Chapter 1

Problem 1.1

A. $25 \div ((1000 \div 100) + (1500 \div 150)) = 1.25\times$

B. $1500 \div (25 \div 1.67 - 10) = 300(km/h)$

Problem 1.2

$(1 - 0.1) \div (\frac{1}{4} - 0.1) = 6\times$

Chapter 2

Problem 2.1

A. `0010 0101 1011 1001 1101 0010`

B. `0xAE49`

C. `1010 1000 1011 0011 1101`

D. `0x322D96`

Problem 2.2

n	2^n(decimal)	2^n(hexadecimal)
5	32	0x20
23	8388608	0x80000
15	32768	0x8000
13	8192	0x2000
12	4096	0x1000
6	64	0x40
8	256	0x100

Problem 2.3

Decimal	Binary	Hexadecimal
0	0000 0000	0x00
158	1001 1110	0x9E
76	0100 1100	0x7C
145	1001 0001	0x91
174	1010 1110	0xAE
60	0011 1100	0x3C
241	1111 0001	0xF1
116	0111 0101	0x75
189	1011 1101	0xBD
245	1111 0101	0xF5

Problem 2.4

A. `0x605C+0x5=0x6061`

B. `0x605C-0x20=0x603C`

C. `0x605C+32=0x607C`

D. `0x60FA-0x605C=0x9e`

Problem 2.5

A. Little endian: 78 Big endian: 12

B. Little endian: 78 56 Big endian: 12 34

C. Little endian: 78 56 34 Big endian: 12 34 56

Problem 2.6

A. `0x0027C8F8: 0000 0000 0010 0111 1100 1000 1111 1000`
`0x4A1F23E0: 0100 1010 0001 1111 0010 0011 1110 0000`

Problem 2.7

`6d 6e 6f 70 71 72`

Problem 2.8

Operation	Result
`a`	`01001110`
`b`	`11100001`
`~a`	`10110001`
`~b`	`00011110`
`a&b`	`01000000`
`a|b`	`11101111`
`a^b`	`10101111`

Problem 2.9

A.

Color	Complement
White	Black
Blue	Yellow
Green	Magenta
Cyan	Red

B. Blue|Green = Cyan
Yellow&Cyan = Green
Red^Magenta = Blue

Problem 2.10

Step	`*x`	`*y`
Initially	`a`	`b`
Step 1	`a`	`a^b`
Step 2	`b`	`a^b`
Step 3	`b`	`a`

Problem 2.11

A. first=last=k

B. The funtion implace_swap was given two identical arguments.

C. Change line 4 to "`first < last`".

Problem 2.12

A. `x&0xFF`

B. `x^~0xFF`

C. `x|0xFF`

Problem 2.13

```
bis(x, y)
```

```
bis(bic(x, y), bitc(y, x))
```

Problem 2.14

Expression	Value	Expression	Value
`a&b`	`0x44`	`a&&b`	1
`a|b`	`0x57`	`a||b`	1
`~a|~b`	`0xBB`	`!a||!b`	0
`a&!b`	`0`	`a&&~b`	1

Problem 2.15

```
bool equal(int x, int y) {
    return !(x ^ y);
}
```

Problem 2.16

`a`		`a<<2`		`a>>3(Logical)`		`a>>3(Arithmetic)`	
Hex	Binary	Hex	Binary	Hex	Binary	Hex	Binary
`0xD4`	`1101 0100`	`0x50`	`0101 0000`	`0x1A`	`0001 1010`	`0xFA`	`1111 1010`
`0x64`	`0110 0100`	`0x90`	`1001 0000`	`0x0C`	`0000 1100`	`0x0C`	`0000 1100`
`0x72`	`0111 0010`	`0xC8`	`1100 1000`	`0x0E`	`0000 1110`	`0x0E`	`0000 1110`
`0x44`	`0100 0100`	`0x10`	`0001 0000`	`0x08`	`0000 1000`	`0x08`	`0000 1000`

Problem 2.17

Hexadecimal	Binary	$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
`0xA`	`1010`	$2^3 + 2^1 = 10$	$-2^3 + 2^1 = -6$
`0x1`	`0001`	$2^0 = 1$	$2^0 = 1$
`0xB`	`1011`	$2^3 + 2^1 + 2^0 = 11$	$-2^3 + 2^1 + 2^0 = -5$
`0x2`	`0010`	$2^1 = 2$	$2^1 = 2$
`0x7`	`0111`	$2^2 + 2^1 + 2^0 = 7$	$2^2 + 2^1 + 2^0 = 7$
`0xC`	`1100`	$2^3 + 2^2 = 12$	$-2^3 + 2^2 = -4$

Problem 2.18

A. `0x2e0=736`

B. `-0x58=-88`

C. `0x28=40`

D. `-0x30=-48`

E. `0x78=120`

F. `0x88=136`

G. `0x1f8=504`

H. `0xc0=192`

I. `-0x48=-72`

Problem 2.19

x	$T2U_4(x)$
-1	15
-5	11
-6	10
-4	12
1	1
8	8

Problem 2.20

pass

Problem 2.21

Expression	Type	Evaluation
`-2147483647-1 == 2147483648U`	Unsigned	1
`-2147483647-1 < 2147483647`	Signed	1
`-2147483647-1U < 2147483647`	Unsigned	0
`-2147483647-1 < -2147483647`	Signed	1
`-2147483647-1U < -2147483647`	Unsigned	1

Problem 2.22

A. $1100_2 = -2^3 + 2^2 = -4$

B. $11100_2 = -2^4 + 2^3 + 2^2 = -4$

C. $111100_2 = -2^5 + 2^4 + 2^3 + 2^2 = -4$

Problem 2.23

A.

w	fun1(w)	fun2(w)
`0x00000076`	`0x00000076`	`0x00000076`
`0x87654321`	`0x00000021`	`0x00000021`
`0x000000C9`	`0x000000C9`	`0xFFFFFFC9`
`0xEDCBA987`	`0x00000087`	`0xFFFFFF87`

B. fun1 return the zero extension of the least significant byte of w.
fun2 return the sign extension of the least significant byte of w.

Problem 2.24

Hex		Unsigned		Signed	
Original	Truncated	Original	Truncated	Original	Truncated
1	1	1	1	1	1
3	3	3	3	3	3
5	5	5	5	5	5
C	4	12	4	-4	-4
E	6	14	6	-2	-2

Problem 2.25

Reason: When length equals 0, length minus 1 equals $UMax_{32}$, so the expression $i \leq UMax_{32}$ holds for any unsigned i and hence the for loop would never stop.

Correction: Change the expression `i <= length - 1` to `i < length`.

Problem 2.26

A. When string s is shorter than string t.

B. The data type of strlen(s)-strlen(t) is unsigned, so it will be greater than 0 for any different strlen(s) and strlen(t). So when s is short than t, this function will return a wrong answer.

C. Change the return value to strlen(s) > strlen(t).

Problem 2.27

```
int uadd_ok(unsigned x, unsigned y) {
    return x + y < x || x + y < y;
}
```

Problem 2.28

x		$-^u_4 x$	
Hex	Decimal	Decimal	Hex
1	1	15	F
4	4	11	B
7	7	9	9
A	10	6	6
E	14	2	2

Problem 2.29

x	y	$x+y$	$x +^t_5 y$	Case
-12	-15	-27	5	1
10100	10001	100101	00101	1
-8	-8	-16	-16	2
11000	11000	110000	10000	2
-9	8	-1	-1	2
10111	01000	11111	11111	2
2	5	7	7	3
00010	00101	00111	00111	3
12	4	16	-16	4
01100	00100	10000	10000	4

Problem 2.30

```
int tadd_ok(int x, int y) {
    int z = x + y;
    return !((x > 0 && y > 0 && z <= 0) || (x < 0 && y < 0 && z >= 0));
}
```

Problem 2.31

Signed addition is associative and commutative, so (x+y)-x = y+(x-x)=y and hence whether or not there is an overflow, this function will always return 1.

Problem 2.32

For any x and y = $TMin$, this function will give incorrect results.

Problem 2.33

x		$-^t_4 x$	
Hex	Decimal	Decimal	Hex
2	2	-2	E
3	3	-3	D
9	9	-9	7
B	11	-11	5
C	12	-12	4

The bit patterns generated by two's complement and unsigned negation are identical.

Problem 2.34

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
	Hex	Binary	Hex	Binary	Hex	Binary	Hex	Binary
Unsigned	4	100	5	101	20	010100	4	100
Two's complement	-4	100	-3	101	12	001100	-4	100
Unsigned	2	010	7	111	14	001110	6	110
Two's complement	2	010	-1	111	-2	111110	-2	110
Unsigned	6	110	6	110	36	100100	4	110
Two's complement	-2	110	-2	110	4	000100	-4	100

Problem 2.35

pass

Problem 2.36

```
int tmult_ok(int x, int y) {
    int64_t z1 = (int64_t)x * y;
    int z2 = x * y;
    return (int64_t)z2 == z1;
}
```

Problem 2.37

A. No improvement at all. Although variable asize is 64-bit and its value is accurate, when it is passed to malloc as a parameter with type size_t, it will still be truncated to 32 bit as well.

B. Since the parameter of malloc is size_t with 32 bit, it's impossible to allocate more than 2^{32} bytes. What we can do is to determine whether there is an overflow before malloc. If there is, do not call malloc and return NULL.

Problem 2.38

A power of $2(2^k, for\ any\ k > 0)$ or A power of 2 plus $1(2^k + 1, for\ any\ k > 0)$.

Problem 2.39

$-(x << m)$

Problem 2.40

K	Shifts	Add/Subs	Expression
7	1	1	$(x << 3) - x$
30	4	3	$(x << 4) + (x << 3) + (x << 2) + (x << 1)$
28	2	1	$(x << 5) - (x << 2)$
55	2	2	$(x << 6) - (x << 3) - x$

Problem 2.41

When m = n and m + 1 = n, choose form A, otherwise form B.

Problem 2.42

```
int div16(int x) {
    return (x + ((x >> 31) & 0xF)) >> 4;
}
```

Problem 2.43

M = 31, N = 8.

Problem 2.44

A. False for x = -2147483648.

B. True. If (x & 7) != 7 is false, namely (x & 7) == 7, the least 3 significant bits must be [111]. So the most 3 significant bits of x « 29 will be 111 and hence x « 29 < 0.

C. False for x = 50000 where the value of x * x equals 2500000000 > 2147483647, causes positive overflow and yields a negative value.

D. True. For any x >= 0, -x must be smaller than or equal to 0. Negation of a nonnegative value will never cause an overflow.

E. False. This is true for any value of type `int` except -2147483648. Negation of -2147483648 is still -2147483648 and is smaller than 0.

F. True. Two's complement addition has the same bit-level represetation as unsigned.

G. True