# An Overview of Floating Point Numbers

*Harry H. Porter III*

HHPorter3@gmail.com

*19 September 2022*

This document introduces and describes floating point numbers. It starts with the basics, and is appropriate for someone first encountering floating point. This document then proceeds into greater and greater detail.

Floating point numbers are complex and the IEEE specification is lengthy. This document summarizes and simplifies the topic.

# Table of Contents

# Chapter 1: Floating Point Numbers

## The Basic Idea

Integer values are represented in a computer with binary numbers. For example, in many programming languages a value of type "int" will be represented in exactly 32 bits. With only a fixed number of bits, there is a limit to how many different values can be represented.

For example, the range of values that can be represented with 32-bit integers is:

-2,147,483,648 … +2,147,483,647

To represent fractional values, humans often use scientific notation, such as:

$6.02214076 \times 10^{23}$

Floating point representation is an attempt to represent numbers like this in a fixed and small number of bits. Typically, each floating point number will be represented in 32 bits, although some programs will use 64 bits for each number.

With integers, there is a limitation: very large and very negative values simply cannot be represented.

With floating point, we have these limitations:

- The range of the exponents is limited.
- The amount of precision is limited.

For example:

| | |
|---|---|
| $18.0 \times 10^2$ | ← *can be represented* |
| $18.00000000000000001 \times 10^2$ | ← *can NOT be represented* |

With scientific notation, humans often express the amount precision or accuracy of a value, showing the accuracy by the number of digits. Measurements that are more accurate have more digits, and less accurate values are rounded to values with fewer digits.

For example, these two values are exactly equal, but they suggest different confidences in their accuracy:

$7.25 \times 10^6$
$7.2500 \times 10^6$

With floating point, these two numbers are represented identically. After all, they are really the same number.

- Each floating point value is nothing more than a value. There is no information about the accuracy of that value.

Since there is a finite number of bits available for each number, there are only a finite number of values that can be represented. As such:

- Many values cannot be represented.

Instead, we must make-do with numbers that are nearby and about the same as desired value. The value that the bits of a floating point value represent will be the closest approximation to the true, correct value. At least we hope so!

Floating point representation is fundamentally a binary representation, not a decimal representation. As a consequence:

- Many simple decimal values cannot be represented.

For example, the following commonly used number can be represented simply and exactly in decimal, but cannot be represented exactly in floating point:

0.3

The closest we can come with floating point is:

0.300000011920928955078125

Because of the limitations of floating point, almost every operation (such as +, -, ×, and ÷) will introduce errors. And the more operations that are performed, the greater the inaccuracy of the final result.

- Arithmetic operations are usually inexact and introduce errors.
- Errors tend to get larger as more operations are performed.

Because of these factors, you must learn about floating point point if you wish to write reliable, correct code:

- If accurate numerical results are required, the programmer must understand floating point.

## The IEEE 754-2008 Standard

The IEEE 754-2008 standard describes how floating point numbers are to be represented and how floating point operations are to be executed by computers.

The standard is complicated and detailed. This document is meant to be an introduction and is not an exhaustive description. Most modern processor Instruction Set Architectures (ISAs) implement the IEEE 754-2008 specification, but the specification has options and some parts are not fully implemented on most computers.

## Single and Double Data Types

The specification defines two main data types:

| | |
|---|---|
| **Single Precision** | 32-bit “float” values |
| **Double Precision** | 64-bit “double” values |

These are the important data types that you need to be aware of. You should probably ignore the other data types.

These two data types are available in most common languages, such as C, C++, C#, Objective C, and Java[1]:

| Data Type in Programming Language | Implementation |
|---|---|
| “float” | single precision (32 bits) |
| “double” | double precision (64 bits) |

Some computers implement only **single** precision in hardware; other computers implement both **single** and **double**. Some computers do not implement either.

For simpler processors, the implementation of floating point occurs purely in software. That is, floating point is “emulated”. This is generally transparent and the programmer will not be aware of whether the processor is implementing the floating point operations in hardware (which is much faster) or in software (which is much slower).

In both single and double representation, the idea is to represent a real rational number in a way similar to scientific notation. For example, the following number is given in scientific notation:

$6.022 \times 10^{23}$ (an approximation to Avogadro’s constant)

With only 32 bits for single (or 64 bits for double), there are limits to the amount of precision and the size of the exponents. The available bits are used as follows:

[1] Python has a type called “float” which is implemented with IEEE double (64 bit) representation. KPL has a “double” type, but not a “single” type.

Number of bits used for…

| | Single | Double |
|---|---|---|
| Sign | 1 | 1 |
| Exponent | 8 | 11 |
| Value | 23 | 52 |
| **Total** | **32** | **64** |

## Fixed Point Numbers

Furthermore, with floating point a numerical value is represented in binary (not decimal) and this introduces some subtleties when going back and forth between the internal bit patterns and decimal representations which humans can read.

Every positive integer can be represented with a finite number of digits and a finite number of bits. For example, here is the same number, represented both ways. Of course, this number requires a few more characters in binary, but the represented value is equal.

| | |
|---|---|
| 2,468 | (decimal) |
| 100110100100 | (binary) |

We commonly represent rational numbers in decimal using a "decimal point", as in:

123.456

We can also represent rational numbers in binary using a "binary point", as in:

101.0101

With decimal numbers, the position of each digit is important and we talk about the place value of the digits. The place values are all powers of 10:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **…** | **1000** | **100** | **10** | **1** | **1/10** | **1/100** | **1/100** | **1/1000** | **…** |
| … | $10^3$ | $10^2$ | $10^1$ | $10^0$ | $10^{-1}$ | $10^{-2}$ | $10^{-3}$ | $10^{-4}$ | … |

Consider this number:

123.456

We can use the place values to compute the value of a number, as you learned in primary school:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **...** | **1000** | **100** | **10** | **1** | **1/10** | **1/100** | **1/100** | **1/1000** | **...** |
| ... | | 1 | 2 | 3 | 4 | 5 | 6 | | ... |

We can multiply this out to determine the value:

$$\begin{aligned}
&= (\mathbf{1} \times 10^{2}) + (\mathbf{2} \times 10^{1}) + (\mathbf{3} \times 10^{0}) + (\mathbf{4} \times 10^{-1}) + (\mathbf{5} \times 10^{-2}) + (\mathbf{6} \times 10^{-3}) \\
&= (\mathbf{1} \times 100) + (\mathbf{2} \times 10) + (\mathbf{3} \times 1) + (\mathbf{4} \times 0.1) + (\mathbf{5} \times 0.01) + (\mathbf{6} \times 0.001) \\
&= 100 + 20 + 3 + 4/10 + 5/100 + 6/1000 \\
&= 123.456
\end{aligned}$$

The same system works with binary numbers. That is, the value of each bit is scaled according to the place value of the bit. However, with binary numbers, the place values are all powers of 2:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **...** | **8** | **4** | **2** | **1** | **1/2** | **1/4** | **1/8** | **1/16** | **...** |
| ... | $2^{3}$ | $2^{2}$ | $2^{1}$ | $2^{0}$ | $2^{-1}$ | $2^{-2}$ | $2^{-3}$ | $2^{-4}$ | ... |

Consider this binary number:

101.0101

Using this, we can convert binary numbers into decimal numbers:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **...** | **8** | **4** | **2** | **1** | **1/2** | **1/4** | **1/8** | **1/16** | **...** |
| ... | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | ... |

$$\begin{aligned}
&= (\mathbf{1} \times 2^{2}) + (\mathbf{0} \times 2^{1}) + (\mathbf{1} \times 2^{0}) + (\mathbf{0} \times 2^{-1}) + (\mathbf{1} \times 2^{-2}) + (\mathbf{0} \times 2^{-3}) + (\mathbf{1} \times 2^{-4}) \\
&= (\mathbf{1} \times 4) + (\mathbf{0} \times 2) + (\mathbf{1} \times 1) + (\mathbf{0} \times 1/2) + (\mathbf{1} \times 1/4) + (\mathbf{0} \times 1/8) + (\mathbf{1} \times 1/16) \\
&= 4 + 1 + 1/4 + 1/16 \\
&= 5.3125
\end{aligned}$$

## Exponents

With a decimal number, we can multiply by a power of 10 to shift the position of the decimal point.

$71.234 = 7123.4 \times 10^{-2}$

Here are some examples showing that the same sequence of digits can represent different numbers, when the base (10) is raised to different powers:

| Scientific Notation | Equal Value |
|---|---|
| $7.1234 \times 10^{-1}$ | `.71234` |
| $7.1234 \times 10^{0}$ | `7.1234` |
| $7.1234 \times 10^{1}$ | `71.234` |
| $7.1234 \times 10^{2}$ | `712.34` |
| $7.1234 \times 10^{3}$ | `7123.4` |
| $7.1234 \times 10^{4}$ | `71234.` |

A number is represented with two parts. The "**mantissa**" is the numerical portion and the "**exponent**" is the power on the base.

For $7.1234 \times 10^{3}$ we have:

| | |
|---|---|
| Mantissa: | 7.1234 |
| Exponent: | 3 |

The same thing works with binary numbers. For example:

$100.111110101 = 10011111.0101 \times 2^{-5}$

Here are some other examples:

| **Scientific Notation** | **Equal Value** |
|---|---|
| $1.10110 \times 2^{-1}$ | `.110110` |
| $1.10110 \times 2^{-0}$ | `1.10110` |
| $1.10110 \times 2^{-2}$ | `11.0110` |
| $1.10110 \times 2^{-3}$ | `110.110` |
| $1.10110 \times 2^{-4}$ | `1101.10` |
| $1.10110 \times 2^{-5}$ | `11011.0` |
| $1.10110 \times 2^{-6}$ | `110110.` |

For $1.10110 \times 2^{-5}$ we have:

| | |
|---|---|
| Mantissa: | 1.10110 |
| Exponent: | -5 |

In specifying a decimal number such as:

$$7.1234 \times 10^{3}$$

we naturally show the mantissa, the exponent, and the base (10) in decimal.

In specifying a binary number such as:

$$1.10110 \times 2^{5}$$

we show the mantissa in binary. But we show the exponent (5) and the base (2) in decimal. Note that 2, when written in binary, is "10". Showing the base is binary would be especially confusing! Consider how ambiguous this would be:

$1.10110 \times 10^{101}$ ← *Avoid specifying base and exponent in binary!!!*

The basic ideas with floating point representation are…

We break the number into mantissa and exponent, such that

- The binary point is at a fixed, unchanging position.
- We represent the mantissa as a binary value.
- We represent the exponent as a binary value.
- We pack the mantissa bits and the exponent bits and a sign bit all together.

A single precision floating point number is represented with 32 bits. We will we use 23 bits for the mantissa and 8 bits for the exponent. This leaves 1 bit for the sign.

For double precision floating point, we will use 52 bits for the mantissa and 11 bits for the exponent.

## Differences Between Decimal and Binary

Some rational numbers require an infinite number of digits in their decimal representation. For example:

1/3 = 0.33333…

There are a couple of different notations that mathematicians use to represent repeating decimals:

0.3(3)*
$0.\overline{3}$

Likewise, some rational numbers may require an infinite number of bits in their binary representation.

But regardless of whether we represent a rational number in decimal or binary, the infinite strings of digits/bits will settle into a simple repeating pattern. This is true of all rational numbers, but irrational numbers (e.g., $\pi$, $\sqrt{2}$) do not have such simple decimal or binary representations. Neither their decimal nor their binary expansions will ever exhibit a repeating pattern.

Some numbers many have a finite representation in decimal but require an infinite sequence in binary. For example, the following number:

4.3

requires an infinite binary expansion to represent it, namely:

100.01001100110011… = 100.01(0011)*

It turns out that every binary number without a repeating part can be represented with a finite number of decimal digits. Furthermore, the number of digits to the right of the decimal point will never exceed the number of places to the right of the binary point. For example:

101.1101 *(binary)* = 5.8125 *(decimal)*

Turning to floating point representation, we have limited number of bits available, which means we cannot accommodate arbitrary precision. Not every number is representable, so we must round numbers to a nearby number that is representable.

For example, the number $6.022 \times 10^{23}$ can only be represented approximately, even though it appears not to have a great amount of precision. Here is the closest number that can be represented using a single precision floating point:

$6.0220001312414749845094 \times 10^{23}$

On the other hand, it turns out that this number:

$\underline{2.383496}609792 \times 10^{12}$

can be represented exactly using only 32 bit single precision. The next largest value that can be represented exactly happens to be:

$\underline{2.383496}871936 \times 10^{12}$

The underlining shows the commonality in these numbers. This example demonstrates that we can represent numbers accurately up to about 7 decimal digits with single precision floating point.

No number between these two values can be represented exactly with a single precision floating point number. For such values, we'll have to choose one of these two nearby numbers.

The ideal thing to do is "round" our desired value to the nearest number that can be represented, and then use that. Of course, any computation will be off by a little, due to this rounding.

Working with floating point involves dealing with inaccuracy and this is very tricky. To predict the expected accuracy of a computation is not at all trivial, yet may be crucial.

## Floating Point Truths

We can make the following statements about IEEE 754-2008 floating point number representation:

- Every floating point numbers has a sign. Every number is either positive or negative.

- There are two representations for zero: positive zero (i.e., +0.0) and negative zero (i.e., -0.0).

- There are two representations of infinity: positive infinity (+∞ or +inf) and negative infinity  (-∞ or -inf)

- The exponent may be positive or negative, allowing both very large numbers and very small numbers.

- There is a special representation called "not a number" ("NaN"). This value can represent a missing value or the result of a undefined operation, such as divide by zero. In some implementations there are two variations, called "quiet NaN" and "signaling NaN".

- Every 32-bit integer (i.e., every integer in the range -2,147,483,648 to +2,147,483,647) can be represented exactly with a 64 bit double precision floating point number, but not with a single precision float. In fact, the integer range is a little greater: every 54-bit integer can be represented exactly in double precision floating point. Almost all larger integers will get rounded.

- Every 25-bit integer (i.e., every integer in the range -16,777,216 to +16,777,215) can be represented exactly with a 32 bit single precision floating point number. Almost all integers outside of this range will get rounded.

As mentioned earlier, not every value in the above ranges can be represented.[2]

Floating point arithmetic is meant to mimic mathematical arithmetic, but it must be remembered that they are only approximately the same:

- The exact value or result of an operation is **not** always representable, so the computed answer is often **not** mathematically correct.
- Floating point addition is **not** always associative, due to rounding errors. That is, (x + y) + z is not always equal to x + (y + z).
- Floating point multiplication is **not** always associative. That is, (x * y) * z is not always equal to x * (y * z).
- Floating point multiplication does **not** always distribute over addition with the exact same results. That is, x * (y + z) is not always equal to (x * y) + (x * z).

However, we can say this:

- Floating point addition and multiplication are commutative, like math. For example, x+y = y+x, so you don't have to worry about the order of operands for a single operation.

[2] Recall there is a countable infinity of rational numbers between any two numbers, yet with only 32 or 64 bits, we only have a small number of unique representations.

## A Nasty Example

As an example of the dangers of not understanding floating point, consider this "C" code

```
    d1 = 123.0 + 1.0e+57 - 1.0e+57;
    printf ("d1 = %g\n", d1);
```

It prints "0", while the mathematically correct value is "123.0".

Why? The computer performs the addition first and is forced to round the value to 1.0e+57 because the exact answer cannot be represented as a double precision floating point number.

By simply inserting parentheses to force the subtraction to be done first, the following results in the mathematically correct answer.

```
    d1 = 123.0 + (1.0e+57 - 1.0e+57);
```

## Range of Values

Here is the range of values that can be represented. (We use decimal notation here and approximate the exact values.)

**Single Precision**

| | |
|---|---|
| Largest value: | $\sim 3.40282347 \times 10^{+38}$ |
| Smallest normalized value above 0: | $\sim 1.17549440 \times 10^{-38}$ |
| Smallest denormalized value above 0: | $\sim 1.40129846432 \times 10^{-45}$ |
| Digits of accuracy: | about 7 |

**Double Precision**

| | |
|---|---|
| Largest value: | $\sim 1.7976931348623157 \times 10^{+308}$ |
| Smallest normalized value above 0: | $\sim 2.2250738585072014 \times 10^{-308}$ |
| Smallest denormalized value above 0: | $\sim 5 \times 10^{-324}$ |
| Digits of accuracy: | about 16 |

## Special Values

Here are the different types of things that can be represented in a floating point bit pattern:

- Positive zero (+0.0)
- Negative zero (-0.0)
- Positive infinity (+∞ or +inf)
- Negative infinity (-∞ or -inf)
- Not-a-number (NaN)
  - Quiet Nan (qNaN)
  - Signaling Nan (sNaN)
- Normal numbers (or "normalized numbers")
- Denormalized numbers (or "denormals")

### Zero – Positive and Negative

There are exactly two ways to represent zero, one is positive and the other is negative. This is unlike math, where there is only a single number called zero and it is unsigned.

Here are some interesting behaviors:

1/+0 yields +∞
1/-0 yields –∞
+0 will normally compare as equal to -0 (e.g., the == in the "C" language)
Some languages provide a way to distinguish +0 and -0.

There are additional behaviors, such as:

-0/-∞ yields +0

Although +0 and -0 may compare as equal, they may also result in different outcomes in some computations. This challenges our understanding of the meaning of "equal", to say the least.

The bit pattern representation of zero is:

| | **single** | **double** |
|---|---|---|
| +0.0 | `0x00000000` | `0x0000000000000000` |
| -0.0 | `0x80000000` | `0x8000000000000000` |

Note that the floating point representation for +0.0 is bit-for-bit identical to the representation for 0 in binary integer representation (both signed and unsigned).

It happens to be true that -0.0 is represented identically to the most negative signed integer, but this is less useful.

### Infinity

There are two infinities which are represented as follows:

| | **single** | **double** |
|---|---|---|
| +infinity | `0x7F800000` | `0x7FF0000000000000` |
| -infinity | `0xFF800000` | `0xFFF0000000000000` |

## Not-a-Number (NaN)

There is a special value called “not-a-number”, which is often abbreviated “NaN”. Some arithmetic operations are considered to be “undefined” and, when attempted, will result in a NaN result, to indicate that the result is undefined. Here are some examples of operations that with yield “not-a-number”.

$0/0$
$\infty / \infty$
$0 * \infty$

Other operations are mathematically defined but give a complex result. Complex numbers are not handled by floating point, so operations such as the following will return NaN.

Square root of a negative number
Log of a negative number

Another use of NaN is to represent an uninitialized or missing value. If a variable is used before it is initialized, spurious incorrect results might occur, but this can be avoided if the variable contains NaN.

The IEEE spec actually mentions two kinds of NaN: “signaling NaN” and “quiet NaN”. But usually, we just talk about NaN without making any distinction about whether it is signaling or quiet.

A “signaling NaN” is supposed to cause a break in the flow of execution when it is encountered in a computation. That is, a trap or exception of some sort will occur, and the normal instruction sequence will be interrupted immediately. Signaling NaNs might reasonably used for uninitialized values: their use may represent a program bug which needs attention. In theory, signaling NaNs might also be used as placeholders for values (such as complex numbers) which require special handling.

The idea with a “quiet NaN”, is that it can be used as an operand in arithmetic operations . Furthermore, a quiet NaN will be propagated. That is, if one of the operands to some operation is a quiet NaN, the result will also be a quiet NaN. This allows a lengthy sequence of operations to be performed quickly with no special testing for problems. Once a NaN appears, as a result of some error, it will persist in the chain of computations. Each subsequent operation will complete normally, without causing an exception or trap even though some sort of error occurred earlier in the sequence. If any problems occur at any step of the computation, the final result will be a quiet NaN. Therefore, it is sufficient to perform only a single test for NaN after the entire computation to see if any errors arose at any stage of the computation.

The spec does not require signaling NaNs; they are optional. One implementation approach is for the hardware to interpret all NaN values identically, basically as quiet NaNs.

What generally happens with C is that the “Invalid” flag will be set if either operand is “signaling NaN”. However, if the operands are only “quiet NaNs”, the result will be a NaN but the “Invalid” flag will not be modified.

There are several bit patterns that can be used to represent NaNs, so there is not a single bit pattern for NaN.

A value is defined to represent NaN if (1) the exponent field is all 1's, and (2) the bits of the fraction field are not all zero. (If the fraction bits are all zero, then the value is either +∞ or –∞.) The sign bit of a NaN value is ignored.

If a distinction between quiet and signaling NaN is implemented, then one of the bits in the fraction field will be used to distinguish between quiet and signaling.

The exact bit patterns for NaN are not fully specified and can vary between implementations.

We can say that a value with all bits set to one (i.e., the representation for the signed integer -1 which is 0xFFFF FFFF or 0xFFFF FFFF FFFF FFFF) will definitely represent a NaN and will almost certainly represent a quiet NaN. For example, the all-ones pattern will be a quiet NaN for Intel, AMD, SPARC, ARM, RISC-V, etc.

**Mixing Single and Double Precision Using NaN**

There are many bits in the fraction field, and the only requirement for NaN is that they cannot all be zero. Thus, there is room to store some additional data within the NaN. So a NaN can carry a sort of "payload" value in the fraction bits. This capability may or may not be used in a particular implementation of IEEE 754-2008.

For example, the fraction field in a double is 52 bits. Assume that one bit is reserved to be always set to indicate that this is a NaN, and assume that a second bit is reserved and used to distinguish between a quiet NaN and a signaling NaN. This leaves 50 bits that can be used to store a arbitrary value. Notice that this is enough room to store an entire single precision floating point number.

Imagine a machine that implements double precision arithmetic and uses 64-bit registers to store floating point values. How might this machine store 32-bit single precision values in these same registers?

Any 64-bit value in which the high order 32 bits are set, will be always recognized as a NaN. One approach to storing a single precision value in a 64 bit register is to store the single precision value in the least significant bits 32 bits and all 1s in the most significant 32 bits.

All single precision operations will only look at the least significant 32-bits of the operands and, for the result value, will always set the most-significant 32 bits to 1s.

Any accidental attempt to perform a double precision operation on a register containing a single precision value, will interpret that operand as a NaN.

## Normalized and Denormalized Numbers

Not every number is representable and the representable numbers are spaced out on the number line. So each possible floating point value is separated by a numerical distance from the next smallest number and from the next largest number. As the numbers get smaller and closer to zero, the spacing gets smaller and the numbers are closer together. As the numbers get larger, the spacing is farther apart.

For example, the following numbers differ by a very small amount:

$4.567 \times 10^{-25}$
$4.568 \times 10^{-25}$

On the other hand, these two numbers differ by a very large amount:

$4.567 \times 10^{+25}$
$4.568 \times 10^{+25}$

However in both examples above, the accuracy is the same: 4 digits of precision.

However, there is only a limited number of bits available to represent the exponents. Exponents cannot continue to get more negative and we cannot represent smaller and smaller numbers, ever more close to zero. Therefore, this pattern of the floating point numbers becoming spaced ever more closely as they get closer and closer to zero cannot continue. Something has to change as the numbers get smaller and approach zero.

What happens is that below some size, the representable values are simply spaced uniformly all the way down to zero. This is the role of denormalized numbers.

Most floating point numbers are “normal” numbers. Normal numbers have about 7 digits of accuracy (for single precision) and 16 digits of accuracy (for double precision). In other words, we can approximate any desired value with about 7 (or 16) digits of accuracy.

Another way to look at denormalized numbers is this: For very small values, we cannot approximate the value with full accuracy. As we get closer and closer to zero, we can approximate the true value with fewer and fewer places of accuracy. For really tiny values, we may even be forced to use 0.0 to represent the value, essentially losing all accuracy.

We can make the following statements about denormalized numbers:

- All denormalized numbers are very close to zero.
- Denormalized numbers extend on both the positive and negative sides of zero.
- +0.0 and -0.0 are themselves represented as denormalized numbers.
- All denormalized numbers are regularly and evenly spaced. (Exception: +0.0 and -0.0 have an infinitesimal difference and are considered equal.)
- The largest denormalized number is just less than the smallest positive normal number.
- Likewise, the most negative denormalized number is just greater than the least negative normal number.
- It is generally safe to ignore the distinction between normalized and denormalized numbers when using floating point in your applications.

There are rules for determining the precision of the results of an arithmetic calculation involving scientific notation. But if very small values (i.e., denormalized numbers) arise during a computation, then your assumptions about precision will be violated and the final results will have reduced precision. In some cases, the final result will be a meaningless, incorrect value.

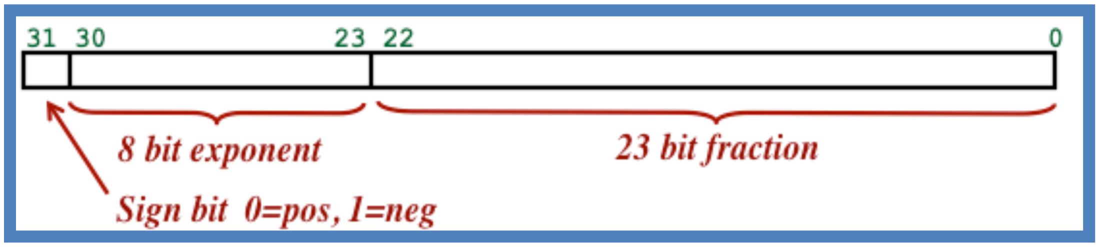
**Warning:** Always remember that numbers as represented in computers are NOT true mathematical numbers. Computer arithmetic is NOT mathematical arithmetic. Remember: "int"s are not integers and "floats" are not real or rational numbers.

Computer values and computation are mere approximations of mathematically pure ideals. A good programmer knows how important it is to understand and remember their differences in creating reliable software.

# Chapter 2: Single Precision

## Representation in 32 Bits

A single precision floating point number is represented with a 32-bit word as shown here:



How can we interpret the 32 bits representing a floating point number in single precision?

Let "**sign**" be the most significant bit. Let "**exponent**" be the bit pattern in bits 30:23. Let "**fraction**" be the bit pattern in bits 22:0.

The number represented is:

$$(-1)^{\mathbf{sign}} \times 1.\mathbf{fraction} \times 2^{\mathbf{exponent}}$$

The first term simply gives the sign of the number: 0=positive and 1=negative. Note that the most significant bit holds the sign bit for both floating point numbers and signed integers.[3]

[3] If you are considering using a two's complement instruction to check the sign bit, that will work as long as the value is not the special "not-a-number" NaN value.

There are 8 bits in the exponent field. The interpretation of the “exponent” bit patterns is:

| Bit Pattern | Meaning of Exponent Field |
|---|---|
| `0000 0000` | -126 — *Denormalized Numbers, including zero* |
| `0000 0001` | -126 |
| . . . | . . . |
| `0111 1110` | -1 |
| `0111 1111` | 0 |
| `1000 0000` | +1 |
| . . . | . . . |
| `1111 1110` | +127 |
| `1111 1111` | *Infinity, Not-a-Number* |

**Example** Let’s convert 1010.111 into floating point format.

First, we shift the binary point to just after the leftmost 1 bit:

$1.010111 \times 2^3$

Every number (except zero) will always contain at least a single 1 bit. Thus, the most significant bit must be a 1 and representing it is redundant. This explains why we prefix the fractional part with “1.”. (This trick of making one bit implicit doesn’t work with decimal numbers: the leading digit can be anything except 0, so we cannot make it implicit.)

This gives a mantissa, which we extend to 23 bits by adding zeros on the right:

```
0 1 0 1 1 1 1   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Next, we convert the exponent using the above chart

$2^3 \rightarrow$ `1000 0010`

Putting **sign**, **exponent**, and **fraction** together:

```
0  1 0 0 0 0 0 0 0  0 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Regrouping and converting to hex:

```
  0100 0000 0010 1111 0000 0000 0000 0000
  0x402F0000
```

For normalized numbers, the exponent has an effective range of -126 .. +127.

The **smallest positive normalized number** is:

In binary:

$1.00000000000000000000000 \times 2^{-126}$ (There are 23 zeros)

In bits:

0x00800000 = 0 00000001 00000000000000000000000

Decimal approximation:

$1.17549435 \times 10^{-38}$

Exact value:

0.000000000000000000000000000000000000011754943508
222875079687365372222456778186655567720875215087517062784172594547271728515625

The **largest normalized number** is:

In binary:

$1.11111111111111111111111 \times 2^{+127}$ (There are 1+23 ones)

In bits:

0x7F7FFFFF = 0 11111110 11111111111111111111111

Decimal approximation:

$3.4028235 \times 10^{+38}$

If the exponent is all ones (i.e., 11111111), then the value of the fraction matters. If the fraction is all zeros, then the value is +∞ or –∞ depending on the sign bit.

+∞:

0x7F800000 = 0 11111111 00000000000000000000000

-∞:

0xFF800000 = 1 11111111 00000000000000000000000

If the exponent is all ones (i.e., 11111111) and the value of the fraction is not all zeros, then NaN is represented. There are multiple representations that are to be interpreted as NaN values. The canonical, preferred representation of NaN is often this:

NaN (typical):
0xFFFFFFFF = 1 11111111 11111111111111111111111

If the exponent field is all zeros (i.e., 00000000), then the value is a denormalized number. The value of the number is:

$$N = (-1)^{\mathbf{sign}} \times 0.\mathbf{fraction} \times 2^{-126}$$

Notice that the leading implicit "1" bit is no longer assumed; it is now "0". Also the exponent is always -126, which happens to be the smallest exponent for normalized numbers.

Here are some sample numbers that may help explain denormalized numbers:

**Smallest normalized number:**
$1.00000000000000000000000 \times 2^{-126}$ (24 bits of precision)
**Largest denormalized number:**
$0.11111111111111111111111 \times 2^{-126}$ (23 bits of precision)
…
**Random denormalized number:**
$0.00000000001100101110101 \times 2^{-126}$ (13 bits of precision)
…
**Smallest denormalized number:**
$0.00000000000000000000001 \times 2^{-126}$ (1 bit of precision)
**+0.0:**
$0.00000000000000000000000 \times 2^{-126}$ (0 bits of precision)

The word "precision" above may be misleading. Each floating point value is an exact value. Precision and accuracy are more meaningful when talking about measurements. In that case, we have the true value and we use the terms "precision" and "accuracy" to describe the relationship between the number we obtained and the unknown true value.

## Exponent Bias

Notice that the exponent is not represented in standard “two’s complement” fashion. Normally, the integer 3 is represented as 0000 0011. For single precision, an exponent of 3 is represented as 1000 0010, which would be 130 if interpreted using two’s complement.

Sometimes we speak of a “bias” being added to the exponent. For single precision, this “bias” value is 127. We can use this number to quickly convert the exponent to/ from two’s complement.

| **Actual exponent** | | **As represented in a single precision value** | |
|---|---|---|---|
| -126 | +127 = | 1 | 0000 0001 |
| … | | | |
| 0 | +127 = | 127 | 0111 1111 |
| … | | | |
| 3 | +127 = | 130 | 1000 0010 |
| … | | | |
| +127 | +127 = | 254 | 1111 1110 |

When implementing the floating point operations using more primitive bit-based operations, it is important to remember that we cannot simply add exponents. For example

$2^3 + 2^4 = 2^7$
$3 + 4 = 7$

The correct bit pattern of the result is:

7 + 127 = 134 (binary: 10000110)

But if we simply added the exponent bits as we find them, we get the wrong result. In fact, the value overflows our 8 bit limit:

1000,0010 + 1000,0011 = 1,0000,0101 (= 261)

Notice that by subtracting the bias (and using enough bits to avoid overflow issues), we get the correct result:

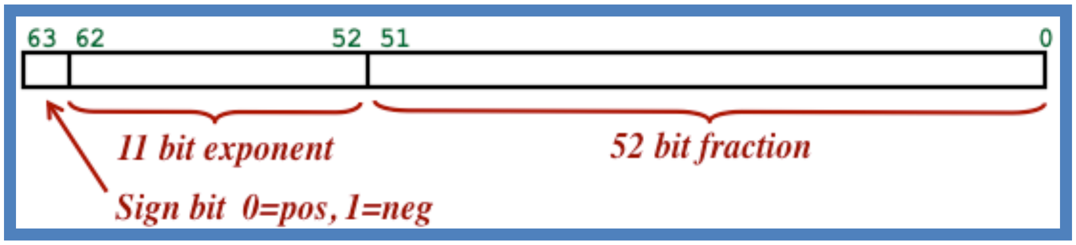
261 - 127 = 134

# Chapter3: Double Precision

## Representation in 64 Bits

Double precision floating point numbers are represented using an analogous scheme to single precision. The only difference is the number of bits in the "exponent" and "fraction" fields.

Here is the representation of a 64-bit double precision floating point value:



There are 11 bits in the exponent field, instead of 8 as in single precision. The interpretation of the "exponent" bit patterns is:

| Bit Pattern | Meaning of Exponent Field |
|---|---|
| 000 0000 0000 | -1022 — *Denormalized Numbers, including zero* |
| 000 0000 0001 | -1022 |
| . . . | . . . |
| 011 1111 1110 | -1 |
| 011 1111 1111 | 0 |
| 100 0000 0000 | +1 |
| . . . | . . . |
| 111 1111 1110 | +1023 |
| 111 1111 1111 | *Infinity, Not-a-Number* |

If the exponent is all ones (i.e., 11111111111), then the value of the fraction matters. If the fraction is all zeros, then the value is +∞ or −∞ depending on the sign bit.

```
+∞:
    0x7FF0000000000000 =
        0 11111111111 0000000000000000000000000000000000000000000000000000
-∞:
    0xFFF0000000000000 =
        1 11111111111 0000000000000000000000000000000000000000000000000000
```

If the exponent is all ones (i.e., 111_1111_1111) and the value of the fraction is not all zeros, then NaN is represented. There are multiple representations that are to be interpreted as NaN values. Here are two common representations for NaN:

```
NaN:
    0xFFFFFFFFFFFFFFFF =
        1 11111111111 1111111111111111111111111111111111111111111111111111
    0x7FF8000000000000 =
        0 11111111111 1000000000000000000000000000000000000000000000000000
```

If the exponent field is all zeros (i.e., 00000000000), then the value is a denormalized number. The value of the number is:

$$N = (-1)^{\mathbf{sign}} \times 0.\mathbf{fraction} \times 2^{\mathbf{-1022}}$$

Notice that the leading implicit “1” bit is no longer assumed; it is now “0”. Also the exponent is always -1022, which happens to be the smallest exponent for normalized numbers.

For normalized numbers, the exponent has an effective range of -1022 … +1023.[4]

The “bias” is +1023.

The largest normalized number is:

In binary:

$1.111111111...1111111111 \times 2^{+1023}$ (There are 1+52 ones)

Representation:

[4] The following names are used: **emin** =-1022, **emax** = +1023

`0x7FEF FFFF FFFF FFFF`

Decimal approximation:

$1.7976931348623157 \times 10^{+308}$

The smallest positive normalized number is:

In binary:

`1.000000000...00000000000` $\times 2^{-1022}$ (There are 52 zeros)

Representation:

`0x0010 0000 0000 0000`

Decimal approximation:

$2.2250738585072014 \times 10^{-308}$

The largest denormalized number is:

In binary:

`0.111111111...11111111111` $\times 2^{-1022}$

Representation:

`0x000F FFFF FFFF FFFF`

Decimal approximation:

$2.2250738585072009 \times 10^{-308}$

The smallest positive denormalized number is:

In binary:

`0.000000000...00000000001` $\times 2^{-1022}$

Representation:

`0x0000 0000 0000 0001`

Decimal approximation:

$4.9406564584124654 \times 10^{-324}$

## Other Sizes Beyond Single and Double

In addition to the well known single precision (32-bit) and double precision (64-bit) sizes, the IEEE 754-2008 standard also describes these floating point sizes. They are much less common.

16 bits (half precision)
128 bits (quadruple precision)
256 bits (octuple precision)

There is also mention of decimal-based representations.

# Chapter 4: Rounding

## Introduction

Rounding is necessary when an exact value requires too many bits and we need to alter the representation (and value) of the number to fit into some desired format with fewer bits.

Rounding is necessary after floating point operations such as addition and multiplication. The exact answer almost always requires more bits than the operands required. The result of a computation will almost always be a number that is not precisely representable.

For example, the addition of two double numbers may not be exactly representable as a double.

To see why this can happen, look at the following simple addition. We'll use decimal numbers, but the same effect happens with binary numbers.

Here are two numbers with 5 digits of precision. Their sum requires 8 digits to represent.

```
  12.345          = 1.2345 × 10^1
+    .067891      = 6.7891 × 10^-2
  12.412891       = 1.2412891 × 10^1
```

The IEEE spec says that the exact result should be "rounded" to a number that can be represented. For example, when two doubles are added, their result will be some double value.

In the above example, to bring the result back down to 5 digits of precision, some accuracy must be sacrificed.

Just as in this example, when floating point operations are performed, there may be a loss of accuracy as a result of rounding.

The IEEE spec lists several ways that a value can be rounded to something that can be represented:

- Round to the nearest number
  (For a tie, the value with a zero in the least significant bit is chosen.)
- Round toward zero (i.e., truncate)
- Round toward positive infinity (i.e., round up)
- Round toward negative infinity (i.e., round down)

In order to perform rounding correctly, a computer may need to perform calculations (e.g., multiplication) with greater precision to first compute the correct value. Then, as the final step in the calculation, the value must be properly rounded to fit into the available floating point bits.

Here is another example, showing that the entire effect of an operation can be lost as the result of rounding.

```
   12.345            
+    .00000067891    
   12.34500067891    
```

$= 1.2345 \times 10^{1}$

$= 6.7891 \times 10^{-7}$

$= 1.234500067891 \times 10^{1}$

Rounding to a value with the same precision (either rounding down, rounding toward zero, or rounding to the nearest) gives the initial operand, unchanged. Here is the rounded result:

```
   12.345
```

$= 1.2345 \times 10^{1}$

Rounding up or rounding away from zero would give a different result:

```
   12.346
```

By the way, in common speaking, the term “rounding” usually means “round-to-the-nearest”. In the floating point world, “rounding” is a more general term that also includes “round-toward-zero”, “round-up”, and “round-down”.

## Rounding Toward Zero (Truncation)

With signed integers (that is, “two’s complement” representation), truncation will round the value down, toward -infinity. In this example, we’ll round down the last 3 bits, effectively going to the nearest multiple of $2^3 = 8$. The bits being eliminated are underlined:

| **Binary** | **Decimal** | | **Rounded** | **Decimal** |
|---|---|---|---|---|
| 00001$\underline{001}$ | +9 | → | 00001$\underline{000}$ | +8 |
| 11101$\underline{001}$ | -23 | → | 11101$\underline{000}$ | -24 |

The implementation is simple: we just clear the underlined bits.

With regards to sign, truncation works differently with floating point numbers. Floats are represented as a magnitude and a sign bit. Truncation affects the magnitude only. For example +7.125 and -7.125 have the same magnitude. Truncation works only on the magnitude. So truncation takes the magnitude from 7.125 to 7.000, regardless of sign.

So for floating point, “**truncation**" and “**rounding toward zero**” mean the same thing.

This form of rounding is the easiest to implement, since all we do is change the unwanted bits to zero.

Truncation can never result in overflow.

## Rounding Away from Zero

Rounding away from zero is more complicated to implement.

As an example, imagine that we want to round off the last 3 bits, that is, round to a multiple of 8. For integers that are already a multiple of 8, this is easy; there is no change:

| **Binary** | **Decimal** | | **Rounded** | **Decimal** |
|---|---|---|---|---|
| $00001\underline{000}$ | +8 | → | $00001\underline{000}$ | +8 |
| $00101\underline{000}$ | +40 | → | $00101\underline{000}$ | +40 |

But for numbers that have "1"s in the underlined area, we need to clear the last three bits and then add "1000":

| **Binary** | **Decimal** | | **Rounded** | **Decimal** |
|---|---|---|---|---|
| $00001\underline{001}$ | +9 | → | $00010\underline{000}$ | +16 |
| $00101\underline{011}$ | +43 | → | $00110\underline{000}$ | +48 |

Rounding away from zero can result in overflow.

For example, consider rounding the following value to 5 bits and adjusting the exponent to account for the lost bits.

| **Binary** | **Decimal** | | **Rounded** | **Decimal** |
|---|---|---|---|---|
| $11111\underline{001}$ | +249 | → | $100000\underline{000}$ | +256 |

The resulting rounded value now contains 6 bits, which will now be larger than 5 bits. To get the value to fit into 5 bits, we need to round again and adjust the exponent by 1. Fortunately, the bit we must remove (shown in red) will always be zero.

Thus, whenever we have an overflow like this, we simply the rounded value right by 1 bit and adjust the exponent. Of course, the adjustment to the exponent can also overflow, in which case we have an overflow situation.

## Rounding Up and Rounding Down

By "rounding up", we mean rounding numbers in the direction of +infinity. By "rounding down", we mean in the direction of -infinity.

Assuming we can round a number towards zero or away from zero, we can implement “rounding up” and “rounding down” simply. But we have to look at the sign to know what to do.

We implement “**rounding up**” as follows:

```
If the number is > 0
    Round  the number away from zero
If the number is < 0
    Round  the number toward zero
```

We implement “**rounding down**” as follows:

```
If the number is > 0
    Round  the number toward zero
If the number is < 0
    Round  the number away from zero
```

## Round-to-Nearest

Next, look at how we round a number “to the nearest”. With decimal, there are three cases:

If the dropped digit is < 5
$83.\underline{2} \rightarrow 83.$ Drop the digit

If the dropped digit is = 5
$83.\underline{5} \rightarrow 83.\ /\ 84.$ Exactly in the middle; could go either way.

If the dropped digit is > 5
$83.\underline{9} \rightarrow 84.$ Drop the digit and add 1 in the next place

Let’s ignore the location of the decimal point and just talk about rounding to a given place. We show the digits we want to eliminate with underlining, as in:

$83\underline{284} \rightarrow 83$

In the case of “5”, we need to look at the digits to the right of the “5”.

Here are the cases:

If the first dropped digit is < 5
  Drop the digits
    83<u>**2**84</u> → 83
Otherwise, if the dropped digits are “500000…”
  Exactly in the middle; could go either way.
    83<u>**500**</u> → 83 / 84
Otherwise
  Drop the digits and add 1
    83<u>**5**01</u> → 84
    83<u>**9**89</u> → 84

With binary, we only have two bits, “0” and “1”. The “1” bit functions like decimal “5”. Note that:

```
110.0 = 6.0
110.1 = 6.5
```

Each successive bit divides by 1/2:

```
110.00 = 6.00
110.01 = 6.25

110.000 = 6.000
110.001 = 6.125

110.0000 = 6.0000
110.0001 = 6.0625
```

So if the number ends with “1” it is exactly halfway between the next shorter numbers:

```
110.00 = 110.000 = 6.0
         110.001 = 6.125   ← Halfway between 6.0 and 6.25
110.01 = 110.010 = 6.25
```

Here are the cases for binary:

If the first dropped bit is = 0
  Drop the bits
    $\texttt{110}\underline{\texttt{0110101}} \rightarrow \texttt{110}$
Otherwise, if the dropped bits are "100000…"
  Exactly in the middle; it could go either way.
    $\texttt{110}\underline{\texttt{1000000}} \rightarrow \texttt{110/111}$
Otherwise
  Drop the bits and add 1
    $\texttt{110}\underline{\texttt{1000010}} \rightarrow \texttt{111}$

**Example** Let's look at this number and round it to 6 bits. The first dropped bit is shown in red.

$\texttt{1.00011}\underline{\texttt{0011}}$

Since it is zero in this example, we round down. We drop bits, giving:

```
1.00011
```

**Example** Here the key bit is "1" and there are other dropped bits that are "1".

$\texttt{1.00011}\underline{\texttt{1011}}$

In this case, we must round up. This means we add 1 to the least significant position:

```
  1.00011
+       1
---------
  1.00100
```

**Example** Here the key bit is "1" but all the other dropped bits are "0".

$\texttt{1.00011}\underline{\texttt{1000}}$

We could either round up or truncate the bits.

## Rounding Ties

In the case where we are right in the middle, which way do we go? We can refer to this as "**the problem of rounding .5**" or "**rounding ties**":

Decimal:
```
        83.50000 → 83 or 84?
```
Binary:
```
        11010000 → 110 or 111?
```

Here are the choices:

- Always round up.
- Always round down.
- Sometimes up, sometimes down.

"**Rounding ties up**" is the easiest. We always add 1 in the next place.

```
    83.50000 → 84.
```

Note that this option allows us to avoid looking at all other digits. We simply look at the key digit and ignore everything to the right of it.

```
    83.50000 → 84.
    83.50073 → 84.
```

Here it is in binary. We can ignore all bits to the right of the key bit.

```
    11010000 → 111
    11010001 → 111
```

"**Rounding ties down**" is next the easiest. We still need to examine all bits to right of the key bit to determine whether we round down or up.

```
    11010000 → 110          If all zeros, round down
    11010001 → 111          Otherwise, round up
```

## Rounding Ties to Even (or Odd)

The final implementation is to determine whether to round up or down based on the previous digit position.

Decimal digits are either odd or even. Likewise, bits are either odd or even. By looking at the next digit or bit, we are essentially doing a "coin toss" and using that digit/bit to determine which way to round. Assuming that digits/bits are randomly distributed, we'll round ties up half the time and down the other half of the time.

Let's look at "**round to even**" with decimal. Here, we'll look at the digit in blue to decide the tie. If odd, we round up. If even, we round down. This results in the blue digit always becoming even.

| | |
|---|---|
| `83.5000` → `84.` | Odd → round up |
| `86.5000` → `86.` | Even → round down |

With binary, "**round to even**" always results in a value where the last bit is zero. Whenever there is a tie, we get a number with a "0" in the least significant bit.

| | |
|---|---|
| `10000.1000` → `10000.` | Even → round down |
| `10001.1000` → `10010.` | Odd → round up |
| `10010.1000` → `10010.` | Even → round down |
| `10011.1000` → `10100.` | Odd → round up |
| `10100.1000` → `10100.` | Even → round down |
| `10101.1000` → `10110.` | Odd → round up |

With binary, there is also a "**round to odd**", which is similar. Whenever there is a tie, we get a number with a "1" in the least significant bit.

| | |
|---|---|
| `10000.1000` → `10001.` | Even → round up |
| `10001.1000` → `10001.` | Odd → round down |
| `10010.1000` → `10011.` | Even → round up |
| `10011.1000` → `10011.` | Odd → round down |
| `10100.1000` → `10101.` | Even → round up |
| `10101.1000` → `10101.` | Odd → round down |

# Chapter 5: Error Conditions[5]

## Not-A-Number

If there is a problem with some floating point operation, the result will be not-a-number (**NaN**).

There are three general reasons for such an error:

- One of the operands was non-a-number to start with.
- The result is mathematically undefined. For example, 0/0.
- The result is a complex number. For example, “square root of a negative”.

Whenever a **NaN** is is first encountered — that is, when a result is **NaN** although the arguments were themselves okay — the IEEE spec requires that the “**invalid operation**” flag be set.

However, when one argument is already **NaN**, we need to determine whether the NaN is a “**signaling NaN**” or “**quiet NaN**”. If **signaling**, then the “**invalid operation**” flag will be set. If **quiet**, then the “**invalid operation**” flag will not be set.

[5] Much of the information in this document comes from Wikipedia.

## When the Result is Undefined

The following operations will result in **NaN** and the “**invalid operation**” flag will be set.

- $+\infty + -\infty$
- $-\infty + +\infty$
- $+\infty - +\infty$
- $-\infty - -\infty$
- $\pm 0 \times \pm\infty$
- $\pm\infty \times \pm 0$
- $\pm 0 \ / \ \pm 0$
- $\pm\infty \ / \ \pm\infty$
- $\pm\infty \ \% \ y$ (remainder function)
- $x \ \% \ \pm 0$
- The standard has alternative requirements for the “power” function:
  - The standard **pow** function and the integer exponent **pown** function define $0^0$, $1^\infty$, and $\infty^0$ as 1.
  - The **powr** function defines $0^0$, $1^\infty$, and $\infty^0$ as **NaN**.

## When the Result is a Complex Number

Here are some operations that result in a complex number. The IEEE spec says that these operations will raise an error. That is, the result will be **NaN** and the “**invalid operation**” flag will be set.

- The square root of a negative number
- The logarithm of a negative number
- The inverse sine or cosine of a number that is less than −1 or greater than 1

Some hardware ISAs may implement SQUARE ROOT in hardware, but it is probable that some or all of these operations will be implemented in software, not hardware.

## Conversion Between Integers and Floating Point

### Floating Point → Integer

Any computer architecture that supports floating point generally includes an instruction to convert from floating point to integer. For example, the Blitz-64 machine instruction converts from double precision to a 64-bit integer:

```
    fcvtif   r4,r5     # Floating ConVerT to Integer from Floating
```

What if the argument is **NaN**? The general rule that "**NaN**s are always propagated" cannot be followed since the result — a 64-bit signed, twos complement integer — cannot represent a **NaN** value.

Typically, the machine architecture will set the "**invalid operation**" flag. The integer result might be specified as "0" or left "undefined".

There is also the question of what happens when the floating point number exceeds the range of the integers: Is this an error, or overflow, or treated by just setting the result integer to the maximal value.

Note that the integer 9,223,372,036,854,775,808 is representable exactly as a double precision float since it is $1.0 \times 2^{63}$. This integer can be expressed as the unsigned number 0x8000,0000,0000,0000, which is one greater than we can represent as a signed 64 bit integer. (Of course, we can represented its negation -9,223,372,036,854,775,808 exactly as a signed integer as 0x8000,0000,0000,0000.)

Using double precision floats, the adjacent values to this large number differ by a substantial amount, due to the loss of precision at this magnitude. The integer representation has 63 zero bits while the double precision representation has only 52 zero bits. We lost 11 bits. Note that $2^{11}$ = 2,048 and binary 1000_0000_0000 is decimal 2,048. Thus, going to the next greatest number, will add 2,048 to the value. Going down instead, we must decrement the exponent and go to:

$$1.1111111111111111111111111111111111111111111111111111 \times 2^{62}$$

which is 1,024 less.

Here are the values around the largest signed integer ( 0x7FFF,FFFF,FFFF,FFFF = 9,223,372,036,854,775,807) that we can represent exactly with double precision floats:

+9,223,372,036,854,774,784.0
+9,223,372,036,854,775,808.0
+9,223,372,036,854,777,856.0

For Blitz-64, the exact test for overflow for FCVTIF is this:

If the floating value is greater than +9,223,372,036,854,775,808.0 then the "**overflow flag**" (OV) will be set and +9,223,372,036,854,775,807 (i.e., 0x7FFF,FFFF,FFFF,FFFF) will be placed in RegD. We intentionally use > and not ≥ because it is reasonable to round +9,223,372,036,854,775,808.0 to +9,223,372,036,854,775,807 (i.e., to 0x7FFF,FFFF,FFFF,FFFF).

## Integer → Floating Point

The computer will also have an instruction for converting in the other direction, i.e., from integer to floating point.

For example, the following Blitz-64 machine instruction converts from 64-bit signed integer to double precision floating point:

```
     fcvtfi   r4,r5     # Floating ConVerT to Floating from Integer
```

With this conversion, the range of floating point numbers is much greater than integers, so there is no possibility of overflow. However, not all integers can be represented exactly. In some cases, the value must be rounded to the nearest floating point value.

All integers in this range can be represented exactly as floating point numbers:

| Decimal | | 64-bit Integer | Double Precision |
|---|---|---|---|
| $-2^{53}$ | -9,007,199,254,740,992 | 0xFFE0,0000,0000,0000 | 0xC340,0000,0000,0000 |
| | … | | |
| $+2^{53}$ | +9,007,199,254,740,992 | 0x0020,0000,0000,0000 | 0x4340,0000,0000,0000 |

Most integers outside this range must be rounded, and the rounding rules will be used.

( Note that $2^{53}$ is represented as a binary integer as a "1" followed by 53 "0"s. In other words, it requires 54 bits to represent. Recall that double precision floating point accommodates only 52 bits after the leading "1"; how can we accommodate 54 bit numbers? But notice This: $2^{53}$ can be represented exactly, since it is an even power of 2. Although we can represent this number exactly, we cannot represent all 54 bit numbers exactly. And all numbers smaller than $2^{53}$ can be represented with only 53 bits. In a double precision number , the leading "1" bit is implicit and there is enough room for up to 52 additional bits. )

For other conversions, we have:

| **From** | **To** | |
|---|---|---|
| 32 bit integer | Single Precision | May round |
| 32 bit integer | Double Precision | Always exact |
| 64 bit integer | Single Precision | May round |
| 64 bit integer | Double Precision | May round |

When converting from a 32 bit integer to a double precision floating point, there will never be any rounding. The 52 bits of mantissa are more than enough to represent all possible 32 bit integers with perfect accuracy.

## Relational Operations with NaN

There are some strange and unexpected behaviors when one of the operands to a relational comparison is NaN.

Normally, an operation in which one operand is NaN is required to yield NaN as a result. For relational operations, the result is normally a "boolean" or a branch, so it is not possible to yield a NaN.

So what happens? Here is the rule:

***Every NaN shall compare unordered with everything, including itself.***

In particular, we have:

| Operation | Result | Invalid Operation |
|---|---|---|
| NaN < x | false | yes |
| NaN <= x | false | yes |
| NaN > x | false | yes |
| NaN >= x | false | yes |
| NaN == x | false | no |
| NaN != x | true | no |

Note that this implies the following very **unexpected results**:

| Operation | Result | Invalid Operation |
|---|---|---|
| NaN == NaN | false | no |
| NaN != NaN | true | no |

Normally, we accept these equivalences:

| This: | is the same as: |
|---|---|
| x < y | NOT (x >= y) |
| x <= y | NOT (x > y) |
| x > y | NOT (x <= y) |
| x >= y | NOT (x < y) |

However, these are not true when one of the operands is NaN!

# Chapter 6: Implementing Addition

## How to Add Floating Point Numbers

Let's make the assumption that the two numbers we want to add are positive. If we have two numbers of different signs, we will actually use a subtraction algorithm. If we have two numbers that are negative, then we'll add and then mark the result negative.

It may be that one of the numbers we are adding is one of the following values. We will assume these special cases are dealt with separately.

+0.0  
-0.0  
NaN  
+infinity  
-infinity

In particular, we assume that neither "x" nor "y" is zero, which means that they will have at least one "1" bit.

The answer will need to be rounded and we have these possible ways to round:

**round-down**  
**round-up**  
**rounding-towards-nearest**

If we require "round toward +infinity" or "round toward -infinity", we'll have to look at the signs of the numbers we are adding:

| | **round toward +infinity** | **round toward -infinity** |
|---|---|---|
| **x, y are positive** | use “round-up” | use “round-down” |
| **x, y are negative** | use “round-down” | use “round-up” |

Let’s assume we are implementing single precision and are given two non-zero numbers “x” and “y”. Singe precision has 1 implicit bit and 23 bits of fractional mantissa.

To shorten our examples, we’ll use 1 implicit bit and 23 bits of fractional mantissa.

Here is an example. We make the implicit 1 bit explicit:

x: $1.10110 \times 2^{17}$
y: $1.00111 \times 2^{14}$

If we happen to have a denormalized number, we’ll make the implicit 0 bit explicit:

denorm example: $0.00010 \times 2^{-126}$

First, we can shift the second number left or right to make the exponents equal:

y: $1.00111 \times 2^{14}$
y (shifted): $0.00100111 \times 2^{17}$

Now we can add them:

x: $1.10110 \times 2^{17}$
y (shifted): $0.00100111 \times 2^{17}$
sum: $01.11010111 \times 2^{17}$

Whenever we add two binary numbers, the result may require a single additional bit for a carry.

In this example, there was no carry. If we are truncating (rounding positives toward zero), then there is nothing more to do. We just grab the bits relevant bits:

sum: $01.11010111 \times 2^{17}$

If there had been a carry, then we just shift the bits right one bit and increase the exponent. For example:

|  |  |
|---|---|
| x: | $1.10001 \times 2^{17}$ |
| y: | $1.01011 \times 2^{17}$ |
| sum: | $10.11100 \times 2^{17}$ |
| sum (shifted): | $1.011100 \times 2^{18}$ |

If the exponent was +127 and is incremented to +128, then we have to signal overflow.

In previous examples, the numbers had similar exponents, but consider this example:

|  |  |
|---|---|
| x: | $1.10110 \times 2^{17}$ |
| y: | $1.00111 \times 2^{6}$ |

Shifting "y", we get:

|  |  |
|---|---|
| x: | $1.10110 \times 2^{17}$ |
| y( shifted): | $0.0000000000100111 \times 2^{17}$ |
| sum: | $01.1011000000100111 \times 2^{17}$ |
| sum (rounded): | $01.10110 \times 2^{17}$ |

After rounding (either down or to nearest), we see that the sum is just "x" itself. In other words, when the exponents are very different, the answer is simply the largest of the two values.

How different must the exponents be for us to ignore the smaller number? It depends on how we will be rounding.

Here is the limiting case for rounding down.

| | |
|---|---|
| x: | `1.10110` × $2^{17}$ |
| y: | `1.00111` × $2^{11}$ |
| y( shifted): | `0.00000100111` × $2^{17}$ |
| sum: | `1.10110100111` × $2^{17}$ |
| sum (rounded): | `1.10110` × $2^{17}$ |

The difference in exponents is just 6 bits, i.e., 5+1. With single precision, the required difference is 1+23 = 24. If the difference in exponents is equal or greater than this, the answer is just the largest number.

If the difference in exponents is less than this, then the numbers will overlap.

In this next example, the bits will overlap. We will shift the smaller number to the right until the exponents are the same.

If we are rounding down toward zero, we do not need to keep the bits we are shifting out.

| | |
|---|---|
| x: | `1.10110` × $2^{17}$ |
| y: | `1.00111` × $2^{14}$ |
| | `0.100111` × $2^{15}$ |
| | `0.0100111` × $2^{16}$ |
| | `0.00100111` × $2^{17}$ |

| | |
|---|---|
| x: | `1.10110` × $2^{17}$ |
| y: | `0.00100` × $2^{17}$ |
| sum: | `1.11010` × $2^{17}$ |

If we are rounding up, the case is similar. If the exponents differ by 24 (for single precision), then the answer is just the largest, EXCEPT… We must add 1 in the least significant place to the larger number. Since we have assumed that both numbers are non-zero, we must round the result up.

This rounding up may cause a bit of a problem, and may result in a carry. So, perhaps it is simplest if — when rounding up — we substitute a fixed value for the smaller number. So we replace the smaller number "y" by "0.00001":

| | |
|---|---|
| y: | 1.00111 $\times 2^{11}$ |
| y (shifted): | 0.00000100111 $\times 2^{17}$ |
| substituted incr: | 0.00001 |
| | |
| x: | 1.10110 $\times 2^{17}$ |
| substituted incr: | 0.00001 |
| sum: | 1.10111 $\times 2^{17}$ |
| sum (rounded): | 1.10111 $\times 2^{17}$ |

Another way we can think of this is that we are shifting "y" to the right, but we summarize all the bits shifted to the right. We show in red the bits shifted out. We can lose them altogether; the summary bit is all we will need. The summary bit is a sticky bit, initialized to the rightmost bit of "y".

| | |
|---|---|
| y: | 1.0010**0** $\times 2^{11}$ |
| | 0.1001**0**0 $\times 2^{12}$ |
| | 0.0100**1**00 $\times 2^{13}$ |
| | 0.0010**1**100 $\times 2^{14}$ |
| | 0.0001**1**0100 $\times 2^{15}$ |
| | 0.0000**1**00100 $\times 2^{16}$ |
| | 0.0000**1**100100 $\times 2^{17}$ |

Here is an example in which the exponent is not so extreme. We will just shift until the exponents are equal. In this example, we only lose part of the number:

| | |
|---|---|
| y: | 1.0010**0** $\times 2^{14}$ |
| | 0.1001**0**0 $\times 2^{15}$ |
| | 0.0100**1**00 $\times 2^{16}$ |
| | 0.0010**1**100 $\times 2^{17}$ |

With "round-to-nearest", things are trickier. We need to remember whether any of the bits shifted out to the right were non-zero.

# Chapter 7: Implementing Subtraction

## Addition, Subtraction, and Signs

Addition and subtraction are closely related when the values can be either positive or negative. For example, we can use addition when we are subtracting a negative number. Likewise, the operation of subtraction can be used when we are adding a positive number to a negative number.

```
x  -  -y  =  x  +  y
x  +  -y  =  x  -  y
```

Here are the cases. (In this table, "+x" and "-x" represent positive and negative numbers, respectively.)

| Input | | | Sign of Result | Operation to Perform |
|---|---|---|---|---|
| +x | + | +y | + | addition |
| +x | + | -y | + or - | subtraction |
| -x | + | +y | + or - | subtraction |
| -x | + | -y | - | addition |
| | | | | |
| +x | - | +y | + or - | subtraction |
| +x | - | -y | + | addition |
| -x | - | +y | - | addition |
| -x | - | -y | + or - | subtraction |

The first step is obviously to sort out which case we have and determine whether we will use the addition or subtraction operation[6]

[6] In hardware, to speed things up, we might perform both the addition and subtraction operations in parallel. While these operations are being performed, other circuitry can determine which of these cases applies and then select which answer to return.

With subtraction, we must look at the magnitudes of “x” and “y” to determine the sign of the result.

With signed integers represented in two’s complement, the subtraction operation is simple and the sign of the result does not require special-casing. With floating point representation, we must evaluate the magnitude to determine which way to subtract.

In other words, the subtraction operation must compute the **difference**. With the mantissas, we are not working with two’s complement representations. This means we must determine which number is larger and either compute “x-y” or “y-x”.

| Input | Sign of Result | Order of Subtraction |
|---|---|---|
| 5 - 2 | + | x - y |
| 2 - 5 | - | y - x |

Our approach will be to determine which value is larger. First, we compare the exponents. If one exponent is larger than the other, then that floating point value is larger. But if the exponents happen to be equal, we must look at the mantissas.

To compare two integers “a” and “b", we can compute “a-b” and use the sign of the result to tell us which was larger. If we want the difference between two numbers without knowing which is larger, then we can compute both “a-b” and “b-a” in parallel, and then use the sign bit to select which result to deliver.

Other than the above comments above, subtracting floating point numbers is similar to adding them. The binary points must be shifted, the exponents must be adjusted, and the result must be rounded.

To subtract one positive binary number from another, addition can be used. To compute

$$x - y$$

we negate “y” and perform addition. To negate “y”, we flip the bits at add 1. We can flip the bits of a binary number easily and quickly. We can add 1 at the same time we perform the main add by asserting a “carry-in” to the least significant bit.

# Chapter 8: Implementing Multiplication

## Overview

Multiplication of two floating point numbers is fairly straightforward.

First, we can deal with special cases separately. These include all cases when an argument is:

NaN
± 0.0
± infinity

Next, we can compute the sign of the result:

signOfResult ← signOfX **XOR** signOfY

We will also consider the rounding mode. We will need to know how to round the result:

round to nearest
round toward negative infinity
round toward positive infinity
round up
round down

At this stage we can choose one of the following rounding modes, which will be applied to the final result.

round to nearest
round up
round down

For the cases of rounding-toward-infinity, we can choose either round-up or round-down based on the sign of the result.

Recall that when you multiply two N bit numbers, the result will have at most 2N bits:

```
       x:         000011
       y:   ×     000010
                  ------
 product:   000000000110

       x:         111111
       y:   ×     111111
                  ------
 product:   111110000001
```

With floating point, we have the exponents, which will be added.

We will first move the arguments and shift them so that all have the binary point to the right of the first “1” bit.

For normalized numbers, there will be no change. But we will make the missing “1” explicit. In our examples, we’ll show the binary point, although a hardware implementation would obviously not represent it.

input: $01011 \times 2^{57}$

adjusted: $1.01011 \times 2^{57}$

For denormalized numbers, we’ll add in zeros on the right. As we shift the point, we’ll increment the exponent, causing the exponent to go below the -126 limit.

input: $0.00001 \times 2^{-126}$ smallest denormalized

adjusted: $1.00000 \times 2^{-131}$

input: $0.00101 \times 2^{-126}$ typical

adjusted: $1.01000 \times 2^{-129}$

input: $0.11111 \times 2^{-126}$ largest denormalized

adjusted: $1.11110 \times 2^{-127}$

Now consider we can perform the multiplication:

|  |  |
|---|---|
| x: | `1.00111` $\times 2^{-11}$ |
| y: | `1.01001` $\times 2^{+37}$ |
| product: | `01.1000111111` $\times 2^{+26}$ |

After computing the product, we need to possibly shift the binary point (adjusting the exponent), round the value, deal with denormalized results, and detect overflow and underflow.

## Examples

Let's look at some different examples. We'll use "round-down".

**Case 1:**

|  |  |  |
|---|---|---|
| x: | `1.00111` | $\times 2^{-11}$ |
| y: | `1.01001` | $\times 2^{+37}$ |
| product: | `01.1000111111` | $\times 2^{+26}$ |
| rounding: | `1.10001` | $\times 2^{+26}$ |

**Case 2:**

|  |  |  |
|---|---|---|
| x: | `1.11111` | $\times 2^{12}$ |
| y: | `1.11111` | $\times 2^{34}$ |
| product: | `11.1110000001` | $\times 2^{46}$ |
| shifting right: | `1.11110000001` | $\times 2^{47}$ |
| rounding: | `1.11110` | $\times 2^{47}$ |

**Case 3:**

|  |  |  |
|---|---|---|
| x: | `1.11111` | $\times 2^{127}$ |
| y: | `1.11111` | $\times 2^{127}$ |
| product: | `11.1110000001` | $\times 2^{254}$ |
| shifting right: | `1.11110000001` | $\times 2^{255}$ |
| result: | …overflow… |  |

**Case 4:**

| | | | |
|---|---|---|---|
| x: | `0.00001` | $\times 2^{-126}$ | smallest denormalized |
| y: | `1.00000` | $\times 2^{-126}$ | smallest normalized |
| product: | `00.0000100000` | $\times 2^{-252}$ | |
| shifting left: | `1.00000000000` | $\times 2^{-252}$ | |
| result: | ...underflow... | | |

**Case 5:**

| | | |
|---|---|---|
| x: | `0.00001` | $\times 2^{-126}$ |
| y: | `0.00001` | $\times 2^{-126}$ |
| product: | `00.0000000001` | $\times 2^{-252}$ |
| shifting left: | `1.00000000000` | $\times 2^{-262}$ |
| result: | ...underflow... | |

**Case 6:**

| | | |
|---|---|---|
| x: | `1.11111` | $\times 2^{127}$ |
| y: | `0.00001` | $\times 2^{-126}$ |
| product: | `00.0000111111` | $\times 2^{1}$ |
| shifting left: | `01.1111100000` | $\times 2^{-4}$ |
| rounding: | `1.11111` | $\times 2^{-4}$ |

## Detecting Denormalized Results

In all cases, we need to normalize the result, which means:

- Shift the binary point to just to the right of the leftmost 1 bit, adjusting the exponent.
- Detect overflow / underflow

After shifting the binary point, we must detect when the result must be turned into a denormalized number. We can do this by looking at the exponent. If it is less than -126 (but -131 or more), then we shift right, incrementing the exponent, until it is -126.

After shifting the binary point, we must round the number appropriately as required by the current rounding mode, e.g., round-to-nearest, round-up, round-down, to eliminate the bits on the right end.

In this example, the result is the largest denormalized number.

| | |
|---|---|
| x: | *x.xxxxx* × $2^{-xx}$ |
| y: | *y.yyyyy* × $2^{-yy}$ |
| product: | `0.00011111zz` × $2^{-131}$ |
| shifting left: | `1.1111zz0000` × $2^{-127}$ |
| as a denormal: | `0.11111zz0000` × $2^{-126}$ |
| after rounding: | `0.11111` × $2^{-126}$ |

In this example, the result is the smallest denormalized number.

| | |
|---|---|
| x: | *x.xxxxx* × $2^{-xx}$ |
| y: | *y.yyyyy* × $2^{-yy}$ |
| product: | 0.100000$\underline{\texttt{zzzz}}$ × $2^{-130}$ |
| shifting left: | 1.00000$\underline{\texttt{zzzz}}$0 × $2^{-131}$ |
| as a denormal: | `0.0000100000` × $2^{-126}$ |
| after rounding: | `0.00001` × $2^{-126}$ |

In the last case, note that we shifted several bits out during the denormalizing process. These bits are underlined and represented as $\underline{\text{zzzz}}$. If the rounding mode is round-to-nearest or round-up, these bits can affect the rounding result, so they must not be ignored.

# Chapter 9: Miscellaneous Topics

## Named Values

The standard specifies several predefined names:

The “**machine epsilon**” (eps) is the distance from 1.0 to the next larger floating point number.

The smallest positive normalized floating point number is called “**realmin**”.

The largest floating point number is called “**realmax**”.

For double precision:

| **Name** | **Value** | **Approx value** |
|---|---|---|
| eps | $2^{-52}$ | $2.2204 \times 10^{-16}$ |
| realmin | $2^{-1022}$ | $2.2251 \times 10^{-308}$ |
| realmax | $(2\text{-eps})^{1023}$ | $1.7977 \times 10^{+308}$ |
| emin | -1022 | |
| emax | 1023 | |

## Exceptions

The five possible exceptions are[7]:

**Invalid operation**
The result is mathematically undefined, e.g., the square root of a negative number. By default, returns qNaN.

**Division by zero**
An operation on finite operands gives an exact infinite result, e.g., 1/0 or log(0). By default, returns ±infinity.

**Overflow**
A result is too large to be represented correctly (i.e., its exponent with an unbounded exponent range would be larger than emax). By default, returns ±infinity for the round-to-nearest mode (and follows the rounding rules for the directed rounding modes).

**Underflow**
A result is very small (outside the normal range) and is inexact. By default, returns a subnormal or zero (following the rounding rules).

**Inexact**
The exact (i.e., unrounded) result is not representable exactly. By default, returns the correctly rounded result.

## Conversion to Decimal[8]

Conversions to and from a decimal character format are required for all formats.

Conversion to an external character sequence must be such that conversion back using round to even will recover the original number. There is no requirement to preserve the payload of a NaN or signaling NaN, and conversion from the external character sequence may turn a signaling NaN into a quiet NaN.

[7] Source: Wikipedia

[8] Source: Wikipedia

The original binary value will be preserved by converting to decimal and back again using:

5 decimal digits for half precision (16 bits)
9 decimal digits for single precision
17 decimal digits for double precision

# About This Document

## Document Revision History / Permission to Copy

Version numbers are not used to identify revisions to this document. Instead the date and the author's name is used. The document history is:

| **Date** | **Author** | |
|---|---|---|
| 2018-2019 | Harry H. Porter III | <From my RISC-V document> |
| 24 March 2021 | Harry H. Porter III | Document completed |
| 19 September 2022 | Harry H. Porter III | Minor corrections |

In the spirit of the open-source and free software movements, the author grants permission to freely copy and/or modify this document, with the following requirement:

> ***You must not alter this section, except to add to the revision history. You must append your date/name to the revision history.***

Any material lifted should be referenced.

## Corrections and Errors

Please contact the author if you find…

- Inaccurate information that you can correct
- Incomplete information that you can fill in
- Confusing text that needs to be reworded

***Thanks!***

# About the Author

Professor Harry H. Porter III teaches in the Department of Computer Science at Portland State University. He has produced several video courses, notably on the Theory of Computation. Recently he built a complete computer using the relay technology of the 1940s. The computer has eight general purpose 8 bit registers, a 16 bit program counter, and a complete instruction set, all housed in mahogany cabinets as shown. Porter also designed and constructed the Blitz System, a collection of software designed to support a university-level course on Operating Systems. Using the software, students implement a small, but complete, time-sliced, VM-based operating system kernel. Porter has habit of designing and implementing programming languages, the most recent being a language specifically targeted at kernel implementation.

Porter holds an Sc.B. from Brown University and a Ph.D. from the Oregon Graduate Center.

Porter lives in Portland, Oregon. When not trying to figure out how his computer works, he skis, hikes, travels, and spends time with his children building things.

Professor Porter's website: www.cecs.pdx.edu/~harry