- Two’s Complement Representation
- Signed arithmetic in C
- Floating-point arithmetic
- Fixed-point arithmetic in C99: ISO TR 18037 (may become an ISO standard)
- Implementing fixed-point arithmetic using normal integers in C
Recall that nonnegative integer numbers are represented by an \( n \)-tuple 
\[
(x_{n-1}, x_{n-2}, \ldots, x_1, x_0)
\]
where each \( x_i \in \{0, 1\} \) and the tuple represents the value

\[
X = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ldots + x_12 + x_0 = \sum_{i=0}^{n-1} x_i2^i
\]

Arithmetic with unsigned binary numbers is performed in the same way we do arithmetic on numbers in base 10.

For example, with \( n = 8 \), adding \( X = 12 \) and \( Y = 9 \) gives \( S = X + Y \) as follows

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
+ & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\hline
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0
\end{array}
\]

12

9

21
If we assume instead that $n = 4$, an arithmetic overflow is detected when there is a carry out from the adding of $x_3$ and $y_3$, and only the incorrect result 5 is retained in the four bits.

\[
\begin{array}{cccc}
1 & 1 & 0 & 0 \\
+ & 1 & 0 & 0 & 1 \\
\hline
1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

Recall C performs unsigned integer arithmetic performed modulo $2^n$, so by definition there cannot be an overflow.

But a program can of course think it is an overflow since the machine gets the wrong result!

To discover overflow in C/C++ we can simply test whether the resulting sum is smaller than either of the operands.
Multiplication and Divide by a Power of Two

- An unsigned number can be multiplied by $2^k$ by shifting it $k$ positions to the left and inserting $k$ zeroes from the right.
- For instance $101_2 \times 2^3 = 101000_2$.
- An **unsigned number** can be divided by $2^k$ by shifting it to the right $k$ positions and inserting zeroes from the left.
- In general it does not work for signed numbers! See slides below.
- Shifting is much faster than division, but should generally be left to compilers in order to preserve the readability of the source code.
- However, if the programmer knows the divisor is a power of two, but the compiler cannot figure that out it can be worthwhile to use shift.
- Check the assembler code to see what the compiler produced!
- Compile with -S to get assembler code.
There are three main ways to represent signed integers using $n$ bits.

- The most obvious might be to use one bit as a sign bit and $n - 1$ bits for an unsigned number.
- This method is called the **signed-magnitude method** and is problematic.
- For instance there are two different zeroes, and the operands must be compared before an operation can be performed.
- In $X + Y$ where $Y$ is negative, depending on the magnitudes of $X$ and $Y$, either $X - |Y|$ or $- (|Y| - X)$ should be computed.
A better representation called **radix complement**, or **two’s complement** for binary systems,

This avoids both of the two previous problems.

A third method called **diminished radix complement**, or **ones’s complement** for binary systems, is possible but rarely used.

Note that the C standard does not specify how signed integers are represented, but it is almost always safe to assume the two's complement representation.
In the two’s complement representation, nonnegative numbers are represented exactly as normal binary numbers.

To represent the negative $-Y$ the number $2^n - Y$ is stored in the $n$ bits instead.

Recall, for instance in a four bit signed integer, $-5$ is represented by

$$2^4 - 5 = 10000_2 - 0101_2 = 1111_2 + 1 - 0101_2 = 1010_2 + 1 = 1011_2$$

Also recall that the procedure to convert a negative number to its two’s complement representation is to subtract its absolute value from $2^n$.

$$2^n = 1 + \sum_{i=0}^{n-1} 2^i$$

is 1 plus the binary number represented by $n$ ones.

Subtracting a binary number $Y$, say $0101$, from $1111$ gives the bitwise negation of $Y$, i.e. a one becomes a zero and a zero becomes a one.

Hence, we calculate $2^n - Y$ by bitwise negation of $Y$ and add one, as we did above.
Negation and Subtraction

- We would expect \(-(-Y) = Y\), and negating \(-Y\) correctly gives \(Y\):
  \[ 2^n - (2^n - Y) = Y \]

- To compute \(X - Y\), the correct result is given by \(X + (-Y)\), without having to consider the relative magnitudes of \(X\) and \(Y\):
  \[ X + (2^n - Y) \]

- If \(X > Y\), the result should be \(X - Y\), which is also what is stored since \(2^n\) is discarded when only \(n\) bits are stored.

- If on the other hand \(X < Y\), the value stored should be \(- (Y - X)\), i.e. \(2^n - (Y - X)\), which it also is.

- Finally, if \(X = Y\) only zeroes are stored since again \(2^n\) is discarded.
The smallest number that can be represented is $-2^{n-1}$ and the largest is $2^{n-1} - 1$.

Thus we cannot negate the smallest number.

For bytes, the range is $[-128, +127]$. The leftmost bit, at position $n - 1$ can be used to test the sign of a number, and its value is either zero or $-2^{n-1}$.

Therefore, to interpret what e.g. 1110 means we compute $-8 + 4 + 2 = -2$
Similarly as for unsigned integers, numbers on two’s complement representation can be multiplied by $2^k$ by shifting it left $k$ positions.

- **Arithmetic shift right** inserts $k$ copies the most significant bit (the ”sign bit”) from the left.

When we divide a negative number by $2^k$ using arithmetic shift right, if any nonzero bit is shifted out, we must add one to the result of the shift, which we will explain after introducing fixed point numbers.
We can also use the two’s complement representation to store rational numbers.

Sometimes the most suitable representation is that of a pair of integers, the numerator and denominator.

Another representation of either rational or irrational real numbers is to approximate using one part of the $n$-tuple for the integer and the other for the fraction.

The equivalent of the decimal point for base 10 is the **binary point**, and a binary digit to the right of it at position $k$ has a weight of $2^{-k}$.

For **fixed point numbers** the binary point is at a certain position, for instance we might divide a 16 bit storage item into 12 bits for the integer part and 4 bits for the fraction.
An advantage of fixed point numbers is that they can be used on simpler hardware without support for floating point numbers. The reason for this is that the processor’s normal add instruction can be used to add fixed point numbers and no special instruction is required. We must however then scale the results of multiplication and division.
Let us calculate \(11.8125 + -3.75\).

\[
1011.1101_2 = 2^3 + 2 + 1 + 2^{-1} + 2^{-2} + 2^{-4} = 11.8125_{10}
\]

How is \(-3.75_{10}\) represented?

\[
\begin{array}{cccccccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
- & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
= & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Adding these two numbers results in:

\[
\begin{array}{cccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
+ & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
\hline
= & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
\end{array}
\]

Adding these two numbers results in:

\[
\begin{array}{cccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
+ & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
\hline
= & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
\end{array}
\]

Jonas Skeppstedt (jonasskeppstedt.net)
Now we can explain why using arithmetic shift to perform a divide by a power of two not always works out correctly for negative numbers.

Consider calculating \(- \frac{1}{2} = 0\) with integers.

Let us write \(-1\) as a fixed point number:

\[
\begin{array}{cccccc}
1 & 1 & 1 & 1 & . & 0 & 0 & 0 & 0
\end{array} = -1.0
\]

With arithmetic shift by one we get the correct value \(-0.5\):

\[
\begin{array}{cccccc}
1 & 1 & 1 & 1 & . & 1 & 0 & 0 & 0
\end{array} = -0.5
\]

which we can write as \(-8 + 4 + 2 + 1 + 0.5 = -0.5\).

However, with integers the 0.5 is lost and the result becomes \(-1\).

The error comes from rounding down \(-0.5\) with truncation while integer division should round toward zero in C.

If any nonzero bit is shifted out, the rounding by truncation is wrong, and we must compensate by adding 1.

In general, the result of arithmetic shift of \(x\) by \(k\) is:

\[
x >> k = \lfloor \frac{x}{2^k} \rfloor
\]
Let us look at one more example and divide $-108 = -128 + 16 + 4$ by 16 using arithmetic shift by 4:

$$\begin{array}{cccccccccccccccc}
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & . & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} = -108.0$$

We get the correct quotient $-6.75$:

$$\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & . & 0 & 1 & 0 & 0 & 0 & 0 \\
\end{array} = -6.75$$

For simplicity we write $-6.75 = -8 + 1 + 0.25$. As we again can see, when truncating the nonzero fractional part we get the wrong quotient $-7$, and we must compensate by adding 1 to get $-108/16 = -6$. 
Floating point numbers have been used since Konrad Zuse’s Z1 computer which he patented 1936 when he was 26 years old.

Zuse introduced representations for the special values of infinity, not defined (now called not-a-number, or NaN), and imaginary, and his machines supported operations including addition, subtraction, multiplication, division, root extraction, squaring, calculation of the reciprocal, multiplication by $\pi$, 10, 2, and $1/2$ and the minimum and maximum of two numbers.

Imagine designing a computer that can do this — without using another computer!
A **floating point number** can be described by the following numbers:

- $s$: the sign
- $r$: the radix
- $e$: the exponent: $E_{\text{min}} \leq e \leq E_{\text{max}}$
- $p$: the precision (number of significand digits)
- $f_k$: the significand digits: $0 \leq f_k < r$

It has the value:

$$X = (-1)^s r^e (1 + \sum_{k=1}^{p} f_k r^{-k})$$
Due to differences in the floating point formats of different machines, it used to be very difficult to write numerical software which produces the same result on different platforms.

Under the leadership of Wilhelm Kahan, of UC Berkeley, the academia and industry standardized a format now known as IEEE 754, or IEC 60559 — the latter is the international standard.

Kahan received the 1989 Turing Award for this and other contributions to numerical computing.

Except for small embedded processor with no hardware support for floating point computing, almost every processor has supported IEEE 754 since the 1990’s.

As usual, software is behind and it is only more recently that programming languages have started to support this standard.

C99 is one of the few languages which support the standard.

Java has partial support for the standard.
The IEEE 754 was first standardized in 1985 and revised in 2008.

- It specifies three binary floating point formats, where the first two, `binary32 format` and `binary64 format`, correspond to the types `float` and `double` in Java (and C/C++) and the third, `binary128 format` corresponds to `long double` in C/C++.
- These were previously called the `single format`, `double format`, and `extended format`. 
The IEEE Single Format

- The Binary32, or Single Format, specifies a 32-bit floating point number with an eight bit exponent and a 23-bit mantissa, and a sign bit.

- There are three special values: ±∞ and NaN, or not-a-number.

- The latter is for example in C the result of \( \sqrt{-1.0} \) which attempts to calculate \( \sqrt{-1} \).

- They are represented by the following:

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent bits</th>
<th>Mantissa bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11111111</td>
<td>all zeroes</td>
<td>+∞</td>
</tr>
<tr>
<td>1</td>
<td>11111111</td>
<td>all zeroes</td>
<td>−∞</td>
</tr>
<tr>
<td>-</td>
<td>11111111</td>
<td>any bit a one</td>
<td>NaN</td>
</tr>
</tbody>
</table>

- The sign bit of a NaN is ignored.
There are both positive and negative zeroes, and a positive zero is represented by all 32 bits as zeroes.

For other values, the exponent, i.e. a signed integer, is stored in a formed called a **biased** representation.

If the exponent is \( e \), then \( e + 127 \) is stored in memory or a register.

The purpose of this representation is to simplify comparisons.
A zero or a value with a mantissa of the form $1.fraction$ is called a **normalized** value, and when possible values are always stored in this form.

To increase the precision of the mantissa, the initial (integer) one is not stored explicitly. It is called a **hidden one** and is implicit for exponents in the range $-126 \leq e \leq 127$.

For a number with the exponent $e = -127$ and any bit of the mantissa nonzero, the value is $(-1)^s \times 0.fraction \times 2^{-127}$ and the number is called a **subnormal value**.

Subnormal numbers and zero have zeroes in the exponent field.

For example the number 1.0 is stored as:

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent bits</th>
<th>Mantissa bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01111111</td>
<td>all zeroes</td>
<td>1.0</td>
</tr>
</tbody>
</table>
The other formats

- The IEEE Binary64, or Double Format, specifies a 64-bit floating point number with an 11 bit exponent field and 52 bit mantissa field.
- The IEEE Binary128 specifies a 128-bit floating point number with an 16 bit exponent field and 112 bit mantissa field.
- The earlier extended format was not specified to have 128 bit encodings and for example Intel instead used 80 bits.
Ranges

<table>
<thead>
<tr>
<th>Format</th>
<th>$E_{\text{min}}$</th>
<th>$E_{\text{max}}$</th>
<th>$N_{\text{min}}$</th>
<th>$\approx N_{\text{min}}$</th>
<th>$N_{\text{max}}$</th>
<th>$\approx N_{\text{max}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary32</td>
<td>$-126$</td>
<td>127</td>
<td>$2^{-126}$</td>
<td>$1.18 \times 10^{-38}$</td>
<td>$2^{128} - 2^{104}$</td>
<td>$3.40 \times 10^{38}$</td>
</tr>
<tr>
<td>Binary64</td>
<td>$-1022$</td>
<td>1023</td>
<td>$2^{-1022}$</td>
<td>$2.23 \times 10^{-308}$</td>
<td>$2^{1024} - 2^{971}$</td>
<td>$1.80 \times 10^{308}$</td>
</tr>
<tr>
<td>Binary128</td>
<td>$-32766$</td>
<td>32767</td>
<td>$2^{-32766}$</td>
<td>$2.82 \times 10^{-9864}$</td>
<td>$2^{32768} - 2^{32655}$</td>
<td>$1.42 \times 10^{9864}$</td>
</tr>
</tbody>
</table>
Machine Epsilon

- The gap between 1.0 and the next value is called the **machine epsilon**, $\epsilon$:
  \[ \epsilon = 0.00...01_2 = 2^{-(p-1)} \]

- The **unit in the least position**, of a floating point number $x$, commonly denoted $ulp(x)$, is:
  \[ ulp(x) = \epsilon \times 2^E \]

- The $\epsilon$ of each binary formats is:

<table>
<thead>
<tr>
<th>Format</th>
<th>Precision</th>
<th>Machine Epsilon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary32</td>
<td>$p = 24$</td>
<td>$\epsilon = 2^{-23} \approx 1.2 \times 10^{-7}$</td>
</tr>
<tr>
<td>Binary64</td>
<td>$p = 53$</td>
<td>$\epsilon = 2^{-52} \approx 2.2 \times 10^{-16}$</td>
</tr>
<tr>
<td>Binary128</td>
<td>$p = 113$</td>
<td>$\epsilon = 2^{-112} \approx 1.9 \times 10^{-34}$</td>
</tr>
</tbody>
</table>

- Except for subnormal numbers, a computed floating point result is **guaranteed** by the IEEE Standard to be exact within a factor of $(1 + \epsilon)$. 
Consider a positive number \( x = 1.b_1b_2...b_{p-1}b_p b_{p+1}... \times 2^E \) which cannot be represented exactly as a floating point number with precision \( p \).

We define \( x_- \) as the truncation of \( b_p b_{p+1}... \) from \( x \) i.e. \( x_- = 1.b_1b_2...b_{p-1} \times 2^E \), and we define \( x_+ \) as \( x_- + ulp(x) \). If \( x > N_{max} \) then \( x_- = N_{max} \) and \( x_+ = \infty \).

For negative numbers \( x_- \) and \( x_+ \) are defined in the opposite way.

Numbers which cannot be represented exactly are rounded according to one of the following four rounding modes, expressed as the function \( round(x) \).

- For round down \( round(x) = x_- \).
- For round up \( round(x) = x_+ \).
- These are sometimes called round towards \(-\infty\) and \( \infty \), respectively.
- For round toward zero \( round(x) = x_- \) for \( x > 0 \), and \( x_+ \) for \( x < 0 \).
The default rounding mode of the IEEE Standard, and hence also of C and Java, is **round to nearest**.

- If $x_- = N_{max}$ and $b_p = 1$ then $\text{round}(x) = \infty$, and otherwise $\text{round}(x)$ is the nearest to $x$ of $x_-$ and $x_+$.
- If they are equally near, the standard specifies that the one with the least significant bit equal to zero is chosen.
Guarantees

- The IEEE 754 Standard requires that systems perform all rounding according to this standard.

- By system is here meant the hardware chip performing basic arithmetic, the library which implements mathematical functions, and the compiler which translates the source code of a program to machine code.

- However, as a rule, normally you should never compare two floating point numbers for equality.
After correctness, the most basic rule in all forms of optimization is to make the common case fast, be it related to computers or not.

Suppose we want to compute the Euclidean norm of a vector:

\[ \|x\| = \sqrt{\sum_{i=1}^{n} x_i^2} \, . \]

Even though \( \|x\| \) itself can be represented, rounded or not, as a finite floating point number, a straightforward implementation can go wrong since computing \( x_i^2 \) may result in either an overflow or an underflow and adding the terms may also result in an overflow.

With knowledge about the input data, we might know that no overflow or underflow can occur, however, if we e.g. are implementing a robust and secure library we must take such values into account.
The IEEE standard defines **five exceptions** including **overflow** and **underflow**.

An example: $2^{-127}/4 = ?$ (not underflow!)

For each type of exception, there is a boolean flag in the processor which is automatically set when an exception occurs.

The flags can also be set by software such as in the math functions.

Processors can be programmed to jump to a trap handler when the exception occurs or to simply proceed.

The default behavior of a system which supports the IEEE standard is to proceed and not taking the trap.
To write a robust library routine to compute e.g. the Euclidean norm, we can either use
- a slower algorithm which always works, or
- a fast and check if an exception occurred, and only then use the slower algorithm.

The latter approach is an example of making the common case fast.
Other Exceptions

- The three other exceptions are
  - division by zero,
  - invalid operation, and
  - inexact.

- The result of an operation for which an exception occurs can be thought of as consisting of two parts:
  - the numerical value, and
  - a flag being set.

- For overflow and underflow, the value depends on the rounding mode.

- For a division by zero operation, say $x/0$, both the exception type and the value depends on $x$.

- If $x$ is finite and $x \neq 0$ the value is $\pm \infty$ and the exception type is division by zero, while if $x = 0$ the value is NaN, and the exception type is invalid operation.
Using $\infty$ and NaN

Using $\infty$ in operations may or may not raise an exception depending on whether the operation makes mathematical sense.

$\infty/0$, $0/\infty$, $\infty + \infty$, $\infty \times \infty$ or $x - \infty$ with a finite $x$ raise no exception.

However, $\infty - \infty$, $\infty \times 0$, and $\infty/\infty$ raise the invalid operation exception and produce the value NaN as result.

Any operation with a NaN operand always results in a NaN value.

Likewise any comparison with a NaN operand results in the false value.

The last exception type, inexact is raised whenever the result of a floating point operation is rounded.

Recall, the default behaviour at an exception is to set a bit in a status register and proceed.
Consider the representation of special values again:

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent bits</th>
<th>Mantissa bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11111111</td>
<td>all zeroes</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>1</td>
<td>11111111</td>
<td>all zeroes</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>-</td>
<td>11111111</td>
<td>any bit a one</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Except for NaN, the bits for $-\infty$ is the smallest negative integer and $\infty$ is the largest signed integer.

If our input contains no NaN we can thus use integer comparisons to sort floating pointer numbers.

This becomes somewhat faster than floating point compare.

Note: avoid two floating point instructions which will be executed if you simply subtract the floating point numbers:

- first the subtract, then
- second the convert floating point to integer.
IEEE 754 is now known as IEC 60559:1989.

If the compiler defines `__STDC_IEC_559__`, then the `float` type is the IEEE single-precision format and `double` is the IEEE double-precision format.

The are also optional complex and imaginary data types in C99 and if the compiler defines `__STDC_IEC_559__`, it indicates that these types are supported. They become available when including `<complex.h>`.

Implementing complex multiplication and division is not as trivial as might first be expected.

When the programmer uses `#pragma STDC CX_LIMITED_RANGE ON`, then the compiler is free to use the obvious math formulas.
The Standard does not specify what happens when an overflow occurs. Usually it is simply ignored and (wrong) calculations continue. Eg for + this of course is the result of using an add machine instruction which does not trigger exceptions, and simply taking the "result" of the add as the true sum.

Is this desired? There are in fact some algorithms (eg some hash-functions) which can create overflows but still work correctly.

-fsanitize=signed-integer-overflow makes gcc and clang generate code that checks for overflow.

-fsanitize=unsigned-integer-overflow not enabled by default since there is no such thing as "unsigned integer overflow" according to the C standard but may still be useful.
Despite ignoring overflows, the compiler must assure that the value of a variable must be in the range that can be represented by its type.

For example, with 8-bit chars, the range of a signed char $c$ is $-128..127$ if two-complement representation is used (which it almost always is).

Therefore, we can never get the value $+128$ in the variable $c$.

What if the compiler allocates $c$ in a 32-bit register?

The compiler is responsible for fixing this: If $c$ is register-allocated, what to do depends on the machine (eg extsb (extend signed byte) for Power).
The same performance penalty is introduced for short.

This is why C has a type int which is defined to be the fastest integer data type.

However, on 64-bit machines, int usually still are 32 bit despite 64 would have been faster, in order to use less memory.

Exactly the same performance problem exists for unsigned char and unsigned short.

What about long long? It is 64-bits and is efficient on 64-bit machines but slow on 32-bit machines which must emulate the 64-bit data type.