

NUMBER AND CHARACTER REPRESENTATIONS

J. Wunderlich PhD

ASCII is a 7-bit code, but most computers manipulate an 8-bit quantity as a single unit called a *byte*. Therefore, ASCII characters most often are stored one per byte, with the most significant bit set to 0. The extra bit is sometimes used for specific purposes, depending on the application. For example, some printers recognize an additional 128 8-bit characters, with the most significant bit set to 1. These characters

CHARACTERS

TABLE 1-4

American Standard Code for Information Interchange (ASCII)

B ₇ B ₆ B ₅ B ₄ B ₃ B ₂ B ₁	B ₇ B ₆ B ₅							
	000	001	010	011	100	101	110	111
0000	NULL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	\$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(8	H	X	h	x
1001	HT	EM)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[k	{
1100	FF	FS	,	<	L	\	l	
1101	CR	GS	-	=	M]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Control Characters:

NULL	NULL	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End of transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete



BASE 10	BINARY (BASE 2)	HEX (BASE 16)
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10

NUMBERS

J. HUPPERMAN
PhD

← LEARN ABOUT
IN OTHER COURSES
(CS121, 122, 332, ETC.)

TABLE 1-3
Signed Binary Numbers

Decimal	Signed-2's complement	Signed-1's complement	Signed- magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	—	1111	1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1001	1110
-7	1001	1000	1111
-8	1000	—	—



CONVERT BINARY 111
TO DECIMAL

$$\begin{aligned}
 & (111)_2 \xleftarrow{\text{BASE 2}} \\
 & = (1 * 2^2) + (1 * 2^1) + (1 * 2^0) \\
 & = (1 * 4) + (1 * 2) + (1 * 1) \\
 & = 4 + 2 + 1 \\
 & = \boxed{(7)_{10}} \xleftarrow{\text{BASE 10}}
 \end{aligned}$$

CONVERT $(9)_{10}$ TO BINARY

→ LOOK AT POSSIBLE PARTS, AND "PATCH" TOGETHER PARTS

MSB \square_n ...

LSB ← LEAST SIGNIFICANT BIT

7	6	5	4	3	2	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

← IF BIT = 1

1 0 0 1 ← SO $(9)_{10} = (1001)_2$

FIND BIGGEST # THAT FITS FIRST THEN GO FROM LEFT TO RIGHT

$\boxed{(1001)_2}$



" ,
1

Assignment: Homework #10

Course: EGR/CS 230 "MicrocomputerArchitecture"

Instructor: J. Wunderlich

Wednesday: **May 4 at 12:30pm**

Late penalty: -25% per class period

Revised: 4/13/11, 4/18/11

Show all the bits for IEEE BFP (Binary Floating Point) in single precision for:

- a) $\frac{1}{4}$
- b) $\frac{1}{2}$
- c) **1/1024**
- d) $\frac{3}{4}$
- e) **-3/4**
- f) **1.25**
- g) **The biggest possible number**
- h) **The smallest possible number**
- i) 1
- j) 0

" ,
1

Show all the bits for IEEE BFP (Binary Floating Point) in double precision for:

- a) **One trillion**
- b) **One Terabyte (i.e., just Tera of addressable bytes)**

In addition to class handout and notes written on board,
READ THIS:

L



IEEE Standard 754 Floating Point Numbers

Steve Hollasch / Last update 2005-Feb-24
Available on-line at <http://steve.hollasch.net/cgindex/coding/ieeefloat.html>

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.
4. 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:

1. Negative numbers less than $-(2-2^{-23}) \times 2^{127}$ (*negative overflow*)
2. Negative numbers greater than -2^{-149} (*negative underflow*)
3. Zero
4. 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 ⋮ 11..11	Positive Denormalized Real $0.f \times 2^{(-b+1)}$
0	00..01 ⋮ 11..10	XX..XX	Positive Normalized Real $1.f \times 2^{(e-b)}$
0	11..11	00..00	+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 $-0.f \times 2^{(-b+1)}$
1	00..01 ⋮ 11..10	XX..XX	Negative Normalized Real $-1.f \times 2^{(e-b)}$
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](http://www.ieee.org)
- *Comparing floating point numbers*, Bruce Dawson <http://www.cygnus-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.cygnus-software.com/papers/x86andinfinity.html>. This is another good article covering performance issues with IEEE specials on X86 architecture.

© 2001-2005 Steve Hollasch



Conversion Examples to help with Fractional Part of IEEE Floating Point J.Wunderlich PhD

(adapted from: <http://sandbox.mc.edu/~bennet/cs110/98/dfof.html>)

(I added **bolds** and underlines for fractional parts, and *italics* and small fonts for exponents)

Convert -1313.**3125** to IEEE 32-bit floating point format.

The integral part is $1313_{10} = 10100100001_2$. The fractional:

0. <u>3125</u>	$\times 2 =$	0.625	0	Generate 0 and continue.
0.625	$\times 2 =$	1.25	1	Generate 1 and continue with the rest.
0.25	$\times 2 =$	0.5	0	Generate 0 and continue.
0.5	$\times 2 =$	1.0	1	Generate 1 and nothing remains.

So $1313.\textbf{3125}_{10} = 10100100001.\textbf{0101}_2$.

Normalize: $10100100001.\textbf{0101}_2 = 1.0100100001\textbf{0101}_2 \times 2^{10}$.

Mantissa is 0100100001**0101**000000000, *exponent is $10 + 127 = 137 = 10001001_2$* , sign bit is 1.

So -1313.**3125** is $1\textbf{000100101001000010101000000000} = \text{c4a42a00}_{16}$

Convert 0.**1015625** to IEEE 32-bit floating point format.

0. <u>1015625</u>	$\times 2 =$	0.203125	0	Generate 0 and continue.
0.203125	$\times 2 =$	0.40625	0	Generate 0 and continue.
0.40625	$\times 2 =$	0.8125	0	Generate 0 and continue.
0.8125	$\times 2 =$	1.625	1	Generate 1 and continue with the rest.
0.625	$\times 2 =$	1.25	1	Generate 1 and continue with the rest.
0.25	$\times 2 =$	0.5	0	Generate 0 and continue.
0.5	$\times 2 =$	1.0	1	Generate 1 and nothing remains.

So $0.\textbf{1015625}_{10} = 0.\textbf{0001101}_2$.

Normalize: $0.\textbf{0001101}_2 = \textbf{1.101}_2 \times 2^{-4}$.

Mantissa is **101**00000000000000000000, *exponent is $-4 + 127 = 123 = 01111011_2$* , sign bit is 0.

So 0.1015625 is $0\textbf{01111011101000000000000000000000} = \text{3dd00000}_{16}$

Convert 39887.**5625** to IEEE 32-bit floating point format.

The integral part is $39887_{10} = 1001101111001111_2$. The fractional:

0. <u>5625</u>	$\times 2 =$	1.125	1	Generate 1 and continue with the rest.
0.125	$\times 2 =$	0.25	0	Generate 0 and continue.
0.25	$\times 2 =$	0.5	0	Generate 0 and continue.
0.5	$\times 2 =$	1.0	1	Generate 1 and nothing remains.

So $39887.\textbf{5625}_{10} = 1001101111001111.\textbf{1001}_2$.

Normalize: $1001101111001111.\textbf{1001}_2 = 1.001101111001111\textbf{1001}_2 \times 2^{15}$.

Mantissa is 001101111001111**1001**0000, *exponent is $15 + 127 = 142 = 10001110_2$* , sign bit is 0.

So 39887.**5625** is $0\textbf{1000111000110111100111110010000} = \text{471bcf90}_{16}$



SOLUTION

a) $1/4 = 1/2^2 = (1.0)_2 \times 2^{-2}$ $\text{expo. } -2 = e - 127$
 $125 = e$

0	01111101	00000000...0000
	8	23

b) $1/2 = 1/2^1 = (1.0)_2 \times 2^{-1}$ $-1 = e - 127$
 $126 = e$

0	01111110	000000...000000
		23

c) $1/1024 = 1/2^{10} = (1.0)_2 \times 2^{-10}$ $-10 = e - 127$
 $117 = e$

0	01110101	0000...0000
		23

d) $3/4 = (11)_2/2^2 = (11)_2 \times 2^{-2} = (1.1)_2 \times 2^{-1}$ $-1 = e - 127$
 $126 = e$

0	01111101	1000...000
		22

e) $-3/4 = -(1.1)_2 \times 2^{-1}$ $-1 = e - 127$
 $126 = e$

1	10111110	10000...0000
		22

f) $1.25 = 5/4 = (101)_2/2^2 = 101 \times 2^{-2} = 1.01 \times 2^0$ $e = 127$

0	10111111	010000...0000
		21

g) biggest # $e = 255 \Rightarrow \text{actual} = 255 - 127 = 128$

0	11111111	0000...0000
		23

OK, BUT
 SHOULD PUT
 ALL 1s
 HERE

OK,
 MOST
 NEGATIVE

h) smallest # neg. $e = 255 \Rightarrow \text{actual} = 128$

1	11111111	0000...0000
		23

AND
 MOST
 "PRECISE"

0	00000000	0...1
---	----------	-------



