Representations of Integer and Floating Points

Course: Introduction to Programming and Data Structures

Laltu Sardar

Institute for Advancing Intelligence (IAI),
TCG Centres for Research and Education in Science and Technology (TCG Crest)

August 20, 2024

Signed Integers: Representations

Storing Positive Numbers in 2's Complement

- Positive numbers are stored in binary.
- The first bit is the sign bit (0 for positive numbers).
- Example: Storing 18 in an 8-bit system:

$$18 \rightarrow 10010 \rightarrow 0001\ 0010$$

- The binary representation of 18 is `0001 0010`.

Storing Negative Numbers in 2's Complement

- Negative numbers are stored by:
  1. Converting the positive number to binary.
  2. Inverting the bits.
  3. Adding 1 to the result.
- Example: Storing -18 in an 8-bit system:

$$18 \rightarrow 0001\ 0010$$

$$\text{Invert: } 1110\ 1101$$

$$\text{Add 1: } 1110\ 1110$$

- The binary representation of -18 is $1110\ \ 1110$.

Conversion Table with 2's Complement

Decimal	Bin	Flipped Bits	2's Complement	-ve
0	0000 0000	1111 1111	0000 0000	0
1	0000 0001	1111 1110	1111 1111	-1
2	0000 0010	1111 1101	1111 1110	-2
64	0100 0000	1011 1111	1100 0000	-64
126	0111 1110	1000 0001	1000 0010	-126
127	0111 1111	1000 0000	1000 0001	-127
128	~~1000 0000~~	0111 1111	1000 0000	-128

Table: Conversion Table with 2's Complement

Adding Positive + Positive Numbers

- Adding 18 and 12:

$$18 \rightarrow 0001\ 0010$$

$$12 \rightarrow 0000\ 1100$$

$$\text{Sum: } 0001\ 1110 \quad (30 \text{ in decimal})$$

Adding Positive + Negative Numbers

- Adding 18 and -12:

$$18 \rightarrow 0001\ 0010$$

$$-12 \rightarrow 1111\ 0100$$

$$\text{Sum: } 0000\ 0110 \quad (6 \text{ in decimal})$$

Adding Negative + Negative Numbers

- Adding -18 and -12:

$$-18 \rightarrow 1110\ 1110$$

$$-12 \rightarrow 1111\ 0100$$

$$\text{Sum: } 1110\ 0010 \quad (-30 \text{ in decimal})$$

Overflow in Positive + Positive Addition

- Adding 70 and 70:

$$70 \rightarrow 0100\ 0110$$

$$70 \rightarrow 0100\ 0110$$

$$\text{Sum: } 1000\ 1100 \quad (\text{This is } -116, \text{overflow!})$$

Overflow in Negative + Negative Addition

- Adding -70 and -70:

$$-70 \rightarrow 1011\ 1010$$

$$-70 \rightarrow 1011\ 1010$$

$$\text{Sum: } 0111\ 0100 \quad (\text{This is } +116, \text{overflow!})$$

Overflow During Multiplication

- Multiplying 20 and 15:

$$20 \rightarrow 0001\ 0100$$

$$15 \rightarrow 0000\ 1111$$

- Result needs more than 8 bits:

$$20 \times 15 = 300 \rightarrow 100101100 \text{ (requires 9 bits)}$$

- Overflow occurs as the result cannot be stored in 8 bits.

How **Overflows** are handled in C?

- Signed int: Undefined
- Unsigned int: Undefined – Wrap around

Does Nothing– Keep as it is
Use Wisely

Floating Point Representation

Why Integers Are Not Sufficient?

- Integers can only represent whole numbers.
- Real-world applications require representation of fractional values, very large numbers, and very small numbers.
- Examples:
  - 3.14 (pi)
  - 0.000001 (small decimal)
  - 1.5 (fractional number)
- Integers cannot represent these values, hence the need for floating-point representation.

- Floating-point representation allows for a wide range of numbers.
- It balances between the range and precision needed in computations.

Main Idea Behind Floating Point Representation

- Floating-point numbers are represented using three components:
  - Sign bit
  - Exponent [Excess-N representation]
  - Significand (Mantissa)
- Similar to scientific notation:

$$\text{value} = (-1)^{\text{sign}} \times \text{base}^{\text{exponent}} \times \text{fraction}$$

- Trade-off between range and precision.

32-bit IEEE Floating Point Format

- 32-bit format, also known as single-precision.
- Components:
  - Sign bit: 1 bit
  - Exponent: 8 bits (with a bias of 127)
  - Mantissa: 23 bits (implied leading 1)
- Representation (Normal):

$$\text{value} = (-1)^{\text{sign}} \times 2^{(\text{exponent}+127)} \times (1.\text{mantissa})$$

- Computation (Normal):

$$\text{value} = (-1)^{\text{sign}} \times 2^{(\text{exponent}-127)} \times (1 + \text{fraction})$$

tcg crest
Inventing Harmonious Future

32-bit IEEE Floating Point Format

- Subnormal form: Represent numbers very close to zero, providing a "smooth" transition to zero, but at the cost of precision.
- Components:
  - Sign bit: 1 bit
  - Exponent: 0000 0000, is fixed at -126
  - Mantissa: 23 bits (implied leading 1)
- Computation (Normal):

$$\text{value} = (-1)^{\text{sign}} \times 2^{-126} \times (0 + \text{fraction})$$

Presenting -18.625

- We'll represent the number -18.625 in the IEEE 754 single-precision (32-bit) floating-point format.
- The process involves
  1. converting the number to binary,
  2. **normalizing** it,
  3. determining the sign bit, exponent, and mantissa, and
  4. combining these components into the final 32-bit representation.

Step 1: Converting the Number to Binary

- Convert the integer part (18) to binary:

$$18_{10} = 10010_2$$

- Convert the fractional part (0.625) to binary:

$$0.625_{10} = 0.101_2$$

- Combine both parts:

$$-18.625_{10} = -10010.101_2$$

Step 2: Normalize the Binary Number

- Normalize 10010.101_2 to the form $1.xxxxx \times 2^n$:

$$10010.101_2 = 1.0010101_2 \times 2^4$$

Step 3: Determine the Sign Bit

- The sign bit is ‘1‘ for negative numbers.
- For -18.625, the sign bit is:

$$\text{Sign bit} = 1$$

Step 4: Determine the Exponent

- The exponent n is 4.
- Add the bias (127 for single-precision):

$$\text{Exponent} = 4 + 127 = 131_{10} = 10000011_2$$

It is called

Excess-127 Representation

Excess to 127

Excess-127 Representation

- The exponent in single precision uses Excess-127.
- The stored exponent is calculated as:

$$E_{\text{stored}} = E_{\text{actual}} + 127$$

- This allows representation of both positive and negative exponents.

Example: Representation of $E_{\text{actual}} = 5$

- Given $E_{\text{actual}} = 5$:

$$E_{\text{stored}} = 5 + 127 = 132$$

- Binary representation of 132:

$$132_{10} = 10000100_2$$

- The exponent field in IEEE 754 format will be 10000100.

Example: Representation of $E_{\text{actual}} = -3$

- Given $E_{\text{actual}} = -3$:

$$E_{\text{stored}} = -3 + 127 = 124$$

- Binary representation of 124:

$$124_{10} = 01111100_2$$

- The exponent field in IEEE 754 format will be 01111100.

2's Complement vs Excess-127

Decimal Values	Binary (2's Complement)	Excess-127 (Stored)	Excess-127 (Actual)
-128	10000000	00000000	-127
-3	11111101	01111110	-3
-1	11111111	01111111	-1
0	00000000	01111111	0
1	00000001	10000000	1
3	00000011	10000010	3
127	01111111	11111110	127
128	——	11111111	128 (Reserved)

Table: Comparison of 2's Complement and Excess-127 Representations for 8-bit Numbers

Step 5: Determine the Mantissa (Significand)

- The mantissa is the fractional part of the normalized number, excluding the leading 1.
- For 1.0010101_2, the mantissa is:

$$\text{Mantissa} = 00101010000000000000000$$

Step 6: Combine the Components

- Combine the sign bit, exponent, and mantissa into a 32-bit binary number:

$$1\ 10000011\ 00101010000000000000000$$

Final Representation

- The IEEE 754 single-precision floating-point representation of -18.625 is:

$$1\ 10000011\ 00101010000000000000000$$

- In hexadecimal, it can be written as:

$$C12A8000_{16}$$

Special Case

- IEEE 754 standard defines the representation of floating-point numbers, including special cases like zero, infinity, and NaN (Not a Number)
- $E_{\text{stored}} = 0 = 00000000_2$: Represents subnormal numbers.
- $E_{\text{stored}} = 255 = 11111111_2$: Represents infinity or NaN (Not a Number).

Zero (± 0) Representation

Single-Precision (32-bit):

- Sign bit: 0 for +0, 1 for -0
- Exponent: All bits are 0 (00000000)
- Mantissa: All bits are 0 (00000000000000000000000)
- **+0:** ‘0 00000000 00000000000000000000000‘ (Hex: ‘0x00000000‘)
- **-0:** ‘1 00000000 00000000000000000000000‘ (Hex: ‘0x80000000‘)

Infinity ($\pm\infty$) Representation

Single-Precision (32-bit):

- Sign bit: 0 for $+\infty$, 1 for $-\infty$
- Exponent: All bits are 1 (11111111)
- Mantissa: All bits are 0 (00000000000000000000000)
- **$+\infty$:** ‘0 11111111 00000000000000000000000‘ (Hex: ‘0x7F800000‘)
- **$-\infty$:** ‘1 11111111 00000000000000000000000‘ (Hex: ‘0xFF800000‘)

Not a Number (NaN) Representation

NaN is used to represent undefined or unrepresentable values, such as the result of 0/0 or sqrt(-1).

- Sign bit: Can be 0 or 1 (doesn't matter for NaN)
- Exponent: All bits are 1 (11111111)
- Mantissa: At least one non-zero bit
- **Quiet NaN:** ‘s 11111111 1xxxxxxxxxxxxxxxxxxxxxx‘ (e.g., Hex: ‘0x7FC00000‘)
- **Signaling NaN:** ‘s 11111111 0xxxxxxxxxxxxxxxxxxxxxx‘ (e.g., Hex: ‘0x7FA00000‘)

Smallest and Largest +**ve** Values in Floating Point

- **Smallest Positive Normalized Number:**
  0 00000001 00000000000000000000000
  - Exponent = (1-127) =-126 ; Mantissa = 0
  - Value $= 2^{-126} \times 1 \approx 1.18 \times 10^{-38}$
- **Largest Positive Normalized Number:**
  0 11111110 11111111111111111111111
  - Exponent = 254-127 = 127 ;
  - Mantissa $= 1.11\ldots1 = 111\ldots1 \times 2^{-23} = (2^{24} - 1) \times 2^{-23}$
  - Value $= (2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$
- **Smallest Positive Denormalized Number:**
  0 00000000 00000000000000000000001
  - Exponent = -126 (Fixed) $\neq (0 - 127)$
  - Mantissa $= 2^{-23}$
  - Value $= 2^{-126} \times 2^{-23} = 2^{-149} \approx 1.4 \times 10^{-45}$

Smallest and Largest -**ve** Values ?

Smallest and Largest -**ve** Values in Floating Point

Smallest $\overrightarrow{\longleftarrow}$ Largest : symbols changed Only

- **Largest Negative Normalized Number**
  1 00000001 00000000000000000000000
  - Exponent = (1-127) =-126 ; Mantissa = 0
  - Value $= -2^{-126} \times 1 \approx -1.18 \times 10^{-38}$
- **Smallest Negative Normalized Number:**
  1 11111110 11111111111111111111111
  - Exponent = 254-127 = 127 ;
  - Mantissa $= 1.11\ldots1 = 111\ldots1 \times 2^{-23} = (2^{24} - 1) \times 2^{-23}$
  - Value $= -(2 - 2^{-23}) \times 2^{127} \approx -3.4 \times 10^{38}$
- **Largest Positive Denormalized Number:**
  1 00000000 00000000000000000000001
  - Exponent = -126 (Fixed) $\neq (0 - 127)$
  - Mantissa $= 2^{-23}$
  - Value $= -2^{-126} \times 2^{-23} = -2^{-149} \approx -1.4 \times 10^{-45}$

Floating Point – Rounding off

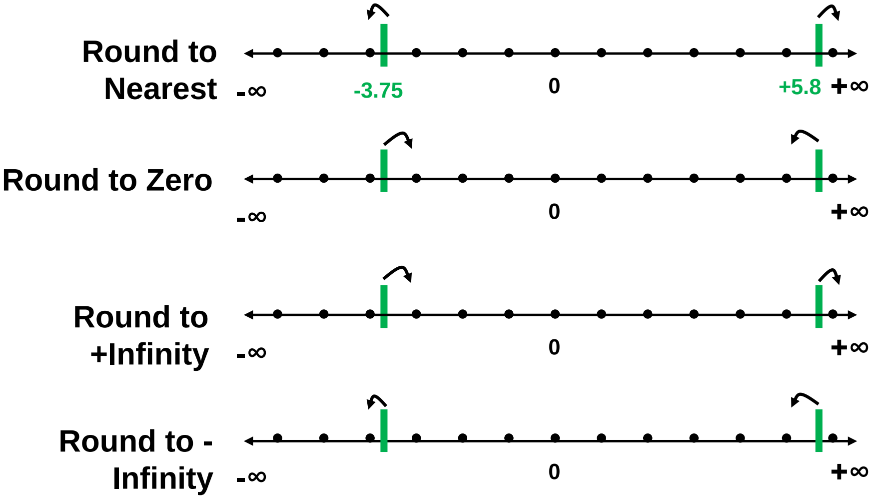
IEEE 754 32-bit Floating Point Format

- 32-bit floating-point representation consists of:
  - 1 bit for the sign
  - 8 bits for the exponent (with a bias of 127)
  - 23 bits for the mantissa (fraction)
- Normalized form: $(-1)^{\text{sign}} \times 1.\text{mantissa} \times 2^{(\text{exponent}-127)}$

Rounding in IEEE 754

- Rounding is necessary when the exact binary representation exceeds 23 bits in the mantissa.
- Common rounding modes:
  - Round to Nearest, ties to Even (default)
  - Round toward Zero (truncation)
  - Round toward Positive Infinity
  - Round toward Negative Infinity

Example


Round to Nearest
-∞
-3.75
0
+5.8
+∞
Round to Zero
-∞
0
+∞
Round to +Infinity
-∞
0
+∞
Round to -Infinity
-∞
0
+∞

Example 1: Rounding 4.7

- Convert 4.7 to binary: $4.7 \approx 100.1011001100110011\ldots_2$
- Normalize: $1.001011001100110011\ldots \times 2^2$
- Fit mantissa into 23 bits:

$$1.00101100110011001100110 \text{ (truncated)}$$

- Assemble IEEE 754 representation:
  - Sign: 0
  - Exponent: $2 + 127 = 129$, binary: 10000001
  - Mantissa: 00101100110011001100110
- Final result: $0 \,|\, 10000001 \,|\, 00101100110011001100110$

Example 2: Rounding 0.1

- Convert 0.1 to binary: $0.1 \approx 0.00011001100110011\ldots_2$
- Normalize: $1.1001100110011\ldots \times 2^{-4}$
- Fit mantissa into 23 bits:

$$1.10011001100110011001110 \text{ (rounded)}$$

- Assemble IEEE 754 representation:
  - Sign: 0
  - Exponent: $-4 + 127 = 123$, binary: 01111011
  - Mantissa: 10011001100110011001110
- Final result: $0 \,|\, 01111011 \,|\, 10011001100110011001110$

Default Method:

- Different versions use different methods
- Default Method: Round-to-Nearest-Even (old)

Round-to-Nearest-Even

1. If the digit after the place you are rounding to is less than 5, you round down. $2.4 \rightarrow 2$
2. If it's 5 or greater, you round up. $2.6 \rightarrow 3$
3. If a number is exactly halfway between two rounding options, it gets rounded to the nearest even number.
    1. 2.5 rounds to 2 (because 2 is even).
    2. 3.5 rounds to 4 (because 4 is even).
    3. 4.5 rounds to 4 (because 4 is even).
    4. 5.5 rounds to 6 (because 6 is even).

Effect of Associativity in Arithmetic Operations

- Due to rounding, **floating-point arithmetic is not strictly associative**.
- Example: Addition
  - $(a + b) + c \neq a + (b + c)$ in floating-point arithmetic.
  - Example: Let $a = 1.0 \times 10^{10}$, $b = -1.0 \times 10^{10}$, $c = 1.0$.
  - $(a + b) + c = 1.0$ (Correct)
  - $a + (b + c) = 0.0$ (Incorrect due to loss of significance)
- Example: Multiplication
  - $(a \times b) \times c \neq a \times (b \times c)$ in floating-point arithmetic.
  - Example: Let $a = 1.0 \times 10^{-5}$, $b = 1.0 \times 10^{5}$, $c = 1.0 \times 10^{-10}$.
  - $(a \times b) \times c = 1.0 \times 10^{-10}$ (Correct)
  - $a \times (b \times c) = 1.0 \times 10^{-15}$ (Incorrect due to rounding)

Examples of Rounding in Operations

- Addition/Subtraction Example:
  - $a = 1.23456789$, $b = 1.0 \times 10^{-7}$
  - Result: $a + b = 1.23456789$ (rounded)
- Multiplication/Division Example:
  - $a = 1.23456789$, $b = 1.23456789$
  - Result: $a \times b = 1.524157875019...$ (rounded)
  - After rounding: 1.52415788