

Programming

2's Complement Representation for Signed Integers

- [Definition](#)
- [Calculation of 2's Complement](#)
- [Addition](#)
- [Subtraction](#)
- [Multiplication](#)
- [Division](#)
- [Sign Extension](#)
- [Other Signed Representations](#)
- [Notes](#)

Definition

Property

Two's complement representation allows the use of binary arithmetic operations on signed integers, yielding the correct 2's complement results.

Positive Numbers

Positive 2's complement numbers are represented as the simple binary.

Negative Numbers

Negative 2's complement numbers are represented as the binary number that when added to a positive number of the same magnitude equals zero.

Integer		2's Complement
Signed	Unsigned	
5	5	0000 0101
4	4	0000 0100
3	3	0000 0011
2	2	0000 0010
1	1	0000 0001
0	0	0000 0000
-1	255	1111 1111
-2	254	1111 1110
-3	253	1111 1101
-4	252	1111 1100

-5	251	1111 1011
----	-----	-----------

Note: The most significant (leftmost) bit indicates the sign of the integer; therefore it is sometimes called the sign bit.

If the sign bit is zero,
then the number is greater than or equal to zero, or positive.

If the sign bit is one,
then the number is less than zero, or negative.

Calculation of 2's Complement

To calculate the 2's complement of an integer, invert the binary equivalent of the number by changing all of the ones to zeroes and all of the zeroes to ones (also called **1's complement**), and then add one.

For example,

$$0001\ 0001 \text{ (binary 17)} \Rightarrow 1110\ 1111 \text{ (two's complement -17)}$$

$$\text{NOT}(0001\ 0001) = 1110\ 1110 \text{ (Invert bits)}$$

$$1110\ 1110 + 0000\ 0001 = 1110\ 1111 \text{ (Add 1)}$$

2's Complement Addition

Two's complement addition follows the same rules as **binary addition**.

For example,

$$\begin{array}{r} 5 + (-3) = 2 \quad 0000\ 0101 = +5 \\ \quad \quad \quad + 1111\ 1101 = -3 \\ \hline \quad \quad \quad 0000\ 0010 = +2 \end{array}$$

2's Complement Subtraction

Two's complement subtraction is the **binary addition** of the minuend to the 2's complement of the subtrahend (adding a negative number is the same as subtracting a positive one).

For example,

$$7 - 12 = (-5) \quad 0000\ 0111 = +7$$

$$\begin{array}{r}
 + 1111\ 0100 = -12 \\
 \hline
 1111\ 1011 = -5
 \end{array}$$

2's Complement Multiplication

Two's complement multiplication follows the same rules as [binary multiplication](#).

For example,

$$\begin{array}{r}
 (-4) \times 4 = (-16) \quad 1111\ 1100 = -4 \\
 \times 0000\ 0100 = +4 \\
 \hline
 1111\ 0000 = -16
 \end{array}$$

2's Complement Division

Two's complement division is repeated **2's complement subtraction**. The 2's complement of the divisor is calculated, then added to the dividend. For the next subtraction cycle, the quotient replaces the dividend. This repeats until the quotient is too small for subtraction or is zero, then it becomes the remainder. The final answer is the total of subtraction cycles plus the remainder.

For example,

$$\begin{array}{r}
 7 \div 3 = 2 \text{ remainder } 1 \quad 0000\ 0111 = +7 \quad 0000\ 0100 = +4 \\
 + 1111\ 1101 = -3 \quad + 1111\ 1101 = -3 \\
 \hline
 0000\ 0100 = +4 \quad 0000\ 0001 = +1 \text{ (remainder)}
 \end{array}$$

Sign Extension

To extend a signed integer from 8 bits to 16 bits or from 16 bits to 32 bits, append additional bits on the left side of the number. Fill each extra bit with the value of the smaller number's most significant bit (the sign bit).

For example,

Signed Integer	8-bit Representation	16-bit Representation
-1	1111 1111	1111 1111 1111 1111
+1	0000 0001	0000 0000 0000 0001

Other Representations of Signed Integers

Sign-Magnitude Representation

Another method of representing negative numbers is sign-magnitude. Sign-magnitude representation also uses the most significant bit of the number to indicate the sign. A negative number is the 7-bit binary representation of the positive number with the most significant bit set to one. The drawbacks to using this method for arithmetic computation are that a different set of rules are required and that zero can have two representations (+0, 0000 0000 and -0, 1000 0000).

Offset Binary Representation

A third method for representing signed numbers is offset binary. Begin calculating a offset binary code by assigning half of the largest possible number as the zero value. A positive integer is the absolute value added to the zero number and a negative integer is subtracted. Offset binary is popular in A/D and D/A conversions, but it is still awkward for arithmetic computation.

For example,

Largest value for 8-bit integer = $2^8 = 256$

Offset binary zero value = $256 \div 2 = 128_{(\text{decimal})} = 1000\ 0000_{(\text{binary})}$

$1000\ 0000_{(\text{offset binary } 0)} + 0001\ 0110_{(\text{binary } 22)} = 1001\ 0110_{(\text{offset binary } +22)}$

$1000\ 0000_{(\text{offset binary } 0)} - 0000\ 0111_{(\text{binary } 7)} = 0111\ 1001_{(\text{offset binary } -7)}$

Signed Integer	Sign Magnitude	Offset Binary
+5	0000 0101	1000 0101
+4	0000 0100	1000 0100
+3	0000 0011	1000 0011
+2	0000 0010	1000 0010
+1	0000 0001	1000 0001
0	0000 0000 1000 0000	1000 0000
-1	1000 0001	0111 1111
-2	1000 0010	0111 1110
-3	1000 0011	0111 1101
-4	1000 0100	0111 1100
-5	1000 0101	0111 1011

Notes

Other Complements

1's Complement = $\text{NOT}(n) = 1111 \ 1111 - n$

9's Complement = $9999 \ 9999 - n$

10's Complement = $(9999 \ 9999 - n) + 1$

Binary Arithmetic

[Addition](#)

[Subtraction](#)

[Multiplication](#)

[Division](#)

[[Index](#) | [Technical Notes](#)]

[DISCLAIMER](#)

Page author: Dawn Rorvik (rorvikd@evergreen.edu)

Last modified: 05/20/2003

IEEE Standard 754 Floating Point Numbers

[Steve Hollasch](#) / Last update 2005-Feb-24

IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most Unix platforms. This article gives a brief overview of IEEE floating point and its representation. Discussion of arithmetic implementation may be found in the book mentioned at the bottom of this article.

What Are Floating Point Numbers?

There are several ways to represent real numbers on computers. Fixed point places a radix point somewhere in the middle of the digits, and is equivalent to using integers that represent portions of some unit. For example, one might represent 1/100ths of a unit; if you have four decimal digits, you could represent 10.82, or 00.01. Another approach is to use rationals, and represent every number as the ratio of two integers.

Floating-point representation - the most common solution - basically represents reals in scientific notation. Scientific notation represents numbers as a base number and an exponent. For example, 123.456 could be represented as 1.23456×10^2 . In hexadecimal, the number 123.abc might be represented as $1.23abc \times 16^2$.

Floating-point solves a number of representation problems. Fixed-point has a fixed window of representation, which limits it from representing very large or very small numbers. Also, fixed-point is prone to a loss of precision when two large numbers are divided.

Floating-point, on the other hand, employs a sort of "sliding window" of precision appropriate to the scale of the number. This allows it to represent numbers from 1,000,000,000,000 to 0.0000000000000001 with ease.

Storage Layout

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the fraction and an implicit leading digit (explained below). The exponent base (2) is implicit and need not be stored.

The following figure shows the layout for single (32-bit) and double (64-bit) precision floating-point values. The number of bits for each field are shown (bit ranges are in square brackets):

	Sign	Exponent	Fraction	Bias
Single Precision	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision	1 [63]	11 [62-52]	52 [51-00]	1023

The Sign Bit

The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

The Exponent

The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of $(200-127)$, or 73. For reasons discussed later, exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers.

For double precision, the exponent field is 11 bits, and has a bias of 1023.

The Mantissa

The mantissa, also known as the significand, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits.

To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways. For example, the number five can be represented as any of these:

$$\begin{aligned} 5.00 &\times 10^0 \\ 0.05 &\times 10^2 \\ 5000 &\times 10^{-3} \end{aligned}$$

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. In normalized form, five is represented as 5.0×10^0 .

A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

Putting it All Together

So, to sum up:

1. The sign bit is 0 for positive, 1 for negative.
2. The exponent's base is two.
3. The exponent field contains 127 plus the true exponent for single-precision, or

1023 plus the true exponent for double precision.

- The first bit of the mantissa is typically assumed to be 1, f , where f is the field of fraction bits.

Ranges of Floating-Point Numbers

Let's consider single-precision floats for a second. Note that we're taking essentially a 32-bit number and re-jiggering the fields to cover a much broader range. Something has to give, and it's precision. For example, regular 32-bit integers, with all precision centered around zero, can precisely store integers with 32-bits of resolution. Single-precision floating-point, on the other hand, is unable to match this resolution with its 24 bits. It does, however, approximate this value by effectively truncating from the lower end. For example:

```

11110000 11001100 10101010 00001111 // 32-bit integer
= +1.1110000 11001100 10101010 x 231 // Single-Precision Float
= 11110000 11001100 10101010 00000000 // Corresponding Value

```

This approximates the 32-bit value, but doesn't yield an exact representation. On the other hand, besides the ability to represent fractional components (which integers lack completely), the floating-point value can represent numbers around 2^{127} , compared to 32-bit integers maximum value around 2^{32} .

The range of positive floating point numbers can be split into normalized numbers (which preserve the full precision of the mantissa), and *denormalized* numbers (discussed later) which use only a portion of the fractions's precision.

	Denormalized	Normalized	Approximate Decimal
Single Precision	$\pm 2^{-149}$ to $(1-2^{-23}) \times 2^{-126}$	$\pm 2^{-126}$ to $(2-2^{-23}) \times 2^{127}$	$\pm \sim 10^{-44.85}$ to $\sim 10^{38.53}$
Double Precision	$\pm 2^{-1074}$ to $(1-2^{-52}) \times 2^{-1022}$	$\pm 2^{-1022}$ to $(2-2^{-52}) \times 2^{1023}$	$\pm \sim 10^{-323.3}$ to $\sim 10^{308.3}$

Since the sign of floating point numbers is given by a special leading bit, the range for negative numbers is given by the negation of the above values.

There are five distinct numerical ranges that single-precision floating-point numbers are **not** able to represent:

- Negative numbers less than $-(2-2^{-23}) \times 2^{127}$ (negative overflow)
- Negative numbers greater than -2^{-149} (negative underflow)
- Zero
- Positive numbers less than 2^{-149} (positive underflow)

5. Positive numbers greater than $(2-2^{-23}) \times 2^{127}$ (positive overflow)

Overflow means that values have grown too large for the representation, much in the same way that you can overflow integers. Underflow is a less serious problem because it just denotes a loss of precision, which is guaranteed to be closely approximated by zero.

Here's a table of the effective range (excluding infinite values) of IEEE floating-point numbers:

	Binary	Decimal
Single	$\pm (2-2^{-23}) \times 2^{127}$	$\sim \pm 10^{38.53}$
Double	$\pm (2-2^{-52}) \times 2^{1023}$	$\sim \pm 10^{308.25}$

Note that the extreme values occur (regardless of sign) when the exponent is at the maximum value for finite numbers (2^{127} for single-precision, 2^{1023} for double), and the mantissa is filled with 1s (including the normalizing 1 bit).

Special Values

IEEE reserves exponent field values of all 0s and all 1s to denote special values in the floating-point scheme.

Zero

As mentioned above, zero is not directly representable in the straight format, due to the assumption of a leading 1 (we'd need to specify a true zero mantissa to yield a value of zero). Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and $+0$ are distinct values, though they both compare as equal.

Denormalized

If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a denormalized number, which does *not* have an assumed leading 1 before the binary point. Thus, this represents a number $(-1)^s \times 0.f \times 2^{-126}$, where s is the sign bit and f is the fraction. For double precision, denormalized numbers are of the form $(-1)^s \times 0.f \times 2^{-1022}$. From this you can interpret zero as a special type of denormalized number.

Infinity

The values $+\infty$ and $-\infty$ are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. *Operations with infinite values are well defined in IEEE floating point.*

Not A Number

The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signalling NaN).

A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined.

An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage.

Semantically, QNaN's denote *indeterminate* operations, while SNaN's denote *invalid* operations.

Special Operations

Operations on special numbers are well-defined by IEEE. In the simplest case, any operation with a NaN yields a NaN result. Other operations are as follows:

Operation	Result
$n \div \pm\text{Infinity}$	0
$\pm\text{Infinity} \times \pm\text{Infinity}$	$\pm\text{Infinity}$
$\pm\text{nonzero} \div 0$	$\pm\text{Infinity}$
$\text{Infinity} + \text{Infinity}$	Infinity
$\pm 0 \div \pm 0$	NaN
$\text{Infinity} - \text{Infinity}$	NaN
$\pm\text{Infinity} \div \pm\text{Infinity}$	NaN
$\pm\text{Infinity} \times 0$	NaN

Summary

To sum up, the following are the corresponding values for a given representation:

Float Values ($b = \text{bias}$)

Sign	Exponent (e)	Fraction (f)	Value
0	00..00	00..00	+0
0	00..00	00..01	Positive Denormalized Real

0	00..00 : 11..11	0.f × 2 ^(-b+1)	
0	00..01 : 11..10	Positive Normalized Real 1.f × 2 ^(e-b)	
0	11..11	+Infinity	
0	11..11	00..01 : 01..11	SNaN
0	11..11	10..00 : 11..11	QNaN
1	00..00	00..00	-0
1	00..00	00..01 : 11..11	Negative Denormalized Real -0.f × 2 ^(-b+1)
1	00..01 : 11..10	XX..XX	Negative Normalized Real -1.f × 2 ^(e-b)
1	11..11	00..00	-Infinity
1	11..11	00..01 : 01..11	SNaN
1	11..11	10..00 : 11..11	QNaN

References

A lot of this stuff was observed from small programs I wrote to go back and forth between hex and floating point (*printf*-style), and to examine the results of various operations. The bulk of this material, however, was lifted from Stallings' book.

1. *Computer Organization and Architecture*, William Stallings, pp. 222-234 Macmillan Publishing Company, ISBN 0-02-415480-6
2. IEEE Computer Society (1985), *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Std 754-1985.
3. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, (a PDF document downloaded from intel.com.)

See Also

- [IEEE Standards Site](#)
- *Comparing floating point numbers*, Bruce Dawson, <http://www.cygnum-software.com/papers/comparingfloats/comparingfloats.htm>. This is an excellent article on the traps, pitfalls and solutions for comparing floating point numbers. Hint — epsilon comparison is usually the *wrong* solution.
- *x86 Processors and Infinity*, Bruce Dawson, <http://www.cygnum-software.com/papers/x86andinfinity.html>. This is another good article covering performance issues with IEEE specials on X86 architecture.

© 2001-2005 [Steve Hollasch](#)

Integer (computer science)

From Wikipedia, the free encyclopedia

In computer science, the term **integer** is used to refer to any data type which can represent some subset of the mathematical integers. These are also known as **integral data types**.

Contents

- 1 Value and representation
- 2 Common integral data types
- 3 Data type names
- 4 Pointers
- 5 Bytes and octets
- 6 Words

Value and representation

The *value* of a datum with an integral type is the mathematical integer that it corresponds to. The *representation* of this datum is the way the value is stored in the computer's memory. Integral types may be *unsigned* (capable of representing only non-negative integers) or *signed* (capable of representing negative integers as well).

The most common representation of a positive integer is a string of bits, using the binary numeral system. The order of the bits varies; see Endianness. The *width* or *precision* of an integral type is the number of bits in its representation. An integral type with n bits can encode 2^n numbers; for example an unsigned type typically represents the non-negative values 0 through 2^n-1 .

There are three different ways to represent negative numbers in a binary numeral system. The most common is two's complement, which allows a signed integral type with n bits to represent numbers from $-2^{(n-1)}$ through $2^{(n-1)}-1$. Two's complement arithmetic is convenient because there is a perfect one-to-one correspondence between representations and values, and because addition and subtraction do not need to distinguish between signed and unsigned types. The other possibilities are sign-magnitude and ones' complement. See Signed number representations for details.

Another, rather different, representation for integers is binary-coded decimal, which is still commonly used in mainframe financial applications and in databases.

Common integral data types

Bits	Name	Range	Uses
8	byte, octet	<i>Signed</i> : −128 to +127 <i>Unsigned</i> : 0 to +255	ASCII characters, C int8_t, Java byte

16	halfword, word	<i>Signed</i> : −32,768 to +32,767 <i>Unsigned</i> : 0 to +65,535	UCS-2 characters, C int16_t, Java char, Java short
32	word, doubleword, longword	<i>Signed</i> : −2,147,483,648 to +2,147,483,647 <i>Unsigned</i> : 0 to +4,294,967,295	UCS-4 characters, Truecolor with alpha, C int32_t, Java int
64	doubleword, longword, quadword	<i>Signed</i> : −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 <i>Unsigned</i> : 0 to +18,446,744,073,709,551,615	C int64_t, Java long
128		<i>Signed</i> : −170,141,183,460,469,231,731,687,303,715,884,105,728 to +170,141,183,460,469,231,731,687,303,715,884,105,727 <i>Unsigned</i> : 0 to +340,282,366,920,938,463,463,374,607,431,768,211,455	C only available as non-standard compiler-specific extension
<i>n</i>	<i>n</i> -bit integer	<i>Signed</i> : -2^{n-1} to $2^{n-1} - 1$ <i>Unsigned</i> : 0 to $2^n - 1$	

Different CPUs support different integral data types. Typically, hardware will support both signed and unsigned types but only a small, fixed set of widths.

The table above lists integral type widths that are supported in hardware by common processors. High level programming languages provide more possibilities. It is common to have a ‘double width’ integral type that has twice as many bits as the biggest hardware-supported type. Many languages also have *bit-field* types (a specified number of bits, usually constrained to be less than the maximum hardware-supported width) and *range* types (which can represent only the integers in a specified range).

Some languages, such as Lisp, REXX and Haskell, support *arbitrary precision* integers (also known as *infinite precision integers* or *bignums*). Other languages which do not support this concept as a top-level construct may have libraries available to represent very large numbers using arrays of smaller variables, such as Java's BigInteger class or Perl's "bigint" package. These use as much of the computer's memory as is necessary to store the numbers; however, a computer has only a finite amount of storage, so they too can only represent a finite subset of the mathematical integers. These schemes support very large numbers, for example one kilobyte of memory could be used to store numbers up to about 2560 digits long.

A Boolean or Flag type is a type which can represent only two values: 0 and 1, usually identified with *false* and *true* respectively. This type can be stored in memory using a single bit, but is often given a full byte for convenience of addressing and speed of access.

A four-bit quantity is known as a *nibble* (when eating, being smaller than a *bite*) or *nybble* (being a pun on the form of the word *byte*). One nibble corresponds to one digit in hexadecimal and holds one digit or a sign code in binary-coded decimal.

Data type names

Bits	Signed	Java	C#	SQL92	vbScript	C
8	Yes	byte	sbyte			int8_t, signed char
16	Yes	short	short, Int16	smallint, int2	int	int16_t, int (on C89), short(on C99)
				integer, int		

32	Yes	int	int, Int32	integer, int, int4	long	int32_t, long(on C89), int(on C99)
64	Yes	long	long, Int64	bigint, int8		int64_t, long long (on C99)
8	No		byte	tinyint, int1	byte	uint8_t, unsigned char
16	No	char	ushort, UInt16			uint16_t,
32	No		uint, UInt32			uint32_t, unsigned int(on C89), unsigned short(on C99)
64	No		ulong, UInt64			uint64_t, unsigned long long (on C99)

Note: C++ has no compiler-independent integer types with fixed bit widths.

Pointers

A pointer is often, but not always, represented by an unsigned integer of specified width. This is often, but not always, the widest integer that the hardware supports directly. The value of this integer is the *memory address* of whatever the pointer points to.

Bytes and octets

The term *byte* initially meant ‘the smallest addressable unit of memory’. In the past, 5-, 6-, 7-, 8-, and 9-bit bytes have all been used. There have also been computers that could address individual bits (‘bit-addressed machine’), or that could only address 16- or 32-bit quantities (‘word-addressed machine’). The term *byte* was usually not used at all in connection with bit- and word-addressed machines.

The term *octet* always refers to an 8-bit quantity. It is mostly used in the field of computer networking, where computers with different byte widths might have to communicate.

In modern usage *byte* almost invariably means eight bits, since all other sizes have fallen into disuse; thus *byte* has come to be synonymous with *octet*.

Words

The term *word* is used for a small group of bits which are handled simultaneously by processors of a particular architecture. The size of a word is thus CPU-specific. Many different word sizes have been used, including 6-, 8-, 12-, 16-, 18-, 24-, 32-, 36-, 39-, 48-, 60-, and 64-bit. Since it is architectural, the size of a *word* is usually set by the first CPU in a family, rather than the characteristics of a later compatible CPU. The meanings of terms derived from *word*, such as *longword*, *doubleword*, *quadword*, and *halfword*, also vary with the CPU and OS.

As of 2006, 32-bit word sizes are most common among general-purpose computers, with 64-bit machines used mostly for large installations. Embedded processors with 8- and 16-bit word size are still common. The 36-bit word length was common in the early days of computers, but word sizes that are not a multiple of 8 have vanished along with non-8-bit bytes.

Retrieved from "http://en.wikipedia.org/wiki/Integer_%28computer_science%29"

Categories: Data types | Computer arithmetic

- This page was last modified 02:30, 26 March 2007.
- All text is available under the terms of the GNU Free Documentation License. (See **Copyrights** for details.)
Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a US-registered 501(c)(3) tax-deductible nonprofit charity.

Programming

2's Complement Representation for Signed Integers

- [Definition](#)
- [Calculation of 2's Complement](#)
- [Addition](#)
- [Subtraction](#)
- [Multiplication](#)
- [Division](#)
- [Sign Extension](#)
- [Other Signed Representations](#)
- [Notes](#)

Definition

Property

Two's complement representation allows the use of binary arithmetic operations on signed integers, yielding the correct 2's complement results.

Positive Numbers

Positive 2's complement numbers are represented as the simple binary.

Negative Numbers

Negative 2's complement numbers are represented as the binary number that when added to a positive number of the same magnitude equals zero.

Integer		2's Complement
Signed	Unsigned	
5	5	0000 0101
4	4	0000 0100
3	3	0000 0011
2	2	0000 0010
1	1	0000 0001
0	0	0000 0000
-1	255	1111 1111
-2	254	1111 1110
-3	253	1111 1101
-4	252	1111 1100

-5	251	1111 1011
----	-----	-----------

Note: The most significant (leftmost) bit indicates the sign of the integer; therefore it is sometimes called the sign bit.

If the sign bit is zero,
then the number is greater than or equal to zero, or positive.

If the sign bit is one,
then the number is less than zero, or negative.

Calculation of 2's Complement

To calculate the 2's complement of an integer, invert the binary equivalent of the number by changing all of the ones to zeroes and all of the zeroes to ones (also called **1's complement**), and then add one.

For example,

$$0001\ 0001 \text{ (binary 17)} \Rightarrow 1110\ 1111 \text{ (two's complement -17)}$$

$$\text{NOT}(0001\ 0001) = 1110\ 1110 \text{ (Invert bits)}$$

$$1110\ 1110 + 0000\ 0001 = 1110\ 1111 \text{ (Add 1)}$$

2's Complement Addition

Two's complement addition follows the same rules as **binary addition**.

For example,

$$\begin{array}{r} 5 + (-3) = 2 \quad 0000\ 0101 = +5 \\ \quad \quad \quad + 1111\ 1101 = -3 \\ \hline \quad \quad \quad 0000\ 0010 = +2 \end{array}$$

2's Complement Subtraction

Two's complement subtraction is the **binary addition** of the minuend to the 2's complement of the subtrahend (adding a negative number is the same as subtracting a positive one).

For example,

$$7 - 12 = (-5) \quad 0000\ 0111 = +7$$

$$\begin{array}{r}
 + 1111\ 0100 = -12 \\
 \hline
 1111\ 1011 = -5
 \end{array}$$

2's Complement Multiplication

Two's complement multiplication follows the same rules as [binary multiplication](#).

For example,

$$\begin{array}{r}
 (-4) \times 4 = (-16) \quad 1111\ 1100 = -4 \\
 \times 0000\ 0100 = +4 \\
 \hline
 1111\ 0000 = -16
 \end{array}$$

2's Complement Division

Two's complement division is repeated **2's complement subtraction**. The 2's complement of the divisor is calculated, then added to the dividend. For the next subtraction cycle, the quotient replaces the dividend. This repeats until the quotient is too small for subtraction or is zero, then it becomes the remainder. The final answer is the total of subtraction cycles plus the remainder.

For example,

$$\begin{array}{r}
 7 \div 3 = 2 \text{ remainder } 1 \quad 0000\ 0111 = +7 \quad 0000\ 0100 = +4 \\
 + 1111\ 1101 = -3 \quad + 1111\ 1101 = -3 \\
 \hline
 0000\ 0100 = +4 \quad 0000\ 0001 = +1 \text{ (remainder)}
 \end{array}$$

Sign Extension

To extend a signed integer from 8 bits to 16 bits or from 16 bits to 32 bits, append additional bits on the left side of the number. Fill each extra bit with the value of the smaller number's most significant bit (the sign bit).

For example,

Signed Integer	8-bit Representation	16-bit Representation
-1	1111 1111	1111 1111 1111 1111
+1	0000 0001	0000 0000 0000 0001

Other Representations of Signed Integers

Sign-Magnitude Representation

Another method of representing negative numbers is sign-magnitude. Sign-magnitude representation also uses the most significant bit of the number to indicate the sign. A negative number is the 7-bit binary representation of the positive number with the most significant bit set to one. The drawbacks to using this method for arithmetic computation are that a different set of rules are required and that zero can have two representations (+0, 0000 0000 and -0, 1000 0000).

Offset Binary Representation

A third method for representing signed numbers is offset binary. Begin calculating a offset binary code by assigning half of the largest possible number as the zero value. A positive integer is the absolute value added to the zero number and a negative integer is subtracted. Offset binary is popular in A/D and D/A conversions, but it is still awkward for arithmetic computation.

For example,

Largest value for 8-bit integer = $2^8 = 256$

Offset binary zero value = $256 \div 2 = 128_{(\text{decimal})} = 1000\ 0000_{(\text{binary})}$

$1000\ 0000_{(\text{offset binary } 0)} + 0001\ 0110_{(\text{binary } 22)} = 1001\ 0110_{(\text{offset binary } +22)}$

$1000\ 0000_{(\text{offset binary } 0)} - 0000\ 0111_{(\text{binary } 7)} = 0111\ 1001_{(\text{offset binary } -7)}$

Signed Integer	Sign Magnitude	Offset Binary
+5	0000 0101	1000 0101
+4	0000 0100	1000 0100
+3	0000 0011	1000 0011
+2	0000 0010	1000 0010
+1	0000 0001	1000 0001
0	0000 0000 1000 0000	1000 0000
-1	1000 0001	0111 1111
-2	1000 0010	0111 1110
-3	1000 0011	0111 1101
-4	1000 0100	0111 1100
-5	1000 0101	0111 1011

IEEE Standard 754 Floating Point Numbers

[Steve Hollasch](#) / Last update 2005-Feb-24

IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most Unix platforms. This article gives a brief overview of IEEE floating point and its representation. Discussion of arithmetic implementation may be found in the book mentioned at the bottom of this article.

What Are Floating Point Numbers?

There are several ways to represent real numbers on computers. Fixed point places a radix point somewhere in the middle of the digits, and is equivalent to using integers that represent portions of some unit. For example, one might represent 1/100ths of a unit; if you have four decimal digits, you could represent 10.82, or 00.01. Another approach is to use rationals, and represent every number as the ratio of two integers.

Floating-point representation - the most common solution - basically represents reals in scientific notation. Scientific notation represents numbers as a base number and an exponent. For example, 123.456 could be represented as 1.23456×10^2 . In hexadecimal, the number 123.abc might be represented as $1.23abc \times 16^2$.

Floating-point solves a number of representation problems. Fixed-point has a fixed window of representation, which limits it from representing very large or very small numbers. Also, fixed-point is prone to a loss of precision when two large numbers are divided.

Floating-point, on the other hand, employs a sort of "sliding window" of precision appropriate to the scale of the number. This allows it to represent numbers from 1,000,000,000,000 to 0.0000000000000001 with ease.

Storage Layout

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the fraction and an implicit leading digit (explained below). The exponent base (2) is implicit and need not be stored.

The following figure shows the layout for single (32-bit) and double (64-bit) precision floating-point values. The number of bits for each field are shown (bit ranges are in square brackets):

	Sign	Exponent	Fraction	Bias
Single Precision	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision	1 [63]	11 [62-52]	52 [51-00]	1023

The Sign Bit

The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

The Exponent

The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of $(200-127)$, or 73. For reasons discussed later, exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers.

For double precision, the exponent field is 11 bits, and has a bias of 1023.

The Mantissa

The mantissa, also known as the significand, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits.

To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways. For example, the number five can be represented as any of these:

$$\begin{aligned} 5.00 &\times 10^0 \\ 0.05 &\times 10^2 \\ 5000 &\times 10^{-3} \end{aligned}$$

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. In normalized form, five is represented as 5.0×10^0 .

A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

Putting it All Together

So, to sum up:

1. The sign bit is 0 for positive, 1 for negative.
2. The exponent's base is two.
3. The exponent field contains 127 plus the true exponent for single-precision, or

1023 plus the true exponent for double precision.

- The first bit of the mantissa is typically assumed to be 1, f , where f is the field of fraction bits.

Ranges of Floating-Point Numbers

Let's consider single-precision floats for a second. Note that we're taking essentially a 32-bit number and re-jiggering the fields to cover a much broader range. Something has to give, and it's precision. For example, regular 32-bit integers, with all precision centered around zero, can precisely store integers with 32-bits of resolution. Single-precision floating-point, on the other hand, is unable to match this resolution with its 24 bits. It does, however, approximate this value by effectively truncating from the lower end. For example:

```

11110000 11001100 10101010 00001111 // 32-bit integer
= +1.1110000 11001100 10101010 x 231 // Single-Precision Float
= 11110000 11001100 10101010 00000000 // Corresponding Value

```

This approximates the 32-bit value, but doesn't yield an exact representation. On the other hand, besides the ability to represent fractional components (which integers lack completely), the floating-point value can represent numbers around 2^{127} , compared to 32-bit integers maximum value around 2^{32} .

The range of positive floating point numbers can be split into normalized numbers (which preserve the full precision of the mantissa), and *denormalized* numbers (discussed later) which use only a portion of the fractions's precision.

	Denormalized	Normalized	Approximate Decimal
Single Precision	$\pm 2^{-149}$ to $(1-2^{-23}) \times 2^{-126}$	$\pm 2^{-126}$ to $(2-2^{-23}) \times 2^{127}$	$\pm \sim 10^{-44.85}$ to $\sim 10^{38.53}$
Double Precision	$\pm 2^{-1074}$ to $(1-2^{-52}) \times 2^{-1022}$	$\pm 2^{-1022}$ to $(2-2^{-52}) \times 2^{1023}$	$\pm \sim 10^{-323.3}$ to $\sim 10^{308.3}$

Since the sign of floating point numbers is given by a special leading bit, the range for negative numbers is given by the negation of the above values.

There are five distinct numerical ranges that single-precision floating-point numbers are **not** able to represent:

- Negative numbers less than $-(2-2^{-23}) \times 2^{127}$ (negative overflow)
- Negative numbers greater than -2^{-149} (negative underflow)
- Zero
- Positive numbers less than 2^{-149} (positive underflow)

5. Positive numbers greater than $(2-2^{-23}) \times 2^{127}$ (positive overflow)

Overflow means that values have grown too large for the representation, much in the same way that you can overflow integers. Underflow is a less serious problem because it just denotes a loss of precision, which is guaranteed to be closely approximated by zero.

Here's a table of the effective range (excluding infinite values) of IEEE floating-point numbers:

	Binary	Decimal
Single	$\pm (2-2^{-23}) \times 2^{127}$	$\sim \pm 10^{38.53}$
Double	$\pm (2-2^{-52}) \times 2^{1023}$	$\sim \pm 10^{308.25}$

Note that the extreme values occur (regardless of sign) when the exponent is at the maximum value for finite numbers (2^{127} for single-precision, 2^{1023} for double), and the mantissa is filled with 1s (including the normalizing 1 bit).

Special Values

IEEE reserves exponent field values of all 0s and all 1s to denote special values in the floating-point scheme.

Zero

As mentioned above, zero is not directly representable in the straight format, due to the assumption of a leading 1 (we'd need to specify a true zero mantissa to yield a value of zero). Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

Denormalized

If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a denormalized number, which does *not* have an assumed leading 1 before the binary point. Thus, this represents a number $(-1)^s \times 0.f \times 2^{-126}$, where s is the sign bit and f is the fraction. For double precision, denormalized numbers are of the form $(-1)^s \times 0.f \times 2^{-1022}$. From this you can interpret zero as a special type of denormalized number.

Infinity

The values +infinity and -infinity are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. *Operations with infinite values are well defined in IEEE floating point.*

Not A Number

The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signalling NaN).

A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined.

An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage.

Semantically, QNaN's denote *indeterminate* operations, while SNaN's denote *invalid* operations.

Special Operations

Operations on special numbers are well-defined by IEEE. In the simplest case, any operation with a NaN yields a NaN result. Other operations are as follows:

Operation	Result
$n \div \pm\text{Infinity}$	0
$\pm\text{Infinity} \times \pm\text{Infinity}$	$\pm\text{Infinity}$
$\pm\text{nonzero} \div 0$	$\pm\text{Infinity}$
$\text{Infinity} + \text{Infinity}$	Infinity
$\pm 0 \div \pm 0$	NaN
$\text{Infinity} - \text{Infinity}$	NaN
$\pm\text{Infinity} \div \pm\text{Infinity}$	NaN
$\pm\text{Infinity} \times 0$	NaN

Summary

To sum up, the following are the corresponding values for a given representation:

Float Values ($b = \text{bias}$)

Sign	Exponent (e)	Fraction (f)	Value
0	00..00	00..00	+0
0	00..00	00..01	Positive Denormalized Real

0	00..00 : 11..11	0.f × 2 ^(-b+1)	
0	00..01 : 11..10	Positive Normalized Real 1.f × 2 ^(e-b)	
0	11..11	+Infinity	
0	11..11	00..01 : 01..11	SNaN
0	11..11	10..00 : 11..11	QNaN
1	00..00	00..00	-0
1	00..00	00..01 : 11..11	Negative Denormalized Real -0.f × 2 ^(-b+1)
1	00..01 : 11..10	XX..XX	Negative Normalized Real -1.f × 2 ^(e-b)
1	11..11	00..00	-Infinity
1	11..11	00..01 : 01..11	SNaN
1	11..11	10..00 : 11..11	QNaN

References

A lot of this stuff was observed from small programs I wrote to go back and forth between hex and floating point (*printf*-style), and to examine the results of various operations. The bulk of this material, however, was lifted from Stallings' book.

1. *Computer Organization and Architecture*, William Stallings, pp. 222-234 Macmillan Publishing Company, ISBN 0-02-415480-6
2. IEEE Computer Society (1985), *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Std 754-1985.
3. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, (a PDF document downloaded from intel.com.)

[Art and Comics](#) [Critical Mass](#) [Emacs](#) [Free Stuff](#) [Games](#) [Gnu/Linux](#) [Home](#) [Learn Japanese](#) [Links](#) [Montreal](#) [Music](#) [News](#) [Some Essays](#) [Statistics](#)

Node: Integer variables, Next: [Declarations](#), Previous: [Variables and declarations](#), Up: [Variables and declarations](#)

Integer variables

C has five kinds of integer. An integer is a whole number (a number without a fractional part). In C, there are a limited number of integers possible; how many depends on the type of integer. In arithmetic, you can have as large a number as you like, but C integer types always have a largest (and smallest) possible number.

- `char`: A single byte, usually one ASCII character. (See the section on the `char` type below.)
- `short`: A short integer (16 bits long on most GNU systems). Also called `short int`. Rarely used.
- `int`: A standard integer (32 bits long on most GNU systems).
- `long`: A long integer (32 bits long on most GNU systems, the same as `int`). Also called `long int`.
- `long long`: A long long integer (64 bits long on most GNU systems). Also called `long long int`.

64-bit operating systems are now appearing in which long integers are 64 bits long. With GCC, long integers are normally 32 bits long and long long integers are 64 bits long, but it varies with the computer hardware and implementation of GCC, so check your system's documentation.

These integer types differ in the size of the integer they can hold and the amount of storage required for them. The sizes of these variables depend on the hardware and operating system of the computer. On a typical 32-bit GNU system, the sizes of the integer types are as follows.

Type	Bits	Possible Values
<code>char</code>	8	-127 to 127
<code>unsigned char</code>	8	0 to 255
<code>short</code>	16	-32,767 to 32,767
<code>unsigned short</code>	16	0 to 65,535
<code>int</code>	32	-2,147,483,647 to 2,147,483,647
<code>unsigned int</code>	32	0 to 4,294,967,295
<code>long</code>	32	-2,147,483,647 to 2,147,483,647
<code>unsigned long</code>	32	0 to 4,294,967,295
<code>long long</code>	64	-9,223,372,036,854,775,807 to 9,223,372,036,854,775,807

`unsigned long long` 64 0 to 18,446,744,073,709,551,615

On some computers, the lowest possible value may be 1 less than shown here; for example, the smallest possible `short` may be -32,768 rather than -32,767.

The word `unsigned`, when placed in front of integer types, means that only positive or zero values can be used in that variable (i.e. it cannot have a minus sign). The advantage is that larger numbers can then be stored in the same variable. The ANSI standard also allows the word `signed` to be placed before an integer, to indicate the opposite of `unsigned`.

- [The char type:](#)
- [Floating point variables:](#)

The IEEE standard for floating point arithmetic

The IEEE (Institute of Electrical and Electronics Engineers) has produced a standard for floating point arithmetic. This standard specifies how single precision (32 bit) and double precision (64 bit) floating point numbers are to be represented, as well as how arithmetic should be carried out on them.

Because many of our users may have occasion to transfer unformatted or "binary" data between an IEEE machine and the Cray or the VAX/VMS, it is worth noting the details of this format for comparison with the Cray and VAX representations. The differences in the formats also affect the accuracy of floating point computations.

Summary:

Single Precision

The IEEE single precision floating point standard representation requires a 32 bit word, which may be represented as numbered from 0 to 31, left to right. The first bit is the sign bit, S, the next eight bits are the exponent bits, 'E', and the final 23 bits are the fraction 'F':

```
S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFF
0 1      8 9                      31
```

The **value V represented** by the word may be determined as follows:

- If E=255 and F is nonzero, then V=NaN ("Not a number")
- If E=255 and F is zero and S is 1, then V=-Infinity
- If E=255 and F is zero and S is 0, then V=Infinity
- If $0 < E < 255$ then $V = (-1)^S * 2^{E-127} * (1.F)$ where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
- If E=0 and F is nonzero, then $V = (-1)^S * 2^{E-126} * (0.F)$ These are "unnormalized" values.
- If E=0 and F is zero and S is 1, then V=-0
- If E=0 and F is zero and S is 0, then V=0

In particular,

```
0 00000000 000000000000000000000000 = 0
1 00000000 000000000000000000000000 = -0

0 11111111 000000000000000000000000 = Infinity
1 11111111 000000000000000000000000 = -Infinity

0 11111111 000001000000000000000000 = NaN
1 11111111 001000100010010101010101 = NaN

0 10000000 000000000000000000000000 = +1 * 2**(128-127) * 1.0 = 2
0 10000001 101000000000000000000000 = +1 * 2**(129-127) * 1.101 = 6.5
1 10000001 101000000000000000000000 = -1 * 2**(129-127) * 1.101 = -6.5

0 00000001 000000000000000000000000 = +1 * 2**(1-127) * 1.0 = 2**(-126)
0 00000000 100000000000000000000000 = +1 * 2**(-126) * 0.1 = 2**(-127)
0 00000000 000000000000000000000001 = +1 * 2**(-126) *
0.00000000000000000000000000000001 =
2**(-149) (Smallest positive value)
```

Double Precision

The IEEE double precision floating point standard representation requires a 64 bit word, which may be represented as numbered from 0 to 63, left to right. The first bit is the sign bit, S, the next eleven bits are the exponent bits, 'E', and the final 52 bits are the fraction 'F':

```
S EEEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFF
0 1      11 12                      63
```

The value V represented by the word may be determined as follows:

- If E=2047 and F is nonzero, then V=NaN ("Not a number")
- If E=2047 and F is zero and S is 1, then V=-Infinity
- If E=2047 and F is zero and S is 0, then V=Infinity
- If $0 < E < 2047$ then $V = (-1)^S * 2^{E-1023} * (1.F)$ where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
- If E=0 and F is nonzero, then $V = (-1)^S * 2^{E-1022} * (0.F)$ These are "unnormalized" values.

- If $E=0$ and F is zero and S is 1, then $V=-0$
- If $E=0$ and F is zero and S is 0, then $V=0$

Reference:

*ANSI/IEEE Standard 754-1985,
Standard for Binary Floating Point Arithmetic*

See also:

- Mathematical and statistical software packages installed on PSC machines.
- Distributed Computing
- Utilities software packages and libraries installed on PSC machines.

© Pittsburgh Supercomputing Center, 300 S. Craig Street, Pittsburgh, PA 15213 Phone: 412.268.4960 Fax: 412.268.5832

Unsigned and Signed Integers

An integer is a number with no fractional part; it can be positive, negative or zero. In ordinary usage, one uses a minus sign to designate a negative integer. However, a computer can only store information in [bits](#), which can only have the values zero or one. We might expect, therefore, that the storage of negative integers in a computer might require some special technique. It is for that reason that we began this section with a discussion of unsigned integers.

As you might imagine, an **unsigned integer** is either positive or zero. Given our discussion in the previous sections about [binary numbers](#), it might seem that there is little more to say about unsigned integers. In fact, there is essentially only one thing, and that is one of the most important things that you will learn in this text. Consider a single digit decimal number: in a single decimal digit, you can write a number between 0 and 9. In two decimal digits, you can write a number between 0 and 99, and so on. Since nine is equivalent to $10^1 - 1$, 99 is equivalent to $10^2 - 1$, etc., in n decimal digits, you can write a number between 0 and $10^n - 1$. Analogously, in the binary number system,

an unsigned integer containing n bits can have a value between 0 and $2^n - 1$ (which is 2^n different values).

This fact is one of the most important and useful things to know about computers. When a computer program is written, the programmer, either explicitly or implicitly, must decide how many bits are used to store any given quantity. Once the decision is made to use n bits to store it, the program has an inherent limitation: that quantity can only have a value between 0 and $2^n - 1$. You will meet these limitations in one form or another in every piece of hardware and software that you will learn about during your career:

- the BIOS (Basic Input Output Software) in older PCs uses 10 bits to store the cylinder number on the hard drive where your operating system begins; therefore those PCs cannot boot an operating system from a cylinder greater than $2^{10} - 1$, or 1023.
- a FAT12 file system (used on Windows diskettes), which allocates file space in units called "clusters", uses 12 bits to store cluster numbers; therefore there can be no more than $2^{12} - 1$ or 4,095 clusters in such a file system.
- a UNIX system keeps track of the processes (programs) it runs using a PID (Process IDentifier); for typical memory sizes, the PID is 16 bits long and so after $2^{16} - 1$ or 65,535 processes, the PIDs must start over at the lowest number not currently in use.

These are just a few examples of this basic principle that you will meet in your future studies.

Most modern computers store memory in units of **8 bits, called a "byte"** (also called an "octet"). Arithmetic in such computers can be done in bytes, but is more often done in larger units called "**(short) integers**" (16 bits), "**long integers**" (32 bits) or "**double integers**" (64 bits). Short integers can be used to store numbers between 0 and $2^{16} - 1$, or 65,535. Long integers can be used to store numbers between 0 and $2^{32} - 1$, or 4,294,967,295. and double integers can be used to store numbers between 0 and $2^{64} - 1$, or 18,446,744,073,709,551,615. (Check these!)

When a computer performs an unsigned integer arithmetic operation, there are three possible problems which can occur:

1. if the result is too large to fit into the number of bits assigned to it, an "**overflow**" is said to have occurred. For example if the result of an operation using 16 bit integers is larger than 65,535, an overflow results.
1. in the division of two integers, if the result is not itself an integer, a "**truncation**" is said to have occurred: 10 divided by 3 is truncated to 3, and the extra 1/3 is lost. This is not a problem, of course, if the programmer's intention was to ignore the remainder!
1. any division by zero is an error, since division by zero is not possible in the context of arithmetic.

Signed Integers

Signed integers are stored in a computer using **2's complement**. As you recall, when computing the 2's complement of a number it was necessary to know how many bits were to be used in the final result; **leading zeroes** were appended to the **most significant digit** in order to make the number the appropriate length. Since the process of computing the 2's complement involves first computing the **1's complement**, these leading zeros become leading ones, and the left most bit of a negative number is therefore always 1. **In computers, the left most bit of a signed integer is called the "sign bit".**

Consider an 8 bit signed integer: let us begin with 0000000_2 and start counting by repeatedly adding 1:

- When you get to **127**, the integer has a value of **0111111_2** ; this is easy to see because you know now that a 7 bit integer can contain a value between 0 and $2^7 - 1$, or 127. What happens when we add 1?
- If the integer were **unsigned**, the next value would be **1000000_2** , or **128** (2^7). But since this is a signed integer, 1000000_2 is a negative value: the sign bit is 1!
- Since this is the case, we must ask the question: what is the decimal value corresponding to the signed integer 1000000_2 ? To answer this question, we must take the 2's complement of that value, by first taking the 1's complement and then adding one.
- The 1's complement is 0111111_2 , or decimal 127. Since we must now add 1 to that, our conclusion is that the signed integer **1000000_2 must be equivalent to decimal -128!**

Odd as this may seem, it is in fact the only consistent way to interpret 2's complement signed integers. Let us continue now to "count" by adding 1 to 1000000_2 :

- $1000000_2 + 0000001_2$ is 1000001_2 .
- To find the decimal equivalent of 1000001_2 , we again take the 2's complement: the 1's complement is 0111110_2 and adding 1 we get 0111111_2 (127) so 1000001_2 is equivalent to -127.
- We see then that once we have accepted the fact that 1000000_2 is decimal -128, counting by adding one works as we would expect.
- Note that the most negative number which we can store in an 8 bit signed integer is -128, which

is -2^{8-1} , and that the largest positive signed integer we can store in an 8 bit signed integer is 127, which is $2^{8-1} - 1$.

- The number of integers between -128 and + 127 (inclusive) is 256, which is 2^8 ; this is the same number of values which an unsigned 8 bit integer can contain (from 0 to 255).
- Eventually we will count all the way up to $1\ 1\ 1\ 1\ 1\ 1\ 1\ 1_2$. The 1's complement of this number is obviously 0, so $1\ 1\ 1\ 1\ 1\ 1\ 1\ 1_2$ must be the decimal equivalent of -1.

Using our deliberations on 8 bit signed integers as a guide, we come to the following observations about signed integer arithmetic in general:

- **if a signed integer has n bits, it can contain a number between -2^{n-1} and $(2^{n-1} - 1)$.**
- **since both signed and unsigned integers of n bits in length can represent 2^n different values, there is no inherent way to distinguish signed integers from unsigned integers simply by looking at them; the software designer is responsible for using them correctly.**
- no matter what the length, if a signed integer has a binary value of all 1's, it is equal to decimal -1.

You should verify that a signed short integer can hold decimal values from -32,768 to +32,767, a signed long integer can contain values from -2,147,483,648 to +2,147,483,647 and a signed double integer can represent decimal values from -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.

There is an interesting consequence to the fact that in 2's complement arithmetic, one expects to throw away the final carry: in unsigned arithmetic a carry out of the most significant digit means that there has been an overflow, but in signed arithmetic an overflow is not so easy to detect. In fact, signed arithmetic overflows are detected by checking the consistency of the signs of the operands and the final answer. A signed overflow has occurred in an addition or subtraction if:

- the sum of two positive numbers is negative;
- the sum of two negative numbers is positive;
- subtracting a positive number from a negative one yields a positive result; or
- subtracting a negative number from a positive one yields a negative result.

Integer arithmetic on computers is often called "**fixed point**" arithmetic and the integers themselves are often called fixed point numbers. Real numbers on computers (which may have fractional parts) are often called "floating point" numbers, and they are the subject of the [next section](#).

Go to: [Title Page](#) [Table of Contents](#) [Index](#)

©2002, Kenneth R. Koehler. All Rights Reserved. This document may be freely reproduced provided that this copyright notice is included.

Please send comments or suggestions to [the author](#).