

Floating Point and the IEEE Standard

Ivor Page¹

16.1 Floating Point and the IEEE Standard

Floating Point is the representation most widely used for fractional quantities and values that have wide range. Consider, for example, the need for a system that can represent the mass of an electron, $9.109 \times 10^{-31}kg$ and the age of the universe, 10^{19} seconds. The *range* required is extremely large, yet each quantity need only be stored with relatively small *precision*, say between 8 and 16 decimal digits. Floating point systems separate the range and the precision of a value into two separate values.

There are four components of a floating point number, the sign, the significand, the base, and the exponent. The base is implied and is 2 in most systems including the IEEE standard.

$$x = \pm s \times b^e = \pm \text{significand} \times \text{base}^{\text{exponent}}$$

The significand is also known as the *mantissa*. The sign is a single bit and applies to the significand. The significand is usually a normalized binary fraction. Systems prior to the IEEE standard typically used $0.5 \leq \text{significand} < 1.0$, but the standard requires $1.0 \leq \text{significand} < 2.0$ for normalized numbers. In the standard representation, the sign bit and significand comprise a sign-magnitude number. The sign bit is zero for a positive significand.

A biased exponent system is used. For single-length IEEE numbers the bias is 127. It is 1023 for double-length numbers. A number such as the mass of an electron, 1.1369×2^{-100} , has 0 for its sign bit, a significand of $1.001000110001_2 \dots$, and an exponent field of 27 in the single precision form.

The value $1.75 \times 2^{+100}$ would have a significand of $+1.11000 \dots$ and an exponent field of 227. The exponent fields for these quantities would contain $1023 - 100$ and $1023 + 100$ respectively if they were represented in the double-length format.

¹University of Texas at Dallas

In the standard, a normalized number always has a significand with a leading 1. This leading 1 is not stored in the representation, so 1.75 is represented as $0.11_2 \times 2^0$. The missing 1 is referred to as the *hidden 1*.

The standard allows for denormalized numbers, very small values that are beyond the lower end of the range of normalized values. In the representation of *denorms* the exponent value is 0 and there is no hidden bit in the significand. For example, $0.00111_2 \times 2^{-126}$ is a denorm that represents $1.75_{10} \times 2^{-129}$.

The single precision IEEE standard calls for 1 bit for the sign, 8 bits for the exponent, and 23 bits for the fraction. The five forms of representation for single precision IEEE standard values are as follows:

$$x = \pm s \times 2^e$$

1. $x = NaN$ if $e = 255 \wedge f \neq 0$
2. $x = \pm\infty$ if $e = 255 \wedge f = 0$
3. $x = \pm 2^{e-127} \times 1.f$ if $0 < e < 255$
4. $x = \pm 2^{-126} \times 0.f$ if $e = 0 \wedge f \neq 0$
5. $x = \pm 0$ if $e = 0 \wedge f = 0$

The first representation is for *Not a Number*. A divide by zero or other arithmetic error will produce *NaN* as its result. Any arithmetic operation with a *NaN* as an operand produces *NaN* as its result. *NaNs* are useful for presetting variables in programs so that, if programmers forget to initialize a variable, any operations using that variable will lead to *NaN* results and the error will easily be found.

The second representation is for plus or minus infinity. Operations on either of these values produce the expected results.

The third form is for normalized numbers with hidden bits and exponents in the range -126 to 127.

The fourth form is for denormalized numbers.

The fifth form is for plus or minus zero. Notice that the fraction and the exponent fields contain all 0s. The sign may be 0 or 1.

The double precision IEEE standard calls for 1 bit for the sign, 11 bits for the exponent, and 52 bits for the fraction. The corresponding five forms of representation for double precision IEEE standard values are as follows:

$$x = \pm s \times 2^e$$

1. $x = NaN$ if $e = 2047 \wedge f \neq 0$
2. $x = \pm\infty$ if $e = 2047 \wedge f = 0$
3. $x = \pm 2^{e-1023} \times 1.f$ if $0 < e < 2047$
4. $x = \pm 2^{-1022} \times 0.f$ if $e = 0 \wedge f \neq 0$
5. $x = \pm 0$ if $e = 0 \wedge f = 0$

16.2 Rounding Methods

There are four rounding methods provided in the standard:

- Round to nearest even (the default).
- Round toward zero (inward).
- Round toward $+\infty$ (upward).
- Round toward $-\infty$ (downward).

Let's consider rounding methods in general.

16.2.1 Truncation, Round Toward Zero

Since the standard uses sign magnitude, truncation rounds towards zero - the result's magnitude is always smaller. Figure 1 shows the transfer diagram from x to $chop(x)$. The large dots are the values chosen for the special cases when x is an exact integer on this scale.

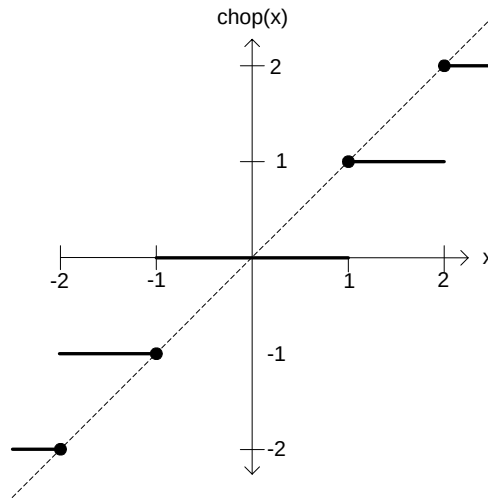


Figure 1: Truncation, Round Toward Zero

The IEEE standard requires that the rounded result of some arithmetic operation should be the same as the result if infinite precision were used, followed by rounding of the result using the prescribed rounding method. The units along the x and y axes represent *ulps*. The unrounded value is represented by any point along the x axis.

Rounding towards zero is biased, as can be shown by the following example from the text. Say we are rounding a $l + 2$ bit value into l binary places (by discarding the two least significant bits). The four settings of those bits result in the following rounding:

$$\begin{aligned}
 x_{-1}x_{-2} &= 00 \text{ Round down } error = 0 \\
 x_{-1}x_{-2} &= 01 \text{ Round down } error = 1/4 \\
 x_{-1}x_{-2} &= 10 \text{ Round down } error = 1/2
 \end{aligned}$$

$$x_{-1}x_{-2} = 11 \text{ Round down } error = 3/4$$

If these four bit settings occur with equal probability the average error is $1/2$ *ulp*.

16.2.2 Downward, Round Toward $-\infty$

Figure 2 shows the transfer diagram from x to $chop(x)$. This system is also biased. Consider the same example as above.

$$\begin{aligned} x_{-1}x_{-2} &= 00 \text{ Round down } error = 0 \\ x_{-1}x_{-2} &= 01 \text{ Round down } error = 1/4 \\ x_{-1}x_{-2} &= 10 \text{ Round down } error = 1/2 \\ x_{-1}x_{-2} &= 11 \text{ Round down } error = 3/4 \end{aligned}$$

If these four bit settings occur with equal probability the average error is $1/2$ *ulp*.

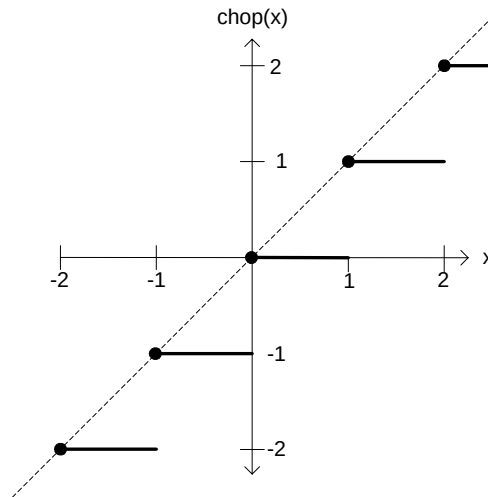


Figure 2: Downward, Round Toward $-\infty$

16.2.3 Round to Nearest

Figure 3 shows the transfer diagram from x to $\text{chop}(x)$. This system is also biased. Consider the same example as above.

$x_{-1}x_{-2}$	=	00	Round down	$\text{error} = 0$
$x_{-1}x_{-2}$	=	01	Round down	$\text{error} = -1/4$
$x_{-1}x_{-2}$	=	10	Round up	$\text{error} = 1/2$
$x_{-1}x_{-2}$	=	11	Round up	$\text{error} = 1/4$

If these four bit settings occur with equal probability the average error is $1/4$ *ulp*.

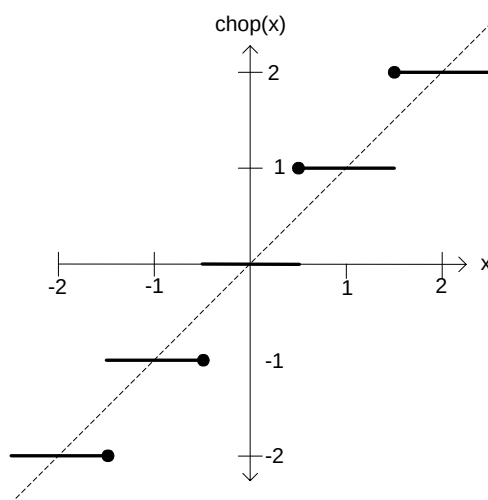


Figure 3: Round to Nearest

16.2.4 Round to Nearest Even

We expect truncation to cause biased results, but rounding to the nearest should imply no bias. For this reason, *Round to Nearest Even* is preferred and is the default rounding method in the IEEE standard.

Figure 4 shows the transfer diagram. This system is unbiased if we average the error over values of x (within the l digits) that are equally likely to be odd or even.

$$\begin{aligned}
 x_{-1}x_{-2} &= 00 \text{ Round down } error = 0 \\
 x_{-1}x_{-2} &= 01 \text{ Round down } error = -1/4 \\
 x_{-1}x_{-2} &= 10 \text{ Round up } error = +1/4 \\
 x_{-1}x_{-2} &= 11 \text{ Round up } error = 1/2
 \end{aligned}$$

If these four bit settings occur with equal probability the average error is 0 *ulp*.

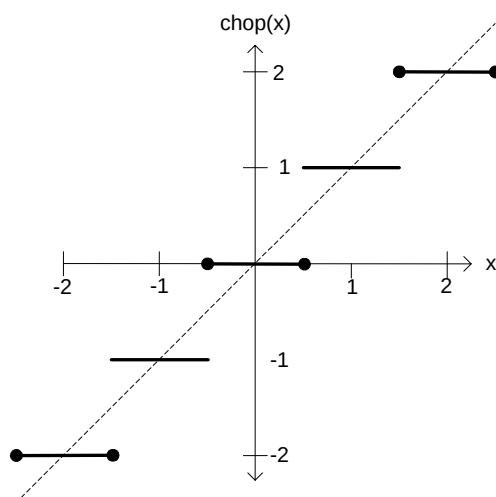


Figure 4: Round to Nearest Even

16.2.5 Von Neumann Rounding, or Jamming

In the previous two varieties of *Round to Nearest Something* a rounding operation could cause carry-propagation over all the bits of the fraction. Von Neumann rounding was designed to prevent this. The method truncates the value and then forces the lsb to 1. The worst case error is double that of

other methods, but it is an unbiased system and it avoids carry propagation. Figure 5 shows the transfer diagram.

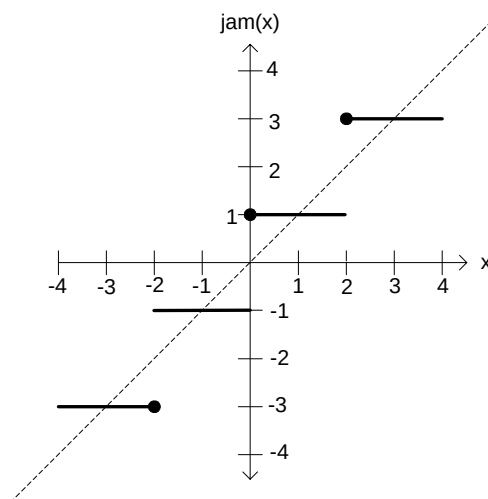


Figure 5: von Neumann Rounding, Jamming

16.3 Floating Point Arithmetic

In addition and subtraction, the steps to be performed are:

1. extract the exponents and the significands of the two arguments,
2. subtract the exponents giving difference p
3. right shift the significand of the value with the smaller exponent p places, inserting its hidden 1, and add p to the corresponding exponent.
4. add or subtract the singificands
5. renormalize the result.

One of the two significands may need complementing, depending on the signs of the operands and the operation required.

If the operation is the addition of two values with the same sign or the subtraction of two values with different signs, the resulting significand may require a single bit left-shift (with a corresponding decrement of the exponent) in order to normalize the result.

If the operation is the addition of values with different signs or the subtraction of values with the same sign, *cancellation* of one or more most significant bits may take place. Consider, for example, the operation (given in decimals for simplicity) $1.112233445566 - 1.11223344543$. Ten digits are cancelled. If the two operands were accurate to 12 places, the result is only accurate to 2 places. This phenomenon is known as *catastrophic cancellation*.

There are many factors left out of this simple sequence. For example, the special cases of demorms, NaN, ± 0 , and $\pm \infty$ must be caught and each one dealt with specially.

During the right-shift operation to align the exponents, some of the least significant bits that are shifted out of the operand's significand field may be saved. They will be used in post-normalization if a left shift of the result takes place.

The IEEE standard requires that three bits be kept beyond the lsb in the arithmetic unit to enable the rounding schemes discussed above. They are the:

- G:Guard Bit
- R:Round Bit
- S:Sticky Bit

Recall that proper rounding requires that the result should be the same as if infinite precision were used in the arithmetic, followed by application of the prescribed rounding method. These three bits satisfy that requirement.

If one of the significands is right-shifted p places, where p is less than the length of the significand, then the most significant bit shifted out will be captured in the guard bit. If there is such a right-shift, with $p > 1$, then no more than a single left-shift is necessary to post-normalize the result. To see this, assume the sum is $1.00000 \times 2^{10} - 1.11111 \times 2^8$. This is the largest possible significand subtracted from the smallest, where the former must be right shifted two places before the subtraction. After pre-shifting to align

the exponents, we have $(1.00000 - 0.01111|1) \times 2^{10} = 0.10001|1 \times 2^{10}$. After post normalization, the result is 1.00011×2^{10} . The guard bit has “guarded against” an error.

If we repeat the calculation with a 1-bit pre-shift, cancellation occurs, but the result is correct, according to the requirements of the IEEE specification: $1.00000 \times 2^{10} - 1.11111 \times 2^9 = 1.00000 \times 2^{10} - 0.11111|1 \times 2^{10} = 0.00001|1 \times 2^{10} = 1.10000 \times 2^5$

Cancellation may occur when $p < 2$.

A single bit right-shift is also occasionally necessary after an addition or a subtraction.

The round bit is needed to determine whether to add or subtract 1 from the significand, according to the rounding method in use, after post-normalization.

$$\begin{aligned} R=0 & \rightarrow \text{discard part} < ulp/2 \\ R=1 & \rightarrow \text{discard part} \geq ulp/2 \end{aligned}$$

Finally, some rounding schemes need to know if the discard part exactly equals $ulp/2$. The sticky bit provides this information. Its value is the logical OR of all the values that pass through it during pre-shifting.

Round to nearest is achieved as follows: Here are the conditions before and after post-shifting:

Before Post Shifting (z)	z_{-l+1}	z_{-l}	G	R	S
1-bit normalizing right-shift	z_{-l+2}	z_{-l+1}	z_{-l}	G	$R \wedge S$
1-bit normalizing left-shift	z_{-l}	G	R	S	0
After normalization	z_{-l+1}	lsb	G	R	S

After normalization,

if ($G=0$ OR $lsb=R=S=0$) do nothing.
 else add $ulp = 2^{-l}$ to f .

Other rounding algorithms use these bits differently.

Overflow may occur with addition and subtraction operations (when the resulting exponent is larger than the maximum possible in the representation).

Underflow may also occur, resulting in a denorm.

Addition and subtraction with denorms must also be provided for.

Multiplication and division are conceptually simpler operations. The steps for multiplication are:

1. extract the exponents and significands of the two operands.
2. add the exponents and subtract the bias.
3. multiply the significands.
4. normalize the result by a possible right-shift 1 place and the addition of 1 to the exponent.

Multiplication of two l bit significands would normally produce a $2l$ bit result, which is rounded to l bits. The result can, however be kept to $l + 1$ bits by using the round bit and the sticky bit during the multiplication.

Overflow can occur when the sum of the two exponents exceeds the exponent field. Underflow can also occur.

Division is similar in structure:

1. extract the exponents and significands of the two operands.
2. subtract the exponents and add the bias.
3. divide the significands.
4. normalize the result by a possible left-shift 1 place and the subtraction of 1 from the exponent.

During division, $l + 2$ bits of quotient must be generated to act as the guard bit and the round bit.

Overflow and underflow can also occur with division.