IEEE 754 Floating-Point Format

Floating-Point Decimal Number

$$\begin{aligned}
-123456. \times 10^{-1} &= -12345.6 \times 10^{0} \\
&= -1234.56 \times 10^{1} \\
&= -123.456 \times 10^{2} \\
&= -12.3456 \times 10^{3} \\
&= -1.23456 \times 10^{4} \text{(normalised)} \\
&\approx -0.12345 \times 10^{5} \\
&\approx -0.01234 \times 10^{6}
\end{aligned}$$

Note

- There are different representations for the same number and there is no fixed position for the decimal point.
- Given a fixed number of digits, there may be a loss of precision.
- Three pieces of information represents a number: sign of the number, the significant value and the signed exponent of 10.

Note

- Given a fixed number of digits, the floating-point representation covers a wider range of values compared to a fixed-point representation.
- Naturally most of the numbers in the range do not have accurate representation.

Example

- The range of a fixed-point decimal system with six digits, of which two are after the decimal point, is 0.00 to 9999.99. This is same as non-negative integers.
- The range of a floating-point representation of the form $m.mmm \times 10^{ee}$ is $0.0, 0.001 \times 10^{0}$ to 9.999×10^{99}. Note that the radix-10 is implicit.

In a C Program

- Data of type `float` and `double` are represented as binary floating-point numbers.
- These are approximations of real numbers[a] like an `int`, an approximation of integers.

---

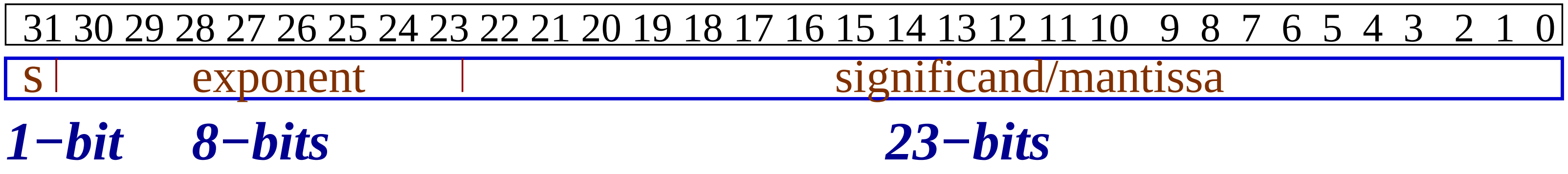
[a] In general a real number may have infinite information content. It cannot be stored in the computer memory and cannot be processed by the CPU.

IEEE 754 Standard

- Most of the binary floating-point representations follow the IEEE-754 standard.
- The data type `float` uses IEEE 32-bit single precision format and the data type `double` uses IEEE 64-bit double precision format.
- A floating-point constant is treated as a double precision number by GCC.

Bit Patterns

- There are 4294967296 patterns for any 32-bit format and 18446744073709551616 patterns for the 64-bit format.
- The number of representable `float` data is same as `int` data. But a wider range can be covered by a floating-point format due to non-uniform distribution of values over the range.

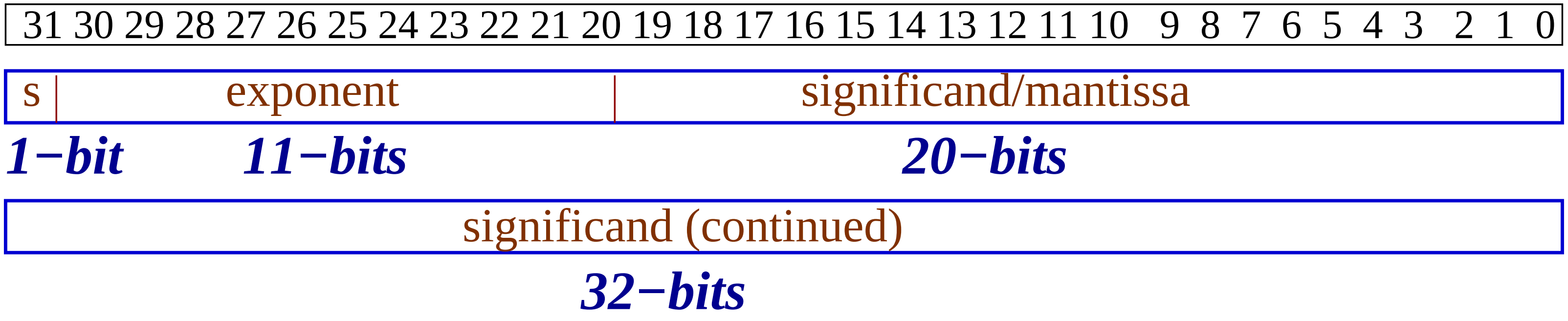


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

s | exponent | significand/mantissa

1−bit **8−bits** **23−bits**

Single Precession (32−bit)



31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

s | exponent | significand/mantissa

1−bit **11−bits** **20−bits**

significand (continued)

32−bits

Double Precession (64−bit)

Bit Pattern

```
#include <stdio.h>
void printFloatBits(float);
int main() // floatBits.c
{
    float x;
    printf("Enter a floating-point numbers: ")
    scanf("%f", &x);
    printf("Bits of %f are:\n", x);
    printFloatBits(x);
```

```
    putchar('\n');

    return 0;
}
void printBits(unsigned int a){
     static int flag = 0;
     if(flag != 32) {
        ++flag;
        printBits(a/2);
        printf("%d ", a%2);
        --flag;
```

```
            if(flag == 31 || flag == 23) putchar('
        }
}
void printFloatBits(float x){
        unsigned int *iP = (unsigned int *)&x;
        printBits(*iP);
}
```

Float Bit Pattern

`float` Data	Bit Pattern
1.0	0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-1.0	1 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1.7	0 0 1 1 1 1 1 1 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0
2.0×10^{-38}	0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1 1 0 0 0 1 1 1 1 1 0 1 1 1 0 1
2.0×10^{-39}	0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0 0 1 1 0 0 0 0

Interpretation of Bits

- The most significant bit indicates the sign of the number - one is negative and zero is positive.
- The next eight bits (11 in case of double precision) store the value of the signed exponent of two ($2^{\mathrm{biasedExp}}$).
- Remaining 23 bits (52 in case of double precision) are for the significand (mantissa).

Types of Data

Data represented in this format are classified in five groups.

- Normalized numbers,
- Zeros,
- Subnormal(denormal) numbers,
- Infinity and not-a-number (nan).

Single Precision Data: Interpretation

Single Precision		Data Type
Exponent	Significand	
0	0	± 0
0	nonzero	± subnormal number
1 - 254	anything	± normalized number
255	0	$\pm\infty$
255	nonzero	NaN (not a number)

Double Precision Data

Double Precision		Data Type
Exponent	Significand	
0	0	± 0
0	nonzero	± subnormal number
1 - 2046	anything	± normalized number
2047	0	$\pm\infty$
2047	nonzero	NaN (not a number)

Different Types of `float`

Not a number: `signaling nan`	0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
Not a number: `quiet nan`	0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
Infinity: `inf`	0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Largest Normal: `3.402823e+38`	0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Smallest Normal: `1.175494e-38`	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Different Types of `float`

Smallest Normal: `1.175494e-38`

0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Largest De-normal: `1.175494e-38`

0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Smallest De-normal: `1.401298e-45`

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Zero: `0.000000e+00`

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Single Precision Normalized Number

Let the sign bit (31) be s, the exponent (30-23) be e and the mantissa (significand or fraction) (22-0) be m. The valid range of the exponents is 1 to 254 (if e is treated as an unsigned number).

- The actual exponent is biased by 127 to get e i.e. the actual value of the exponent is $e - 127$. This gives the range: $2^{1-127} = 2^{-126}$ to $2^{254-127} = 2^{127}$.

Single Precision Normalized Number

- The normalized significand is $1.m$ (binary dot). The binary point is before bit-22 and the 1 (one) is not present explicitly.
- The sign bit $s = 1$ for a −ve number it is 0 for a +ve number.
- The value of a normalized number is

$$\boxed{(-1)^s \times 1.m \times 2^{e-127}}$$

An Example

Consider the following 32-bit pattern

1 1011 0110 011 0000 0000 0000 0000 0000

The value is

$$\begin{aligned} & (-1)^1 \times 2^{10110110-01111111} \times 1.011 \\ = \; & -1.375 \times 2^{55} \\ = \; & -49539595901075456.0 \\ = \; & -4.9539595901075456 \times 10^{16} \end{aligned}$$

An Example

Consider the decimal number: +105.625. The equivalent binary representation is

$$\begin{aligned} & +1101001.101 \\ = & +1.101001101 \times 2^{6} \\ = & +1.101001101 \times 2^{133-127} \\ = & +1.101001101 \times 2^{10000101-01111111} \end{aligned}$$

In IEEE 754 format:

0 1000 0101 101 0011 0100 0000 0000 0000

An Example

Consider the decimal number: $+2.7$. The equivalent binary representation is

$$\begin{aligned} & +10.10\ 1100\ 1100\ 1100 \cdots \\ = & +1.010\ 1100\ 1100 \cdots \times 2^{1} \\ = & +1.010\ 1100\ 1100 \cdots \times 2^{128-127} \\ = & +1.010\ 1100 \cdots \times 2^{10000000-01111111} \end{aligned}$$

In IEEE 754 format (approximate):

$$\boxed{0\ 1000\ 0000\ 010\ 1100\ 1100\ 1100\ 1100\ 1101}$$

Range of Significand

The range of significand for a 32-bit number is 1.0 to $(2.0 - 2^{-23})$.

Count of Numbers

The count of floating point numbers x, $m \times 2^i \leq x < m \times 2^{i+1}$ is 2^{23}, where $-126 \leq i \leq 127$ and $1.0 \leq m \leq 2.0 - 2^{-23}$.

Count of Numbers

The count of floating point numbers within the ranges $[2^{-126}, 2^{-125})$, $\cdots$, $[\frac{1}{4}, \frac{1}{2})$, $[\frac{1}{2}, 1.0)$, $[1.0, 2.0)$, $[2.0, 4.0)$, $\cdots$, $[1024.0, 2048.0)$, $\cdots$, $[2^{126}, 2^{127})$ etc are all equal.

In fact there are also 2^{23} numbers in the range $[2^{127}, \infty)$

Single Precision Subnormal Number

The interpretation of a subnormal[a] number is different. The content of the exponent part (e) is zero and the significand part (m) is non-zero. The value of a subnormal number is

$$\boxed{(-1)^s \times 0.m \times 2^{-126}}$$

There is no implicit one in the significand.

---

[a] This was also know as denormal numbers.

Note

- The smallest magnitude of a normalized number in single precision is $\pm$ 0000 0001 000 0000 0000 0000 0000 0000, whose value is 1.0×2^{-126}.
- The largest magnitude of a normalized number in single precision is $\pm$ 1111 1110 111 1111 1111 1111 1111 1111, whose value is $1.99999988 \times 2^{127} \approx 3.403 \times 10^{38}$.

Note

- The smallest magnitude of a subnormal number in single precision is $\pm$ 0000 0000 000 0000 0000 0000 0000 0001, whose value is $2^{-126+(-23)} = 2^{-149}$.
- The largest magnitude of a subnormal number in single precision is $\pm$ 0000 0000 111 1111 1111 1111 1111 1111, whose value is $0.99999988 \times 2^{-126}$.

Note

- The smallest subnormal 2^{-149} is closer to zero.
- The largest subnormal $0.99999988 \times 2^{-126}$ is closer to the smallest normalized number 1.0×2^{-126}.

Note

Due to the presence of the subnormal numbers, there are 2^{23} numbers within the range $[0.0, 1.0 \times 2^{-126})$.

Note

Infinity:

∞: 1111 1111 000 0000 0000 0000 0000 0000

is greater than (as an unsigned integer) the largest normal number:

1111 1110 111 1111 1111 1111 1111 1111

Note

- The smallest difference between two normalized numbers is 2^{-149}. This is same as the difference between any two consecutive subnormal numbers.
- The largest difference between two consecutive normalized numbers is 2^{104}.

Non-uniform distribution

± Zeros

There are two zeros (±) in the IEEE representation, but testing their equality gives true.

```
#include <stdio.h>
int main() // twoZeros.c
{
    double a = 0.0, b = -0.0 ;

    printf("a: %f, b: %f\n", a, b) ;
    if(a == b) printf("Equal\n");
    else printf("Unequal\n");
    return 0;
}
```

```
$ cc -Wall twoZeros.c
$ a.out
a:  0.000000, b:  -0.000000
Equal
```

Largest $+1 = \infty$

The 32-bit pattern for infinity is

0 1111 1111 000 0000 0000 0000 0000 0000

The largest 32-bit normalized number is

0 1111 1110 111 1111 1111 1111 1111 1111

If we treat the largest normalized number as an `int` data and add one to it, we get ∞.

Largest $+1 = \infty$

```
#include <stdio.h>
int main() // infinity.c
{
    float f = 1.0/0.0 ;
    int *iP ;

    printf("f: %f\n", f);
    iP = (int *)&f;  --(*iP);
    printf("f: %f\n", f);

    return 0 ;
}
```

Largest $+1 = \infty$

```
$ cc -Wall infinity.c
$ ./a.out
f: inf
f: 340282346638528859811704183484516925440.00
```

Note

Infinity can be used in a computation e.g. we can compute $\tan^{-1}\infty$.

Note

```
#include <stdio.h>
#include <math.h>
int main() // infinity1.c
{
     float f ;
     f = 1.0/0.0 ;
     printf("atan(%f) = %f\n",f,atan(f));
     printf("1.0/%f = %f\n", f, 1.0/f) ;
     return 0;
}
```

$$\boxed{\tan^{-1}\infty = \pi/2 \text{ and } 1/\infty = 0}$$

```
$ cc -Wall infinity1.c
$ ./a.out
atan(inf) = 1.570796
1.0/inf = 0.000000
```

Note

The value infinity can be used in comparison. $+\infty$ is larger than any normalized or denormal number. On the other hand nan cannot be used for comparison.

NaN

There are two types of NaNs - quiet NaN and signaling NaN.

A few cases where we get NaN:

$0.0/0.0,\ \pm\infty/\pm\infty,\ 0\times\pm\infty, -\infty+\infty, \texttt{sqrt}(-1.0), \log(-1.0)$

https://en.wikipedia.org/wiki/NaN

NaN

```
#include <stdio.h>
#include <math.h>
int main() // nan.c
{
  printf("0.0/0.0: %f\n", 0.0/0.0);
  printf("inf/inf: %f\n", (1.0/0.0)/(1.0/0.0));
  printf("0.0*inf: %f\n", 0.0*(1.0/0.0));
  printf("-inf + inf: %f\n", (-1.0/0.0) + (1.0/0.0));
  printf("sqrt(-1.0): %f\n", sqrt(-1.0));
  printf("log(-1.0): %f\n", log(-1.0));
  return 0;
}
```

NaN

```
$ cc -Wall nan.c -lm
$ a.out
0.0/0.0: -nan
inf/inf: -nan
0.0*inf: -nan
-inf + inf: -nan
sqrt(-1.0): -nan
log(-1.0): nan
$
```

A Few Programs

`int isInfinity(float)`

```
int isInfinity(float x){ // differentFloatType.c
    int *xP, ess;
    xP = (int *) &x;
    ess = *xP;
    ess = ((ess & 0x7F800000) >> 23); // exponent
    if(ess != 255) return 0;
    ess = *xP;
    ess &= 0x007FFFFF;       // significand
    if(ess != 0) return 0;
    ess = *xP >> 31;         // sign
    if(ess) return -1; return 1;
}
```

`int isNaN(float)`

It is a similar function where `if(ess != 0) return 0;` is replaced by `if(ess == 0) return 0;`.