Chapter 1 A Tour of Computer Systems

Problem 1.1

A. We use the formula with $\alpha = \frac{3}{5}$ and $k = \frac{150}{100} = \frac{3}{2}$.

$$\begin{aligned} S &= \frac{1}{\left(1 - \frac{3}{5}\right) + \frac{3}{5} \cdot \frac{2}{3}} \\ &= \frac{1}{\frac{2}{5} + \frac{2}{5}} \\ &= \frac{5}{4} \\ &= 1.25\times \end{aligned}$$

B. We use the formula and work our way back:

$$\begin{aligned} \frac{5}{3} &= \frac{1}{\left(1 - \frac{3}{5}\right) + \frac{3}{5k}} \\ \frac{3}{5} &= \frac{2}{5} + \frac{3}{5k} \\ \frac{1}{5} &= \frac{3}{5k} \\ 1 &= \frac{3}{k} \\ k &= 3 \end{aligned}$$

So the drive through Montana needs a speedup of $3\times$ which is 300 km/hr.

Problem 1.2

Use the formula with $\alpha = \frac{4}{5}$ and $S = 2$ and solve for k.

$$\begin{aligned} 2 &= \frac{1}{\left(1 - \frac{4}{5}\right) + \frac{4}{5k}} \\ \frac{2}{5} + \frac{8}{5k} &= 1 \\ \frac{8}{5k} &= \frac{3}{5} \\ \frac{1}{k} &= \frac{3}{8} \\ k &= \frac{8}{3} \end{aligned}$$

Chapter 2 Representing and Manipulating Information

Problem 2.1

A. `0x39A7F8` to binary: `0011 1001 1010 0111 1111 1000`

B. `1100100101111011` to hexadecimal: `0xC97B`

C. `0xD5E4C` to binary: `1101 0101 1110 0100 1100`

D. `1001101110011110110101` to hexadecimal: `0x26E7B5`

Problem 2.2

n	2^n (decimal)	2^n (hexadecimal)
`9`	`512`	`0x200`
`19`	`524288`	`0x80000`
`14`	`16384`	`0x4000`
`16`	`65536`	`0x10000`
`17`	`131072`	`0x20000`
`5`	`32`	`0x20`
`7`	`128`	`0x80`

Problem 2.3

Decimal	Binary	Hexadecimal
`0`	`0000 0000`	`0x00`
`167`	`1010 0111`	`0xA7`
`62`	`0011 1110`	`0x3E`
`188`	`1011 1100`	`0xBC`
`55`	`0011 0111`	`0x37`
`136`	`1000 1000`	`0x88`
`243`	`1111 0011`	`0xF3`
`82`	`0101 0010`	`0x52`
`172`	`1010 1100`	`0xAC`

Decimal	Binary	Hexadecimal
`231`	`1110 0111`	`0xE7`

Problem 2.4

A. `0x503c` $+$ `0x8` $=$ `0x5044`

B. `0x503c` $-$ `0x40` $=$ `0x4ffc`

C. `0x503c` $+$ `64` $=$ `0x507c`

D. `0x50ea` $-$ `0x503c` $=$ `0xae`

Problem 2.5

	Little endian	Big endian
A.	`21`	`87`
B.	`21 43`	`87 65`
C.	`21 43 65`	`87 65 43`

Problem 2.6

A.

`0x00359141` in binary: `0000 0000 0011 0101 1001 0001 0100 0001`

`0x4A564504` in binary: `0100 1010 0101 0110 0100 0101 0000 0100`

B.

```
00000000001101011001000101000001
  01001010010101100100010100000100
           *********************
```

There are 21 matching bits.

C.

The whole integer occurs in the float representation, except for the most-significant bit which is a 1. Similarly, some of the most-significant bits of the float representation do not occur in the int representation.

Problem 2.7

It prints `61 62 63 64 65 66` (it does not print the terminating null character because the `strlen` function does not count it).

Problem 2.8

Operation	Result
`a`	`[01101001]`
`b`	`[01010101]`
`~a`	`[10010110]`
`~b`	`[10101010]`
`a & b`	`[01000001]`
`a | b`	`[01111101]`
`a ^ b`	`[00111100]`

Problem 2.9

A. The following colors completent each other:

Black ↔ White
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	`a ^ (a ^ b) = b`	`a ^ b`
Step 3	`b`	`b ^ (a ^ b) = a`

Problem 2.11

A. In the final iteration we have `first = k` and `last = k` (swap the middle element with itself).

B. In this case `*x` and `*y` point to the same address and the steps become:

Step	*x	*y
Initially	`a`	`a`
Step 1	`a ^ a = 0`	`a ^ a = 0`
Step 2	`0 ^ 0 = 0`	`0 ^ 0 = 0`
Step 3	`0 ^ 0 = 0`	`0 ^ 0 = 0`

C. We can fix it by changing the condition to `first < last` since the middle element does not need to be swapped anyway.

Problem 2.12

A. `x & 0xFF` leaves the least significant byte and sets everything else to zero.

B. `x ^ ~0xFF` inverts everything except the least significant byte.

C. `x | 0xFF` sets the least significant byte to ones and leaves everything else.

Problem 2.13

`x | y` is equivalent to `bis(x, y)`.

`x ^ y` is equivalent to `bis(bic(x, y), bic(y, x))`.

Problem 2.14

We have `x = 0110 0110` and `y = 0011 1001`.

Expression	Value	Expression	Value
`x & y`	`0010 0000`	`x && y`	`1`
`x | y`	`0111 1111`	`x || y`	`1`
`~x | ~y`	`1111 1111 1111 1111 1111 1111 1101 1111` (assuming 32-bit int)	`!x || !y`	`0`
`x & !y`	`0`	`x && ~y`	`1`

Problem 2.15

`!(x ^ y)` is equivalent to `x == y` because `x ^ y` will be `0` only if all the bits match.

Problem 2.16

x	x << 3	x >> 2 (logical)	x >> 2 (arithmetic)
`0xC3 = 1100 0011`	`0001 1000 = 0x18`	`0011 0000 = 0x30`	`1111 0000 = 0xF0`
`0x75 = 0111 0101`	`1010 1000 = 0xA8`	`0001 1101 = 0x1D`	`0001 1101 = 0x1D`
`0x87 = 1000 0111`	`0011 1000 = 0x38`	`0010 0001 = 0x21`	`1110 0001 = 0xE1`
`0x66 = 0110 0110`	`0011 0000 = 0x30`	`0001 1001 = 0x19`	`0001 1001 = 0x19`

Problem 2.17

Hexadecimal	Binary	$B2U_4(x)$	$B2T_4(x)$
`0xE`	`[1110]`	$2^3 + 2^2 + 2^1 = 14$	$-2^3 + 2^2 + 2^1 = -2$
`0x0`	`[0000]`	0	0
`0x5`	`[0101]`	$2^2 + 2^0 = 5$	$2^2 + 2^0 = 5$
`0x8`	`[1000]`	$2^3 = 8$	$-2^3 = -8$
`0xD`	`[1101]`	$2^3 + 2^2 + 2^0 = 13$	$-2^3 + 2^2 + 2^0 = -3$
`0xF`	`[1111]`	$2^3 + 2^2 + 2^1 + 2^0 = 15$	$-2^3 + 2^2 + 2^1 + 2^0 = -1$

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	$T2U_4(x)$
-8	8
-3	$2^3 + 2^2 + 2^0 = 13$
-2	$2^3 + 2^2 + 2^1 = 14$
-1	$2^3 + 2^2 + 2^1 + 2^0 = 15$
0	0
5	5

Problem 2.20

Equation 2.5 can be used to solve the previous problem. Since $\omega = 4$, we need to add $2^4 = 16$ to all negative numbers in Two's Complement. For example, $-8 + 16 = 8$ and $-1 + 16 = 15$. Positive numbers (and zero) stay the same.

Problem 2.21

Expression	**Type**	**Evaluation**
$-2147483647 - 1 == 2147483648\text{U}$	Unsigned	1
$-2147483647 - 1 < 2147483647$	Signed	1
$-2147483647 - 1\text{U} < 2147483647$	Unsigned	0
$-2147483647 - 1 < -2147483647$	Signed	1
$-2147483647 - 1\text{U} < -2147483647$	Unsigned	1

Problem 2.22

A. $[1011] = -2^3 + 2^1 + 2^0 = -5$
B. $[11011] = -2^4 + 2^3 + 2^1 + 2^0 = -5$
C. $[111011] = -2^5 + 2^4 + 2^3 + 2^1 + 2^0 = -5$

Problem 2.23

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

`fun1` keeps only the least significant byte and sets the other three to all zeroes, resulting in a value between 0 and 255. `fun2` also extracts the least significant byte, but it performs sign extension instead of zero extension, which results in a value between −128 and 127.

Problem 2.24

Hex		Unsigned		Two's complement	
Original	Truncated	Original	Truncated	Original	Truncated
0	0	0	0	0	0
2	2	2	2	2	2
9	1	9	1	−7	1
B	3	11	3	−5	3
F	7	15	7	−1	−1

We can use the equations to verify these results. For example, in hex F truncates to 7, in unsigned $B2U_4(1111) \bmod 2^3 = 7$ and in two's complement $U2T_3(B2U_4(1111) \bmod 2^3) = -1$.

Problem 2.25

Because `length` is unsigned the expression $0 - 1$ evaluates to UMax. The comparison has an unsigned integer on one side, which means the other side will also be treated as unsigned. Of course every unsigned number is $\leq$ UMax and so we try to access invalid array elements.

We can fix it by changing the condition to `i < length` or changing `length` to a signed integer.

Problem 2.26

A. The function returns wrong results in case `t` is longer than `s`.

B. The problem is that `strlen` returns a `size_t` which is unsigned. When calculating `strlen(s) - strlen(t)` where `t` is longer than `s` unsigned arithmetic is used, resulting in a number close to UMax instead of a negative number. This is obviously greater than 0 so the function incorrectly says that `s` is longer.

C. We can fix it by changing the condition to `strlen(s) > strlen(t)`.

Problem 2.27

```c
/* Determine whether arguments can be added without overflow */
int uadd_ok(unsigned x, unsigned y) {
  return x + y >= x;
}
```

Problem 2.28

x		$-^u_\omega x$	
Hex	**Decimal**	**Decimal**	**Hex**
0	0	0	0
5	5	11	B
8	8	8	8
D	13	3	3
F	15	1	1

Problem 2.29

x	y	$x + y$	$x +^t_5 y$	Case
−12 [10100]	−15 [10001]	−27 [100101]	5 [00101]	1
−8 [11000]	−8 [11000]	−16 [110000]	−16 [10000]	2
−9 [10111]	8 [01000]	−1 [111111]	−1 [11111]	2
2 [00010]	5 [00101]	7 [000111]	7 [00111]	3
12 [01100]	4 [00100]	16 [010000]	−16 [10000]	4

Problem 2.30

```c
/* Determine whether arguments can be added without overflow */
int tadd_ok(int x, int y) {
  int sum = x + y;

  if (x > 0 && y > 0) {
    return sum > 0;
  }

  if (x < 0 && y < 0) {
    return sum < 0;
  }

  return 1;
}
```

Problem 2.31

sum $-$ x can overflow again, since it's another two's complement addition. For example, if x and y are large positive numbers whose sum overflows to a negative number, then sum $-$ x will cause a negative overflow "wrapping back around" to y. So this check will not detect the overflow.

Problem 2.32

The function will be incorrect for $y = \text{TMin}_\omega$. This is because the two's complement representation is not symmetric. $-y = -\text{TMin}_\omega = \text{TMin}_\omega$ causes an overflow possibly resulting in an incorrect return value.

Problem 2.33

x		$-^t_4 x$	
Hex	Decimal	Decimal	Hex
0	0	0	0
5	5	−5	B
8	−8	−8	8
D	−3	3	3
F	−1	1	1

The bit patterns for two's complement and unsigned negation are the same.

Problem 2.34

Mode	x	y	$x \cdot y$	Truncated $x \cdot y$
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	$6 = [110]$	$6 = [110]$	$36 = [100100]$	$4 = [100]$
Two's complement	$-2 = [110]$	$-2 = [110]$	$4 = [000100]$	$-4 = [100]$

Problem 2.35

1. Let $t = u + p_{\omega-1}$ where u is the two's complement number represented by the ω upper bits of the 2ω-bit representation of $x \cdot y$. Since $p_{\omega-1}$ is either 0 or 1, there are two possibilities for t to equal 0.

1. If $p_{\omega-1} = 0$ then it must be that $u = 0$ (upper ω bits are all 0s).
2. If $p_{\omega-1} = 1$ then it must be that $u = -1$ (upper ω bits are all 1s).

So $t = 0$ if the upper $\omega + 1$ bits are all 0s or all 1s. These are exactly the cases where the multiplication does not overflow. All other cases do overflow.

This means we can write $x \cdot y = p + t2^\omega$ which overflows iff $t \neq 0$.

2. To show that p can be written in the form $p = x \cdot q + r$, where $|r| < |q|$ we consider integer division. Dividing p by nonzero x gives a quotiont q and remainder r, such that $|r| < |q|$.

3. By plugging in we get $x \cdot y = x \cdot q + r + t2^\omega$. If $r + t2^\omega = 0$ then $q = y$. Since $|r| < |q| < 2^\omega$ this can only hold if $r = t = 0$.

Problem 2.36

```
/* Determine whether arguments can be multiplied without overflow */
int tmult_ok(int x, int y) {
  int64_t prod = ((int64_t)x) * y;
  int64_t upper = prod >> 31;
  // if the upper 33 bits are all 1s or 0s the number fits into 32 bits
  return upper == 0 || upper == -1;
}
```

Problem 2.37

A. The new code does not improve the situation since the 64-bit number will be truncated to 32 bits when passed to `malloc`. This truncation is the same that also happens when the multiplication overflows.

B. Check the multiplication for overflow (by one of the previous methods) and if it overflows immediately abort and don't allocate any memory.

Problem 2.38

A single LEA instruction can compute the following multiples:

k	b	$(a \ll k) + b$
0	0	$(2^0 + 0)a = 1a$
0	a	$(2^0 + 1)a = 2a$
1	0	$(2^1 + 0)a = 2a$
1	a	$(2^1 + 1)a = 3a$
2	0	$(2^2 + 0)a = 4a$
2	a	$(2^2 + 1)a = 5a$
3	0	$(2^3 + 0)a = 8a$
3	a	$(2^3 + 1)a = 9a$

Problem 2.39

In this case the expression simplifies to $-(x \ll m)$. This is because shifting by $n + 1 = \omega$ to the left results in 0 so we can ignore the first term.

Problem 2.40

K	Shifts	Add/Subs	Expression
6	2	1	$(x \ll 2) + (x \ll 1)$
31	1	1	$(x \ll 5) - x$
-6	2	1	$(x \ll 1) - (x \ll 3)$
55	2	2	$(x \ll 6) - (x \ll 3) - x$

Problem 2.41

Consider two cases:

- $m > 0$: In this case form A requires $n - m + 1$ shifts and $n - m$ additions while form B requires 2 shifts and 1 subtraction. So if $n = m$, form A is favorable since it requires only 1 shift and 0 additions which is less than the constant numbers required for form B. If $n = m + 1$, form A takes 2 shifts and 1 addition, so either form is equally efficient in this case. If $n > m + 1$ form A takes $n - m + 1 > 2$ shifts and $n - m > 1$ additions so form B is favorable in this case.
- $m = 0$: In this case the last bit does not cause a shift so form A requires n shifts and n additions, while form B takes 1 shift and 1 subtraction. The same three cases apply in the same way here, so the above analysis extends to this case as well.

Problem 2.42

```c
int div16(int x) {
  int bias = (x >> 31) & 15;
  return (x + bias) >> 4;
}
```

Problem 2.43

x is shifted to the left by 5 which is equivalent to multiplying by $2^5 = 32$. Then, one x is subtracted from it, so we end up with $31x$ and $M = 31$.

If y is negative, a bias of $7 = 8 - 1$ is added. Then y is shifted right arithmetically by 3 which is equivalent to dividing by $2^3 = 8$. This means that $N = 8$.

Problem 2.44

A. `false` for $x = -2^{31}$ which is `INT32_MIN`. x is obviously not greater than 0, and $x - 1$ overflows to `INT32_MAX` which is not lower than 0.

B. Always `true`. The expressions are connected by `OR` so both parts would need to evaluate to 0. Let x_2 be x's third bit from the right. For the first part to evaluate to 0 we need $x_2 = 1$. For the second part, x_2 will become the sign so it needs to be 0 to represent a positive number. This is obviously not possible at the same time so at least one part always evaluates to 1.

C. `false` for $x = 2^{16} - 1$.

D. Always `true`. For all negative numbers the first part of the expression evaluates to 1. For 0 the second part evaluates to 1. Every positive number can be negated and still fits into a 32-bit integer so in this case also the second parts evaluates to 1.

E. `false` for $x = -2^{31}$ which is `INT32_MIN`. It's not greater than 0 and negating it causes an overflow (`-INT32_MIN = INT32_MIN`) which is still lower than 0.

F. Always `true`. Addition works the same on the bit level for both unsigned and two's complement numbers. Since the right side is unsigned, C will interpret both sides as unsigned numbers for the comparison.

G. Always `true`. `~y` is equal to $-y - 1$. Also, on the bit level `uy * ux` is the same as `x * y`. Plugging in we get $x(-y-1) + xy = -xy - x + xy = -x$.

Problem 2.45

Fractional value	Binary representation	Decimal representation
$\frac{1}{8}$	0.001	0.125
$\frac{3}{4}$	0.11	0.75
$\frac{25}{16}$	1.1001	1.5625
$\frac{43}{16}$	10.1011	2.6875
$\frac{9}{8}$	1.001	1.125
$\frac{47}{8}$	101.111	5.875
$\frac{51}{16}$	11.0011	3.1875

Problem 2.46

A. Binary representation of $0.1 - x$ is: 0.000000000000000000000000011001100...

B. The above is the binary representation of 0.1 with the binary point shifted to the left by 20, so it is equal to $0.1 \times 2^{-20} \approx 9.54 \times 10^{-8}$.

C. After 100 hours the clock is behind by $9.54 \times 10^{-8} \cdot 100 \cdot 60 \cdot 60 \cdot 10 \approx 0.343$ seconds.

D. $2000\text{m/s} \cdot 0.343\text{s} \approx 686\text{m}$

Problem 2.47

Bits	e	E	2^E	f	M	$2^E \times M$	V	Decimal
0 00 00	0	0	1	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	0	0.0
0 00 01	0	0	1	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	0.25
0 00 10	0	0	1	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{1}{2}$	0.5

Bits	e	E	2^E	f	M	$2^E \times M$	V	Decimal
0 00 11	0	0	1	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	0.75
0 01 00	1	0	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{4}{4}$	1	1.0
0 01 01	1	0	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	1.25
0 01 10	1	0	1	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{6}{4}$	$\frac{3}{2}$	1.5
0 01 11	1	0	1	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{7}{4}$	$\frac{7}{4}$	1.75
0 10 00	2	1	2	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$	2	2.0
0 10 01	2	1	2	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{10}{4}$	$\frac{5}{2}$	2.5
0 10 10	2	1	2	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{12}{4}$	3	3.0
0 10 11	2	1	2	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{14}{4}$	$\frac{7}{2}$	3.5
0 11 00	-	-	-	-	-	-	∞	-
0 11 01	-	-	-	-	-	-	NaN	-
0 11 10	-	-	-	-	-	-	NaN	-
0 11 11	-	-	-	-	-	-	NaN	-

Problem 2.48

$3'510'593$ in binary is 11 0101 1001 0001 0100 0001 which equals $1.1010110010001010000001 \times 2^{21}$. So the biased exponent is $21 + 2^7 - 1 = 148$.

The complete single-precision floating-point representation is:

0 10010100 10101100100010100000100 or 0x4A564504

Comparing the two we see that part of it overlaps:

```
00000000001101011001000101000001
           *********************
  01001010010101100100010100000100
```

Problem 2.49

A. $2^{n+1} + 1$

B. $2^{24} + 1 = 16'777'217$

Problem 2.50

Exact		Rounded	
Binary	Decimal	Binary	Decimal
10.010_2	$2\frac{1}{4}$	10.0_2	2
10.011_2	$2\frac{3}{8}$	10.1_2	$2\frac{1}{2}$
10.110_2	$2\frac{3}{4}$	11.0_2	3
11.001_2	$3\frac{1}{8}$	11.0_2	3

Problem 2.51

A. 0.00011001100110011001101

B.

```
  0.00011001100110011001101
- 0.00011001100110011001100110011 0011 0011...
  ----------------------------------------
  0.00000000000000000000000[1100]
```

This is equal to $\frac{1}{10} \times 2^{-22} \approx 2.38 \times 10^{-8}$.

C. After 100 hours the clock is ahead by $2.38 \times 10^{-8} \cdot 100 \cdot 60 \cdot 60 \cdot 10 \approx 0.086$ seconds.

D. $2000\text{m/s} \cdot 0.086\text{s} \approx 172\text{m}$

Problem 2.52

Format A		Format B	
Bits	Value	Bits	Value
011 0000	1	0111 000	1
101 1110	$\frac{15}{2}$	1001 111	$\frac{15}{2}$
010 1001	$\frac{25}{32}$	0110 100	$\frac{3}{4}$
110 1111	$\frac{31}{2}$	1011 000	16
000 0001	$\frac{1}{64}$	0001 000	$\frac{1}{64}$

Problem 2.53

```c
#define POS_INFIFINTY 1e400 // overflows
#define NEG_INFINITY (-POS_INFINITY)
#define NEG_ZERO (1.0/NEG_INFINITY)
```

Problem 2.54

A. Always `true`. `double` has enough range to represent all `int` values and it does not need to round. This also means no rounding when casting back.

B. `false` for $2^{24} + 1$. It is not true for all numbers with 24 or more significant bits, since `float` uses 23 bits for representing the fraction, which means these values get rounded and lose precision.

C. `false` for $2^{24} + 1$. The same reasoning as above applies here.

D. Always `true`. `double` has enough range and precision to exactly represent any `float`.

E. Always `true`. Negating the `float` only flips the sign bit and loses no information. So flipping it back results in the original value.

F. Always `true`. The integers get converted to `float` first.

G. Always `true`. If the multiplication overflows it results in $\infty > 0$.

H. `false` for $f = 10^{20}$ and $d = 1.0$. All values are promoted to `double` but even then the precision is not enough to represent $10^{20} + 1.0$. This is because $\log_2 10^{20} > 66$ and `double` has only `52` bits for representing the fraction. So when we set the least significant bit to 1 we would need about `66` bits to store the precise number. This means that adding `1.0` will have no effect on the very large number.

Problem 2.58

```
int is_little_endian() {
  // set only least significant byte to 0x01
  uint16_t x = 1;
  // char pointer reads only first byte, so it will be:
  // 0x00 on big endian systems
  // 0x01 on little endian systems
  return *(char *)&x;
}
```

Problem 2.59

```
int main() {
  int x = 0x89ABCDEF;
  int y = 0x76543210;
  int mask = 0xFF;
  printf("%d\n", ((x & mask) | (y & ~mask)) == 0x765432EF);
}
```

Problem 2.60

```
unsigned replace_byte(unsigned x, int i, unsigned char b) {
  unsigned mask = ~(0xFF << (i << 3));
  return (x & mask) | (b << (i << 3));
}

int main() {
  printf("%d\n", replace_byte(0x12345678, 2, 0xAB) == 0x12AB5678);
  printf("%d\n", replace_byte(0x12345678, 0, 0xAB) == 0x123456AB);
}
```

Problem 2.61

```
  int A = !!x;
  int B = !!~x;
  int C = !!(x & 0xFF);
  int D = !!~(x >> ((sizeof(int) - 1) << 3));
```

Problem 2.62

```
int int_shifts_are_arithmetic() {
  return (-1 >> 1) == -1;
}
```

Problem 2.63

```
unsigned srl(unsigned x, int k) {
  /* Perform shift arithmetically */
  unsigned xsra = (int)x >> k;

  int w = sizeof(int) << 3;
  int mask = ~(-1 << (w - k));
  return xsra & mask;
}

int sra(int x, int k) {
  /* Perform shift logically */
  int xsrl = (unsigned) x >> k;

  int w = sizeof(int) << 3;
  int mask = -1 << (w - k);
  int sign = -(x & (1 << (w - 1)));
  return xsrl | (mask & sign);
}
```

Problem 2.64

```
int any_odd_one(unsigned x) {
  int mask = 0b10101010101010101010101010101010;
  return !!(x & mask);
}
```

Problem 2.65

```c
int odd_ones(unsigned x) {
  x ^= x >> 16;
  x ^= x >> 8;
  x ^= x >> 4;
  x ^= x >> 2;
  x ^= x >> 1;
  return x & 1;
}
```

Problem 2.66

```c
int leftmost_one(unsigned x) {
  // Spread leftmost 1 to the right
  x |= x >> 1;
  x |= x >> 2;
  x |= x >> 4;
  x |= x >> 8;
  x |= x >> 16;
  return x ^ (x >> 1);
}
```

Problem 2.67

Shifting by an amount $\geq$ the width of the type is undefined for signed integers.

```c
int int_size_is(unsigned w) {
  unsigned set_msb = 1U << (w - 1);
  unsigned beyond_msb = set_msb << 1;
  return set_msb && !beyond_msb;
}
```

Problem 2.68

```c
int lower_one_mask(int n) {
  int w = sizeof(int) << 3;
  return ~0U >> (w - n);
}
```

Problem 2.69

```
unsigned rotate_left(unsigned x, int n) {
  unsigned w = sizeof(unsigned) << 3;
  unsigned rotated_bits = x >> (w - n - 1) >> 1;
  return (x << n) | rotated_bits;
}
```

Problem 2.70

```
int fits_bits(int x, int n) {
  int w = sizeof(int) << 3;
  int shifted = x << (w - n) >> (w - n);
  return shifted == x;
}
```

Problem 2.71

A. The upper three bytes are always all zeroes, which means we can't represent negative numbers.

B. Correct implementation:

```
typedef unsigned packed_t;

int xbyte(packed_t word, int bytenum) {
  // cut off unwanted bytes to the left
  int shifted = word << ((3 - bytenum) << 3);
  // cut off unwanted bytes to the right
  // number will be sign extended because type is int
  return shifted >> 24;
}
```

Problem 2.72

A. Because `sizeof` returns a `size_t` (which is unsigned), the result of `maxbytes - sizeof(val)` is also unsigned and thus always ≥ 0.

B. Correct implementation:

```
void copy_int(int val, void *buf, int maxbytes) {
  if (maxbytes >= sizeof(val))
    memcpy(buf, (void *)&val, sizeof(val));
}
```

Problem 2.73

```c
int saturating_add(int x, int y) {
  int sum = x + y;
  int offset = (sizeof(int) << 3) - 1;

  // create masks with all 0s or all 1s
  int x_neg = x >> offset;
  int y_neg = y >> offset;
  int sum_neg = sum >> offset;

  int pos_overflow = ~x_neg & ~y_neg & sum_neg;
  int neg_overflow = x_neg & y_neg & ~sum_neg;

  // use masks to choose between sum and bounds
  return (INT32_MAX & pos_overflow) | (INT32_MIN & neg_overflow) |
         (sum & ~(pos_overflow | neg_overflow));
}
```

Problem 2.74

```c
int tsub_ok(int x, int y) {
  int offset = (sizeof(int) << 3) - 1;
  int x_neg = x >> offset;
  int y_neg = y >> offset;
  int diff_neg = (x - y) >> offset;

  int pos_overflow = ~x_neg & y_neg & diff_neg;
  int neg_overflow = x_neg & ~y_neg & ~diff_neg;
  return !(pos_overflow | neg_overflow);
}
```

Problem 2.75

```c
unsigned unsigned_high_prod(unsigned x, unsigned y) {
  unsigned prod = signed_high_prod(x, y);
  unsigned offset = (sizeof(unsigned) << 3) - 1;
  unsigned x_sign = x >> offset;
  unsigned y_sign = y >> offset;
  return prod + x_sign * y + y_sign * x;
}
```

Problem 2.76

```
void *calloc(size_t nmemb, size_t size) {
  if (nmemb == 0 || size == 0) {
    return NULL;
  }

  size_t total_size = nmemb * size;

  if (total_size / size != nmemb) {
    // Overflow occurred
    return NULL;
  }

  void *ptr = malloc(total_size);

  if (ptr == NULL) {
    // Allocation failed
    return NULL;
  }

  memset(ptr, 0, total_size);
  return ptr;
}
```

Problem 2.77

```
int times_17 = (x << 4) + x;
int times_neg7 = x - (x << 3);
int times_60 = (x << 6) - (x << 2);
int times_neg112 = (x << 4) - (x << 7);
```

Problem 2.78

```
int divide_power(int x, int k) {
  int w = sizeof(int) << 3;
  int mask = x >> (w - 1);
  int bias = mask & ((1 << k) - 1);
  return (x + bias) >> k;
}
```

Problem 2.79

```c
int mul3div4(int x) {
  int mul3 = (x << 1) + x;
  int w = sizeof(int) << 3;
  int mask = mul3 >> (w - 1);
  int bias = mask & 3;
  return (mul3 + bias) >> 2;
}
```

Problem 2.80

```c
int threefourths(int x) {
  // calculate result for higher bits that are not relevant for rounding
  int high_div4 = x >> 2;
  int high_result = (high_div4 << 1) + high_div4;

  // for the remainder we can safely multiply first since it cannot overflow
  int rem = x - (high_div4 << 2);
  int rem_mul3 = (rem << 1) + rem;
  int mask = x >> ((sizeof(int) << 3) - 1);
  int bias = mask & 3;
  int low_result = (rem_mul3 + bias) >> 2;

  return high_result + low_result;
}
```

Problem 2.81

```c
int main() {
  // example values
  int k = 9;
  int j = 5;

  int A = -1 << k;
  int B = ((1 << k) - 1) << j;
}
```

Problem 2.82

A. `false` for $x = -2^{31}$ which is TMin_{32} because $-\text{TMin} = \text{TMin}$ (since it overflows).

B. Always `true`. If any overflows happen they will be the same on both sides. (ring properties of two's complement arithmetic)

C. Always `true`. `~x` is equal to `-x - 1`. So we can write it as

`-x - 1 - y - 1 + 1 = -(x+y) - 1` which simplifies to `-x - y - 1 = -x - y - 1`.

D. Always `true`. By negating we turn $y - x$ into $x - y$. Regardless of the negation, both sides are treated as unsigned for the comparison. And unsigned and two's complement arithmetic are the same on the bit level.

E. Always `true`. This basically sets the 2 lower bits to 0. These bits have a positive weight regardless of the sign bit, so setting them to 0 can not make the number larger.

Problem 2.83

A. Let V be the value of the string. Shifting the binary point k to the right results in y.yyyy... which equals both $V + Y$ and $V \times 2^k$. These can now be equated:

$$\begin{aligned} V + Y &= V \times 2^k \\ Y &= V \times 2^k - V \\ Y &= V(2^k - 1) \\ V &= \frac{Y}{2^k - 1} \end{aligned}$$

B.

(a) $V = \frac{5}{2^3-1} = \frac{5}{7}$

(b) $V = \frac{6}{2^4-1} = \frac{2}{5}$

(c) $V = \frac{19}{2^6-1} = \frac{19}{63}$

Problem 2.84

```
return ((ux << 1) == 0 && (uy << 1) == 0) // +0 and -0 are considered equal
        || (!sx && !sy && ux <= uy)        // both positive
        || (sx && sy && ux >= uy)          // both negative
        || (sx > sy);                      // different signs
```

Problem 2.85

A. The number 7.0 will have exponent $E = 2$, significand $M = 1.11_2 = \frac{7}{4}$, fraction $f = 0.11_2 = \frac{3}{4}$ and value $V = 7$. The exponent will be represented as `10...01` (bias is $2^{k-1} - 1 = 01...1_2$) and the fraction as `110...0`.

B. The largest odd integer that can be represented exactly will have exponent $E = n$, significand $M = 1.1...1 = 2 - \frac{1}{2^n}$, fraction $f = 0.1...1_2 = 1 - \frac{1}{2^n}$ and value $V = 2^{n+1} - 1$. The exponent will be the binary representation of $n + 2^{k-1} - 1$ and the fraction will be represented as 1...1.

C. The smallest positive normalized value has exponent $2-2^{k-1}$ and fraction 0, giving a value of $1.0 \times 2^{2-2^{k-1}}$. The reciprocal of this is $V = 2^{2^{k-1}-2}$. It will have exponent $E = 2^{k-1}-2$, significand $M = 1$ and fraction $f = 0$. The biased exponent will be $2^{k-1}-2-2^{k-1}-1 = -3$ represented by `1...101` and the fraction by `0...0`.

Problem 2.86

	Extended precision	
Description	Value	Decimal
Smallest positive denormalized	$2^{-63} \times 2^{2-2^{14}}$	2.645×10^{-4951}
Smallest positive normalized	$2^{2-2^{14}}$	3.362×10^{-4932}
Largest normalized	$(2-2^{-63}) \times 2^{2^{14}-1}$	1.190×10^{4932}

Problem 2.87

Description	Hex	M	E	V	D
-0	`0x8000`	0	-14	-0	-0.0
Smallest value > 2	`0x4001`	$\frac{1025}{1024}$	1	1025×2^{-9}	2.001953125
512	`0x6000`	1	9	512	512.0
Largest denormalized	`0x03FF`	$\frac{1023}{1024}$	-14	1023×2^{-24}	0.0000609756
$-\infty$	`0xFC00`	-	-	$-\infty$	$-\infty$
Number with hex representation `3BB0`	`0x3BB0`	$\frac{123}{64}$	-1	123×2^{-7}	0.9609375

Problem 2.88

Format A		Format B	
Bits	Value	Bits	Value
`1 01111 001`	$\frac{-9}{8}$	`1 0111 0010`	$\frac{-9}{8}$
`0 10110 011`	176	`0 1110 0110`	176
`1 00111 010`	$\frac{-5}{1024}$	`1 0000 0101`	$\frac{-5}{1024}$
`0 00000 111`	$\frac{7}{2^{17}}$	`0 0000 0001`	$\frac{1}{1024}$
`1 11100 000`	-2^{13}	`1 1110 1111`	-248
`0 10111 100`	384	`0 1111 0000`	∞

Problem 2.89

A. Always `true`. `float` cannot represent every 32-bit integer value in full precision, but `double` can, so they will be rounded in the same way.

B. `false` for $x = 0$ and $y = \mathrm{TMin}_{32}$. The integer result will overflow before it is cast to `double` while the operation with `double`s has a wider range and does not overflow.

C. Always `true`. The result of adding two floats in the 32-bit integer range cannot overflow.

D. (Apparently not always true, cannot reproduce solution tho)

E. `false` for $x = 1$ and $z = 0$. Dividing by 0 results in `NaN`.

Problem 2.90

```
float fpwr2(int x) {
  /* Result exponent and fraction */
  unsigned exp, frac;
  unsigned u;

  if (exp < -126-23) {
    /* Too small. Return 0.0 */
    exp = 0;
    frac = 0;
  } else if (exp < -126) {
    /* Demornalized result */
    exp = 0;
    frac = 1 << (exp + 126 + 23);
  } else if (exp <= 127) {
    /* Normalized result */
    exp = exp + 127;
    frac = 0;
  } else {
    /* Too big. Return +inf */
    exp = 0xFF;
    frac = 0;
  }

  /* Pack exp and frac into 32 bits */
  u = exp << 23 | frac;
  /* Return as float */
  return u2f(u);
}
```

Problem 2.91

A. Binary representation: `0 10000000 10010010000111111011011`. So the exponent is $128 - 127 = 1$ and the fractional number is $11.0010010000111111011011_2$.

B. $\frac{22}{7} = 3\frac{1}{7} = 11.001001001001001\ldots_2$

C. The two approximations diverge starting from the 9th bit after the binary point.