CHAPTER 4

Operations On Data

(Solutions to Practice Set)

Review Questions

1. Arithmetic operations interpret bit patterns as numbers. Logical operations interpret each bit as a logical value (*true* or *false*).
2. The leftmost carry is discarded.
3. The bit allocation can be 1. In this case, the data type normally represents a logical value.
4. Overflow happens when the result of an arithmetic operation is outside the range of allocated values.
5. The decimal point of the number with the smaller exponent is shifted to the left until the exponents are equal.
6. A unary operation takes a single operand. A binary operation takes two operands.
7. The common logical binary operations are: AND, OR, and XOR.
8. A truth table lists all possible input combinations with the corresponding outputs.
9. The NOT operation inverts logical values (bits): it changes *true* to *false* and *false* to *true*.
10. The result of an AND operation is true when both of the operands are true.
11. The result of an OR operation is true when one or both of the operands are true.
12. The result of an XOR operator is true when the operands are different.
13. An important property of the AND operator is that if one of the operands is false, the result is false.
14. An important property of the OR operator is that if one of the operands is true, the result is true.
15. An important property of the XOR operator is that if one of the operands is true, the result will be the inverse of the other operand.
16. The OR operator can be used to set bits. Set the desired positions in the mask to 1.
17. The AND operator can be used to clear bits. Set the desired positions in the mask to 0.

18. The XOR operator can be used to invert bits. Set the desired positions in the mask to 1.

19. The logical shift operation is applied to a pattern that does not represent a signed number. The arithmetic shift operation assumes that the bit pattern is a signed number in two's complement format.

Multiple-Choice Questions

20. c	21. d	22. c	23. c	24. c	25. b
26. d	27. b	28. a	29. c	30. d	31. c
32. d	33. c	34. b	35. a	36. a	37. c
38. a	39. b				

Exercises

40.

a.	**NOT $(99)_{16}$**	=	**NOT $(10011001)_2$**	=	**$(01100110)_2$**	=	**$(99)_{16}$**
b.	**NOT $(FF)_{16}$**	=	**NOT $(11111111)_2$**	=	**$(00000000)_2$**	=	**$(00)_{16}$**
c.	**NOT $(00)_{16}$**	=	**NOT $(00000000)_2$**	=	**$(11111111)_2$**	=	**$(FF)_{16}$**
d.	**NOT $(01)_{16}$**	=	**NOT $(00000001)_2$**	=	**$(11111110)_2$**	=	**$(FE)_{16}$**

41.

a.	**$(99)_{16}$ AND $(99)_{16}$**	=	**$(10011001)_2$ AND $(10011001)_2$**	=	**$(10011001)_2$**	=	**$(99)_{16}$**
b.	**$(99)_{16}$ AND $(00)_{16}$**	=	**$(10011001)_2$ AND $(00000000)_2$**	=	**$00000000)_2$**	=	**$(00)_{16}$**
c.	**$(99)_{16}$ AND $(FF)_{16}$**	=	**$(10011001)_2$ AND $(11111111)_2$**	=	**$(10011001)_2$**	=	**$(99)_{16}$**
d.	**$(99)_{16}$ AND $(FF)_{16}$**	=	**$(11111111)_2$ AND $(11111111)_2$**	=	**$(11111111)_2$**	=	**$(FF)_{16}$**

42.

a.	**$(99)_{16}$ OR $(99)_{16}$**	=	**$(10011001)_2$ OR $(10011001)_2$**	=	**$(10011001)_2$**	=	**$(99)_{16}$**
b.	**$(99)_{16}$ OR $(00)_{16}$**	=	**$(10011001)_2$ OR $(00000000)_2$**	=	**$(10011001)_2$**	=	**$(99)_{16}$**
c.	**$(99)_{16}$ OR $(FF)_{16}$**	=	**$(10011001)_2$ OR $(11111111)_2$**	=	**$(11111111)_2$**	=	**$(FF)_{16}$**
d.	**$(FF)_{16}$ OR $(FF)_{16}$**	=	**$(11111111)_2$ OR $(11111111)_2$**	=	**$(11111111)_2$**	=	**$(FF)_{16}$**

43.

a.

$$\textbf{NOT}[(99)_{16}\ \textbf{OR}\ (99)1_6] = \textbf{NOT}\ [(10011001)_2\ \textbf{OR}\ (10011001)_2] = (01100110)_2 = (66)_{16}$$

b.

$$(99)_{16}\ \textbf{OR}\ [\textbf{NOT}\ (00)_{16}] = (10011001)_2\ \textbf{OR}\ [\textbf{NOT}\ (00000000)_2] = (10011001)_2\ \textbf{OR}\ (11111111)_2 = (11111111)_2 = (FF)_{16}$$

c.

$$[(99)_{16} \textbf{ AND } (33)_{16}] \textbf{ OR } [(00)_{16} \textbf{ AND } (FF)_{16})$$
$$= [(10011001)_2 \textbf{ AND } (00110011)_2] \textbf{ OR } [(00000000)_2 \textbf{ AND } (11111111)_2]$$
$$= (00010001)_2 \textbf{ OR } (00000000)_2 = (00010001)_2 = (11)_{16}$$

d.

$$[(99)_{16} \textbf{ OR } (33)_{16}] \textbf{ AND } [(00)_{16} \textbf{ OR } (FF)_{16}]$$
$$= [(10011001)_2 \textbf{ OR } (00110011)_2] \textbf{ AND } [(00000000)_2 \textbf{ OR } (11111111)_2]$$
$$= (10111011)_2 \textbf{ AND } (11111111)_2 = (10111011)_2 = (BB)_{16}$$

44.

Mask = $(00001111)_2$

Operation: Mask **AND** $(xxxxxxxx)_2 = (0000xxxx)_2$

45.

Mask = $(00001111)_2$

Operation: Mask **OR** $(xxxxxxxx)_2 = (xxxx1111)_2$

46.

Mask: $(11000111)_2$

Operation: Mask **XOR** $(xxxxxxxx)_2 = (yyxxxyyy)_2$, where y is NOT x

47.

Mask1= $(00011111)_2$ Mask2 = $(00000011)_2$

Operation: [Mask1 **AND** $(xxxxxxxx)_2$] **OR** Mask2 = $(000xxx11)_2$

48. Arithmetic right shift divides an integer by 2 (the result is truncated to a smaller integer). To divide an integer by 4, we apply the arithmetic right shift operation twice.
49. Arithmetic left shift multiplies an integer by 2. To multiply an integer by 8, we apply the arithmetic left shift operation three times.
50. We assume that extraction is for bits 4 and 5 from left. Let the integer in question be (***abcdefgh***)$_2$.
    a. Apply logical right shift operation on (***abcdefgh***)$_2$ three times to obtain (**000*abcde***)$_2$.
    b. Let (**000*abcde***)$_2$ AND (**00000001**)$_2$ to extract the fifth bit: (0000000***e***)
    c. Apply logical right shift operation on (**000*abcde***)$_2$ once to obtain (**0000*abcd***)$_2$
    d. Let (**0000*abcd***)$_2$ AND (**00000001**)$_2$ to extract the fourth bit: (0000000***d***)

51.

a. 00010011 + 00010111

									Carry	Decimal
			1		1	1	1			
	0	0	0	1	0	0	1	1		19
+	0	0	0	1	0	1	1	1		23
	0	0	1	0	1	0	1	0		42

b. 00010011 – 00010111 = 000010011 + (–00010111) = 00010011 + 11101001 =

									Carry	Decimal
						1	1			
	0	0	0	1	0	0	1	1		19
+	1	1	1	0	1	0	0	1		–23
	1	1	1	1	1	1	0	0		–4

c. (–00010011) + 00010111 = 11101101 + 00010111

									Carry	Decimal
1	1	1	1	1	1	1	1			
	1	1	1	0	1	1	0	1		–19
+	0	0	0	1	0	1	1	1		23
	0	0	0	0	0	1	0	0		4

d. (–00010011) – 00010111 = (–00010011) + (–00010111) = 11101101 + 11101001 =

									Carry	Decimal
1	1	1		1			1			
	1	1	1	0	1	1	0	1		–19
+	1	1	1	0	1	0	0	1		–23
	1	1	0	1	0	1	1	0		–42

52.

a. 00000000 10100001 + 00000011 11111111 =

			Carry	Decimal
	1 1 1	1 1 1 1 1 1 1		
	0 0 0 0 0 0 0 0	1 0 1 0 0 0 0 1		161
+	0 0 0 0 0 0 1 1	1 1 1 1 1 1 1 1		1023
	0 0 0 0 0 1 0 0	1 0 1 0 0 0 0 0		1184

b. 00000000 10100001 – 00000011 11111111 = 00000000 10100001 + (–00000011 11111111) = 00000000 10100001 + 11111100 00000001 =

			Carry	Decimal
		1		
	0 0 0 0 0 0 0 0	1 0 1 0 0 0 0 1		161
+	1 1 1 1 1 1 0 0	0 0 0 0 0 0 0 1		–1023
	1 1 1 1 1 1 0 0	1 0 1 0 0 0 1 0		–862

c. (–00000000 10100001) + 00000011 11111111 = 11111111 01011111 + 00000011 11111111 =

1	**1 1 1 1 1 1 1 1**	**1 1 1 1 1 1 1**	**Carry**	Decimal
	1 1 1 1 1 1 1 1	0 1 0 1 1 1 1 1		–161
+	0 0 0 0 0 0 1 1	1 1 1 1 1 1 1 1		1023
	0 0 0 0 0 0 1 1	0 1 0 1 1 1 1 0		862

d. (-00000000 10100001) – 00000011 11111111 = (–00000000 10100001) + (–00000011 11111111) = 11111111 01011111 + 11111100 00000001 =

1	**1 1 1 1 1**	**1 1 1 1 1**	**Carry**	Decimal
	1 1 1 1 1 1 1 1	0 1 0 1 1 1 1 1		–161
+	1 1 1 1 1 1 0 0	0 0 0 0 0 0 0 1		–1023
	1 1 1 1 1 0 1 1	0 1 1 0 0 0 0 0		–1184

53. Addition of two integers does not create overflow if the result is in the range (–128 to +127).

a. Addition does not create overflow because (–62) + (+63) = 1 (in the range).

b. Addition does not create overflow because (+2) + (+63) = 65 (in the range).

c. Addition does not create overflow because (–62) + (–1) = –63 (in the range).

d. Addition does not create overflow because (+2) + (–1) = 1 (in the range).

54.

a. There is overflow because 32 + 105 = 137 is not in the range (–128 to +127).

b. There is no overflow because 32 – 105 = –73 is in the range (–128 to +127).

c. There is no overflow because –32 + 105 = 73 is in the range (–128 to +127).

d. There is overflow because –32 – 105 = –137 is not in the range (–128 to +127).

55.

a.

			1 1	**1 1**	**Carry**	Hexadecimal
	0 0 0 0	0 0 0 1	0 0 1 0	1 0 1 0		012A
+	0 0 0 0	1 1 1 0	0 0 1 0	0 1 1 1		0E27
	0 0 0 0	1 1 1 1	0 1 0 1	0 0 0 1		0F51

b.

1	**1 1 1**				**Carry**	Hexadecimal
	0 1 1 1	0 0 0 1	0 0 1 0	1 0 1 0		712A
+	1 0 0 1	1 1 1 0	0 0 0 0	0 0 0 0		9E00
	0 0 0 0	1 1 1 1	0 0 1 0	1 0 1 0		**1**0F2A

Note that the result is not valid because of overflow.

c.

					Carry	Hexadecimal
				1		
	1 0 0 0	0 0 0 0	0 0 0 1	0 0 0 1		8011
+	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 1		0001
	1 0 0 0	0 0 0 0	0 0 0 1	0 0 1 0		8012

d.

					Carry	Hexadecimal
1			1	1 1 1		
	1 1 1 0	0 0 0 1	0 0 1 0	1 0 1 0		E12A
+	1 0 0 1	1 1 1 0	0 0 1 0	0 1 1 1		9E27
	0 1 1 1	1 1 1 1	0 1 0 1	0 0 0 1		17F51

Note that the result is not valid because of overflow.

56. Number are stored in sign-and-magnitude format

a. $19 + 23 \rightarrow A = 19 = (00010011)_2$ and $B = 23 = (00010111)_2$.
Operation is addition; sign of B is not changed. $S = A_S$ XOR $B_S = 0$, $R_M = A_M + B_M$ and $R_S = A_S$

No overflow										Carry
			1		1	1	1			
A_S	0		0	0	1	0	0	1	1	A_M
B_S	0	+	0	0	1	0	1	1	1	B_M
R_S	0		0	1	0	1	0	1	0	R_M

The result is $(00101010)_2 = 42$ as expected.

b. $19 - 23 \rightarrow A = 19 = (00010011)_2$ and $B = 23 = (00010111)_2$. Operation is subtraction, sign of B is changed. $B_S = \overline{B_S}$, $S = A_S$ XOR $B_S = 1$, $R_M = A_M + (\overline{B_M} + 1)$. Since there is no overflow $R_M = (\overline{R_M} + 1)$ and $R_S = B_S$

No overflow										Carry
							1	1		
A_S	0		0	0	1	0	0	1	1	A_M
B_S	1	+	1	1	0	1	0	0	1	$(\overline{B_M} + 1)$
			1	1	1	1	1	0	0	R_M
R_S	1		0	0	0	0	1	0	0	$R_M = (\overline{R_M} + 1)$

The result is $(10000100)_2 = -4$ as expected.

c. $-19 + 23 \rightarrow A = -19 = (10010011)_2$ and $B = 23 = (00010111)_2$. Operation is addition, sign of B is not changed. $S = A_S$ XOR $B_S = 1$, $R_M = A_M + (\overline{B_M} + 1)$. Since there is no overflow $R_M = (\overline{R_M} + 1)$ and $R_S = B_S$

No overflow										Carry
							1	1		
A_S	1		0	0	1	0	0	1	1	A_M
B_S	0	+	1	1	0	1	0	0	1	$(\overline{B_M} + 1)$
			1	1	1	1	1	0	0	R_M
R_S	0		0	0	0	0	1	0	0	$R_M = (\overline{R_M} + 1)$

The result is $(00000100)_2 = 4$ as expected.

d. $-19 - 23 \rightarrow A = -19 = (10010011)_2$ and $B = 23 = (00010111)_2$. Operation is subtraction, sign of B is changed. $S = A_S$ XOR $B_S = 0$, $R_M = A_M + B_M$ and $R_S = A_S$

		No overflow			1		1	1	1		Carry
A_S	1			0	0	1	0	0	1	1	A_M
B_S	1		+	0	0	1	0	1	1	1	B_M
R_S	1			0	1	0	1	0	1	0	R_M

The result is $(10101010)_2 = -42$ as expected.

57.

a. $34.75 + 23.125 = (100010.11)_2 + (10111.001)_2 = 2^5 \times (1.0001011)_2 + 2^4 \times (1.0111001)_2$. These two numbers are stored in floating-point format as shown, but we need to remember that each number has a hidden 1 (which is not stored, but assumed). $E_1 = 127 + 5 = 132 = (10000100)_2$ and $E_2 = 127 + 4 = 131 = (10000011)_2$. The first few steps in UML diagram is not needed. We move to denormalization. We denormalize the numbers by adding the hidden 1's to the mantissa and incrementing the exponent.

	S	E	M
A	0	10000100	00010110000000000000000
B	0	10000011	01110010000000000000000

Now both denormalized mantissas are 24 bits and include the hidden 1's. They should store in a location to hold all 24 bits. Each exponent is incremented.

	S	E	Denormalized M
A	0	10000101	**1**00010110000000000000000
B	0	10000100	**1**01110010000000000000000

We align the mantissas. We increment the second exponent by 1 and shift its mantissa to the right once.

	S	E	Denormalized M
A	0	10000101	**1**00010110000000000000000
B	0	10000101	0**1**01110010000000000000000

Now we do sign-and-magnitude addition treating the sign and the mantissa of each number as one integer stored in sign-and-magnitude representation.

	S	E	Denormalized M
R	0	10000101	111001111000000000000000

There is no overflow in mantissa, so we normalized.

	S	E	M
R	0	10000100	11001111000000000000000

The mantissa is only 23 bits because there is no overflow, no rounding is needed.

$$E = (10000100)_2 = 132,\ M = 11001111$$

In other words, the result is

$$(1.11001111)_2 \times 2^{132-127} = (111001.111)_2 = 57.875$$

b. $-12.625 + 451 = -(1100.101)_2 + (111000011)_2 = -2^3 \times (1.100101)_2 + 2^8 \times (1.11000011)_2$. These two numbers are stored in floating-point format as shown, but we need to remember that each number has a hidden 1 (which is not stored, but assumed). $E_1 = 127 + 3 = 130 = (10000010)_2$ and $E_2 = 127 + 8 = 135 = (10000111)_2$

	S	E	M
A	1	10000010	10010100000000000000000
B	0	10000111	11000011000000000000000

The first few steps in UML diagram is not needed. We move to denormalization. We denormalize the numbers by adding the hidden 1's to the mantissa and incrementing the exponent. Now both denormalized mantissas are 24 bits and include the hidden 1's. They should store in a location to hold all 24 bits. Each exponent is incremented.

	S	E	Denormalized M
A	1	10000011	**1**10010100000000000000000
B	0	10001000	**1**11000011000000000000000

We align the mantissas. We increment the first exponent by 5 and shift its mantissa to the right five times.

	S	E	Denormalized M
A	1	10001000	00000**1**100101000000000000
B	0	10001000	**1**11000011000000000000000

Now we do sign-and-magnitude addition treating the sign and the mantissa of each number as one integer stored in sign-and-magnitude representation.

	S	E	Denormalized M
R	0	10001000	110110110011000000000000

There is no overflow in mantissa, so we normalized.

	S	E	M
R	0	10000111	10110110011000000000000

The mantissa is only 23 bits because there is no overflow, no rounding is needed.

$$E = (10000111)_2 = 135,\ M = 10110110011$$

In other words, the result is

$$(1.10110110011)_2 \times 2^{135-127} = (110110110.011)_2 = 438.375$$

c. $33.1875 - 0.4375 = (100001.0011)_2 - (0.0111)_2 = 2^5 \times (1.000010011)_2 - 2^{-2} \times (1.11)_2$. These two numbers are stored in floating-point format as shown, but we need to remember that each number has a hidden 1 (which is not stored, but assumed). $E_1 = 127 + 5 = 132 = (10000100)_2$ and $E_2 = 127 + (-2) = 125 = (01111101)_2$

	S	E	M
A	0	10000100	00001001100000000000000
B	0	01111101	11000000000000000000000

The first two steps in UML diagram is not needed. Since the operation is subtraction, we change the sing of the second number.

	S	E	M
A	0	10000100	00001001100000000000000
B	1	01111101	11000000000000000000000

We denormalize the numbers by adding the hidden 1's to the mantissa and incrementing the exponent. Now both denormalized mantissas are 24 bits and include the hidden 1's. They should store in a location to hold all 24 bits. Each exponent is incremented.

	S	E	Denormalized M
A	0	10000101	**1**00001001100000000000000
B	1	01111110	**1**11000000000000000000000

We align the mantissas. We increment the second exponent by 7 and shift its mantissa to the right seven times.

	S	E	Denormalized M
A	0	10000101	**1**00001001100000000000000
B	1	10000101	0000000**1**1100000000000000

Now we do sign-and-magnitude addition treating the sign and the mantissa of each number as one integer stored in sign-and-magnitude representation.

	S	E	Denormalized M
R	0	10000101	100000110000000000000000

There is no overflow in mantissa, so we normalized.

	S	E	M
R	0	10000100	00000110000000000000000

The mantissa is only 23 bits because there is no overflow, no rounding is needed.

$$E = (10000100)_2 = 132,\ M = 0000011$$

The result is

$$(1.0000011)_2 \times 2^{132-127} = (100000.11)_2 = 32.75$$

d. $-344.3125 - 123.5625 = -(101011000.0101)_2 - (1111011.1001)_2 = 2^8 \times (1.010110000101)_2 - 2^6 \times (1.1110111001)_2$. These two numbers are stored in floating-point format as shown, but we need to remember that each number has a hidden 1 (which is not stored, but assumed). $E_1 = 127 + 8 = 135 = (10000111)_2$ and $E_2 = 127 + 6 = 133 = (10000101)_2$

	S	E	M
A	1	10000111	01011000010100000000000
B	0	10000101	11101110010000000000000

The first two steps in UML diagram is not needed. Since the operation is subtraction, we change the sing of the second number.

	S	E	M
A	1	10000111	01011000010100000000000
B	1	10000101	11101110010000000000000

We denormalize the numbers by adding the hidden 1's to the mantissa and incrementing the exponent. Now both denormalized mantissas are 24 bits and include the hidden 1's. They should store in a location to hold all 24 bits. Each exponent is incremented.

	S	E	M
A	1	10001000	**1**01011000010100000000000
B	1	10000110	**1**11101110010000000000000

We align the mantissas. We increment the second exponent by 7 and shift its mantissa to the right seven times.

	S	E	M
A	1	10001000	**1**01011000010100000000000
B	1	10001000	00**1**111011100100000000000

Now we do sign-and-magnitude addition treating the sign and the mantissa of each number as one integer stored in sign-and-magnitude representation.

	S	E	Denormalized M
R	1	10001000	111010011111000000000000

There is no overflow in mantissa, so we normalized.

	S	E	Denormalized M
R	1	10000111	11010011111000000000000

The mantissa is only 23 bits because there is no overflow, no rounding is needed.

$$E = (10000111)_2 = 135, \; M = 11010011111$$

The result is

$$(1.11010011111)_2 \times 2^{135-127} = (111010011.111)_2 = 467.875$$

58. We assume that both operands are in the presentable range.
   a. Overflow can occur because the magnitude of the result is greater than the magnitude of each number and could fall out of the presentable range.
   b. Overflow does not occur because the magnitude of the result is smaller than one of the numbers; the result is in the presentable range.
   a. When we subtract a positive integer from a negative integer, the magnitudes of the numbers are added. This is the negative version of case *a*. Overflow can occur.
   b. When we subtract two negative numbers, the magnitudes are subtracted from each other. This is the negative version of case b. Overflow does not occur.
59. The result is a number with all 1's which has the value of –0. For example, if we add number $(10110101)_2$ in 8-bit allocation to its one's complement $(01001010)_2$ we obtain

									Decimal equivalent
	1	0	1	1	0	1	0	1	–74
+	0	1	0	0	1	0	1	0	+74
	1	1	1	1	1	1	1	1	–0

   We use this fact in the Internet checksum in Chapter 6.
60. The result is a number with all 0's which has the value of 0. For example, if we add number $(10110101)_2$ in 8-bit allocation to its two's complement $(01001011)_2$ we obtain

									Decimal equivalent
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**Carry**	
	1	0	1	1	0	1	0	1	–74
+	0	1	0	0	1	0	1	1	+74
	0	0	0	0	0	0	0	0	0

   We use this fact in normal mathematical calculation in the computers.