CHAPTER 4

Operations On Data

(Solutions to Review Questions and Problems)

Review Questions

Q4-1. Arithmetic operations interpret bit patterns as numbers. Logical operations interpret each bit as a logical value (*true* or *false*).

Q4-3. The bit allocation can be 1. In this case, the data type normally represents a logical value.

Q4-5. The decimal point of the number with the smaller exponent is shifted to the left until the exponents are equal.

Q4-7. The common logical binary operations are: AND, OR, and XOR.

Q4-9. The NOT operation inverts logical values (bits): it changes *true* to *false* and *false* to *true*.

Q4-11. The result of an OR operation is true when one or both of the operands are true.

Q4-13. An important property of the AND operator is that if one of the operands is false, the result is false.

Q4-15. 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.

Q4-17. The AND operator can be used to clear bits. Set the desired positions in the mask to 0.

Q4-19. 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.

Problems

P4-1.

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 $(00000001)_2$	=	$(11111110)_2$	=	$(FE)_{16}$

P4-3.

$(99)_{16}$ OR $(99)_{16}$ = $(10011001)_2$ OR $(10011001)_2$ = $(10011001)_2$ = $(99)_{16}$

a.

$(99)_{16}$ OR $(00)_{16}$ = $(10011001)_2$ OR $(00000000)_2$ = $(10011001)_2$ = $(99)_{16}$

b.

$(99)_{16}$ OR $(FF)_{16}$ = $(10011001)_2$ OR $(11111111)_2$ = $(11111111)_2$ = $(FF)_{16}$

c.

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

P4-5.

Mask = $(00001111)_2$
Operation: Mask **AND** $(xxxxxxxx)_2 = (0000xxxx)_2$

P4-7.

Mask: $(11000111)_2$
Operation: Mask **XOR** $(xxxxxxxx)_2 = (yyxxxyyy)_2$, where y is NOT x

P4-9. 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.

P4-11. We assume that extraction is for bits 4 and 5 from left. Let the integer in question be $(\boldsymbol{abcdefgh})_2$.

a. Apply logical right shift operation on $(\boldsymbol{abcdefgh})_2$ three times to obtain $(\mathbf{000}\boldsymbol{abcde})_2$.

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***)

P4-13.

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 =

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

																	Carry	Decimal
1	1	1	1	1	1						1	1	1	1	1			
	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

P4-15.

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).

P4-17. 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					1	1		Carry
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.

P4-19. 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.