CHAPTER 4

Operations On Data

(Solutions to Odd-Numbered Problems)

Review Questions

1. Arithmetic operations interpret bit patterns as numbers. Logical operations interpret each bit as a logical values (*true* or *false*).
3. The bit allocation can be 1. In this case, the data type normally represents a logical value.
5. The decimal point of the number with the smaller exponent is shifted to the left until the exponents are equal.
7. The common logical binary operations are: AND, OR, and XOR.
9. The NOT operation inverts logical values (bits): it changes *true* to *false* and *false* to *true*.
11. The result of an OR operation is true when one or both of the operands are true.
13. An important property of the AND operator is that if one of the operands is false, the result is false.
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.
17. The AND operator can be used to clear bits. Set the desired positions in the mask to 0.
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

21. d	23. c	25. b	27. b	29. c	31. c
33. c	35. a	37. c	39. b		

Exercises

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 $(11111111)_2$	=	$(11111111)_2$	=	$(FF)_{16}$

43.

a.

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

b.

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

c.

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

d.

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

45.

Mask = $(00001111)_2$
Operation: Mask **OR** $(xxxxxxxx)_2 = (xxxx1111)_2$

47.

Mask1= $(00011111)_2$ Mask2 = $(00000011)_2$
Operation: [Mask1 **AND** $(xxxxxxxx)_2$] **OR** Mask2 = $(000xxx11)_2$

49. Arithmetic left shift multiplies an integer by 2. To multiply an integer by 8, we apply the arithmetic left shift operation three times.

51.

a. 00010011 + 00010111

			1		**1**	**1**	**1**		**Carry**	Decimal
	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 =

						1	**1**		**Carry**	Decimal
	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

1	**1**	**1**	**1**	**1**	**1**	**1**	**1**		**Carry**	Decimal
	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 =

1	**1**	**1**		**1**			**1**		**Carry**	Decimal
	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

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).

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.

					Carry	Hexadecimal
1	**1 1 1**					
	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		**1**7F51

Note that the result is not valid because of overflow

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**0111001000000000000000

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	11100111100000000000000

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**1001010000000000000
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**11101110010000000000

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	11101001111000000000000

There is no overflow in mantissa, so we normalized.

	S	E	Denormalized M
R	1	10000111	1101001111000000000000

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

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

The result is

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

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.