The Library

Reproduced from:
Computer Systems – A Programmer's Perspective, Bryant; O'Hallaron, pp. 108-126, Prentice Hall, 2011,

This copy was made pursuant to the Fair Dealing Policy of the University of Guelph. The copy may only be used for the purpose of research, private study, criticism, review, news reporting, education, satire or parody. If the copy is used for the purpose of review, criticism or news reporting, the source and the name of the author must be mentioned. The use of this copy for any other purpose may require the permission of the copyright owner.

Additional information about University of Guelph's copyright policies

M: The value of the significand

$2^E \times M$: The (unreduced) fractional value of the number

V: The reduced fractional value of the number

Decimal: The decimal representation of the number

Express the values of 2^E, f, M, $2^E \times M$, and V either as integers (when possible) or as fractions of the form $\frac{x}{y}$, where y is a power of 2. You need not fill in entries marked "—".

Bits	e	E	2^E	f	M	$2^E \times M$	V	Decimal
0 00 00	—	—	—	—	—	—	—	—
0 00 01	—	—	—	—	—	—	—	—
0 00 10	—	—	—	—	—	—	—	—
0 00 11	—	—	—	—	—	—	—	—
0 01 00	—	—	—	—	—	—	—	—
0 01 01	1	0	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	1.25
0 01 10	—	—	—	—	—	—	—	—
0 01 11	—	—	—	—	—	—	—	—
0 10 00	—	—	—	—	—	—	—	—
0 10 01	—	—	—	—	—	—	—	—
0 10 10	—	—	—	—	—	—	—	—
0 10 11	—	—	—	—	—	—	—	—
0 11 00	—	—	—	—	—	—	—	—
0 11 01	—	—	—	—	—	—	—	—
0 11 10	—	—	—	—	—	—	—	—
0 11 11	—	—	—	—	—	—	—	—

Figure 2.35 shows the representations and numeric values of some important single- and double-precision floating-point numbers. As with the 8-bit format shown in Figure 2.34, we can see some general properties for a floating-point representation with a k-bit exponent and an n-bit fraction:

- The value +0.0 always has a bit representation of all zeros.
- The smallest positive denormalized value has a bit representation consisting of a 1 in the least significant bit position and otherwise all zeros. It has a fraction (and significand) value $M = f = 2^{-n}$ and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = 2^{-n-2^{k-1}+2}$.
- The largest denormalized value has a bit representation consisting of an exponent field of all zeros and a fraction field of all ones. It has a fraction (and significand) value $M = f = 1 - 2^{-n}$ (which we have written $1 - \epsilon$) and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = (1 - 2^{-n}) \times 2^{-2^{k-1}+2}$, which is just slightly smaller than the smallest normalized value.

Description	exp	frac	Single precision Value	Single precision Decimal	Double precision Value	Double precision Decimal
Zero	00···00	0···00	0	0.0	0	0.0
Smallest denorm.	00···00	0···01	$2^{-23} \times 2^{-126}$	1.4×10^{-45}	$2^{-52} \times 2^{-1022}$	4.9×10^{-324}
Largest denorm.	00···00	1···11	$(1-\epsilon) \times 2^{-126}$	1.2×10^{-38}	$(1-\epsilon) \times 2^{-1022}$	2.2×10^{-308}
Smallest norm.	00···01	0···00	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
One	01···11	0···00	1×2^{0}	1.0	1×2^{0}	1.0
Largest norm.	11···10	1···11	$(2-\epsilon) \times 2^{127}$	3.4×10^{38}	$(2-\epsilon) \times 2^{1023}$	1.8×10^{308}

Figure 2.35 **Examples of nonnegative floating-point numbers.**

- The smallest positive normalized value has a bit representation with a 1 in the least significant bit of the exponent field and otherwise all zeros. It has a significand value $M = 1$ and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = 2^{-2^{k-1}+2}$.
- The value 1.0 has a bit representation with all but the most significant bit of the exponent field equal to 1 and all other bits equal to 0. Its significand value is $M = 1$ and its exponent value is $E = 0$.
- The largest normalized value has a bit representation with a sign bit of 0, the least significant bit of the exponent equal to 0, and all other bits equal to 1. It has a fraction value of $f = 1 - 2^{-n}$, giving a significand $M = 2 - 2^{-n}$ (which we have written $2 - \epsilon$). It has an exponent value $E = 2^{k-1} - 1$, giving a numeric value $V = (2 - 2^{-n}) \times 2^{2^{k-1}-1} = (1 - 2^{-n-1}) \times 2^{2^{k-1}}$.

One useful exercise for understanding floating-point representations is to convert sample integer values into floating-point form. For example, we saw in Figure 2.14 that 12,345 has binary representation [11000000111001]. We create a normalized representation of this by shifting 13 positions to the right of a binary point, giving $12345 = 1.1000000111001_2 \times 2^{13}$. To encode this in IEEE single-precision format, we construct the fraction field by dropping the leading 1 and adding 10 zeros to the end, giving binary representation [10000001110010000000000]. To construct the exponent field, we add bias 127 to 13, giving 140, which has binary representation [10001100]. We combine this with a sign bit of 0 to get the floating-point representation in binary of [01000110010000001110010000000000]. Recall from Section 2.1.4 that we observed the following correlation in the bit-level representations of the integer value 12345 (0x3039) and the single-precision floating-point value 12345.0 (0x4640E400):

```
  0   0   0   0   3   0   3   9
00000000000000000011000000111001
            ************
    4   6   4   0   E   4   0   0
01000110010000001110010000000000
```

We can now see that the region of correlation corresponds to the low-order bits of the integer, stopping just before the most significant bit equal to 1 (this bit forms the implied leading 1), matching the high-order bits in the fraction part of the floating-point representation.

Practice Problem 2.48

As mentioned in Problem 2.6, the integer 3,510,593 has hexadecimal representation `0x00359141`, while the single-precision, floating-point number 3510593.0 has hexadecimal representation `0x4A564504`. Derive this floating-point representation and explain the correlation between the bits of the integer and floating-point representations.

Practice Problem 2.49

A. For a floating-point format with an n-bit fraction, give a formula for the smallest positive integer that cannot be represented exactly (because it would require an $n+1$-bit fraction to be exact). Assume the exponent field size k is large enough that the range of representable exponents does not provide a limitation for this problem.

B. What is the numeric value of this integer for single-precision format ($n = 23$)?

2.4.4 Rounding

Floating-point arithmetic can only approximate real arithmetic, since the representation has limited range and precision. Thus, for a value x, we generally want a systematic method of finding the "closest" matching value x' that can be represented in the desired floating-point format. This is the task of the *rounding* operation. One key problem is to define the direction to round a value that is halfway between two possibilities. For example, if I have \$1.50 and want to round it to the nearest dollar, should the result be \$1 or \$2? An alternative approach is to maintain a lower and an upper bound on the actual number. For example, we could determine representable values x^- and x^+ such that the value x is guaranteed to lie between them: $x^- \leq x \leq x^+$. The IEEE floating-point format defines four different *rounding modes*. The default method finds a closest match, while the other three can be used for computing upper and lower bounds.

Figure 2.36 illustrates the four rounding modes applied to the problem of rounding a monetary amount to the nearest whole dollar. Round-to-even (also called round-to-nearest) is the default mode. It attempts to find a closest match. Thus, it rounds \$1.40 to \$1 and \$1.60 to \$2, since these are the closest whole dollar values. The only design decision is to determine the effect of rounding values that are halfway between two possible results. Round-to-even mode adopts the

Mode	$1.40	$1.60	$1.50	$2.50	$−1.50
Round-to-even	$1	$2	$2	$2	$−2
Round-toward-zero	$1	$1	$1	$2	$−1
Round-down	$1	$1	$1	$2	$−2
Round-up	$2	$2	$2	$3	$−1

Figure 2.36 **Illustration of rounding modes for dollar rounding.** The first rounds to a nearest value, while the other three bound the result above or below.

convention that it rounds the number either upward or downward such that the least significant digit of the result is even. Thus, it rounds both \$1.50 and \$2.50 to \$2.

The other three modes produce guaranteed bounds on the actual value. These can be useful in some numerical applications. Round-toward-zero mode rounds positive numbers downward and negative numbers upward, giving a value $\hat{x}$ such that $|\hat{x}| \leq |x|$. Round-down mode rounds both positive and negative numbers downward, giving a value x^- such that $x^- \leq x$. Round-up mode rounds both positive and negative numbers upward, giving a value x^+ such that $x \leq x^+$.

Round-to-even at first seems like it has a rather arbitrary goal—why is there any reason to prefer even numbers? Why not consistently round values halfway between two representable values upward? The problem with such a convention is that one can easily imagine scenarios in which rounding a set of data values would then introduce a statistical bias into the computation of an average of the values. The average of a set of numbers that we rounded by this means would be slightly higher than the average of the numbers themselves. Conversely, if we always rounded numbers halfway between downward, the average of a set of rounded numbers would be slightly lower than the average of the numbers themselves. Rounding toward even numbers avoids this statistical bias in most real-life situations. It will round upward about 50% of the time and round downward about 50% of the time.

Round-to-even rounding can be applied even when we are not rounding to a whole number. We simply consider whether the least significant digit is even or odd. For example, suppose we want to round decimal numbers to the nearest hundredth. We would round 1.2349999 to 1.23 and 1.2350001 to 1.24, regardless of rounding mode, since they are not halfway between 1.23 and 1.24. On the other hand, we would round both 1.2350000 and 1.2450000 to 1.24, since 4 is even.

Similarly, round-to-even rounding can be applied to binary fractional numbers. We consider least significant bit value 0 to be even and 1 to be odd. In general, the rounding mode is only significant when we have a bit pattern of the form $XX\cdots X.YY\cdots Y100\cdots$, where X and Y denote arbitrary bit values with the rightmost Y being the position to which we wish to round. Only bit patterns of this form denote values that are halfway between two possible results. As examples, consider the problem of rounding values to the nearest quarter (i.e., 2 bits to the right of the binary point). We would round 10.00011_2 ($2\frac{3}{32}$) down to 10.00_2 (2),

and 10.00110_2 ($2\frac{3}{16}$) up to 10.01_2 ($2\frac{1}{4}$), because these values are not halfway between two possible values. We would round 10.11100_2 ($2\frac{7}{8}$) up to 11.00_2 (3) and 10.10100_2 ($2\frac{5}{8}$) down to 10.10_2 ($2\frac{1}{2}$), since these values are halfway between two possible results, and we prefer to have the least significant bit equal to zero.

Practice Problem 2.50

Show how the following binary fractional values would be rounded to the nearest half (1 bit to the right of the binary point), according to the round-to-even rule. In each case, show the numeric values, both before and after rounding.

A. 10.010_2

B. 10.011_2

C. 10.110_2

D. 11.001_2

Practice Problem 2.51

We saw in Problem 2.46 that the Patriot missile software approximated 0.1 as $x = 0.00011001100110011001100_2$. Suppose instead that they had used IEEE round-to-even mode to determine an approximation x' to 0.1 with 23 bits to the right of the binary point.

A. What is the binary representation of x'?

B. What is the approximate decimal value of $x' - 0.1$?

C. How far off would the computed clock have been after 100 hours of operation?

D. How far off would the program's prediction of the position of the Scud missile have been?

Practice Problem 2.52

Consider the following two 7-bit floating-point representations based on the IEEE floating point format. Neither has a sign bit—they can only represent nonnegative numbers.

1. Format A
   - There are $k = 3$ exponent bits. The exponent bias is 3.
   - There are $n = 4$ fraction bits.
2. Format B
   - There are $k = 4$ exponent bits. The exponent bias is 7.
   - There are $n = 3$ fraction bits.

Below, you are given some bit patterns in Format A, and your task is to convert them to the closest value in Format B. If necessary, you should apply the round-to-even rounding rule. In addition, give the values of numbers given by the Format A

and Format B bit patterns. Give these as whole numbers (e.g., 17) or as fractions (e.g., 17/64).

Format A		Format B	
Bits	Value	Bits	Value
011 0000	1	0111 000	1
101 1110			
010 1001			
110 1111			
000 0001			

2.4.5 Floating-Point Operations

The IEEE standard specifies a simple rule for determining the result of an arithmetic operation such as addition or multiplication. Viewing floating-point values x and y as real numbers, and some operation $\odot$ defined over real numbers, the computation should yield $Round(x \odot y)$, the result of applying rounding to the exact result of the real operation. In practice, there are clever tricks floating-point unit designers use to avoid performing this exact computation, since the computation need only be sufficiently precise to guarantee a correctly rounded result. When one of the arguments is a special value such as -0, ∞, or NaN, the standard specifies conventions that attempt to be reasonable. For example, $1/-0$ is defined to yield $-\infty$, while $1/+0$ is defined to yield $+\infty$.

One strength of the IEEE standard's method of specifying the behavior of floating-point operations is that it is independent of any particular hardware or software realization. Thus, we can examine its abstract mathematical properties without considering how it is actually implemented.

We saw earlier that integer addition, both unsigned and two's complement, forms an abelian group. Addition over real numbers also forms an abelian group, but we must consider what effect rounding has on these properties. Let us define $x +^f y$ to be $Round(x + y)$. This operation is defined for all values of x and y, although it may yield infinity even when both x and y are real numbers due to overflow. The operation is commutative, with $x +^f y = y +^f x$ for all values of x and y. On the other hand, the operation is not associative. For example, with single-precision floating point the expression `(3.14+1e10)-1e10` evaluates to `0.0`—the value 3.14 is lost due to rounding. On the other hand, the expression `3.14+(1e10-1e10)` evaluates to `3.14`. As with an abelian group, most values have inverses under floating-point addition, that is, $x +^f -x = 0$. The exceptions are infinities (since $+\infty - \infty = NaN$), and NaN's, since $NaN +^f x = NaN$ for any x.

The lack of associativity in floating-point addition is the most important group property that is lacking. It has important implications for scientific programmers and compiler writers. For example, suppose a compiler is given the following code fragment:

```
x = a + b + c;
y = b + c + d;
```

The compiler might be tempted to save one floating-point addition by generating the following code:

```
t = b + c;
x = a + t;
y = t + d;
```

However, this computation might yield a different value for `x` than would the original, since it uses a different association of the addition operations. In most applications, the difference would be so small as to be inconsequential. Unfortunately, compilers have no way of knowing what trade-offs the user is willing to make between efficiency and faithfulness to the exact behavior of the original program. As a result, they tend to be very conservative, avoiding any optimizations that could have even the slightest effect on functionality.

On the other hand, floating-point addition satisfies the following monotonicity property: if $a \geq b$ then $x + a \geq x + b$ for any values of a, b, and x other than NaN. This property of real (and integer) addition is not obeyed by unsigned or two's-complement addition.

Floating-point multiplication also obeys many of the properties one normally associates with multiplication. Let us define $x *^{\mathrm{f}} y$ to be $Round(x \times y)$. This operation is closed under multiplication (although possibly yielding infinity or NaN), it is commutative, and it has 1.0 as a multiplicative identity. On the other hand, it is not associative, due to the possibility of overflow or the loss of precision due to rounding. For example, with single-precision floating point, the expression `(1e20*1e20)*1e-20` evaluates to $+\infty$, while `1e20*(1e20*1e-20)` evaluates to `1e20`. In addition, floating-point multiplication does not distribute over addition. For example, with single-precision floating point, the expression `1e20*(1e20-1e20)` evaluates to `0.0`, while `1e20*1e20-1e20*1e20` evaluates to `NaN`.

On the other hand, floating-point multiplication satisfies the following monotonicity properties for any values of a, b, and c other than NaN:

$$a \geq b \text{ and } c \geq 0 \Rightarrow a *^{\mathrm{f}} c \geq b *^{\mathrm{f}} c$$

$$a \geq b \text{ and } c \leq 0 \Rightarrow a *^{\mathrm{f}} c \leq b *^{\mathrm{f}} c$$

In addition, we are also guaranteed that $a *^{\mathrm{f}} a \geq 0$, as long as $a \neq NaN$. As we saw earlier, none of these monotonicity properties hold for unsigned or two's-complement multiplication.

This lack of associativity and distributivity is of serious concern to scientific programmers and to compiler writers. Even such a seemingly simple task as writing code to determine whether two lines intersect in 3-dimensional space can be a major challenge.

2.4.6 Floating Point in C

All versions of C provide two different floating-point data types: `float` and `double`. On machines that support IEEE floating point, these data types correspond to single- and double-precision floating point. In addition, the machines use

the round-to-even rounding mode. Unfortunately, since the C standards do not require the machine to use IEEE floating point, there are no standard methods to change the rounding mode or to get special values such as -0, $+\infty$, $-\infty$, or *NaN*. Most systems provide a combination of include ('.h') files and procedure libraries to provide access to these features, but the details vary from one system to another. For example, the GNU compiler GCC defines program constants `INFINITY` (for $+\infty$) and `NAN` (for *NaN*) when the following sequence occurs in the program file:

```
#define _GNU_SOURCE 1
#include <math.h>
```

More recent versions of C, including ISO C99, include a third floating-point data type, `long double`. For many machines and compilers, this data type is equivalent to the `double` data type. For Intel-compatible machines, however, GCC implements this data type using an 80-bit "extended precision" format, providing a much larger range and precision than does the standard 64-bit format. The properties of this format are investigated in Problem 2.85.

Practice Problem 2.53

Fill in the following macro definitions to generate the double-precision values $+\infty$, $-\infty$, and 0:

```
#define POS_INFINITY
#define NEG_INFINITY
#define NEG_ZERO
```

You cannot use any include files (such as `math.h`), but you can make use of the fact that the largest finite number that can be represented with double precision is around 1.8×10^{308}.

When casting values between `int`, `float`, and `double` formats, the program changes the numeric values and the bit representations as follows (assuming a 32-bit `int`):

- From `int` to `float`, the number cannot overflow, but it may be rounded.
- From `int` or `float` to `double`, the exact numeric value can be preserved because `double` has both greater range (i.e., the range of representable values), as well as greater precision (i.e., the number of significant bits).
- From `double` to `float`, the value can overflow to $+\infty$ or $-\infty$, since the range is smaller. Otherwise, it may be rounded, because the precision is smaller.
- From `float` or `double` to `int` the value will be rounded toward zero. For example, 1.999 will be converted to 1, while -1.999 will be converted to -1. Furthermore, the value may overflow. The C standards do not specify a fixed result for this case. Intel-compatible microprocessors designate the

bit pattern $[10\cdots00]$ ($TMin_w$ for word size w) as an *integer indefinite* value. Any conversion from floating point to integer that cannot assign a reasonable integer approximation yields this value. Thus, the expression `(int) +1e10` yields `-21483648`, generating a negative value from a positive one.

Web Aside DATA:IA32-FP Intel IA32 floating-point arithmetic

In the next chapter, we will begin an in-depth study of Intel IA32 processors, the processor found in many of today's personal computers. Here we highlight an idiosyncrasy of these machines that can seriously affect the behavior of programs operating on floating-point numbers when compiled with GCC.

IA32 processors, like most other processors, have special memory elements called *registers* for holding floating-point values as they are being computed and used. The unusual feature of IA32 is that the floating-point registers use a special 80-bit *extended-precision* format to provide a greater range and precision than the normal 32-bit single-precision and 64-bit double-precision formats used for values held in memory. (See Problem 2.85.) All single- and double-precision numbers are converted to this format as they are loaded from memory into floating-point registers. The arithmetic is always performed in extended precision. Numbers are converted from extended precision to single- or double-precision format as they are stored in memory.

This extension to 80 bits for all register data and then contraction to a smaller format for memory data has some undesirable consequences for programmers. It means that storing a number from a register to memory and then retrieving it back into the register can cause it to change, due to rounding, underflow, or overflow. This storing and retrieving is not always visible to the C programmer, leading to some very peculiar results.

More recent versions of Intel processors, including both IA32 and newer 64-bit machines, provide direct hardware support for single- and double-precision floating-point operations. The peculiarities of the historic IA32 approach will diminish in importance with new hardware and with compilers that generate code based on the newer floating-point instructions.

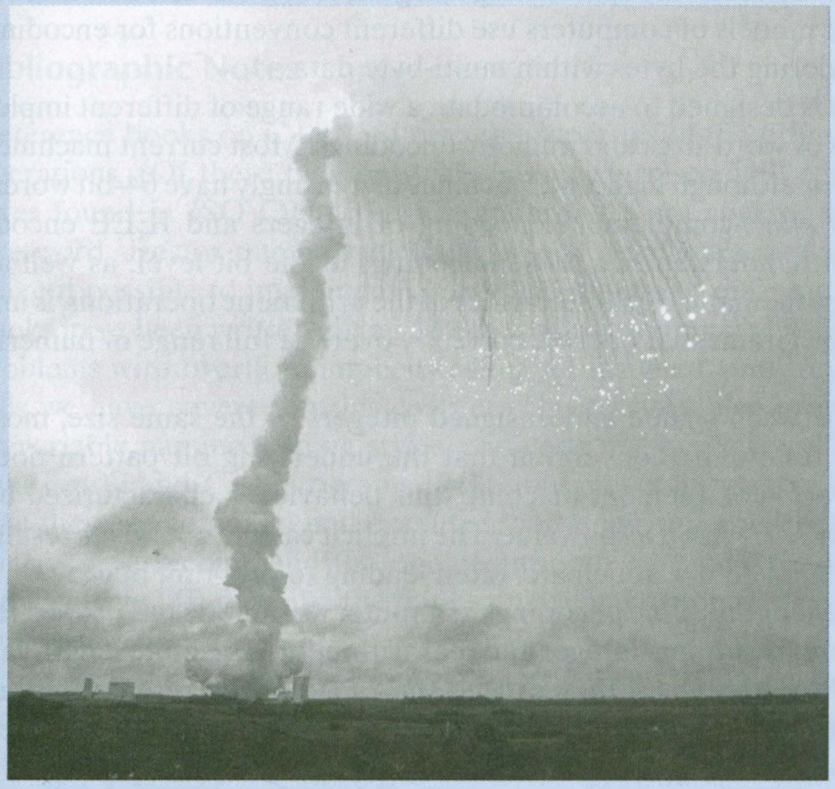
Aside Ariane 5: the high cost of floating-point overflow

Converting large floating-point numbers to integers is a common source of programming errors. Such an error had disastrous consequences for the maiden voyage of the Ariane 5 rocket, on June 4, 1996. Just 37 seconds after liftoff, the rocket veered off its flight path, broke up, and exploded. Communication satellites valued at $500 million were on board the rocket.

A later investigation [69, 39] showed that the computer controlling the inertial navigation system had sent invalid data to the computer controlling the engine nozzles. Instead of sending flight control information, it had sent a diagnostic bit pattern indicating that an overflow had occurred during the conversion of a 64-bit floating-point number to a 16-bit signed integer.

The value that overflowed measured the horizontal velocity of the rocket, which could be more than 5 times higher than that achieved by the earlier Ariane 4 rocket. In the design of the Ariane 4 software, they had carefully analyzed the numeric values and determined that the horizontal velocity

would never overflow a 16-bit number. Unfortunately, they simply reused this part of the software in the Ariane 5 without checking the assumptions on which it had been based.



© Fourmy/REA/SABA/Corbis

Practice Problem 2.54

Assume variables `x`, `f`, and `d` are of type `int`, `float`, and `double`, respectively. Their values are arbitrary, except that neither `f` nor `d` equals $+\infty$, $-\infty$, or *NaN*. For each of the following C expressions, either argue that it will always be true (i.e., evaluate to 1) or give a value for the variables such that it is not true (i.e., evaluates to 0).

A. `x == (int)(double) x`

B. `x == (int)(float) x`

C. `d == (double)(float) d`

D. `f == (float)(double) f`

E. `f == -(-f)`

F. `1.0/2 == 1/2.0`

G. `d*d >= 0.0`

H. `(f+d)-f == d`

2.5 Summary

Computers encode information as bits, generally organized as sequences of bytes. Different encodings are used for representing integers, real numbers, and character strings. Different models of computers use different conventions for encoding numbers and for ordering the bytes within multi-byte data.

The C language is designed to accommodate a wide range of different implementations in terms of word sizes and numeric encodings. Most current machines have 32-bit word sizes, although high-end machines increasingly have 64-bit words. Most machines use two's-complement encoding of integers and IEEE encoding of floating point. Understanding these encodings at the bit level, as well as understanding the mathematical characteristics of the arithmetic operations, is important for writing programs that operate correctly over the full range of numeric values.

When casting between signed and unsigned integers of the same size, most C implementations follow the convention that the underlying bit pattern does not change. On a two's-complement machine, this behavior is characterized by functions $T2U_w$ and $U2T_w$, for a w-bit value. The implicit casting of C gives results that many programmers do not anticipate, often leading to program bugs.

Due to the finite lengths of the encodings, computer arithmetic has properties quite different from conventional integer and real arithmetic. The finite length can cause numbers to overflow, when they exceed the range of the representation. Floating-point values can also underflow, when they are so close to 0.0 that they are changed to zero.

The finite integer arithmetic implemented by C, as well as most other programming languages, has some peculiar properties compared to true integer arithmetic. For example, the expression `x*x` can evaluate to a negative number due to overflow. Nonetheless, both unsigned and two's-complement arithmetic satisfy many of the other properties of integer arithmetic, including associativity, commutativity, and distributivity. This allows compilers to do many optimizations. For example, in replacing the expression `7*x` by `(x<<3)-x`, we make use of the associative, commutative, and distributive properties, along with the relationship between shifting and multiplying by powers of 2.

We have seen several clever ways to exploit combinations of bit-level operations and arithmetic operations. For example, we saw that with two's-complement arithmetic `~x+1` is equivalent to `-x`. As another example, suppose we want a bit pattern of the form $[0, \ldots, 0, 1, \ldots, 1]$, consisting of $w - k$ zeros followed by k ones. Such bit patterns are useful for masking operations. This pattern can be generated by the C expression `(1<<k)-1`, exploiting the property that the desired bit pattern has numeric value $2^k - 1$. For example, the expression `(1<<8)-1` will generate the bit pattern `0xFF`.

Floating-point representations approximate real numbers by encoding numbers of the form $x \times 2^y$. The most common floating-point representation is defined by IEEE Standard 754. It provides for several different precisions, with the most common being single (32 bits) and double (64 bits). IEEE floating point also has representations for special values representing plus and minus infinity, as well as not-a-number.

Floating-point arithmetic must be used very carefully, because it has only limited range and precision, and because it does not obey common mathematical properties such as associativity.

Bibliographic Notes

Reference books on C [48, 58] discuss properties of the different data types and operations. (Of these two, only Steele and Harbison [48] cover the newer features found in ISO C99.) The C standards do not specify details such as precise word sizes or numeric encodings. Such details are intentionally omitted to make it possible to implement C on a wide range of different machines. Several books have been written giving advice to C programmers [59, 70] that warn about problems with overflow, implicit casting to unsigned, and some of the other pitfalls we have covered in this chapter. These books also provide helpful advice on variable naming, coding styles, and code testing. Seacord's book on security issues in C and C++ programs [94], combines information about C programs, how they are compiled and executed, and how vulnerabilities may arise. Books on Java (we recommend the one coauthored by James Gosling, the creator of the language [4]) describe the data formats and arithmetic operations supported by Java.

Most books on logic design [56, 115] have a section on encodings and arithmetic operations. Such books describe different ways of implementing arithmetic circuits. Overton's book on IEEE floating point [78] provides a detailed description of the format as well as the properties from the perspective of a numerical applications programmer.

Homework Problems

2.55 ◆

Compile and run the sample code that uses `show_bytes` (file `show-bytes.c`) on different machines to which you have access. Determine the byte orderings used by these machines.

2.56 ◆

Try running the code for `show_bytes` for different sample values.

2.57 ◆

Write procedures `show_short`, `show_long`, and `show_double` that print the byte representations of C objects of types `short int`, `long int`, and `double`, respectively. Try these out on several machines.

2.58 ◆◆

Write a procedure `is_little_endian` that will return 1 when compiled and run on a little-endian machine, and will return 0 when compiled and run on a big-endian machine. This program should run on any machine, regardless of its word size.

2.59 ◆◆

Write a C expression that will yield a word consisting of the least significant byte of x, and the remaining bytes of y. For operands x = 0x89ABCDEF and y = 0x76543210, this would give 0x765432EF.

2.60 ◆◆

Suppose we number the bytes in a w-bit word from 0 (least significant) to $w/8 - 1$ (most significant). Write code for the following C function, which will return an unsigned value in which byte i of argument x has been replaced by byte b:

```
unsigned replace_byte (unsigned x, int i, unsigned char b);
```

Here are some examples showing how the function should work:

```
replace_byte(0x12345678, 2, 0xAB) --> 0x12AB5678
replace_byte(0x12345678, 0, 0xAB) --> 0x123456AB
```

Bit-level integer coding rules

In several of the following problems, we will artificially restrict what programming constructs you can use to help you gain a better understanding of the bit-level, logic, and arithmetic operations of C. In answering these problems, your code must follow these rules:

- Assumptions
  - Integers are represented in two's-complement form.
  - Right shifts of signed data are performed arithmetically.
  - Data type `int` is w bits long. For some of the problems, you will be given a specific value for w, but otherwise your code should work as long as w is a multiple of 8. You can use the expression `sizeof(int)<<3` to compute w.
- Forbidden
  - Conditionals (`if` or `?:`), loops, switch statements, function calls, and macro invocations.
  - Division, modulus, and multiplication.
  - Relative comparison operators (<, >, <=, and >=).
  - Casting, either explicit or implicit.
- Allowed operations
  - All bit-level and logic operations.
  - Left and right shifts, but only with shift amounts between 0 and $w - 1$.
  - Addition and subtraction.
  - Equality (==) and inequality (!=) tests. (Some of the problems do not allow these.)
  - Integer constants `INT_MIN` and `INT_MAX`.

Even with these rules, you should try to make your code readable by choosing descriptive variable names and using comments to describe the logic behind your solutions. As an example, the following code extracts the most significant byte from integer argument x:

```
/* Get most significant byte from x */
int get_msb(int x) {
    /* Shift by w-8 */
    int shift_val = (sizeof(int)-1)<<3;
    /* Arithmetic shift */
    int xright = x >> shift_val;
    /* Zero all but LSB */
    return xright & 0xFF;
}
```

2.61 ◆◆

Write C expressions that evaluate to 1 when the following conditions are true, and to 0 when they are false. Assume `x` is of type `int`.

A. Any bit of `x` equals 1.

B. Any bit of `x` equals 0.

C. Any bit in the least significant byte of `x` equals 1.

D. Any bit in the most significant byte of `x` equals 0.

Your code should follow the bit-level integer coding rules (page 120), with the additional restriction that you may not use equality (`==`) or inequality (`!=`) tests.

2.62 ◆◆◆

Write a function `int_shifts_are_arithmetic()` that yields 1 when run on a machine that uses arithmetic right shifts for `int`'s, and 0 otherwise. Your code should work on a machine with any word size. Test your code on several machines.

2.63 ◆◆◆

Fill in code for the following C functions. Function `srl` performs a logical right shift using an arithmetic right shift (given by value `xsra`), followed by other operations not including right shifts or division. Function `sra` performs an arithmetic right shift using a logical right shift (given by value `xsrl`), followed by other operations not including right shifts or division. You may use the computation `8*sizeof(int)` to determine w, the number of bits in data type `int`. The shift amount `k` can range from 0 to $w - 1$.

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

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

2.64 ◆

Write code to implement the following function:

```
/* Return 1 when any odd bit of x equals 1; 0 otherwise.
   Assume w=32. */
int any_odd_one(unsigned x);
```

Your function should follow the bit-level integer coding rules (page 120), except that you may assume that data type `int` has $w = 32$ bits.

2.65 ◆◆◆◆

Write code to implement the following function:

```
/* Return 1 when x contains an odd number of 1s; 0 otherwise.
   Assume w=32. */
int odd_ones(unsigned x);
```

Your function should follow the bit-level integer coding rules (page 120), except that you may assume that data type `int` has $w = 32$ bits.

Your code should contain a total of at most 12 arithmetic, bit-wise, and logical operations.

2.66 ◆◆◆

Write code to implement the following function:

```
/*
 * Generate mask indicating leftmost 1 in x.  Assume w=32.
 * For example 0xFF00 -> 0x8000, and 0x6600 --> 0x4000.
 * If x = 0, then return 0.
 */
int leftmost_one(unsigned x);
```

Your function should follow the bit-level integer coding rules (page 120), except that you may assume that data type `int` has $w = 32$ bits.

Your code should contain a total of at most 15 arithmetic, bit-wise, and logical operations.

Hint: First transform x into a bit vector of the form $[0 \cdots 011 \cdots 1]$.

2.67 ◆◆

You are given the task of writing a procedure `int_size_is_32()` that yields 1 when run on a machine for which an `int` is 32 bits, and yields 0 otherwise. You are not allowed to use the `sizeof` operator. Here is a first attempt:

```
1  /* The following code does not run properly on some machines */
2  int bad_int_size_is_32() {
3      /* Set most significant bit (msb) of 32-bit machine */
4      int set_msb = 1 << 31;
5      /* Shift past msb of 32-bit word */
6      int beyond_msb = 1 << 32;
7
8      /* set_msb is nonzero when word size >= 32
9         beyond_msb is zero when word size <= 32  */
10     return set_msb && !beyond_msb;
11 }
```

When compiled and run on a 32-bit SUN SPARC, however, this procedure returns 0. The following compiler message gives us an indication of the problem:

```
warning: left shift count >= width of type
```

A. In what way does our code fail to comply with the C standard?

B. Modify the code to run properly on any machine for which data type `int` is at least 32 bits.

C. Modify the code to run properly on any machine for which data type `int` is at least 16 bits.

2.68 ◆◆

Write code for a function with the following prototype:

```
/*
 * Mask with least signficant n bits set to 1
 * Examples: n = 6 --> 0x2F, n = 17 --> 0x1FFFF
 * Assume 1 <= n <= w
 */
int lower_one_mask(int n);
```

Your function should follow the bit-level integer coding rules (page 120). Be careful of the case $n = w$.

2.69 ◆◆◆

Write code for a function with the following prototype:

```
/*
 * Do rotating left shift.  Assume 0 <= n < w
 * Examples when x = 0x12345678 and w = 32:
 *    n=4 -> 0x23456781, n=20 -> 0x67812345
 */
unsigned rotate_left(unsigned x, int n);
```

Your function should follow the bit-level integer coding rules (page 120). Be careful of the case $n = 0$.

2.70 ◆◆

Write code for the function with the following prototype:

```
/*
 * Return 1 when x can be represented as an n-bit, 2's complement
 * number; 0 otherwise
 * Assume 1 <= n <= w
 */
int fits_bits(int x, int n);
```

Your function should follow the bit-level integer coding rules (page 120).

2.71 ◆

You just started working for a company that is implementing a set of procedures to operate on a data structure where 4 signed bytes are packed into a 32-bit unsigned. Bytes within the word are numbered from 0 (least significant) to 3 (most significant). You have been assigned the task of implementing a function for a machine using two's-complement arithmetic and arithmetic right shifts with the following prototype:

```
/* Declaration of data type where 4 bytes are packed
   into an unsigned */
typedef unsigned packed_t;

/* Extract byte from word.  Return as signed integer */
int xbyte(packed_t word, int bytenum);
```

That is, the function will extract the designated byte and sign extend it to be a 32-bit int.

Your predecessor (who was fired for incompetence) wrote the following code:

```
/* Failed attempt at xbyte */
int xbyte(packed_t word, int bytenum)
{
    return (word >> (bytenum << 3)) & 0xFF;
}
```

A. What is wrong with this code?

B. Give a correct implementation of the function that uses only left and right shifts, along with one subtraction.

2.72 ◆◆

You are given the task of writing a function that will copy an integer val into a buffer buf, but it should do so only if enough space is available in the buffer.

Here is the code you write:

```
/* Copy integer into buffer if space is available */
/* WARNING: The following code is buggy */
```

```
void copy_int(int val, void *buf, int maxbytes) {
    if (maxbytes-sizeof(val) >= 0)
        memcpy(buf, (void *) &val, sizeof(val));
}
```

This code makes use of the library function `memcpy`. Although its use is a bit artificial here, where we simply want to copy an `int`, it illustrates an approach commonly used to copy larger data structures.

You carefully test the code and discover that it *always* copies the value to the buffer, even when `maxbytes` is too small.

A. Explain why the conditional test in the code always succeeds. **Hint:** The `sizeof` operator returns a value of type `size_t`.

B. Show how you can rewrite the conditional test to make it work properly.

2.73 ◆◆

Write code for a function with the following prototype:

```
/* Addition that saturates to TMin or TMax */
int saturating_add(int x, int y);
```

Instead of overflowing the way normal two's-complement addition does, saturating addition returns *TMax* when there would be positive overflow, and *TMin* when there would be negative overflow. Saturating arithmetic is commonly used in programs that perform digital signal processing.

Your function should follow the bit-level integer coding rules (page 120).

2.74 ◆◆

Write a function with the following prototype:

```
/* Determine whether arguments can be subtracted without overflow */
int tsub_ok(int x, int y);
```

This function should return 1 if the computation `x – y` does not overflow.

2.75 ◆◆◆

Suppose we want to compute the complete $2w$-bit representation of $x \cdot y$, where both x and y are unsigned, on a machine for which data type `unsigned` is w bits. The low-order w bits of the product can be computed with the expression `x*y`, so we only require a procedure with prototype

```
unsigned int unsigned_high_prod(unsigned x, unsigned y);
```

that computes the high-order w bits of $x \cdot y$ for unsigned variables.

We have access to a library function with prototype

```
int signed_high_prod(int x, int y);
```

that computes the high-order w bits of $x \cdot y$ for the case where x and y are in two's-complement form. Write code calling this procedure to implement the function for unsigned arguments. Justify the correctness of your solution.

Hint: Look at the relationship between the signed product $x \cdot y$ and the unsigned product $x' \cdot y'$ in the derivation of Equation 2.18.

2.76 ◆◆

Suppose we are given the task of generating code to multiply integer variable `x` by various different constant factors K. To be efficient, we want to use only the operations `+`, `-`, and `<<`. For the following values of K, write C expressions to perform the multiplication using at most three operations per expression.

A. $K = 17$:
B. $K = -7$:
C. $K = 60$:
D. $K = -112$:

2.77 ◆◆

Write code for a function with the following prototype:

```
/* Divide by power of two.  Assume 0 <= k < w-1 */
int divide_power2(int x, int k);
```

The function should compute $x/2^k$ with correct rounding, and it should follow the bit-level integer coding rules (page 120).

2.78 ◆◆

Write code for a function `mul3div4` that, for integer argument `x`, computes `3*x/4`, but following the bit-level integer coding rules (page 120). Your code should replicate the fact that the computation `3*x` can cause overflow.

2.79 ◆◆◆

Write code for a function `threefourths` which, for integer argument `x`, computes the value of $\frac{3}{4}$`x`, rounded toward zero. It should not overflow. Your function should follow the bit-level integer coding rules (page 120).

2.80 ◆◆

Write C expressions to generate the bit patterns that follow, where a^k represents k repetitions of symbol a. Assume a w-bit data type. Your code may contain references to parameters `j` and `k`, representing the values of j and k, but not a parameter representing w.

A. $1^{w-k}0^k$
B. $0^{w-k-j}1^k0^j$

2.81 ◆

We are running programs on a machine where values of type `int` are 32 bits. They are represented in two's complement, and they are right shifted arithmetically. Values of type `unsigned` are also 32 bits.