Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 2: Representing and Manipulating Information
March 11, 2024

Practice Problems

Exercise 2.1. Perform the following conversions:

(a) `0x39A7F8` to binary.

(b) binary 1100100101111011 to hexadecimal.

(c) `0xD5E4C` to binary.

(d) binary 1001101110011110110101 to hexadecimal.

Solution:

(a) Each hexadecimal digit corresponds to a 4-bit binary number:

Hexadecimal	3	9	A	7	F	8
Binary	0011	1001	1010	0111	1111	1000

When concatenated, we find that `0x39A7F8` $= 001110011010011111111000_2$.

(b) We group the number into 4-bit groups:

Binary	1100	1001	0111	1011
Hexadecimal	C	9	7	B

Hence, $1100100101111011_2 =$ `0xC97B`.

(c) We tabulate the values:

Hexadecimal	D	5	E	4	C
Binary	1101	0101	1110	0100	1100

Hence, `0xD5E4C` $= 11010101111001001100_2$.

(d) The table is below:

10 0110 1110 0111 1011 0101 Binary	10	0110	1110	0111	1011	0101
Hexadecimal	2	6	E	7	B	5

Hence, $1001101110011110110101_2 =$ `0x26E7B5`.

Exercise 2.2. Fill in the blank entries in the following table, giving the decimal and hexadecimal representations of different powers of 2:

n	2^n (decimal)	2^n (hexadecimal)
9	512	`0x200`
19	_____	_____
_____	16,384	_____
_____	_____	`0x1000`
17	_____	_____
_____	32	_____
_____	_____	`0x80`

Solution: As per the text, we write $n = i + 4j$, where $0 \leq i \leq 3$. The i determines the leading hex bit to be 2^i (that is, 1, 2, 4, or 8). The j determines the number of hexadecimal 0s thereafter. Some of the n values from the table are below:

$$\begin{aligned} 19 &= 3 + 4 \cdot 4 \\ 14 &= 2 + 4 \cdot 3 \\ 12 &= 0 + 4 \cdot 3 \\ 17 &= 1 + 4 \cdot 4 \\ 5 &= 1 + 4 \cdot 1 \\ 7 &= 3 + 4 \cdot 1. \end{aligned}$$

The filled-in table follows:

n	2^n (decimal)	2^n (hexadecimal)
9	512	`0x200`
19	**524,288**	`0x80000`
14	16,384	`0x4000`
12	4096	`0x1000`
17	131,072	`0x20000`
5	32	`0x20`
7	128	`0x80`

Exercise 2.3. A single byte can be represented by 2 hexadecimal digits. Fill in the missing entries in the following table, giving the decimal, binary, and hexadecimal values of different byte patterns.

Decimal	Binary	Hexadecimal
0	0000 000	`0x00`
167	_____	_____
62	_____	_____
188	_____	_____
_____	0011 0111	_____
_____	1000 1000	_____
_____	1111 0011	_____
_____	_____	`0x52`
_____	_____	`0xAC`
_____	_____	`0xE7`

Solution: We proceed by repeatedly performing the division algorithm, taking each remainder:

$$\begin{aligned} 167 &= 16 \cdot 10 + 7 \\ 10 &= 16 \cdot 0 + 10 \end{aligned}$$

Since $10_{16} =$ `0xA`, we have $167_{10} =$ `0xA7`. We proceed the same way:

$$\begin{aligned} 62 &= 16 \cdot 3 + 14 \\ 3 &= 16 \cdot 0 + 3 \end{aligned}$$

So $62_{10} =$ `0x3E`.

$$\begin{aligned} 188 &= 16 \cdot 11 + 12 \\ 12 &= 16 \cdot 0 + 12 \end{aligned}$$

So $188_{16} =$ `0xCC`. By representing each hexadecimal digit with 4 bits and concatenating them, we get the binary representation. To convert from hexadecimal to decimal, we multiply by the appropriate power of 16:

$$\begin{aligned} \texttt{0x37} &= 3 \cdot 16^1 + 7 \cdot 16^0 = 55_{10} \\ \texttt{0x88} &= 7 \cdot 16^1 + 8 \cdot 16^0 = 136_{10} \\ \texttt{0xF3} &= 15 \cdot 16^1 + 3 \cdot 16^0 = 243_{10} \\ \texttt{0x52} &= 5 \cdot 16^1 + 2 \cdot 16^0 = 82_{10} \\ \texttt{0xAC} &= 10 \cdot 16^1 + 12 \cdot 16^0 = 172_{10} \\ \texttt{0xE7} &= 14 \cdot 16^1 + 7 \cdot 16^0 = 231_{10} \end{aligned}$$

The complete table is below:

Decimal	Binary	Hexadecimal
0	0000 000	`0x00`
167	1010 0111	`0xA7`
62	0011 1110	`0x3E`
188	1100 1100	`0xCC`
55	0011 0111	`0x37`
136	1000 1000	`0x88`
243	1111 0011	`0xF3`
82	0101 0010	`0x52`
172	1010 1100	`0xAC`
231	1110 0111	`0xE7`

Exercise 2.4. Without converting the numbers to decimal or binary, try to solve the following arithmetic problems, giving the answers in hexadecimal. *Hint*: Just modify the methods you use for performing decimal addition and subtraction to use base 16.

(a) `0x503c` + `0x8` = _____

(b) `0x503c - 0x40` = ______

(c) `0x503c` + 64 = ______

(d) `0x50ea - 0x503c` = ______

Solution:

(a)

$$\begin{array}{ccccc} & & & 1 & \\ 0x & 5 & 0 & 3 & c \\ 0x & & & 0 & 8 \\ + & & & & \\ \hline 0x & 5 & 0 & 4 & 4 \end{array}$$

(b)

$$\begin{array}{ccccc} & 4 & F & F3 & \\ 0x & \cancel{5} & \cancel{0} & \cancel{3} & c \\ 0x & & & 4 & 0 \\ - & & & & \\ \hline 0x & 4 & F & F & C \end{array}$$

(c) Note that 64_{10} = `0x40`:

$$\begin{array}{ccccc} 0x & 5 & 0 & 3 & c \\ 0x & & & 4 & 0 \\ + & & & & \\ \hline 0x & 5 & 0 & 7 & c \end{array}$$

(d)

$$\begin{array}{ccccc} & & & D & FA \\ 0x & 5 & 0 & \cancel{e} & \cancel{a} \\ 0x & 5 & 0 & 3 & c \\ - & & & & \\ \hline 0x & 0 & 0 & A & E \end{array}$$

Exercise 2.5. Consider the following three calls to `show_bytes`:

```c
int val = 0x87654321;
byte_pointer valp = (byte_pointer) &val;
show_bytes(valp, 1); /* A. */
show_bytes(valp, 2); /* B. */
show_bytes(valp, 3); /* C. */
```

Indicate the values that will be printed by each call on a little-endian machine and on a big-endian machine.

Solution: Recall `show_bytes` function accepts an `unsigned char*` and the size of the data type, which it uses to know how many bytes to read. For example, we might pass `sizeof(int32_t)`, which would pass 4 because an `int32_t` takes up 32 bits, or 4 bytes. With that out of the way:

(a) The call with 1 means to take 1 byte, which on a little-endian will be the least significant byte and on big-endian machine will be the most significant byte. A byte is 8 bits, or two hexadecimal numbers. Hence, the most least significant byte is `21`, and the most significant is `87`.

(b) On little-endian, it would be `21 43`, and on big-endian, it would be `87 65`.

(c) On a little-endian it's `21 43 65`, and on big-endian it's `87 65 43`.

Exercise 2.6. Using `show_int` and `show_float`, we determine that the integer 3510593 has hexadecimal representation `0x00359141`, while the floating-point number 3510593.0 has hexadecimal representation `0x4A564504`.

(a) Write the binary representations of these two hexadecimal values.

(b) Shift these two strings relative to one another to maximize the number of matching bits. How many bits match?

(c) What parts of the strings do not match?

Solution:

1.

Hexadecimal	Binary
`0x00359141`	0000 0000 0011 0011 0101 0001 0100 0001
`0x4A564504`	0100 1010 0101 0110 0100 0101 0000 0100

2. The shifted numbers are shown below, with the 21 matching bits shown in bold:

00000000001**101011001000101000001**
010010100**101011001000101000001**00

3. The last two bits in the float do not match. Also, the leading bits in integer, 0000000001, do not match the leading bits in the float: 010010100.

Exercise 2.7. What would be printed as a result of the following call to `show_bytes`?

```c
const char *s = "abcdef";
show_bytes((byte_pointer) s, strlen(s));
```

Note that letters '`a`' through '`z`' have ASCII codes `0x61` through `0x7A`.

Solution: The output for the lowercase characters would be: `0x61 0x62 0x63 0x64 0x65 0x66 0x00` on any system using ASCII as its character code. The `0x00` is the null character used to terminate strings in C.

Exercise 2.8. Fill in the following table showing the results of evaluating Boolean operations on bit vectors.

Operationg	Result
a	[01101001]
b	[01010101]
$\sim a$	____
$\sim b$	____
$a \,\&\, b$	____
$a \mid b$	____
a^b	____

Solution: Treating the bit sequences as bit vectors and noting that $\sim$ is logical NOT, & is logical AND, | is logical OR, and ^ is logical XOR, we get:

Operationg	Result
a	[01101001]
b	[01010101]
$\sim a$	[10010110]
$\sim b$	[10101010]
$a \,\&\, b$	[01000001]
$a \mid b$	[01111101]
a^b	[00111100]

Exercise 2.9. Computers generate color pictures on a video screen or liquid crystal display by mixing three different colors of light: red, green, and blue. Imagine a simple scheme, with three different lights, each of which can be turned on or off, projecting onto a glass screen. We can then create eight different colors based on the absence (0) or presence (1) of light sources R, G, and B:

R	G	B	Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

Each of these colors can be represented as a bit vector of length 3, and we can apply Boolean operations to them.

(a) The complement of a color is formed by turning off the lights that are on and turning on the lights that are off. What would be the complement of each of the eight colors listed above?

(b) Describe the effect of applying Boolean operations on the following colors:

$$\begin{aligned} \text{Blue} \mid \text{Green} &= \underline{\qquad} \\ \text{Yellow} \,\&\, \text{Cyan} &= \underline{\qquad} \\ \text{Red} \wedge \text{Magenta} &= \underline{\qquad} \end{aligned}$$

Solution: (a) The augmented table below shows the complementary colors, obtained by applying the logical NOT operation $\sim$ to each bit vector:

R	G	B	Color	Complement
0	0	0	Black	White
0	0	1	Blue	Yellow
0	1	0	Green	Magenta
0	1	1	Cyan	Red
1	0	0	Red	Cyan
1	0	1	Magenta	Green
1	1	0	Yellow	Blue
1	1	1	White	Black

(b) Blue | Green means we apply the logical OR operation to their corresponding bit vectors. We get 001 — 010 = 011, which is Cyan. Yellow & Cyan means we apply the logical AND operator to the bit vectors, so 110 & 011 = 010, which is Green. Finally, Red ^ Magenta means apply logical XOR to the bit vectors, so 100 ^ 101 = 001, which is Blue, so

$$\begin{aligned} \text{Blue} \mid \text{Green} &= \text{Cyan} \\ \text{Yellow} \,\&\, \text{Cyan} &= \text{Green} \\ \text{Red} \wedge \text{Magenta} &= \text{Blue} \end{aligned}$$

Exercise 10. As an application of the property that $a \wedge a = 0$ for any bit vector a, consider the following program:

```
void inplace_swap(int *x, int *y) {
    *y = *x ^ *y;   /* Step 1 */
    *x = *x ^ *y;   /* Step 2 */
    *y = *x ^ *y;   /* Step 3 */
}
```

As the name suggests, we claim that the effect of this procedure is to swap the values stored at the locations denoted by pointer variables `x` and `y`. Note that unlike the usual technique for swapping two values, we do not need a third location to temporarily store one value while we are moving the other. There is no performance advantage to this way of swapping; it is merely an intellectual amusement.

Staring with values a and b in the locations pointed to by `x` and `y`, respectively, fill int he table that follows, giving the values stored at the two locations after each step of the procedure. Use the properties of ^ to show that the desired effect is achieved. Recall that every element is its own additive inverse (that is, a^$a = 0$).

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

Solution: The completed table is below:

Step	`*x`	`*y`
Initially	a	b
Step 1	a	a^b
Step 2	a^$(a$^$b)$	a^b
Step 3	a^$(a$^$b)$	$[a$^$(a$^$b)]$^$[a$^$b]$

Since a^$(a$^$b) = (a$^$a) \wedge b = 0$^$b = b$, the table evaluates correctly.

Exercise 2.11. Armed with the function `inplace_swap` from Problem 2.10, you decide to write code that will reverse the elements of an array by swapping from opposite ends of the array, working toward the middle:

```
void reverse_array(int a[], int cnt) {
    int first, last;
    for (first = 0, last = cnt-1;
         first <= last;
         first++, last--)
         inplace_swap(&a[first], &a[last]);
}
```

When you apply your function to an array containing elements 1, 2, 3, and 4, you find that the array now has, as expected, elements 4, 3, 2, and 1. When you try it on an array with elements 1, 2, 3, 4, and 5, however, you are surprised to see that the array now has elements 5, 4, 0, 2, and 1. In fact, you discover that the code always works on arrays of even length, but it sets the middle element to 0 whenever the array has odd length.

(a) For an array of odd length `cnt` $= 2k + 1$, what are the values of variables `first` and `last` in the final iteration of function `reverse_array`?

(b) Why does this call to function `inplace_swap` set the array element to 0?

(c) What simple modification to the code for `reverse_array` would eliminate this problem?

Solution:

(a) Their values are the same, and their value is the one at the center of the array, namely, `a[cnt / 2]`.

(b) The XOR operation operates on the same number, and since every element is its own additive inverse with respect to this operation, the result is 0.

(c) Replace the comparison `first <= last` with `first < last`.

Exercise 2.12. Write C expressions, in terms of variable `x`, for the following values. Your code should work for any size $w \geq 8$. For reference, we show the result of evaluating the the expressions for `x = 0x87654321`, with $w = 32$.

(a) The least significant byte of `x`, with all other bits set to 0. [`0x00000021`].

(b) All but the least significant byte of `x` complemented, with the least significant byte left unchanged. [`0x789ABC21`]

(c) The least significant byte set to all ones, and all other bytes of `x` left unchanged. [`0x876543FF`]

Solution:

(a) `x & 0xFF`

(b) The expression is: `(x & 0xFF) | (~x & ~0xFF)`. First, we use the `0xFF` mask to get the first byte of x. Then, we complement `x`, but mask with `~0xFF` instead to get all bits except the last byte.

(c) The expression is: `x |0xFF`. By using the logical OR, we ensure the least significant byte is set to all 1s. The upper bytes are 0, so they do not change. what's in `x`.

Exercise 2.13. The Digital Equipment VAX computer was a very popular machine from the late 1970s until the late 1980s. Rather than instructions for Boolean operations AND and OR, it had instructions `bis` (bit set) and `bic` (bit clear). Both instructions take a data word `x` and a mask word `m`. They generate a result `z` consisting of the bits of `x` modified according to the bits of `m`. With `bis`, the modification involves setting `z` to 1 at each position where `m` is 1. With `bic`, the modification involves setting `z` to 0 at each bit position where `m` is 1.

To see how these operations relate to C bit-level operations, assume we have functions `bis` and `bis` implementing the bit set and bit clear operations, and that we want to use these to implement functions computing bitwise operations `|` and `^`. Fill in the missing code below. Write C expressions for the operations `bis` and `bic`.

```
/* Declarations of functions implementing operatings bis and bic */
int bis(int x, int m);
int bic(int x, int m);

/* Compute x|y using only calls to fucntions bis and bic */
int bool_or(int x, int y) {
```

```
    int result = __________;
    return result;
}

/* Compute x^y using only calls to functions bis and bic */
int bool_xor(int x, int y) {
    int result = ___________;
    return result;
}
```

Solution: For the OR operation, suppose we start with x. The expression `bis(x, 0)` is equivalent to `x`. This is because 0 does not have any 1 bits. If `x` has 1 bits, they remain 1; if they're 0, they remain 0. On the other hand, if `y` had all 1 bits, then `bit(x, y)` will be all 1 bits, regardless of what was in `x`. This suggests the correct way to implement `x` OR `y` is with `bis(x,y)`:

1. If the i-th bit of `x` is 1, then the result is 1 regardless of the value of the i-th bit of `y`.
2. If the i-th bit of `x` is 0, then the result is only 1 if the i-th bit of `y` is 1.

Consider a truth table for the `bic(x, y)` operation:

x	y	bic(x, y)
0	0	0
0	1	0
1	0	1
1	1	0

Note that if `x` is 0, then the result is 0. We could flip the inputs and get:

y	x	bic(y, x)
0	0	0
0	1	0
1	0	1
1	1	0

Therefore, the i-th bit of `bic(x, y)` will be 1 only if the i-th bit of `x` is 1 and the i-th bit of `y` is 0. The opposite is true for `bic(y, x)`. We therefore get `x ^ y` by applying the OR operation (which is just `bis`):

```
/* Declarations of functions implementing operatings bis and bic */
int bis(int x, int m);
int bic(int x, int m);

/* Compute x|y using only calls to fucntions bis and bic */
int bool_or(int x, int y) {
    int result = bis(x, y);
    return result;
```

```
}

/* Compute x^y using only calls to functions bis and bic */
int bool_xor(int x, int y) {
    int result = bis(bic(x, y), bic(y, x));
    return result;
}
```

Exercise 2.14. Suppose that `x` and `y` have byte values `0x66` and `0x39`, respectively. Fill in the following table indicating the byte values of the different C expressions:

Expression	Value	Expression	Value
`x & y`	____	`x && y`	____
`x | y`	____	`x || y`	____
`~ x | ~y`	____	`!x || !y`	____
`x & !y`	____	`x && ~y`	____

Solution: Note that `x = 0x66 = 0110 0110` and `y = 0x39 = 0011 1001`. Also `~x = 1001 1001` and `~y = 1100 0110`.

Expression	Value	Expression	Value
`x & y`	`0010 0000`	`x && y`	`0x01`
`x | y`	`0111 1111`	`x || y`	`0x01`
`~x | ~y`	`1101 1111`	`!x || !y`	`0x00`
`x & !y`	`0100 0110`	`x && ~y`	`0x01`

Exercise 2.15. Using only bit-level and logical operations, write a C expression that is equivalent to `x == y`. That is, it will return 1 when `x` and `y` are equal and 0 otherwise.

Solution: We can use `!(x & ~y)`. Suppose `x` and `y` are the same; then `x & ~y` is 0, so the logical NOT operator `!` makes the result 1. Now suppose they're different, say, they're i-th bit is different. Then the i-th bit of `x` & `~y` is the same, so `x & ~y` is nonzero, which means that applying logical NOT gives a value of 0.

Exercise 2.16. Fill in the table below showing the effects of the different shift operations on single-byte quantities. The best way to think about shift operations is to work with binary representations. Convert the initial values to binary, perform the shifts, and then convert back to hexadecimal. Each of the answers should be 8 binary digits or 2 hexadecimal digits.

`x`	`0xC3`	`0x75`	`0x87`	`0x66`
`x << 3`				
`x >> 2` (Logical)				
`x >> 2` (Arithmetic)				

Solution: We begin by converting each hexadecimal to binary:

$$\texttt{0xC3} = \texttt{1100 0011}$$
$$\texttt{0x75} = \texttt{0111 0101}$$
$$\texttt{0x87} = \texttt{1000 0111}$$
$$\texttt{0x66} = \texttt{0110 0110}$$

From here, shifts are easy:

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

Exercise 2.17. Assuming $w = 4$, we can assign a numeric value to each possible hexadecimal digit, assuming either an unsigned or a two's-complement interpretation. Fill in the following table according to these interpretations by writing out the nonzero powers of 2 in the summations shown in Equations 1 and 2:

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i \tag{1}$$

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \tag{2}$$

where w is a positive integer, $x_i \in \{0, 1\}$, and $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$.

Hexadecimal	Binary	$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
`0xE`	`[1110]`	$2^3 + 2^2 + 2^1 = 14$	$-2^3 + 2^2 + 2^1 = -2$
`0x0`	______	______	______
`0x5`	______	______	______
`0x8`	______	______	______
`0xD`	______	______	______
`0xF`	______	______	______

Solution:

Hexadecimal	Binary	$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
`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]`	$2^3 + 2^2 + 2^1 + 2^0 = 15$	$-2^{-3} + 2^2 + 2^1 + 2^0 = -1$

Exercise 2.18. In Chapter 3, we will look at listings generated by a *diassembler*, a program that converts an executable program file back to a more readable ASCII form. These files contain many hexadecimal numbers, typically representing values in two's-complement form. Being able to recognize these numbers and understand their significance (for example, whether they are negative or positive) is an important skill.

For lines labeled A-I (on the right) in the following listing, convert the hexadecimal values (in 32-bit two's complement form) shown to the right of the instruction names (`sub`, `mov`, and `add`) into their decimal equivalents:

```
4004d0:  48 81 ec e0 02 00 00  sub   $0x2e0,%rsp              A.
4004d7:  48 8b 44 24 a8        mov   -0x58(%rsp),%rax         B.
4004dc:  48 03 47 28           add   0x28(%rdi),%rax          C.
4004e0:  48 89 44 24 d0        mov   %rax,-0x30(%rsp)         D.
4004e5:  48 8b 44 24 78        mov   %0x78(%rsp),%rax         E.
4004ea:  48 89 87 88 00 00 00  mov   %rax,0x88(%rdi)          F.
4004f1:  48 8b 84 24 f8 01 00  mov   0x1f8(%rsp),%rax         G.
4004f8:  00
4004f9:  48 03 44 24 08        add   0x8(%rsp),%rax
4004fe:  48 89 84 24 c0 00 00  mov   %rax,%0xc0(%rsp)         H.
400505:  00
400506:  48 8b 44 d4 b8        mov   -0x48(%rsp,%rdx,8),%rax  I.
```

Solution:

(A) We are given $\bar{x}$ = `0x2e0` = `0010 1110 0000`. Since the number is given in 32-bit two's complement form, the other 20 bits are 0. By using the $B2T_{32}$ function, we get

$$B2T_{32}(\bar{x}) = 2^9 + 2^7 + 2^6 + 2^5 = 736$$

(B) Letting $\bar{x}$ = `0x58` = `0101 1000`, we interpret the `-` as a negative sign and get

$$-B2T_{32}(\bar{x}) = -\left(2^6 + 2^4 + 2^3 = 2,147,483,560\right) = -88$$

(C) Since $\bar{x}$ = `0x28` = `0010 1000`, we get

$$B2T_{32}(\bar{x}) = 2^5 + 2^3 = 40$$

(D) Letting $\bar{x}$ = `0x30` = `0011 0000`, we get

$$-B2T_{32}(\bar{x}) = -\left(2^5 + 2^4\right) = -48$$

(E) Letting $\bar{x}$ = `0x78` = `0111 1000`, we get

$$B2T_{32}(\bar{x}) = 2^6 + 2^5 + 2^4 + 2^3 = 120$$

(F) Letting $\bar{x}$ = `0x88` = `1000 1000`, we get

$$B2T_{32}(\bar{x}) = 2^7 + 2^3 = 136$$

(G) Letting $\bar{x}$ `= 0x1f8 = 0001 1111 1000`, we get

$$B2T_{32}(\bar{x}) = 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^3 = 504$$

(H) Letting $\bar{x}$ `= 0xc0 = 1100 0000`, we get

$$B2T_{32}(\bar{x}) = 2^7 + 2^6 = 192$$

(I) Letting $\bar{x}$ `= 0x48 = 0100 1000`, we get

$$-B2T_{32}(\bar{x}) = -\left(2^6 + 2^3\right) = 72$$

Exercise 2.19. Using the table you filled when solving Problem 2.17, fill in the following table describing the function $T2U_4$:

x	$T2U_4(x)$
-8	_____
-3	_____
-2	_____
-1	_____
0	_____
5	_____

Solution:

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

Exercise 2.20. Explain how Equation 3 applies to the entries in the table you generated when solving Problem 2.19.

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{3}$$

Solution: In Problem 2.19, we had $w = 4$. If the value on the left is non-negative, it remains unchanged. Otherwise, we add $2^4 = 16$ to the corresponding value on the left column.

Exercise 2.21. Assuming the expressions are evaluated when executing a 32-bit program on a machine that uses two's-complement arithmetic, fill in the following table describing the effect of casting and relational operations, in the style of Figure 2.19 (of text).

Expression	Type	Evaluating
`-2147483647-1 == 2147483648U`	_____	_____
`-2147483647-1 == 2147483647`	_____	_____
`-2147483647-1U == 2147483647`	_____	_____
`-2147483647-1 == -2147483647`	_____	_____
`-2147483647-1U == -2147483647`	_____	_____

Solution:

Note that in the operation `-2147483647-1U`, the operand `1U` is unsigned, so C implicitly casts `-2147483647` to the unsigned number $2147483647 + 2^{32} = 2147483649$.

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

Exercise 2.22. Show that each of the following bit vectors is a two's-complement representation of -5 by applying Equation 2:

(a) [1011]

(b) [11011]

(c) [111011]

Observe that the second and third bit vectors can be derived from the first by sign extension.

Solution:

(a) $w = 4$, so $B2T_4([1011]) = -2^3 + 2^1 + 2^0 = -8 + 2 + 1 = -5$.

(b) $w = 5$, so $B2T_5([11011]) = -2^4 + 2^3 + 2^1 + 2^0 = -16 + 8 + 2 + 1 = -5$.

(c) $w = 6$, so $B2T_6([111011]) = -2^5 + 2^4 + 2^3 + 2^1 + 2^0 = -32 + 16 + 8 + 2 + 1 = -5$.

Exercise 2. 2.23 Consider the following C functions:

```c
int fun1(unsigned word) {
   return (int) ((word << 24) >> 24);
}

int fun2(unsigne dword) {
   return ((int) word << 24) >> 24;
}
```

Assume these are executed as a 32-bit program on a machine that uses two's-complement arithmetic. Assume also that right shifts og signed values are performed arithmetically, while right shifts of unsigned values are performed logically.

(a) Fill in the following table showing the effect of these functions for several example arguments. You will find it more convenient to work with a hexadecimal representation. Just remember that hex digits `8` through `F` have their most significant bits equal to 1.

`w`	`fun1(w)`	`fun2(w)`
`0x00000076`	_____	_____
`0x87654321`	_____	_____
`0x000000C9`	_____	_____
`0xEDCBA987`	_____	_____

(b) Describe in words the useful computation each of these functions perform.

Solution:

(a)

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

(b) `fun1` computes the zero-extension of the least significant byte, whereas `fun2` computes the sign-extension of the least significant byte. and the

Exercise 2.24. Suppose we truncate a 4-bit value (represented by hex digits `0` through `F`) to a 3-bit value (represented as hex digits `0` through `7`). Fill in the table below showing the effect of this truncation for some cases, in terms of the unsigned two's-complement interpretation of those bit patterns.

Hex		Unsigned		Two's Complement	
Original	Truncated	Original	Truncated	Original	Truncated
0	0	0	_____	0	_____
2	2	2	_____	2	_____
9	1	9	_____	-7	_____
B	3	11	_____	-5	_____
F	7	15	_____	-1	_____

Explain how Equations 4 and 5 apply to these cases.

$$B2U_k([x_{k-1}, x_{k-2}, \ldots, x_0]) = B2U_w([x_{w-1}, x_{w-2}, \ldots, x_0]) \mod 2^k \tag{4}$$

$$B2T_k([x_{k-1}, x_{k-2}, \ldots, x_0]) = U2T_k)(B2U_w([x_{w-1}, x_{w-2}, \ldots, x_0]) \mod 2^k) \tag{5}$$

Solution:

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

The value under the Unsigned Truncated column is obtained by applying equation 4, where we take apply $x \mod (2^3) = x \mod 8$ to all of the values. The value under the Two's Complement Truncated column ar obtained by mapping applying equation 5, which means we just apply $U2T_3$ to the result of the Unsigned Truncated column.

Exercise 2.25. Consider the following code that attempts to sum the elements of an array `a`, where the number of elements is given by parameter `length`:

```c
/* WARNING: This is buggy code */
float sum_elements(float a[], unsigned length) {
    int i;
    float result = 0;

    for (i = 0; i <= length-1; i++)
        result += a[i];
    return result;
}
```

When run with argument `length` equal to 0, this code should return `0.0`. Instead, it encounters a memory error. Explain why this happens. Show how this can be corrected.

Solution: The argument `length` is unsigned, so in the operation `length-1` of the loop condition, the operation is equivalent to `length-1U` because C implicit casts the signed operand `-1` to unsigned. The result is `0+UINT_MAX`, where `UINT_MAX` is the constant declared in `<limits.h>` that represents the maximum unsigned number that can be represented on the existing machine. When the unsigned comparison then happens, the loop condition is always true because no unsigned number exceeds this one. When the index `i` is then used to index into the array `a` inside the loop, an invalid memory location is accessed, causing the error. We can correct the by changing the condition to from `i <= length -1` to `i < length`.

Exercise 2.26. You are given the assignment of writing a function that determines whether one string is longer than another. You decide to make use of the string library function `strlen` having the following declaration:

```c
/* Prototype for library function strlen */
size_t strlen(const char *s);
```

Here is your first attempt at the function:

```c
/* Determine whether string s is longer than string t */
/* WARNING: This function is buggy */
int strlonger(char *s, char *t) {
    return strlen(s) - strlen(t) > 0;
}
```

When you test this on some simple data, things do not seem to work quite right. You investigate further and determine that, whe compiled as a 32-bit program, data type `size_t` is defined (via `typedef`) in header file `stdio.h` to be `unsigned`.

(a) For what cases will this function produce an incorrect result?

(b) Explain how this incorrect result comes about.

(c) Show how to fix the code so that it will work reliably.

Solution:

(a) The function will fail anytime the difference is negative, which is when `s` is shorter than `t`.

(b) The error occurs because when an expression contains unsigned operands, C will implicitly cast signed operands to unsigned. In this case, the result of `strlen(s) - strlen(t)` is cast to an unsigned number. Therefore, if the result is negative, it is cast to a positive number. If the result is negative, it is then cast to a positive number.

(c) We can fix the problem by replacing the expression after the `return` statement with `return strlen(s) > strlen(t)`.

Exercise 2.27. Write a function with the following prototype:

```c
/* Determine whether arguments can be added without overflow */
int uadd_ok(unsigned x, unsigned y);
```

This function should return 1 if arguments `x` and `y` can be added without overflow.

Solution: See code listing below for `./27-unsigned-addition-overflow/uoverflow.c`:

```c
#include "uoverflow.h"

int uadd_ok(unsigned x, unsigned y) {
    return (x + y) >= x;
}
```

Exercise 2.28. The presents shows that for any number x such that $0 \leq x < 2^w$, its w-bit unsigned negation $-^u_w$ is given by the following:

$$-^u_w x = \begin{cases} x, & x = 0 \\ 2^w - x, & x > 0 \end{cases} \tag{6}$$

We can represent a bit pattern of length $w = 4$ with a single hex digit. For an unsigned interpretation of these digits, use Equation 6 to fill in the following table giving the values and bit representations (in hex) of the unsigned additive inverses of the digits shown.

x		$-^u_w x$	
Hex	Decimal	Decimal	Hex
`0`	______	______	______
`5`	______	______	______
`8`	______	______	______
`D`	______	______	______
`F`	______	______	______

Solution:

x		$-^u_w x$	
Hex	Decimal	Decimal	Hex
`0`	0	0	`0x0`
`5`	5	11	`0xB`
`8`	8	8	`0x8`
`D`	13	3	`0x3`
`F`	15	1	`0x1`

Exercise 2.29. Fill in the following table in the style of Figure 2.25 (of book). Give the integer values of the 5-bit arguments, the values of both their integer and two's-complement sums, the bit-level representation of the two's-complement sum, and the case from the derivation of Equation 2.13.

x	y	$x + y$	$x +^t_5 y$	Case
______	______	______	______	______
[10100]	[10001]	______	______	
______	______	______	______	______
[11000]	[11000]	______	______	
______	______	______	______	______
[10111]	[01000]	______	______	
______	______	______	______	______
[00010]	[00101]	______	______	
______	______	______	______	______
[01100]	[00100]	______	______	

Solution:

x	y	$x + y$	$x +_5^t y$	Case
-12	-15	-27	5	1
[10100]	[10001]	[100101]	[00101]	
-8	-8	-16	-16	2
[11000]	[11000]	[10000]	[10000]	
-9	8	-1	-1	2
[10111]	[01000]	[11111]	[11111]	
2	5	7	7	3
[00010]	[00101]	[00111]	[00111]	
12	4	16	-16	4
[01100]	[00100]	[010000]	[10000]	

Exercise 2.30. Write a function with the following prototype

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

This function should return 1 if arguments `x` and `y` can be added without causing overflow.

Solution: See code listing below for `./30-signed-addition-overflow/toverflow.c`:

```c
#include "toverflow.h"

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

Exercise 2.31. Your coworker gets impatient with your analysis of the overflow conditions for two's-complement addition and presents you with the following implementation of `tadd_ok`:

```c
/* Determine whether arguments can be added without overflow */
/* WARNING: This code is buggy */
int tadd_ok(int x, int y) {
    int sum = x + y;
    return (sum-x == y) && (sum-y == x);
}
```

You look at the code and laugh. Explain why.

Solution: Recall that two's-complement numbers have the same bit-level representation as unsigned numbers. Therefore, two's-complement addition is characterized by converting to unsigned, performing unsigned addition, and converting back to two's-complement. As a result, `(x + y) - x` equals `y`, because unsigned addition is just modulo addition. Converting back to signed, now we are comparing two signed numbers with the same bit-level representation. The result always true; that is, `sum-x == y` is as tautology. Similarly, `sum -y` always equals `x`. Therefore, the function always returns 1.

Exercise 2.32. You are assigned the task of writing code for a function `tsub_ok`, with arguments `x` and `y`, that will return 1 if computing `x-y` does not cause overflow. Having just written the code for Problem 2.30, you write the following:

```
/* Determine whether arguemnts can be subtracted without overflow */
/* WARNING: This code is buggy */
int tsub_ok(int x, int y) {
    return tadd_ok(x, -y);
}
```

For what values of `x` and `y` will this function give incorrect results? Writing a correct version of this function is left as an exercise (Problem 2.74).

Solution: Subtraction overflow occurs when `x >= 0, y < 0`, and `x - y < 0`, or when `x <= 0, y > 0`, and `x - y > 0`. Suppose that $TMin_w$ `< y <=` $TMax_w$; then `-`$TMax_w$ `<= -y <` $-TMin_w$, or equivalently, $TMin_w$ `< -y <=` $TMax_w$. Then, `x + (-y)` causes positive overflow if `x > 0, -y > 0`, and `x + (-y) < 0`, which is equivalent to the first subtraction overflow condition. Similarly, `x + (-y)` causes negative overflow if `x < 0, -y < 0`, and `x + (-y) > 0`, which is precisely the second condition.

However, suppose that `y` is $TMin_w$. Then `-y` is unchanged. Recalling that the bit-level representation for two's-complement numbers is the same as unsigned, we could get the negated value by converting to unsigned, negating, and converting back to two's-complement. In converting to unsigned, `y` becomes $-TMin_w$ (which is a positive value in the range of unsigned numbers), and then applying the inverse operation `-` means we compute $2^w - y$, which remains unchanged because $-TMin_w = 2^{w-1}$, so $2^w - y = 2^w - 2^{w-1} = 2^{w-1} = -TMin_w$. Converting back to signed, we are back with the same number we started; that is, `y == -y`. But now:

(i) If `x` is 0, we get `tadd_ok(0, -y)`, which gives 1 because that addition does not overflow. However, the result of `0 - y` should be a positive number, so the `tsub_ok` ends up reporting the wrong value.

(ii) If `x` is positive, then `tadd_ok(x, -y)` reports 1 again, because there is no problem with adding a positive number and $TMin_w$. However, this is incorrect once again, because $x > 0$, $y < 0$, and $x - y < 0$, so the output should be 0, indicating overflow happens.

(iii) If `x` is negative, then `tadd_ok` receives a negative number being added to $TMin_w$, so the sum is positive and it reports overflow. However, that's incorrect also; for example $-1 - (TMin_w + 1)$ is a valid expression that returns $TMax_w$ (as it should).

In summary, it works correctly when `y` is not $TMin_w$. Suppose `y` is not $TMin_w$. Then

Exercise 2.33. We can represent a bit pattern of length $w = 4$ with a single hex digit. For a two's-complement interpretation of these digits, fill in the following table to determine the additive inverses of the digits shown:

	x	$-^t_4 x$	
Hex	Decimal	Decimal	Hex
`0`	____	____	____
`5`	____	____	____
`8`	____	____	____
`D`	____	____	____
`F`	____	____	____

What do you observe about the bit patterns generated by two's-complement and unsigned (Problem 2.28) negation?

Solution:

	x	$-^t_4 x$	
Hex	Decimal	Decimal	Hex
`0`	0	0	`0x0`
`5`	5	-5	`0xB`
`8`	-8	-8	`0x8`
`D`	-3	3	`0x3`
`F`	-1	1	`0x1`

The bit patterns are the same obtained when doing unsigned negation.

Exercise 2.34. Fill in the following table showing the results of multiplying different 3-bit numbers, in the style of Figure 2.27 (of the book):

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	____	[100]	____	[101]	____	____	____	____
Two's complement	____	[100]	____	[101]	____	____	____	____
Unsigned	____	[010]	____	[111]	____	____	____	____
Two's complement	____	[010]	____	[111]	____	____	____	____
Unsigned	____	[110]	____	[110]	____	____	____	____
Two's complement	____	[110]	____	[110]	____	____	____	____

Solution:

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	4	[100]	5	[101]	20	[10100]	4	[100]
Two's complement	-4	[100]	-3	[101]	12	[01100]	-4	[100]
Unsigned	2	[010]	7	[111]	14	[01110]	6	[110]
Two's complement	2	[010]	-1	[111]	-2	[11110]	-2	[110]
Unsigned	6	[110]	6	[110]	36	[100100]	4	[100]
Two's complement	-2	[110]	-2	[110]	4	[00100]	-4	[100]

Exercise 2.35. You are given the assignment to develop code for a function `tmult_ok` that will determine whether two arguments can be multiplied without causing overflow. Here is your solution:

```
/* Determine whether arguments can be multiplied without overflow */
int tmult_ok(int x, int y) {
    int p = x*y;
    /* Either x is zero, or diving p by x gives y */
    return !x || p/x = y;
}
```

You test this code for a number of values of `x` and `y`, and it seems to work properly. Your coworker challenges you, saying "If I can't use subtraction to test whether addition has overflowed (see Problem 2.31), then how can you use division to test whether multiplication has overflowed?"

Devise a mathematical justification of your approach, along the following lines. First, argue the case $x = 0$ is handled correctly. Otherwise, consider w-bit numbers x ($x \neq 0$), y, p, and q, where p is the result of performing two's-complement multiplication on x and y, and q is the result of dividing p by x.

1. Show that $x \cdot y$, the integer product of x and y, can be written in the form $x \cdot y = p + t2^w$, where $t \neq 0$ if and only if the computation of p overflows.

2. Show that p can be written in the form $p = x \cdot q + r$, where $|r| < |x|$.

3. Show that $q = y$ if and only if $r = t = 0$.

Solution: If $x = 0$, then the product is 0 regardless of the value of y, so there is no overflow because x is in the valid range for two's-complement numbers. Now suppose that $x \neq 0$ and y are w-bit two's-complement numbers.

1. Since p is the result of the unsigned two's-complement multiplication of x and y, we have

   $$p = x *^t_w y = U2T_w((x \cdot y) \mod 2^w)$$

   In particular, $u = (x \cdot y) \mod 2^w$ is in the range $[0, 2^w - 1]$, where $u = (x \cdot y) + s2^w$ form some s, by the definition of modulo arithmetic. Then $U2T_w(u) = -u_{w-1}2^w + u$, where u_{w-1} is its most significant bit. Thus

   $$p = U2T_w(u) = -u_{w-1}2^w + u = (x \cdot y) + (s - u_{w-1})2^w = (x \cdot y) + t2^w$$

   where we have let $t = (x - u_{w-1})$. If p overflows, then $x \cdot y$ is outside the range that can be represented by w bits, and since $x \cdot y = p - t2^w$, this happens if and only if $t \neq 0$, for if $t = 0$, the $x \cdot y = p$, which means they both belong to the range $[TMin_w, TMax_w]$.

2. Since q is the quotient of p and x, there are two cases: if x evenly divides p, the $q = p/x$, and hence, $p = x \cdot q + 0$, with $|0| < |x|$ since $x \neq 0$. On the other hand, if there is a remainder when dividing p by x, then $p = x \cdot q + r$.

3. Combining both equations, we have

   $$x \cdot y = x \cdot q + r + t2^w$$

If $q = y$, then the equation reduces to

$$r = -t2^w$$

However, since $|r| < |x|$ and $|x| < 2^w$, this equation does not have a solution unless $r = t = 0$. On the other hand, if we are given that $r = t = 0$, then the equation becomes

$$x(y - q) = 0$$

Since we know that $x \neq 0$, it follows that $y - q = 0$, so $y = q$.

Exercise 2.36. For the case where data type `int` has 32 bits, devise a version of `tmult_ok` (Problem 2.35) that uses the 64-bit precision of `int64_t`, without using division.

Solution: We declare a variable `p` of type `uint64_t` to hold the result of the product. However, that is not enough; if we compute the product of the two 32-bit int values, a 32-bit two's-complement multiplication takes place, and the result is simply sign-extended into the result variable. Instead, we need to ensure multiplication occurs as a 64-bit multiplication. To achieve this, we cast the operands to `uint64_t`. It's enough to cast one operand, e.g., `x`, since C will implicitly promote the other to be of the larger width. Now, overflows does not occur 32 most significant bits are all 0. If we compare the product `p` to its 32-bit truncated version, we can conclude that no truncation happens if their values are the same.

See code listing for `./36-signed-multiplication-overflow/tmultof.c`:

```c
#include <stdint.h> // int64_t
#include "tmultof.h"

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

Exercise 2.37. You are given the task of patching the vulnerability in the XDR code shown in the aside on page 100 for the case where both data types `int` and `size_t` are 32 bits. You decide to eliminate the possibility of the multiplication overflowing by computing the number of bytes to allocate using data type `uint64_t`. You replace the original call to `malloc` (line 9) as follows:

```c
uint64_t asize = ele_cnt * (uint64_t) ele_size;
void *result = malloc(asize);
```

Recall that the argument to `malloc` has type `size_t`.

1. Does your code provide any improvement over the original?

2. How would you change the code to eliminate the vulnerability?

Solution:

(a) Since the variables of type `int` and `size_t` are both 32 bits, their product fits in a 64-bit number. By using a local variable `uint64_t` and casting the `size_t` operand to `untin64_t`, a zero-extension occurs while a sign-extension occurs for the `int`. In any case, the product perfectly fits, without multiplication overflow occurring. However, since `malloc` expects a `size_t` which has 32 bits, and we are passing `asize` which has 64 bits, truncation happens, so there is still a risk that the overflow causes the allocated structure to be shorter in length than the argument `ele_cnt` that controls our iteration. Hence, the risk remains.

(b) To begin, I would change the parameter of type `int` to also be of type `size_t`. I would compute the product with 64 bit precision as in the previous part. But rather than immediately calling `malloc`, I would attempt to detect overflow, either via a downcast, such as `asize == (size_t) asize`, which would indicate that the most significant 32 bits are 0, or by the method of Problem 2.35, where ensure the quotient of `asize` and `ele_cnt` equals `ele_size`. In the scenario that it does not match, I would exit the function early, perhaps by returning `NULL` just like what happens for the check after the call to `malloc`.

Exercise 2.38. As we will see in Chapter 3, the `LEA` instruction can perform computations of the form `(a<<k) + b`, where `k` is either 0, 1, 2, or 3, and `b` is either 0 or some program value. The compiler uses this instruction to perform multiplications by constant factors. For example, we can compute `3*a` as `(a<<1) + a`. Considering cases where `b` is either 0 or equal to `a`, and all possible values of `k`, what multiples of `a` can be computed with a single `LEA` instruction?

Solution: The possible multiples are:

(a) `a*1`: This is `(a<<0) + a`, or just `a`.

(b) `a*2`: `(a<<1)+0`.

(c) `a*3`: `(a<<1)+a`.

(d) `a*4`: `(a<<2)+0`.

(e) `a*5`: `(a<<2)+a`.

(f) `a*8`: `(a<<3)+0`.

(g) `a*9`: `(a<<3)+a`

Note that `(a<<0)` is just `a`,

Exercise 2.39. According to the text, given a binary representation of K as an alternating sequence of zeros and ones:

$$[(0\dots0)(1\dots1)(0\dots0)\cdots(1\dots1)]$$

and letting n and m be bit positions representing the start and end, respectively, of a run of ones, then we can compute the effect of these bits on the product using either of the following two forms:

$$\text{Form A: } \texttt{(x<<n)(x<<(n-1))} + \cdots + \texttt{(x<<n)}$$
$$\text{Form B: } \texttt{(x<<(n+1))-(x<<m)}$$

For example, 14 can be written as $[(0 \ldots 0)(111)(0)]$. How could we modify the expression for form B for the case where bit position n is the most significant bit?

Solution: If bit position n is the most significant bit, then a shift by $(n+1)$ in form B results in 0, since all of the values are bits are shifted out. We can therefore replace that value with 0, amending form B to: `-(x<<m)`.

Exercise 2.40. For each of the following values of K, find ways to express `x*`K using only the specified number of operations, where we consider both additions and subtractions to have comparable cost. You may need to use some tricks beyond form A and B rules we have considered so far.

K	Shifts	Add/Subs	Expression
6	2	1	_____
31	1	1	_____
-6	2	1	_____
55	2	2	_____

Solution: Note that $55 = 64 - 9 = 2^6 - 2^3 - 1$, so `x*55 = x*(64) - x*(8) - x`:

K (Decimal)	Binary	Shifts	Add/Subs	Expression
6	110	2	1	`(x<<2) + (x<<1)`
31	11111	1	1	`(x<<5) - x`
-6	1010	2	1	`-(x<<3) + (x<<1)`
55	110111	2	2	`(x<<6) - (x<<3) - x`

Exercise 2.41. For a run of ones starting at bit position n down to bit position m ($n \geq m$), we saw that we can generate two forms of code, A and B. How should the compiler decide which form to use?

Solution: Assuming that the cost of addition and subtraction are comparable, and that every left shift is comparable (regardless of the shift amount), then whenever $n - m \geq 2$, meaning there is at least 3 shifts in Form A, then form B should be preferred.

Exercise 2.42. Write a function `div16` that returns the value `x/16` for integer arguments `x`. Your function should not use division, modulus, multiplication, any conditions (`if` or `?:`), any comparison operators (e.g., `<`, `>`, or `==`), or any loops. You may assume that data type `int` is 32 bits long, and uses a two's-complement representation, and that right shifts are performed arithmetically.

Solution: My implementation uses bit operations to see if the value is positive or negative, that way I can decide whether biasing is necessary. See code listing for `./42-div16/div16.c`:

```c
#include "div16.h"

int div16(int x) {
    // bias to ensure rounding up towards 0; note 16=2^4.
    static unsigned POW = 4;
    int bias = (0x10000000 & x) && 1; // bias by 0 (meaning not at all) if number
       is positive.
    return (x + (bias << POW) - bias) >> POW;
}
```

Exercise 2.43. In the following code, we have omitted the definitions of constants `M` and `N`:

```c
#define M  /* Mystery number 1 */
#define N  /* Mystery number 2 */
int arith(int x, int y) {
    int result = 0;
    result = x*M + y/N; /* M and N are mystery numbers */
    return result;
}
```

We compiled this code for particular values of `M` and `N`. The compiler optimized the multiplication and division using the methods we have discussed. The following is a translation of the generated machine code back into C:

```c
/* Translation of assembly code for arith */
int optarith(int x, int y) {
    int t = x;
    x <<= 5;
    x -= t;
    if (y<0) y += 7;
    y >>= 3; /* Arithmetic shift */
    return x+y;
}
```

What are the values of `M` and `N`?

Solution: Here, `x` was replaced by `x << 5 - x`, so `M =` $2^5 - 2^0 = 31$. Meanwhile, `y` was replaced by `y>>3` if it's positive, or by `(y+7)>>3` if it's negative. In this case, it means division by 8, so `N =` $2^3 = 8$.

Exercise 2.44. Assume data type `int` is 32 bits long and uses two's-complement representation for signed values. Right shifts are performed arithmetically for signed values and logically for unsigned values. The variables are declared and initialized as follows:

```c
int x = foo();     /* Arbitrary value */
int y = bar();     /* Arbitrary value */

unsigned ux = x;
unsigned uy = y;
```

For each of the following C expressions, either (1) argue that it is true (evaluates to 1) for all vales of `x` and `y`, or (2) give values of `x` and `y` for which it is false (evaluates to 0):

(a) `(x > 0) || (x-1 < 0)`

(b) `(x & 7) != 7 || (x << 29 < 0)`

(c) `(x * x) >= 0`

(d) `x < 0 || -x <= 0`

(e) `x > 0 || -x >= 0`

(f) `x+y == uy+ux`

(g) `x*~y + uy*ux == -x`

Solution:

(a) The statement is false when `x` is $TMin_w$, the smallest number that may be represented with a two's-complement integer. The expression `x > 0` is false because its value, -2^{w-1}. The expression `x-1` causes a negative overflow, resulting in a position number, and making the expression `x-1 < 0` false. The overall expression is false. The correct way to do this test is: `x>0 || x<1`.

(b) Recall that $7_{10} = 111_2$. Therefore, `(x & 7)` sets all but the three least significant bits to 0. Therefore, if the expression `(x & 7) != 7` is false, then it means at least one of the three least significant bits is 0. Meanwhile, the expression `x << 29` shifts all bits in `x` by 29 places to the left, meaning that the three least significant bits become the three most significant bits. Now, the way to get the expression to evaluate to false is clear: let `x` be any number whose value at bit position 2 is 0. The simplest example is 0, because `x & 7` is 0, so the first expression is false, and `x << 29` is 0, so the second expression is also false. For a less trivial example, let `x` be $3_{10} = 011$. Then `x & 7` is 3, which is not equal to 7. Meanwhile, `x << 29` has a 0 as a most significant bit, so the result is not negative.

(c) The result is flows when the square of `x` overflows. For example, recall that $TMax_{32}$ is $2^{31} - 1$. If `x` is 2^{16}, then `x * x` is 2^{32}, which results in positive overflow, and hence evaluates to a negative value, making the expression false. In general, any integer value of `x` such that `x` $> \sqrt{2^{31} - 1}$ will cause overflow.

(d) This is true for all `x`. If `x` is $TMin_w$, then the expression is true because its value is less than 0. If `x` is not equal to $TMin_w$ but is negative, then the expression is true by `x < 0`. If `x` is non-negative, then its inverse `-x` is a number in the range $TMin_w \leq$ `x` ≤ 0, so the second expression is true.

(e) If `x` is $TMin_w$, then `-x` is unchanged because it is its own inverse in two's-complement. Therefore, both `x > 0` and `-x >= 0` is false.

(f) True. By definition, two's-complement addition consists of converting the arguments to unsigned, performing unsigned addition, and then converting to back to two's-complement. Hence, `x+y` and `uy+ux` have the same binary representation. Therefore, since `x+y` is a compared to the unsigned quantity uy+ux, its type gets promoted to unsigned, and the equivalent bit representations cause the expression to result in true.

(g) Recall that a number can be negated by inverting its bits and adding 1. That is, `-y == ~y + 1`. Therefore, `~y == -y - 1`. This means that `x*~y` is equivalent to `x*-y - x`. Since `x*y` and `ux*uy` are equal (noting that `x*y` has the same representation and gets cast to unsigned), the result is `-x`.