IEEE 754 Floating-Point Format
Floating-Point Decimal Number

\[-12345.6 \times 10^{-1} = 12345.6 \times 10^{0}\]
\[= 1234.56 \times 10^{1}\]
\[= 123.456 \times 10^{2}\]
\[= 12.3456 \times 10^{3}\]
\[= 1.23456 \times 10^{4} (\text{normalised})\]
\[\approx 0.12345 \times 10^{5}\]
\[\approx 0.01234 \times 10^{6}\]
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.
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.

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 \( \times 10^{99} \). 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 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 \texttt{float} uses IEEE 32-bit single precision format and the data type \texttt{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.
Single Precision (32-bit)

Double Precision (64-bit)
Bit Pattern

```c
#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);
    printBits(*iP);
}
## Float Bit Pattern

<table>
<thead>
<tr>
<th>float Data</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>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 0</td>
</tr>
<tr>
<td>-1.0</td>
<td>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 0</td>
</tr>
<tr>
<td>1.7</td>
<td>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 1 1 0 1 1 0 1</td>
</tr>
<tr>
<td>2.0 × 10⁻³⁸</td>
<td>0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 1 0 1 1 1 0 1</td>
</tr>
<tr>
<td>2.0 × 10⁻³⁹</td>
<td>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 1 1 0 0 1 1 0 0 0</td>
</tr>
</tbody>
</table>
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 precession) store the value of the signed exponent of two ($2^{\text{biasedExp}}$).
- Remaining 23 bits (52 in case of double precession) 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).
There are two types of NaNs - quiet NaN and signaling NaN.
A few cases where we get NaN:
0.0/0.0, ±∞/±∞, 0 × ±∞, −∞ + ∞, sqrt(−1.0), log(−1.0)
#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;
}
$ 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
$
### Single Precision Data: Interpretation

<table>
<thead>
<tr>
<th>Single Precision</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exponent</strong></td>
<td><strong>Significand</strong></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
</tr>
<tr>
<td>1 - 254</td>
<td>anything</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
</tr>
</tbody>
</table>
### Double Precession Data

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>±0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>± subnormal number</td>
</tr>
<tr>
<td>1 - 2046</td>
<td>anything</td>
<td>± normalized number</td>
</tr>
<tr>
<td>2047</td>
<td>0</td>
<td>±∞</td>
</tr>
<tr>
<td>2047</td>
<td>nonzero</td>
<td>NaN (not a number)</td>
</tr>
</tbody>
</table>
Different Types of float
<table>
<thead>
<tr>
<th>Not a number: signaling nan</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Not a number: quiet nan</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 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</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Infinity: inf</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Largest Normal: $3.402823e+38$</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Smallest Normal: $1.175494e-38$</th>
</tr>
</thead>
</table>
| 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 0 0
### Different Types of float

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest Normal</td>
<td>1.175494e-38</td>
<td>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</td>
</tr>
<tr>
<td>Largest De-normal</td>
<td>1.175494e-38</td>
<td>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</td>
</tr>
<tr>
<td>Smallest De-normal</td>
<td>1.401298e-45</td>
<td>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</td>
</tr>
<tr>
<td>Zero</td>
<td>0.000000e+00</td>
<td>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</td>
</tr>
</tbody>
</table>
Single Precession 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 $2^{254-127} = 2^{127}$. 

Lect 15

Goutam Biswas
Single Precession 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 is zero $(0)$ for a $+ve$ number.
- The value of a normalized number is
  $$(-1)^s \times 1.m \times 2^{e-127}$$
An Example

Consider the following 32-bit pattern

\[ 1 \ 1011 \ 0110 \ 011 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \]

The value is

\[ (-1)^1 \times 2^{10110110-01111111} \times 1.011 \]

\[ = -1.375 \times 2^{55} \]

\[ = -49539595901075456.0 \]

\[ = -4.9539595901075456 \times 10^{16} \]
An Example

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

\[+1101001.101\]

\[= +1.101001101 \times 2^6\]

\[= +1.101001101 \times 2^{133-127}\]

\[= +1.101001101 \times 2^{10000101-01111111}\]

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

\[ +10.10110011001100 \ldots \]

\[ = +1.01011001100\ldots \times 2^1 \]

\[ = +1.01011001100\ldots \times 2^{128-127} \]

\[ = +1.0101100\ldots \times 2^{10000000-01111111} \]

In IEEE 754 format (approximate):

\[ 01000000001011001100110011001101 \]
The range of \textit{significand} for a 32-bit number is 1.0 to \((2.0 - 2^{-23})\).
The count of floating point numbers $x$, $m \times 2^i \leq x < m \times 2^{i+1}$ is $2^{23}$, where $-126 \leq i \leq 126$ and $1.0 \leq m \leq 2.0 - 2^{-23}$. 
Count of Numbers

The count of floating point numbers within the ranges $[2^{-126}, 2^{-125})$, $[2^{-125}, 2^{-124})$, $\cdots$, $[1, 1.25)$, $[1.25, 2)$, $[2, 4)$, $\cdots$, $[1024, 2048)$, $\cdots$, $[2^{126}, 2^{127})$ etc are all equal. In fact there are also $2^{23}$ numbers in the range $[2^{127}, \infty)$.
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

\[
(-1)^s \times 0.m \times 2^{-126}
\]

There is no implicit one in the significand.

\(^a\)This was also known 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 \( 1.0 \times 2^{-126} \).

- The largest magnitude of a normalized number in single precision is
  \( \pm \ 1111 \ 1110 \ 111 \ 1111 \ 1111 \ 1111 \ 1111 \),
  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
  \[ \pm 0000 \ 0000 \ 000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001, \]
  whose value is \[ 2^{-126} + (-23) = 2^{-149}. \]

- The largest magnitude of a subnormal number in single precision is
  \[ \pm 0000 \ 0000 \ 111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111, \]
  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 $1.0 \times 2^{-126}$. 
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:

\[ \infty: 1111\,1111\,000\,0000\,0000\,0000\,0000\,0000 \]

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
There are two zeros ($\pm$) in the IEEE representation, but testing their equality gives true.
```c
#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
The 32-bit pattern for infinity is
\[0 \ 1111 \ 1111 \ 000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000\]
The largest 32-bit normalized number is
\[0 \ 1111 \ 1110 \ 111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111\]
If we treat the largest normalized number as an int data and add one to it, we get \(\infty\).
```c
#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 ;
}
```
\[ \text{Largest} + 1 = \infty \]

\[
\text{cc -Wall infinity.c}
\]
\[
\text{./a.out}
\]
\[
f: \text{inf}
\]
\[
f: 340282346638528859811704183484516925440.00
\]
**Note**

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

```c
#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;
}
```
\[ \tan^{-1} \infty = \frac{\pi}{2} \quad \text{and} \quad 1/\infty = 0 \]

$\texttt{cc -Wall infinity1.c}$

$\texttt{./a.out}$

atan(inf) = 1.570796
1.0/inf = 0.000000
Note

The value \texttt{infinity} can be used in comparison. \( +\infty \) is larger than any normalized or denormal number. On the other hand \texttt{nan} cannot be used for comparison.
A Few Programs
```c
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 &= 0x007FFFFFFF; // 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;.