and RegWriteW and (not
ForwardBE(1))) else '0';

    ldrStallD <= Match_12D_E and MemtoRegE;

    StallD <= ldrStallD;
    StallF <= ldrStallD or PCWrPendingF;
    FlushE <= ldrStallD or BranchTakenE;
    FlushD <= PCWrPendingF or PCSrcW or BranchTakenE;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity regfile is -- three-port register file
    port(clk:          in  STD_LOGIC;
         we3:          in  STD_LOGIC;
         ra1, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
         wd3, r15:     in  STD_LOGIC_VECTOR(31 downto 0);
         rd1, rd2:     out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
    type ramtype is array (31 downto 0) of
        STD_LOGIC_VECTOR(31 downto 0);
    signal mem: ramtype;
begin
    process(clk) begin
        if falling_edge(clk) then -- write rf on negative edge of clock
            if we3 = '1' then mem(to_integer(wa3)) <= wd3;
            end if;
        end if;
    end process;
    process(all) begin
        if (to_integer(ra1) = 15) then rd1 <= r15;
        else rd1 <= mem(to_integer(ra1));
        end if;
        if (to_integer(ra2) = 15) then rd2 <= r15;
        else rd2 <= mem(to_integer(ra2));
        end if;
    end process;
end;

```

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity adder is -- adder
    port(a, b: in STD_LOGIC_VECTOR(31 downto 0);
         y:      out STD_LOGIC_VECTOR(31 downto 0));
end;

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

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity extend is
    port(Instr: in STD_LOGIC_VECTOR(23 downto 0);
         ImmSrc: in STD_LOGIC_VECTOR(1 downto 0);
         ExtImm: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of extend is
begin
    process(all) begin
        case ImmSrc is
            when "00" => ExtImm <= (X"000000", Instr(7 downto 0));
            when "01" => ExtImm <= (X"00000", Instr(11 downto 0));
            when "10" => ExtImm <= (Instr(23), Instr(23), Instr(23),
                                     Instr(23), Instr(23), Instr(23), Instr(23), Instr(23));
            when others => ExtImm <= X"-----";
        end case;
    end process;
end;

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

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

library IEEE; use IEEE.STD LOGIC 1164.all;
```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

architecture asynchronous of flopenrc is
begin
  process(clk, reset) begin
    if reset then q <= (others => '0');
    elsif rising_edge(clk) then
      if en then
        if clear then

```

```

        q <= (others => '0');
    else
        q <= d;
    end if;
end if;
end if;
end process;
end;

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

architecture behave of mux2 is
begin
    y <= d1 when s else d0;
end;

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

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

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity eqcmp is -- equality comparator
    generic(width: integer);
    port(a, b: in  STD_LOGIC_VECTOR(width-1 downto 0);
         y:      out STD_LOGIC);
end;

architecture behave of eqcmp is
begin
    y <= '1' when a = b else '0';

```

```

end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity alu is
    port(a, b:          in  STD_LOGIC_VECTOR(31 downto 0);
         ALUControl: in  STD_LOGIC_VECTOR(1 downto 0);
         Result:       buffer STD_LOGIC_VECTOR(31 downto 0);
         ALUFlags:     out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture behave of alu is
    signal condinvb: STD_LOGIC_VECTOR(31 downto 0);
    signal sum:      STD_LOGIC_VECTOR(32 downto 0);
    signal neg, zero, carry, overflow: STD_LOGIC;
begin
    condinvb <= not b when ALUControl(0) else b;
    sum <= ('0', a) + ('0', condinvb) + ALUControl(0);

    process(all) begin
        case? ALUControl(1 downto 0) is
            when "0-" => result <= sum(31 downto 0);
            when "10" => result <= a and b;
            when "11" => result <= a or b;
            when others => result <= (others => '-');
        end case?;
    end process;

    neg      <= Result(31);
    zero     <= '1' when (Result = 0) else '0';
    carry    <= (not ALUControl(1)) and sum(32);
    overflow <= (not ALUControl(1)) and
                (not (a(31) xor b(31) xor ALUControl(0))) and
                (a(31) xor sum(31));
    ALUFlags <= (neg, zero, carry, overflow);
end;

```

Exercise 7.41

SystemVerilog

```

module hazard(input  logic      clk, reset,
              input  logic      Match_1E_M, Match_1E_W, Match_2E_M,
              input  logic      Match_2E_W, Match_12D_E,
              input  logic      RegWriteM, RegWriteW,
              input  logic      BranchTakenE, MemtoRegE,
              input  logic      PCWrPendingF, PCSrcW,
              output logic [1:0] ForwardAE, ForwardBE,
              output logic      StallF, StallD,
              output logic      FlushD, FlushE);

    logic ldrStallD;

```

```

// forwarding logic
always_comb begin
    if (Match_1E_M & RegWriteM) ForwardAE = 2'b10;
    else if (Match_1E_W & RegWriteW) ForwardAE = 2'b01;
    else ForwardAE = 2'b00;

    if (Match_2E_M & RegWriteM) ForwardBE = 2'b10;
    else if (Match_2E_W & RegWriteW) ForwardBE = 2'b01;
    else ForwardBE = 2'b00;
end

// stalls and flushes
// Load RAW
//   when an instruction reads a register loaded by the previous,
//   stall in the decode stage until it is ready
// Branch hazard
//   When a branch is taken, flush the incorrectly fetched instrs
//   from decode and execute stages
// PC Write Hazard
//   When the PC might be written, stall all following instructions
//   by stalling the fetch and flushing the decode stage
// when a stage stalls, stall all previous and flush next

assign ldrStallD = Match_12D_E & MemtoRegE;

assign StallD = ldrStallD;
assign StallF = ldrStallD | PCWrPendingF;
assign FlushE = ldrStallD | BranchTakenE;
assign FlushD = PCWrPendingF | PCSrcW | BranchTakenE;

endmodule

```

VHDL

```

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity hazard is
    port (clk, reset:      in  STD_LOGIC;
          Match_1E_M:      in  STD_LOGIC;
          Match_1E_W:      in  STD_LOGIC;
          Match_2E_M:      in  STD_LOGIC;
          Match_2E_W:      in  STD_LOGIC;
          Match_12D_E:     in  STD_LOGIC;
          RegWriteM:       in  STD_LOGIC;
          RegWriteW:       in  STD_LOGIC;
          BranchTakenE:    in  STD_LOGIC;
          MemtoRegE:       in  STD_LOGIC;
          PCWrPendingF:    in  STD_LOGIC;
          PCSrcW:          in  STD_LOGIC;
          ForwardAE:       out STD_LOGIC_VECTOR(1 downto 0);
          ForwardBE:       out STD_LOGIC_VECTOR(1 downto 0);
          StallF, StallD:  out STD_LOGIC;
          FlushD, FlushE:  out STD_LOGIC);
end;

```

```

architecture behave of hazard is
    signal ldrStallD: STD_LOGIC;
begin
    ForwardAE(1) <= '1' when (Match_1E_M and RegWriteM) else '0';
    ForwardAE(0) <= '1' when (Match_1E_W and RegWriteW and (not
ForwardAE(1))) else '0';

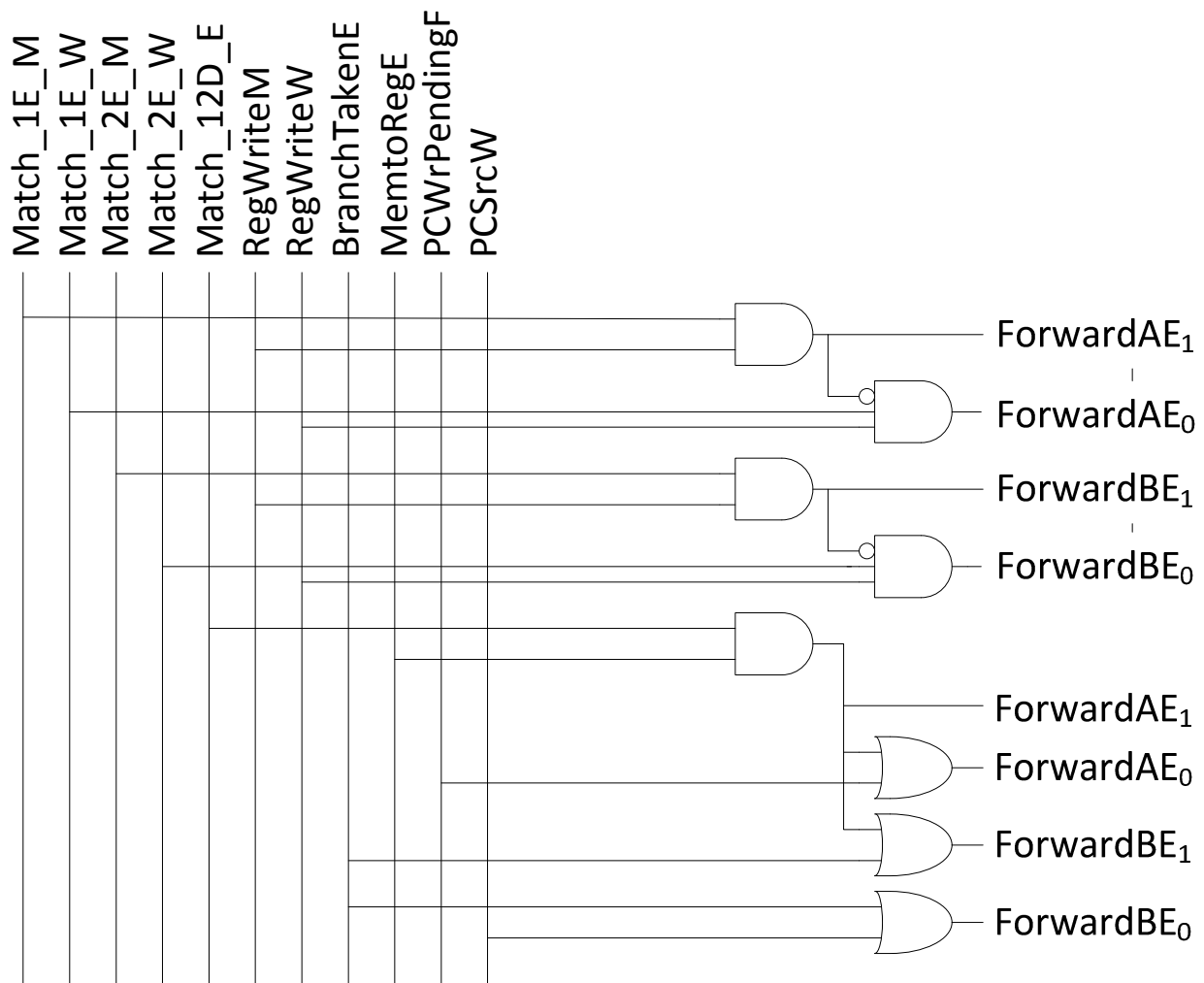
    ForwardBE(1) <= '1' when (Match_2E_M and RegWriteM) else '0';
    ForwardBE(0) <= '1' when (Match_2E_W and RegWriteW and (not
ForwardBE(1))) else '0';

    ldrStallD <= Match_12D_E and MemtoRegE;

    StallD <= ldrStallD;
    StallF <= ldrStallD or PCWrPendingF;
    FlushE <= ldrStallD or BranchTakenE;
    FlushD <= PCWrPendingF or PCSrcW or BranchTakenE;
end;

```

Hazard Unit Schematic



Question 7.1

A pipelined microprocessors with N stages offers an ideal speedup of N over nonpipelined microprocessor. This speedup comes at the cost of little extra hardware: pipeline registers and possibly a hazard unit. The disadvantage of a pipelined processor is added complexity, especially in dealing with data and control hazards.

Question 7.2

While pipelining offers speedup, it still has its costs. The speedup of an N stage processor is not N because of (1) sequencing overhead ($t_{pcq} + t_{setup}$, the delay of inserting a register), (2) unequal delays of pipeline stages, (3) time to fill up the pipeline (at the beginning of a program), (4) time

to drain the pipeline (at the end of a program), and (5) dependencies stalling or flushing the pipeline.

Question 7.3

A hazard in a pipelined microprocessor occurs when the execution of an instruction depends on the result of a previously issued instruction that has not completed executing. Some options for dealing with hazards are:

- (1) to have the **compiler insert nops** to prevent dependencies,
- (2) to have the **compiler reorder the code** to eliminate dependencies (inserting nops when this is impossible),
- (3) to have the hardware **stall** (or **flush**) the pipeline when there is a dependency,
- (4) to have the hardware **forward** results to earlier stages in the pipeline or stall when that is impossible.

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

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

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

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

Question 7.4

A superscalar processor duplicates the datapath hardware to execute multiple instructions (in the same stage of a pipelined processor) at once. Ideally, the fetch stage can fetch multiple instructions per clock cycle. However, due to dependencies, this may be impossible. Thus, the costs of implementing a superscalar processor are (1) more hardware (additional register file and memory ports, additional functional units, more hazard detection and forwarding hardware, etc.), and (2) more complex fetch and commit (execution completion) algorithms. Also, because of dependencies, superscalar processors are often underutilized. Thus, for programs with a large amount of dependencies, superscalar processors can consume more area, power and cost (because of the additional hardware) without providing any speedup.

CHAPTER 8

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 a 100% miss rate with a 2-way set associative cache.

(b) **True.**

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

(c) **Usually true.**

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

Exercise 8.8

- (a) $b \times S \times N \times 4$ bytes
 (b) $[A - (\log_2(S) + \log_2(b) + 2)] \times S \times N$
 (c) $S = 1, N = C/b$
 (d) $S = C/b$

Exercise 8.9

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

Set 15	7C		
	78		
	74		
	70		
	20		
Set 7	9C 1C	7C 9C 1C	78-7C
	98 18	78 98 18	70-74
	94 14	74 94 14	
	90 10	70 90 10	20-24
	4C 8C C	4C 8C C	98-9C 18-1C
	48 88 8	48 88 8	90-94 10-14
	44 84 4	44 84 4	48-4C 88-8C 8-C
Set 0	40 80 0	40 80 0 20	40-44 80-84 0-4
	(a) Direct Mapped	(c) 2-way assoc	(d) direct mapped b=2

- (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

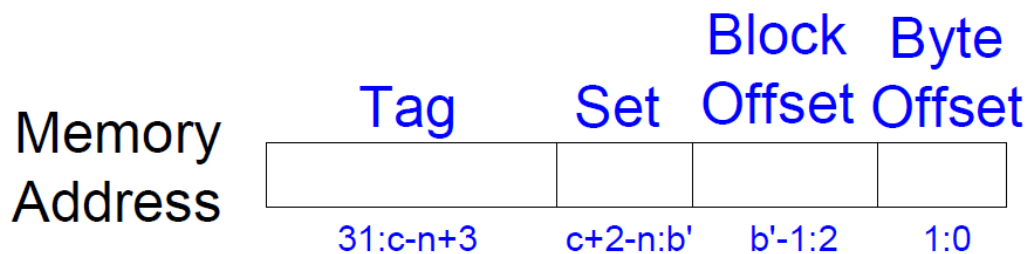
- (a) $11/14 = 79\%$ miss rate
- (b) $12/14 = 86\%$ miss rate
- (c) $6/14 = 43\%$ miss rate
- (d) $7/14 = 50\%$ miss rate

Exercise 8.11

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

Exercise 8.12

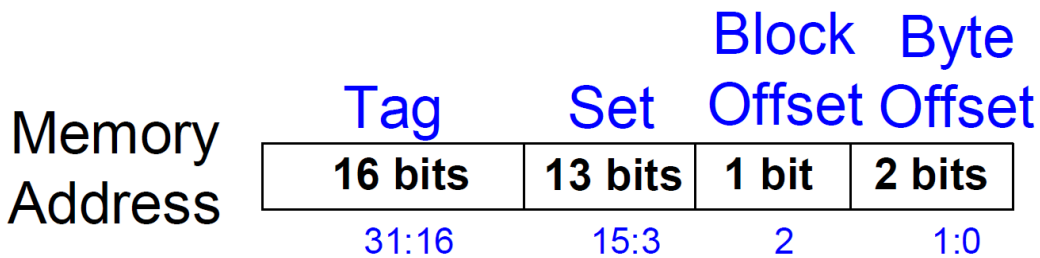
- (a - b)



- (c) Each tag is $32 - (c+2-n)$ bits = $(30 - (c-n))$ bits
- (d) # tag bits \times # blocks = $(30 - (c-n)) \times 2^{c+2-b'}$

Exercise 8.13

(a)



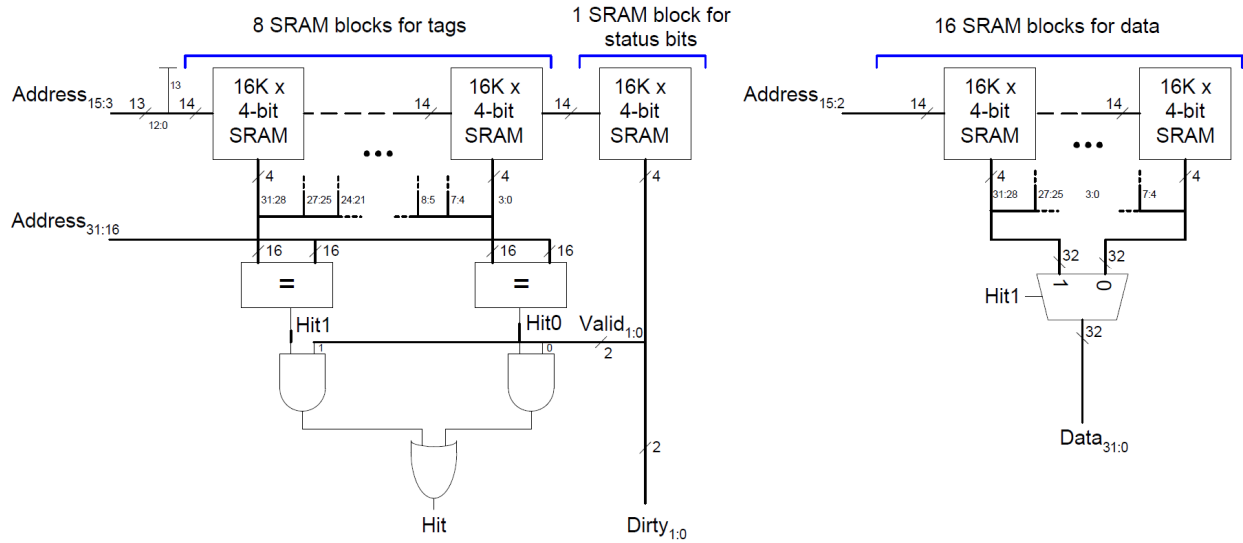
(b) Each tag is 16 bits. There are $32\text{Kwords} / (2 \text{ words / block}) = 16\text{K}$ blocks and each block needs a tag: $16 \times 16\text{K} = 218 = \mathbf{256 \text{ Kbits}}$ of tags.

(c) Each cache block requires: 2 status bits, 16 bits of tag, and 64 data bits, thus each set is $2 \times 82 \text{ bits} = \mathbf{164 \text{ bits}}$.

(d) See figure below. The design must use enough RAM chips to handle both the total capacity and the number of bits that must be read on each cycle. For the data, the SRAM must provide a capacity of 128 KB and must read 64 bits per cycle (one 32-bit word from each way). Thus the design needs at least $128\text{KB} / (8\text{KB}/\text{RAM}) = 16$ RAMs to hold the data and $64 \text{ bits} / (4 \text{ pins}/\text{RAM}) = 16$ 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 $8\text{K} \times 4\text{-bit}$ RAM. We use a $16\text{K} \times 4\text{-bit}$ 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 $1/1 \text{ GHz} = 1 \text{ ns}$,

$$AMAT = 1 \text{ ns} + 0.05(60 \text{ ns}) = 4 \text{ ns}$$

(b) $CPI = 4 + 4 = 8 \text{ cycles}$ (for a load)
 $CPI = 4 + 3 = 7 \text{ cycles}$ (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 = $5.17 + 0.07(60) = \mathbf{9.37}$

Exercise 8.17

(a) $AMAT = t_{\text{cache}} + MR_{\text{cache}} t_{\text{MM}}$

With a cycle time of $1/1 \text{ GHz} = 1 \text{ ns}$,

$$AMAT = 1 \text{ ns} + 0.15(200 \text{ ns}) = \mathbf{31 \text{ ns}}$$

(b) CPI = $31 + 4 = \mathbf{35 \text{ cycles}}$ (for a load)
CPI = $31 + 3 = \mathbf{34 \text{ cycles}}$ (for a store)

(c) Average CPI = $(0.11 + 0.02)(3) + (0.52)(4) + (0.1)(34) + (0.25)(35) = \mathbf{14.6}$

(d) Average CPI = $14.6 + 0.1(200) = \mathbf{34.6}$

Exercise 8.18

$$2^{64} \text{ bytes} = 2^4 \text{ exabytes} = \mathbf{16 \text{ exabytes}}$$

Exercise 8.19

From Figure 8.4, \$1 million will buy about $(\$1 \text{ million} / (\$0.05/\text{GB})) = 20 \text{ million GB}$ of hard disk:

$$20 \text{ million GB} \approx 2^{25} \times 2^{30} \text{ bytes} = 2^{55} \text{ bytes} = 2^5 \text{ petabytes} = \mathbf{32 \text{ petabytes}}$$

\$1 million will buy about $(\$1,000,000 / (\$7/\text{GB})) \approx 143,000 \text{ GB}$ of DRAM.

$$143,000 \text{ GB} \approx 2^7 \times 2^{10} \times 2^{30} = 2^{47} \text{ bytes} = 2^7 \text{ terabytes} = \mathbf{128 \text{ terabytes}}$$

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) $8 \text{ MB} / 4 \text{ KB} = 2^{23}/2^{12} = 2^{11}$ **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.

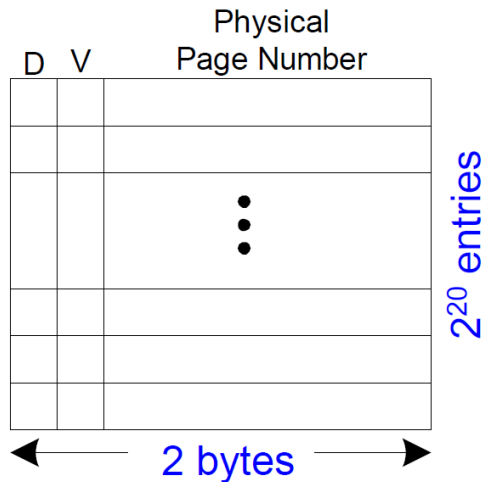
Imagine a program around memory address 0x01000000 operating on data around address 0x00000000. Physical page 0 would constantly be swapped between these two virtual pages, causing severe thrashing.

(f) 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 **2^{21} bytes**.



Exercise 8.21

(a) **31 bits**

(b) $2^{50}/2^{12} = 2^{38}$ **virtual pages**

(c) $2 \text{ GB} / 4 \text{ KB} = 2^{31}/2^{12} = 2^{19}$ **physical pages**

(d) virtual page number: **38 bits**; physical page number = **19 bits**

(e) 2^{38} page table entries (one for each virtual page).

(f) Each entry uses 19 bits of physical page number and 2 bits of status information. Thus, **3 bytes** are needed for each entry (rounding 21 bits up to the nearest number of bytes).

(h) The total table size is **3×2^{38} bytes**.

Exercise 8.22

(a) From Equation 8.2, $AMAT = t_{\text{cache}} + MR_{\text{cache}} (t_{\text{MM}} + MR_{\text{MM}} t_{\text{VM}})$.

However, each data access now requires an address translation (page table or TLB lookup).

Thus,

Without the TLB:

$$AMAT = t_{\text{MM}} + [t_{\text{cache}} + MR_{\text{cache}} (t_{\text{MM}} + MR_{\text{MM}} t_{\text{VM}})]$$

$$AMAT = 100 + [1 + 0.02(100 + 0.000003(1,000,000))] \text{ cycles} = \mathbf{103.06 \text{ cycles}}$$

With the TLB:

$$AMAT = [t_{TLB} + MR_{TLB}(t_{MM})] + [t_{cache} + MR_{cache}(t_{MM} + MR_{MM} t_{VM})]$$

$$AMAT = [1 + 0.0005(100)] + [1 + 0.02(100 + 0.000003 \times 1,000,000)] \text{ cycles}$$

$$= \mathbf{4.11 \text{ cycles}}$$

(b) # bits per entry = valid bit + tag bits + physical page number

1 valid bit

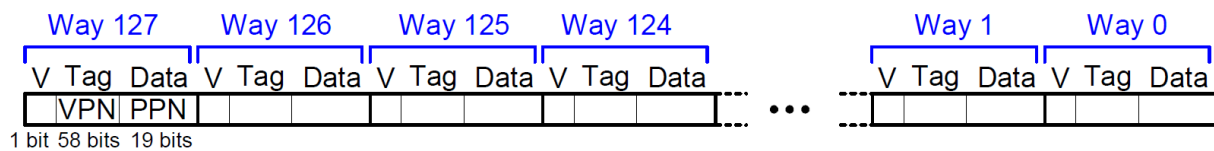
tag bits = virtual page number = 20 bits

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)

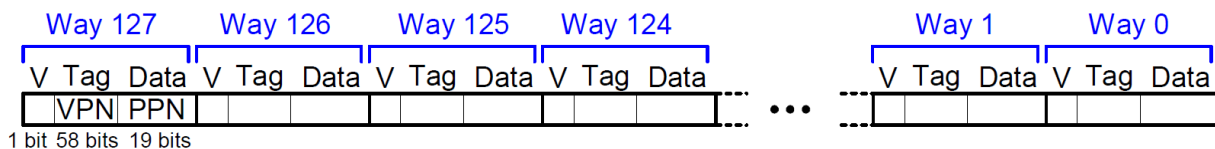


(d) **1 × 2048 bit SRAM**

Exercise 8.23

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

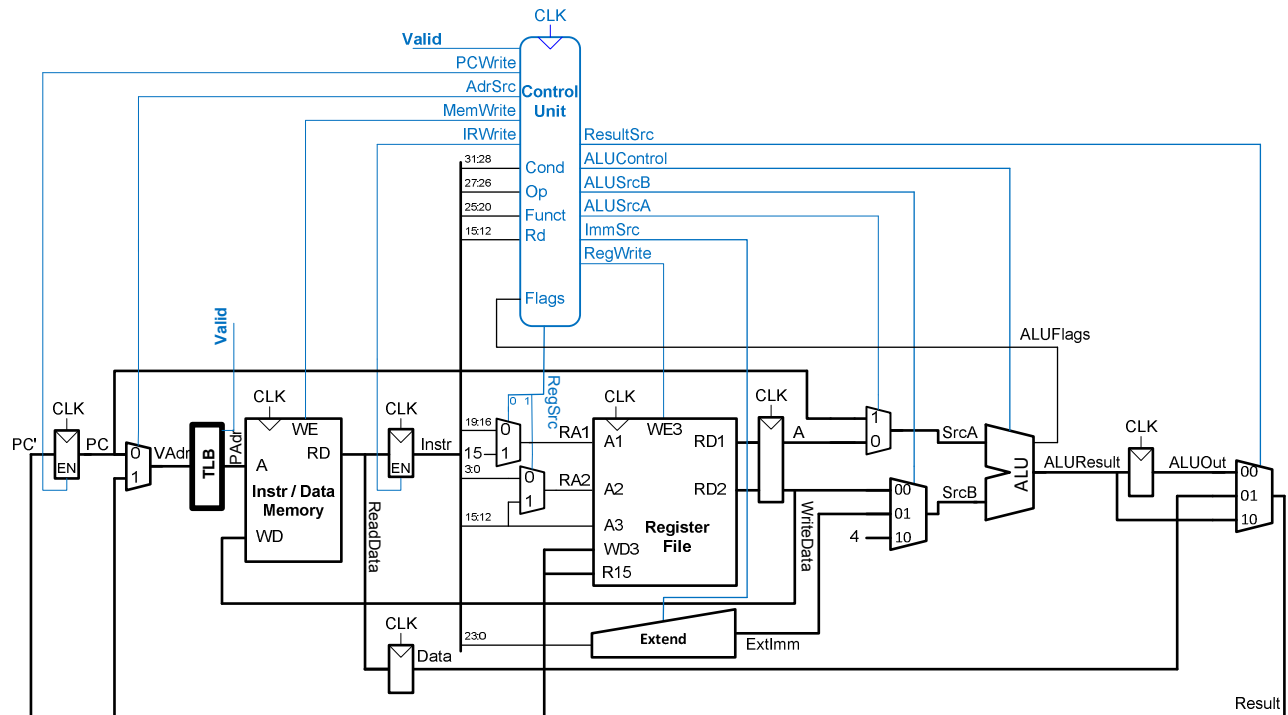
(b)



(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 $2^9 \times 8 \text{ bits} = \mathbf{4096 \text{ bits}}$

(b) This would increase the virtual page number to $25 - 14 = 11$ bits, and the physical page number to $22 - 14 = 8$ bits. This would increase the page table size to:

$$2^{11} \times 10 \text{ bits} = \mathbf{20480 \text{ bits}}$$

This increases the page table by 5 times, wasted valuable hardware to store the extra page table bits.

(c) Yes, this is possible. In order for concurrent access to take place, the number of set + block offset + byte offset bits must be less than the page offset bits.

(d) It is impossible to perform the tag comparison in the on-chip cache concurrently with the page table access because the upper (most significant) bits of the physical address are unknown until after the page table lookup (address translation) completes.

Exercise 8.26

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.