

## 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:

$$19 = 3 + 4 \cdot 4$$

$$14 = 2 + 4 \cdot 3$$

$$12 = 0 + 4 \cdot 3$$

$$17 = 1 + 4 \cdot 4$$

$$5 = 1 + 4 \cdot 1$$

$$7 = 3 + 4 \cdot 1.$$

The filled-in table follows:

$n$	$2^n$ (decimal)	$2^n$ (hexadecimal)
9	512	0x200
19	<b>524,288</b>	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:

$$167 = 16 \cdot 10 + 7$$

$$10 = 16 \cdot 0 + 10$$

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

$$62 = 16 \cdot 3 + 14$$

$$3 = 16 \cdot 0 + 3$$

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

$$188 = 16 \cdot 11 + 12$$

$$12 = 16 \cdot 0 + 12$$

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:

$$0x37 = 3 \cdot 16^1 + 7 \cdot 16^0 = 55_{10}$$

$$0x88 = 8 \cdot 16^1 + 8 \cdot 16^0 = 136_{10}$$

$$0xF3 = 15 \cdot 16^1 + 3 \cdot 16^0 = 243_{10}$$

$$0x52 = 5 \cdot 16^1 + 2 \cdot 16^0 = 82_{10}$$

$$0xAC = 10 \cdot 16^1 + 12 \cdot 16^0 = 172_{10}$$

$$0xE7 = 14 \cdot 16^1 + 7 \cdot 16^0 = 231_{10}$$

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)  $0x503c + 0x8 = \underline{\hspace{2cm}}$



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:

```

000000000001101011001000101000001
010010100110101100100010100000100

```

3. The last two bits in the float do not match. Also, the leading bits in integer, 000000000001, 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`?

---

```

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]
$\sim a$	_____
$\sim b$	_____
$a \& b$	_____
$a   b$	_____
$a^{\wedge} b$	_____

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

Operationg	Result
$a$	[01101001]
$b$	[01010101]
$\sim a$	[10010110]
$\sim b$	[10101010]
$a \& b$	[01000001]
$a   b$	[01111101]
$a^{\wedge} b$	[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:

Blue | Green = \_\_\_\_\_  
 Yellow & Cyan = \_\_\_\_\_  
 Red ^ Magenta = \_\_\_\_\_

**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  $001 \mid 010 = 011$ , 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  $100 \wedge 101 = 001$ , which is Blue, so

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

**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.

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

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	$a$	$a \wedge b$
Step 2	$a \wedge (a \wedge b)$	$a \wedge b$
Step 3	$a \wedge (a \wedge b)$	$[a \wedge (a \wedge b)] \wedge [a \wedge b]$

Since  $a \wedge (a \wedge b) = (a \wedge a) \wedge b = 0 \wedge b = b$ , 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.

- For an array of odd length  $\text{cnt} = 2k + 1$ , what are the values of variables `first` and `last` in the final iteration of function `reverse_array`?
- Why does this call to function `inplace_swap` set the array element to 0?
- 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 `bic` 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 operations bis and bic */
int bis(int x, int m);
int bic(int x, int m);

/* Compute x|y using only calls to functions 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  $x \wedge y$  by applying the OR operation (which is just `bis`):

---

```

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

/* Compute x|y using only calls to functions 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
$x \ \& \ y$	_____	$x \ \&\& \ y$	_____
$x \   \ y$	_____	$x \    \ y$	_____
$\sim x \   \ \sim y$	_____	$!x \    \ !y$	_____
$x \ \& \ !y$	_____	$x \ \&\& \ \sim y$	_____

**Solution:** Note that  $x = 0x66 = 0110 \ 0110$  and  $y = 0x39 = 0011 \ 1001$ . Also  $\sim x = 1001 \ 1001$  and  $\sim y = 1100 \ 0110$ .

Expression	Value	Expression	Value
$x \ \& \ y$	0010 0000	$x \ \&\& \ y$	0x01
$x \   \ y$	0111 1111	$x \    \ y$	0x01
$\sim x \   \ \sim y$	1101 1111	$!x \    \ !y$	0x00
$x \ \& \ !y$	0100 0110	$x \ \&\& \ \sim y$	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  $!(x \ \& \ \sim y)$ . Suppose  $x$  and  $y$  are the same; then  $x \ \& \ \sim y$  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  $x \ \& \ \sim y$  is the same, so  $x \ \& \ \sim y$  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
$x \ll 3$				
$x \gg 2$ (Logical)				
$x \gg 2$ (Arithmetic)				

**Solution:** We begin by converting each hexadecimal to binary:

$$0xC3 = 1100 \ 0011$$

$$0x75 = 0111 \ 0101$$

$$0x87 = 1000 \ 0111$$

$$0x66 = 0110 \ 0110$$

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:

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

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

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

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

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

- (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

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

- (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

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

**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.

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

**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
<code>-2147483647-1 == 2147483648U</code>	_____	_____
<code>-2147483647-1 == 2147483647</code>	_____	_____
<code>-2147483647-1U == 2147483647</code>	_____	_____
<code>-2147483647-1 == -2147483647</code>	_____	_____
<code>-2147483647-1U == -2147483647</code>	_____	_____

**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
<code>-2147483647-1 == 2147483648U</code>	Unsigned	1
<code>-2147483647-1 == 2147483647</code>	Signed	0
<code>-2147483647-1U == 2147483647</code>	Unsigned	0
<code>-2147483647-1 == -2147483647</code>	Signed	1
<code>-2147483647-1U == -2147483647</code>	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:

---

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

int fun2(unsigned 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 of 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	0xFFFFFC9
0xEDCBA987	0x00000087	0xFFFFF87

- (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.

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

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

**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  $x \bmod (2^3) = x \bmod 8$  to all of the values. The value under the Two's Complement Truncated column are 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`:

---

```
/* 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 implicitly 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 this 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:

---

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

---

Here is your first attempt at the function:

---

```
/* 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, when 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)`.