# Announcements

- Assignment 1 has been posted on **Crowdmark**.
  You can get access to **Crowdmark** through **myCourses**.
- TAs' Zoom links have been posted on myCourses.

  TAs will start office hours next week.

  Two TAs are responsible for each week's office hours and each assignment.

# Computer Numbers and Arithmetic

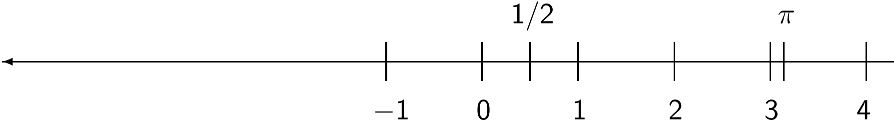
- 3 lectures
- References:
  - Overton, Numerical Computing with IEEE Floating Point Arithmetic, SIAM, 2004.
  - IEEE Computer Society, IEEE Standard for Floating-Point Arithmetic, 2019.
  - Cheney & Kincaid, Numerical Mathematics and Computing, Sections 1.1 and 1.3.

# Today's topics

- Binary and decimal representations of real numbers (Overton Chap 2).
- Computer representation of integers and non-integral real numbers (Overton Chap 3).
- IEEE floating point representation (Overton Chap 4)

# Classes of Real Numbers

All real numbers can be represented by a line:




The Real Line

$$\text{real numbers} \begin{cases} \text{rational numbers} \begin{cases} \text{integers} \\ \text{nonintegral fractions} \end{cases} \\ \text{irrational numbers} \end{cases}$$

# Classes of Real Numbers

- **Rational numbers**:
  all the real numbers which consist of a ratio of two integers.
  e.g., $2/1, 1/3, \ldots$
- **Irrational numbers:**
  Most real numbers are **not** rational, i.e. there is no way of writing them as the ratio of two integers.

  Familiar examples of irrational numbers are:
  $\sqrt{2}, \quad \pi, \quad e \equiv \lim_{n\to\infty}(1+\frac{1}{n})^n.$

## How to represent numbers?

Two basic systems:

- The **decimal**, or **base 10**, system requires 10 symbols, $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

- The **binary**, or **base 2**, system is convenient for electronic computers.
  Every number is represented as a string of $\mathbf{0}$'s and $\mathbf{1}$'s.

## Decimal & binary representations (expansions)

- **Integers:**
  The decimal and binary representation of **integers** requires an expansion in nonnegative powers of the base; e.g.

$$(71)_{10} = 7 \times 10 + 1$$

its **binary equivalent:** $(1000111)_2 =$

$$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

- **Non-integral rational numbers:**

  They have entries to the **right** of the decimal or binary point

# Representations of nonintegral rational numbers

- Both representations are **finite**:

$$\frac{11}{2} = (5.5)_{10} = 5 \times 1 + 5 \times \frac{1}{10},$$

$$\frac{11}{2} = (101.1)_2 = 1 \times 4 + 0 \times 2 + 1 \times 1 + 1 \times \frac{1}{2}$$

- The decimal is finite, but the **binary is infinitely long**

$$1/10 = (0.1)_{10}$$

$$\frac{1}{10} = (0.000110011001100\ldots)_2 \quad \textbf{(it is repeating)}.$$

$$= \frac{1}{16} + \frac{1}{32} + \frac{0}{64} + \frac{0}{128} + \frac{1}{256} + \frac{1}{512} + \cdots.$$

# Representations of nonintegral rational numbers, ctd

- Both representations are **infinite** and **repeating**:

$$1/3 = (0.333\ldots)_{10} = (0.010101\ldots)_2.$$

  If the representation of a **rational** number is *infinite*, it **must** be *repeating*.

- Is it possible that the decimal representation is infinite, but the binary representation is finite?

NO

## Representations of irrational numbers

- **Irrational** numbers:

  **Irrational** numbers always have **infinite, non-repeating** expansions. e.g.

$$
\begin{aligned}
\sqrt{2} &= (1.414213...)_{10}\,, \\
\pi &= (3.141592\ldots)_{10}, \\
e &= (2.71828182845\ldots)_{10}.
\end{aligned}
$$

## Conversion between binary & decimal

- **Binary** $\longrightarrow$ **decimal**:
  Easy. e.g. $(1001.11)_2$ is the decimal number

  $1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 9.75$

- **Decimal** $\longrightarrow$ **binary**:
  Convert the integer and fractional parts separately.
  e.g. if $x$ is a **decimal integer**, we want to find $a_0, a_1, \ldots, a_n$, all 0 or 1 such that $(x)_{10} = (a_n a_{n-1} \cdots a_0)_2$, i.e.,

  $$a_n \times 2^n + a_{n-1} \times 2^{n-1} + \cdots + a_1 \times 2 + a_0 \times 2^0 = x,$$

  Clearly dividing $x$ by 2 gives **remainder** $a_0$, leaving as **quotient**

  $$a_n \times 2^{n-1} + a_{n-1} \times 2^{n-2} + \cdots + a_1 \times 2^0,$$

  and so we can continue to find $a_1$ then $a_2$ etc.

# Converting between binary & decimal

**Q:** What is a similar approach for decimal fractions?

# Computer Representation of Numbers

- **Integers**
  1. **sign-and-modulus**
  2. **2's complement representation**

- **Non-integral real numbers**
  1. **Fixed point**
  2. **Floating point**

# Computer representation of integers

Typically, integers are stored using a 32-bit word.

1. <u>**sign-and-modulus**</u> — a simple approach.

Use 1 bit to represent the **sign**, and store the **binary** representation of the magnitude of the integer. e.g. decimal 71 is stored as the bitstring

| | |
|---|---|
| 0 | 00...01000111 |

**Q.** What is the **largest** magnitude which fits a 32-bit word?

| | |
|---|---|
| 0 | 111111...1111 |

The largest magnitude is $2^{31} - 1$

## Explanation of Ariane 5 Explosion (1996)

The rocket required the conversion of a 64-bit floating point number to a 16 bit signed integer.

The largest integer number which can fit a 16-bit word is

$$2^{15} - 1 = 32,767.$$

But the number was larger than $32,767$.

The conversion failed.

2. <u>**2's complement representation (CR)**</u>

– more convenient & used by most machines.

(i) The **nonnegative** integers 0 to $2^{31}-1$ are stored as before, e.g., 71 is stored as the bitstring $\boxed{000\ldots01000111}$

(ii) A **negative integer** $-x$, where $1 \le x \le 2^{31}$, is stored as the **positive integer** $2^{32}-x$.

e.g. $-71$ is stored as the bitstring $\boxed{111\ldots10111001}$.

**Converting $x$ to the 2's CR $2^{32}-x$ of $-x$:**

$$2^{32} - x = (2^{32} - 1 - x) + 1,$$
$$2^{32} - 1 = (111\ldots111)_2.$$

**Chang all 0 bits of $x$ to 1s, 1 bits to 0 and add 1.**

# 2's complement representation (CR)

**Q:** *What is the quickest way of deciding if a number is negative or nonnegative using 2's CR* **?**

**A:** Check the leading bit.
If it's 1, the number is negative, else the number is nonnegative.

Consider $y + (-x)$, where $1 \leq x \leq 2^{31}$, $0 \leq y \leq 2^{31} - 1$.

2's CR of $y$ is $y$ ;   2's CR of $-x$ is $2^{32} - x$

**Adding** these two **representations** gives

$$y + (2^{32} - x) = 2^{32} + y - x = 2^{32} - (x - y).$$

- If $y \geq x$, the LHS will not fit in a 32-bit word, and the **leading bit** can be dropped, giving the **correct result**, $y - x$.
- If $y < x$, the RHS is **already correct**, since it *represents* $-(x - y)$.

Thus, **no special hardware is needed for integer subtraction**. The addition hardware can be used, once $-x$ has been represented using 2's complement.

# Computer representation of nonintegral real numbers

**Real** numbers are approximately stored using the **binary** representation of the number.

Two possible methods: **fixed point** and **floating point**.

**Fixed point:**
the computer word is divided into **three fields**, one for each of:

- the **sign** of the number
- the number **before** the point
- the number **after** the point.

In a **32-bit word** with field widths of **1,15** and **16**, the number 11/2 would be stored as:

| | | |
|---|---|---|
| 0 | 000000000000101 | 1000000000000000 |

# Fixed point system

The fixed point system has a severe limitation on the **size** of the numbers to be stored.

Smallest magnitude:

| | | |
|---|---|---|
| 0 | 000000000000000 | 0000000000000001 |

$= 2^{-16}$

Largest magnitude:

| | | |
|---|---|---|
| 0 | 111111111111111 | 1111111111111111 |

$= 2^{15} - 2^{-16}$

A fix point system **is inadequate for most scientific computing.**

But it is fast and is used in some real-time applications.

# Explanation of the Patriot Missile Failure (1991)

- The missile system's internal clock stored the time as an integer value in **units of tenths of a second**.
- To produce the time in seconds, multiply the stored integer by $1/10$.
- The binary expansion of $1/10$ is $0.00011001100110011001100|11001100\cdots$
- The system stored the 23 bits after the binary point $0.00011001100110011001100$
- The chopping error is $(0.11001100\ldots)_2 \times 2^{-23} \approx 9.5 \times 10^{-8}$
- After 100 hours, the stored integer is $100 \times 60 \times 60 \times 10$.
- The error in time after 100 hours:

$$100 \times 60 \times 60 \times 10 \times 9.5 \times 10^{-8} \approx 0.34 \;\; \text{seconds}$$

# IEEE Floating Point Standard

Motivations:

- Floating point computation was in standard use by the mid 1950s.
- During the subsequent two decades, each computer manufacturer developed its own floating point system, leading to much inconsistency in how one program might behaviour on different machines.
- It was very difficult to write **portable software** that would work properly on all machines.

## IEEE Floating Point Standard

- **IEEE 754-1985**: a **binary** floating point standard. It was developed through the efforts of **William Kahan** & others.
- **IEEE 854-1987**: radix independent floating point arithmetic. It was motivated by decimal (radix-10) machines, and set the requirement for both binary and decimal floating point arithmetic in a common framework,
- **IEEE 754-2008**: a significant revision of IEEE 754-1985. It extended the previous standard where it was necessary, added decimal arithmetic and formats, and merged in IEEE 854-1987.
- **IEEE 754-2019**: a minor revision of IEEE 754-2008, incorporating mainly clarifications, defect fixes and new recommended operations.

In this course, the "IEEE standard" refers to the binary standard.

# IEEE Floating Point Standard

The standard defines:

- **arithmetic formats:** floating-point formats that can be used to represent floating-point operands or results for the operations of the standard.
- **interchange formats:** formats that have a specific fixed-width encodings (bit strings) that may be used to exchange floating-point data in an efficient and compact form
- **rounding rules:** properties to be satisfied when rounding numbers during arithmetic and conversions
- **operations:** arithmetic and other operations (such as trigonometric functions) on operands
- **exception handling:** indications of exceptional conditions (such as division by zero, overflow, etc.)

# IEEE Floating Point Formats

The IEEE standard has 3 binary floating point basic formats :

binary32, single format, or single precision;

binary64, double format, or double precision;

binary128, quadruple format, or quadruple precision

They have different numbers of bits to represent the significand and exponent.

## Base-2 Normalized Exponential Notation

In base-2 normalized exponential notation, any nonzero binary number $x$ can be written as

$$x = (-1)^s \times m \times 2^E, \quad \text{where } 1 \le m < 2,$$

where

- $s$ is 0 or 1.
- $m$ is called the **significand** or **mantissa**, and its binary expansion is

$$m = (b_0.b_1 b_2 b_3 \ldots)_2, \quad \text{with } b_0 = 1.$$

- $E$ is an **integer**, called the **exponent**.

# IEEE Single Format (binary32)

$$x = (-1)^s \times m \times 2^E, \quad m = (b_0.b_1b_2b_3\ldots)_2, \quad b_0 = 1,$$

**Single format** numbers use **32-bit words**.

| | | |
|---|---|---|
| $s$ | $a_1a_2a_3\ldots a_8$ | $b_1b_2b_3\ldots b_{23}$ |

- **sign field**: **1** bit for $s$.
- **exponent field**: **8** bits $a_1a_2a_3\ldots a_8$ for $E$, $E_{\min} \le E \le E_{\max}$.
- **significand field**: **23** bits for $m$.

$b_0$ is not stored, since it is known that $b_0 = 1$.
This idea is called **hidden bit normalization**.

The significand field is also referred to as the **fraction field**.

## IEEE Single Format, ctd

It may not be possible to store $x = \pm(b_0.b_1b_2b_3\ldots)_2 \times 2^E$ exactly with such a scheme, because

- either $E$ is outside the permissible range.
- or $b_{24}, b_{25}, \ldots$ are **not all zero**.

**Def.** A number is called a **(computer) floating point number** if it can be stored **exactly** this way, e.g.,

$$71 = (1.000111)_2 \times 2^6$$

can be represented by

| | | |
|---|---|---|
| 0 | $E = 6$ | 00011100000000000000000 |

.

If $x$ is not a floating point number, it must be **rounded** before it can be stored on the computer.

# Special Numbers

- $0$. Zero cannot be **normalized**.
- $-0$. $-0$ and $0$ are **two different representations for the same number**
- $\infty$. This allows e.g. $1.0/0.0 \to \infty$, instead of terminating with an **overflow** message.
- $-\infty$. $-\infty$ and $\infty$ represent **two very different numbers**.
- NaN, or **"Not a Number"**, and is an **error pattern**.
- Subnormal numbers (see later)

All **special numbers** are represented by **a special bit pattern** in the **exponent field**.

## IEEE Single Format (binary32) $\boxed{\pm \mid a_1 a_2 a_3 \ldots a_8 \mid b_1 b_2 b_3 \ldots b_{23}}$

| If exponent $a_1 \ldots a_8$ is | Then value is |
|---|---|
| $(00000000)_2 = (0)_{10}$ | $\pm(0.b_1..b_{23})_2 \times 2^{-126}$ |
| $(00000001)_2 = (1)_{10}$ | $\pm(1.b_1..b_{23})_2 \times 2^{-126}$ |
| $(00000010)_2 = (2)_{10}$ | $\pm(1.b_1..b_{23})_2 \times 2^{-125}$ |
| $\downarrow$ | $\downarrow$ |
| $(01111111)_2 = (127)_{10}$ | $\pm(1.b_1..b_{23})_2 \times 2^{0}$ |
| $(10000000)_2 = (128)_{10}$ | $\pm(1.b_1..b_{23})_2 \times 2^{1}$ |
| $\downarrow$ | $\downarrow$ |
| $(11111101)_2 = (253)_{10}$ | $\pm(1.b_1..b_{23})_2 \times 2^{126}$ |
| $(11111110)_2 = (254)_{10}$ | $\pm(1.b_1..b_{23})_2 \times 2^{127}$ |
| $(11111111)_2 = (255)_{10}$ | $\pm\infty$ if $b_1, \ldots, b_{23} = 0$;<br>$\mathrm{NaN}$ otherwise. |

# IEEE Single Format, ctd.

| | | |
|---|---|---|
| $\pm$ | $a_1 a_2 a_3 \dots a_8$ | $b_1 b_2 b_3 \dots b_{23}$ |

All lines *except* *the* *first* *and the* *last* refer to the **normal** numbers, i.e. **not special**.

The exponent representation uses **biased representation**: this bitstring is the binary representation of $E + 127$.

127 is the **exponent bias**. e.g. $1 = (1.000\dots0)_2 \times 2^0$ is stored as

| | | |
|---|---|---|
| 0 | 01111111 | 00000000000000000000000 |

**Exponent range** for normal numbers is 00000001 to 11111110 (1 to 254), representing **actual exponents**

$$\boxed{E_{\min} = -126 \ \text{ to } \ E_{\max} = 127}$$

# IEEE Single Format, ctd.

| | | |
|---|---|---|
| $\pm$ | $a_1 a_2 a_3 \dots a_8$ | $b_1 b_2 b_3 \dots b_{23}$ |

The **smallest normal positive number** is

$$(1.000\dots0)_2 \times 2^{-126} :$$

| | | |
|---|---|---|
| 0 | 00000001 | 00000000000000000000000 |

approximately $1.2 \times 10^{-38}$.

The **largest normal positive number** is

$$(1.111\dots1)_2 \times 2^{127} :$$

| | | |
|---|---|---|
| 0 | 11111110 | 11111111111111111111111 |

approximately $3.4 \times 10^{38}$.

## IEEE Single Format, ctd.

| $\pm$ | $a_1 a_2 a_3 \ldots a_8$ | $b_1 b_2 b_3 \ldots b_{23}$ |
|---|---|---|

*Last line*:

| If exponent $a_1 \ldots a_8$ is | Then value is |
|---|---|
| $(11111111)_2 = (255)_{10}$ | $\pm\infty$ if $b_1, \ldots, b_{23} = 0$;<br>$\mathrm{NaN}$ otherwise |

This shows an **exponent bitstring of all ones** is a special pattern for $\pm\infty$ or $\mathrm{NaN}$, depending on the value of the fraction.

NaN has two types:
quiet NaN (qNaN) if $b_1 = 1$, signaling NaN (sNaN) if $b_1 = 0$.

# IEEE Single Format, ctd.

| | | |
|---|---|---|
| $\pm$ | $a_1 a_2 a_3 \ldots a_8$ | $b_1 b_2 b_3 \ldots b_{23}$ |

*First line*

| | |
|---|---|
| $(00..00)_2 = (0)_{10}$ | $\pm(0.b_1..b_{23})_2 \times 2^{-126}$ |

**zero** requires a zero bitstring for the *exponent* field **as well as** for the *fraction*:

| | | |
|---|---|---|
| 0 | 00000000 | 00000000000000000000000 |

**Initial unstored bit is 0, not 1, in line 1**.

*First line*

| | |
|---|---|
| $(00..00)_2 = (0)_{10}$ | $\pm(0.b_1..b_{23})_2 \times 2^{-126}$ |

If **exponent** is **zero**, but **fraction** is **nonzero**, the number represented is **subnormal**.

Although $2^{-126}$ is the smallest normal positive number, we can represent smaller **subnormal** numbers.

e.g. $2^{-127} = (0.1)_2 \times 2^{-126}$:

| | | |
|---|---|---|
| 0 | 00000000 | 10000000000000000000000 |

and $2^{-149} = (0.0000\ldots01)_2 \times 2^{-126}$:

| | | |
|---|---|---|
| 0 | 00000000 | 00000000000000000000001 |

This is the smallest positive number we can store.

## IEEE Single Format, ctd.

Subnormal numbers cannot be normalized, as this would give exponents which do not fit.

Subnormal numbers are **less accurate** (less room for nonzero bits in the fraction). e.g.,

$$(1/10) \times 2^{-133} = (0.11001100\ldots)_2 \times 2^{-136}$$

is

| | | |
|---|---|---|
| 0 | 00000000 | 00000000001100110011001 |

# IEEE Single Format, ctd.

**Q:** (i) How is 2 represented **??**

| | | |
|---|---|---|
| 0 | 10000000 | 00000000000000000000000 |

(ii) What is the next smallest IEEE single precision number larger than 2 **??**

| | | |
|---|---|---|
| 0 | 10000000 | 00000000000000000000001 |

(iii) What is the gap between 2 and the first IEEE single precision number larger than 2?

$$2^{-23} \times 2 = 2^{-22}.$$

# IEEE Single Format, ctd.

**General Result:**
Let $x = m \times 2^E$ be a single format number. The **gap** between $x$ and the next smallest single format number larger than $x$ is

$$2^{-23} \times 2^E.$$

# Precision, Machine Epsilon

**Def. Precision** : The number of bits in the significand (including the hidden bit) is called the **precision** of the floating point system, denoted by $p$.

In the **single format** system, $p = 24$, the "single precision" is 24. Recall "single precision" also refers to the single format.

**Def. Machine Epsilon** : The gap between the number **1** and the **next larger** floating point number is called the **machine epsilon** of the floating point system, denoted by $\epsilon$.

In the **single format** system, the number after 1 is

$$b_0.b_1 \ldots b_{23} = 1.00000000000000000000001,$$

so $\quad \epsilon = 2^{-23}$.

# IEEE Double Format (binary64)

Each **double format** floating point number is stored in a **64-bit double word**.

Ideas are the same:
Field widths (1, 11 & 52) and exponent bias (1023) different.

$b_1, . ., b_{52}$ can be stored instead of $b_1, . ., b_{23}$.

| | | |
|---|---|---|
| $\pm$ | $a_1 a_2 a_3 \ldots a_{11}$ | $b_1 b_2 b_3 \ldots b_{52}$ |

## IEEE Double Format, ctd

| If exponent is $a_1..a_{11}$ | Then value is |
|---|---|
| $(000..0000)_2 = (0)_{10}$ | $\pm(0.b_1..b_{52})_2 \times 2^{-1022}$ |
| $(000..0001)_2 = (1)_{10}$ | $\pm(1.b_1..b_{52})_2 \times 2^{-1022}$ |
| $(000..0010)_2 = (2)_{10}$ | $\pm(1.b_1..b_{52})_2 \times 2^{-1021}$ |
| $\downarrow$ | $\downarrow$ |
| $(01..111)_2 = (1023)_{10}$ | $\pm(1.b_1..b_{52})_2 \times 2^{0}$ |
| $(10..000)_2 = (1024)_{10}$ | $\pm(1.b_1..b_{52})_2 \times 2^{1}$ |
| $\downarrow$ | $\downarrow$ |
| $(11..101)_2 = (2045)_{10}$ | $\pm(1.b_1..b_{52})_2 \times 2^{1022}$ |
| $(11..110)_2 = (2046)_{10}$ | $\pm(1.b_1..b_{52})_2 \times 2^{1023}$ |
| $(11..111)_2 = (2047)_{10}$ | $\pm\infty$ if $b_1,,b_{52} = 0$;<br>$\mathrm{NaN}$ otherwise |

# IEEE Quadruple Format (binary128)

The **quadruple format (binary128)** uses 128 bits, the field widths are (1, 15 & 112) and exponent bias is $2^{14} - 1 = 16383$.

| | | |
|---|---|---|
| $\pm$ | $a_1 a_2 a_3 \ldots a_{15}$ | $b_1 b_2 b_3 \ldots b_{112}$ |

## IEEE Quadruple Format (binary128), cod

| $\pm$ | $a_1 a_2 a_3 \ldots a_{15}$ | $b_1 b_2 b_3 \ldots b_{112}$ |
|---|---|---|

| If exponent is $a_1..a_{15}$ | Then value is |
|---|---|
| $(000..0000)_2 = (0)_{10}$ | $\pm(0.b_1..b_{112})_2 \times 2^{-16382}$ |
| $(000..0001)_2 = (1)_{10}$<br>$(000..0010)_2 = (2)_{10}$<br>$\downarrow$<br>$(01..111)_2 = (16383)_{10}$<br>$(10..000)_2 = (16384)_{10}$<br>$\downarrow$<br>$(11..101)_2 = (32765)_{10}$<br>$(11..110)_2 = (32766)_{10}$ | $\pm(1.b_1..b_{112})_2 \times 2^{-16382}$<br>$\pm(1.b_1..b_{112})_2 \times 2^{-16381}$<br>$\downarrow$<br>$\pm(1.b_1..b_{112})_2 \times 2^{0}$<br>$\pm(1.b_1..b_{112})_2 \times 2^{1}$<br>$\downarrow$<br>$\pm(1.b_1..b_{112})_2 \times 2^{16382}$<br>$\pm(1.b_1..b_{112})_2 \times 2^{16383}$ |
| $(11..111)_2 = (32767)_{10}$ | $\pm\infty$ if $b_1, \ldots, b_{112} = 0$;<br>NaN otherwise |

## Machine Precision, Epsilon of the 3 Formats

| | | |
|---|---|---|
| Single | $p = 24$ | $\epsilon = 2^{-23} \approx 1.2 \times 10^{-7}$ |
| Double | $p = 53$ | $\epsilon = 2^{-52} \approx 2.2 \times 10^{-16}$ |
| Quadruple | $p = 113$ | $\epsilon = 2^{-112} \approx 1.9 \times 10^{-34}$ |

We use **Floating Point Data** to include

$$\pm 0, \text{ subnormal \& normal FPNs, \& } \pm\infty \text{ and NaNs}$$

in a given format, e.g., single. These form a **finite** set.

$N_{\min}$: the minimum positive **normal** FPN;
$N_{\max}$: the maximum positive **normal** FPN;

A real number $x$ is in the **"normal range"** if

$$N_{\min} \leq |x| \leq N_{\max}$$

## Rounding, ctd.

**Question:**
Let $x$ be a real number and $|x| \leq N_{\max}$.

If $x$ is not a floating point number,

what are two obvious choices for the floating point **approximation** to $x$ **??**

$x_-$ : the closest FPN **less** than $x$,
$x_+$ : the closest FPN **greater** than $x$.

## Rounding, ctd.

Consider the single format.
Let $x$ be positive with

$$x = (b_0.b_1 b_2 \dots b_{23} b_{24} b_{25} \dots)_2 \times 2^E$$

$b_0 = 1$ (normal), or $b_0 = 0$, $E = -126$ (subnormal).
Then **discard** $b_{24}$, $b_{25}$, $\dots$ gives

$$x_- = (b_0.b_1 b_2 \dots b_{23})_2 \times 2^E.$$

An algorithm for $x_+$ is more complicated since it may involve some bit "carries".

$$x_+ = [(b_0.b_1 b_2 \dots b_{23})_2 + (0.00 \dots 1)_2] \times 2^E.$$

If $x$ is **negative**, the situation is reversed: $x_+$ is obtained by dropping bits $b_{24}$, $b_{25}$, etc.

## Correctly Rounded Values

The IEEE standard defines the **correctly rounded value of** $x$, $\mathrm{round}(x)$.

If $x$ **is** a floating point number, $\mathrm{round}(x) = x$.
Otherwise $\mathrm{round}(x)$ depends on the **rounding mode** in effect:

- **Round down:** $\mathrm{round}(x) = x_-$.
- **Round up:** $\mathrm{round}(x) = x_+$.
- **Round towards zero:** $\mathrm{round}(x)$ is either $x_-$ or $x_+$, whichever is between zero and $x$.
- **Round to nearest:** $\mathrm{round}(x)$ is either $x_-$ or $x_+$, whichever is nearer to $x$. In the case of a tie, the one with its **least significant bit (the last bit) equal to zero** is chosen.

  This rounding mode is almost always used.

## Correctly Rounded Values

If $x$ is **positive**, then $x_-$ is between zero and $x$,
so **round down** and **round towards zero** have the same effect;

**round towards zero** simply requires **truncating** the binary expansion, i.e. discarding bits.

**"Round"** with no qualification usually means **"round to nearest"**.

## Absolute Rounding Error

**Def.** The **absolute rounding error** associated with $x$:

$$|\mathrm{round}(x) - x|.$$

Its value depends on mode.

For all modes $\quad |\mathrm{round}(x) - x| < |x_+ - x_-|$.

Suppose $N_{\min} \le x \le N_{\max}$,

$$x = (b_0.b_1 b_2 \ldots b_{23} b_{24} b_{25} \ldots)_2 \times 2^E, \quad b_0 = 1$$

$$\text{IEEE } \textbf{single} \quad x_- = (b_0.b_1 b_2 \ldots b_{23})_2 \times 2^E$$

$$\text{IEEE } \textbf{single} \quad x_+ = x_- + (0.00 \ldots 01)_2 \times 2^E$$

So for **any** mode

$$|\mathrm{round}(x) - x| < 2^{-23} \times 2^E$$

# Absolute Rounding Error

In general for **any rounding mode**:

$$|\text{round}(x) - x| < \epsilon \times 2^E \quad (*)$$

**Q:** Does $(*)$ hold if $0 < x < N_{\min}$, i.e. $E = -126$, $b_0 = 0$ **??**

YES

## Relative Rounding Error, $x \neq 0$

The **relative rounding error** is defined by $|\delta|$,

$$\delta \equiv \frac{\mathrm{round}(x) - x}{x}.$$

Assuming $x$ is in the normal range,

$$x = \pm m \times 2^E, \qquad \text{where } m \geq 1,$$

so $\quad |x| \geq 2^E$.

Since $|\mathrm{round}(x) - x| < \epsilon \times 2^E$, for **all** rounding modes,

$$|\delta| < \frac{\epsilon \times 2^E}{2^E} = \epsilon.$$

## Relative Rounding Error, $x \neq 0$

The relative rounding error is bounded:

$$|\delta| < \epsilon$$

**Q:** Does the bound necessarily hold if $0 < |x| < N_{\min}$,
i.e. $E = -126$ and $b_0 = 0$ **??** Why **??**

**NO**.

Since $\delta = \dfrac{\mathrm{round}(x)}{x} - 1$, for any real $x$ in the **normal range**,

$$\boxed{\mathrm{round}(x) = x(1+\delta), \quad |\delta| < \epsilon}$$

## An Important Idea

$$\text{round}(x) = x(1+\delta),$$

so the **rounded value** of an arbitrary number $x$ in the **normal range** is **equal to** $x(1+\delta)$, where, regardless of the rounding mode,

$$|\delta| < \epsilon.$$

This is very important, because you can think of the stored value of $x$ as **not exact**, but as **exact within a factor** of $1+\epsilon$.

IEEE **single format numbers** are good to a factor of about $1 + 10^{-7}$, i.e., they **have about** 7 **accurate decimal digits**.

## Special Case of Round to Nearest

For **round to nearest**, the **absolute** rounding error can be **no more than <u>half</u> the gap between** $x_-$ **and** $x_+$.

In IEEE single, for all $|x| = (b_0.b_1 b_2 \ldots)_2 \times 2^E \leq N_{\max}$,

$$|\text{round}(x) - x| \leq 2^{-24} \times 2^E,$$

and in general

$$|\text{round}(x) - x| \leq \frac{1}{2}\epsilon \times 2^E.$$

For $x$ in the **normal range** (so $b_0 = 1$)

$$\text{round}(x) = x(1+\delta), \quad |\delta| \leq \frac{\frac{1}{2}\epsilon \times 2^E}{2^E} = \frac{1}{2}\epsilon.$$

# Recap

- IEEE floating point representation (Overton's Chap 4)

| Format | Sign | Exponent | Fraction |
|---|---|---|---|
| Single (binary32) | 1 | 8 | 23 |
| double (binary64) | 1 | 11 | 52 |
| quadruple (binary128) | 1 | 15 | 112 |

  **Concepts:** hidden bit, exponent bias, machine precision, machine epsilon

- Rounding (Overton's Chap 5)
  Four rounding modes: down, up, towards zero, to nearest

$$\frac{|\mathrm{round}(x) - x|}{|x|} \begin{cases} < \epsilon, & \text{all rounding modes} \\ \leq \frac{1}{2}\epsilon, & \text{round to nearest} \end{cases}$$

  where $N_{\min} \leq |x| \leq N_{\max}$.

## Today's topics

- Floating point operations (Overton's Chap 6)
- Exceptional situations (Overton's Chap 7)
- Floating point in C (Overton's Chap 10)

## Operations on Floating Point Numbers

The IEEE standard requires correctly rounded operations:

- correctly rounded basic arithmetic operations ($+$, $-$, $*$, $/$);
- correctly rounded remainder and square root operations;
- correctly rounded format conversions.

**Correctly rounded** means rounded to fit the destination of the result, using rounding mode in effect.

### IEEE Rule for Rounding

The exact result of an operation may **not** be a floating point number, e.g. the **multiplication** of two **24-bit** significands generally gives a **48-bit** significand.

The IEEE standard requires that the computed result be the **correctly** rounded value of the **exact** result

## IEEE Rule for a Floating Point Operation

Let $x$ and $y$ be floating point numbers, and let $\oplus,\ominus,\otimes,\oslash$ denote the **implementations** of $+,-,*,/$ on the computer.
The **IEEE rule** for the basic arithmetic operations is then precisely:

$$
\begin{aligned}
x \oplus y &= \mathrm{round}(x+y),\\
x \ominus y &= \mathrm{round}(x-y),\\
x \otimes y &= \mathrm{round}(x\times y),\\
x \oslash y &= \mathrm{round}(x/y).
\end{aligned}
$$

Therefore when $x+y$ is in the **normal range**,

$$\boxed{x \oplus y = (x+y)(1+\delta), \quad |\delta| < \epsilon}$$

for **all** rounding modes. Similarly for $\ominus$, $\otimes$ and $\oslash$.

(Note that $|\delta| \le \epsilon/2$ for **round to nearest**).

# Format Conversion

The IEEE standard requires support for correctly rounded format conversions:

- Conversion between floating point numbers.
- Conversion between floating point and integer formats.
- Rounding a floating point number to an integral value (not an integer format).
- Binary to decimal and decimal to binary conversion.

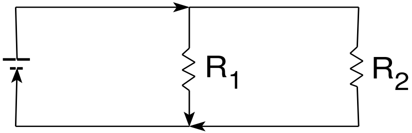
# Exceptional Situations

When a reasonable response to exceptional data is possible, it should be used.

<u>**Division by zero**</u>. Two **earlier** standard responses :

- Generate the **largest FPN** as the result.
  <u>Rationale</u>: user would notice the large number in the output and conclude something had gone wrong.
  **Disaster:** e.g. $2/0 - 1/0$ would then have a result of 0, which is **completely meaningless**. In general the user might **not even notice** that any error had taken place.
- Generate a **program interrupt**, e.g.
  **"fatal error — division by zero"**.
  The burden was on the programmer to make sure that division by zero would **never** occur.

## Example: computing the **total resistance**




The **total resistance** $T = \frac{1}{\frac{1}{R_1}+\frac{1}{R_2}}$.

What if $R_1 = 0$? If one resistor offers no resistance, **all** the current will flow through that and avoid the other; therefore, the total resistance in the circuit is **zero**.

The formula for $T$ also makes perfect sense **mathematically**:

$$T = \frac{1}{\frac{1}{0}+\frac{1}{R_2}} = \frac{1}{\infty+\frac{1}{R_2}} = \frac{1}{\infty} = 0.$$

## The IEEE FPS Solution

Why should a **programmer** have to worry about treating division by zero as an exceptional situation here?

In **IEEE floating point arithmetic**, **division by zero** does not generate an interrupt, but **gives an infinite result**, program execution continuing normally.

In the case of **the parallel resistance formula** this leads to a final **correct result** of $1/\infty = 0$, following the mathematical concepts **exactly**:

$$T = \frac{1}{\frac{1}{0} + \frac{1}{R_2}} = \frac{1}{\infty + \frac{1}{R_2}} = \frac{1}{\infty} = 0.$$

## Other uses of $\infty$

We used some of the following:

$$
\begin{aligned}
a > 0 \quad &: \quad a/0 \to \infty, \qquad a * \infty \to \infty, \\
a \text{ finite} \quad &: \quad a + \infty \to \infty, \qquad a - \infty \to -\infty, \\
&\quad\;\; a/\infty \to 0, \qquad \infty + \infty \to \infty.
\end{aligned}
$$

## Uses of NaN

- The operations $\infty * 0, \quad 0/0, \quad \infty/\infty, \quad \infty - \infty$ are indefinite. Computing any of these is called an **invalid operation**, and the IEEE standard sets the result to NaN.
- A real operation with a complex result, e.g., the square root of a negative number, produces NaN.
- **Almost all** arithmetic operations with at least one $\mathrm{NaN}$ operand **also** produce NaN.
- Whenever a $\mathrm{NaN}$ is discovered in the output, the programmer **knows something has gone wrong**. An $\infty$ in the output may or may not indicate an error, depending on the context.
- sNaN generates interruption while qNaN does not.
  The application decides if it generates qNaN or sNaN.
  For instance, GCC C compiler always generates qNaN unless explicitly specified to behave the other way around.

## Overflow and Underflow

**Overflow** is said to occur when

$$N_{\max} < |\text{ true result }| < \infty,$$

where $N_{\max}$ is the **largest** normal FPN.

Two **pre-IEEE** standard treatments:

(i) Set the result to $(\pm)$ $N_{\max}$, or
(ii) Interrupt with an **error message**.

In IEEE arithmetic, the standard response depends on the **rounding mode**.

## Overflow and Underflow, ctd

Suppose that the overflowed value is **positive**. Then

| rounding mode | result |
| --- | --- |
| **round up** | $\infty$ |
| **round down** | $N_{\max}$ |
| **round towards zero** | $N_{\max}$ |
| **round to nearest** | $\infty$ |

**Round to nearest** is the **default** rounding mode and any other choice may lead to very misleading final computational results.

## Overflow and Underflow, ctd

- **Underflow** is said to occur when

$$0 < |\ \text{true result}\ | < N_{\min},$$

  where $N_{\min}$ is the **smallest** normal floating point number.

- Historically the response was usually:

  **replace the result by zero**.

- In **IEEE standard**, the result may be a **subnormal** number instead of zero – allowing results **much smaller** than $N_{\min}$.
- But there may still be a significant loss of accuracy, since subnormal numbers have fewer bits of format.

## Floating Point in C

In C, the type **float** refers to a **single format** FP variable.

*Example*. **read** in a FPN, using the standard input routine `scanf`, and **print** it out again, using `printf`:

```
main ()      /* echo.c:  echo the input */
{
    float x;
    scanf("%f", &x);
    printf("x = %f", x);
}
```

The 2nd argument `&x` to `scanf` is the **address** of `x`.
The routine `scanf` needs to know **where** to store the value read.

The 2nd argument `x` to `printf` is the **value** of `x`.

The 1st argument `"%f"` to both routines is a **control string**.

## Using Different Output Formats in "printf"

The following result was for a Sun 4.
With input 0.66666666666666666666 :

```
#cc -o echoinput echo.c
#echoinput
0.66666666666666666666      (typed in)
x= 0.666667                 (printed out)
```

## Format Codes

The two standard **format codes** used for specifying floating point numbers in these control strings are:

- `%f`, for **fixed decimal format**
- `%e`, for **exponential decimal format**

The two formats have **identical** effects in `scanf`,
which can process input in a **fixed** decimal format (e.g. 0.666) or an **exponential** decimal format (e.g. `6.66e-1`),

but have different effects in `printf`.

## Using Different Output Formats in "printf"

| Output format | Output |
|---|---|
| `%f` | `0.666667` |
| `%e` | `6.666667e-01` |
| `%8.3f` | `0.667` |
| `%8.3e` | `6.667e-01` |
| `%20.15f` | `0.666666686534882` |
| `%20.15e` | `6.666666865348816e-01` |

- The input is rounded to about 6 or 7 digits of precision, so `%f` and `%e` print, **by default**, 6 digits after the decimal point.
- The next two lines print to **less** precision.
- In the last two lines about half the digits have **no significance**. Regardless of the **output format**, the floating point variables are **always** stored in the **IEEE formats**.

## Format Conversions: scanf and printf

- The scanf routine calls a decimal to binary conversion routine to convert the input decimal format to internal binary floating point representation;
- The printf routine calls a binary to decimal conversion routine to convert the binary floating point representation to the output decimal format.
- Both conversion routines use the rounding mode that is in effect to correctly round the results.

## A note on the %f format code

Using the `%f` format code is **NOT** a good idea, unless

it is known the numbers are neither too small nor too large.

e.g., if the input is `1.0e-10`, the output using `%f` is 0.000000.

In numerical computing, usually the `%e` format code is used.

## Double or Long Float

Double precision variables are declared in C using **double** or **long float**. But changing **float** to **double** in the previous program:

```
main ()      /* echo.c:  echo the input */
{ double x;
  scanf("%f",&x);
  printf("%e",x);
}
```

gives `-6.392091e-236`. **Q:** Why **??**

`scanf` reads the input and stores it in **single format** in the first half of the **double word** allocated to $x$, but when $x$ is **printed**, its value is read assuming it is **stored in double format**.

## **Double** or **Long Float**, ctd

- When `scanf` reads a **double precision** variable **we must use the format** `%lf` (for long float), so that it stores the result in **double** format.
- `printf` **expects** double precision, and single format variables are **automatically** converted to double before being passed to it.

  Since it always receives **long float** arguments, it treats `%e` and `%le` identically;
  likewise `%f` and `%lf`, `%g` and `%lg`.

## Story: Effect of output format on a parliament election

Parliamentary election in Schleswig-Holstein, Germany, April 5, 1992.

- In the elections, a party with less than 5.0% of the vote cannot be seated.
- It was announced the Greens had a cliff-hanging 5.0% the evening of the election.
- It was discovered after midnight that they really had only 4.97%. The printout had one digit after the decimal point, and the actual percentage was rounded to 5.0%.

## A program to "test" if $x$ is "zero"

```
main() /* loop1.c: generate small numbers*/
{ float x; int n;
  n = 0; x = 1;   /* x = 2^0 */
  while (x != 0){
    n++;
    x = x/2;     /* x = 2^(-n) */
    printf("\n n= %d  x=%e", n,x); }
}
```

Initializes $x$ to 1 and repeatedly divides by 2 until it rounds to 0.

## A program to "test" if $x$ is "zero", ctd

Thus $x$ becomes $1/2$, $1/4$, $1/8$, ..., through **subnormal** $2^{-127}$, $2^{-128}$, ... to **the smallest subnormal** $2^{-149}$.

The last value is 0, since $2^{-150}$ is **not** representable.

**Output (various machines with various compliers)**:

```
n= 1   x=5.000000e-01
n= 2   x=2.500000e-01
. . .
n= 149   x=1.401298e-45
n= 150   x=0.000000e+00
```

## Another "test" if $x$ is "zero"

```
main() /* loop2.c: this loop stops sooner*/
{ float x,y; int n;
  n = 0; y = 2; x = 1;      /* x = 2^0 */
  while (y != 1) {
    n++;
    x = x/2;       /* x = 2^(-n) */
    y = 1 + x;      /* y = 1 + 2^(-n) */
    printf("\n n= %d x= %e y= %e",n,x,y);
  }
}
```

**Initializes** $x$ to 1 and repeatedly divides by 2, **terminating** when $y = 1 + x$ is 1.

## Another "test" if $x$ is "zero"

This occurs **much sooner**, since $1 + 2^{-24}$ is **not** a floating point number, rounds to 1. Note $1 + 2^{-23}$ does **not** round to 1.

**Output (various machines with various compliers)**:

```
n= 1 x= 5.000000e-01 y= 1.500000e+00
  . . .
n= 23 x= 1.192093e-07 y= 1.000000e+00
n= 24 x= 5.960464e-08 y= 1.000000e+00
```

## Yet another "test" if $x$ is "zero"

Now instead of using the variable $y$, change the **test** `while (y != 1)` to `while (1 + x != 1)`:

```
n=0;  x=1;      /* x = 2^0 */
while(1 + x != 1){... /*same as before*/}
```

It stops at

- $n = 24$ on a PC using Visual Studio or online GBD, and Mac. It uses registers with **the single precision** $p = 24$
- $n = 53$ on a PC using Visual C++ It uses registers with **the double precision** $p = 24$
- $n = 64$ on a PC using gcc It uses registers with **the extended precision** $p = 64$.

**Conclusion**: Running the same program on different machines with different compliers may still give different results