Binary Arithmetic

Cezar Ionescu
cezar.ionescu@th-deg.de

WS 2022-23

Outline

Admin

Questions?

Binary Arithmetic

“Arithmetization”

What did the semiconductor revolution reduce the cost of? In a word: arithmetic.

Before semiconductors, “computers” were humans who were employed to do arithmetic problems. Digital computers made arithmetic inexpensive, which eventually resulted in thousands of new applications for everything from data storage to word processing to photography.

Ajay Agrawal, Joshua S. Gans, and Avi Goldfarb

https://sloanreview.mit.edu/article/what-to-expect-from-artificial-intelligence/

Old Computers

- Babbage's *Analytic Engine* used the standard *base-10* representation.
- This led to complicated cogwheel mechanisms prone to mechanical failures.
- *ENIAC* (~1945) was probably the last computer to do "serious work" using the decimal system.

Decimal Representation

- The standard notation uses base 10
- The notation $a_n a_{n-1} \ldots a_2 a_1 a_0$ represents the number

$$a_n \times 10^n + a_{n-1} \times 10^{n-1} + \ldots + a_2 \times 10^2 + a_1 \times 10 + a_0$$
$$= \sum_{i=0}^{n} a_i \times 10^i$$

- We can use any natural number $b > 1$ as the base for a number system. The digits will then be elements of $\{0, 1, \ldots, b-1\}$ and

$$a_n a_{n-1} \ldots a_2 a_1 a_0$$
$$= a_n \times b^n + a_{n-1} \times b^{n-1} + \ldots + a_2 \times b^2 + a_1 \times b + a_0$$
$$= \sum_{i=0}^{n} a_i \times b^i$$

Decimal Representation

- The standard notation uses base 10
- The notation $a_n a_{n-1} \ldots a_2 a_1 a_0$ represents the number

$$
\begin{aligned}
& a_n \times 10^n + a_{n-1} \times 10^{n-1} + \ldots + a_2 \times 10^2 + a_1 \times 10 + a_0 \\
= & \sum_{i=0}^{n} a_i \times 10^i
\end{aligned}
$$

- We can use any natural number $b > 1$ as the base for a number system. The digits will then be elements of $\{0, 1, \ldots, b-1\}$ and

$$
\begin{aligned}
& a_n a_{n-1} \ldots a_2 a_1 a_0 \\
= & a_n \times b^n + a_{n-1} \times b^{n-1} + \ldots + a_2 \times b^2 + a_1 \times b + a_0 \\
= & \sum_{i=0}^{n} a_i \times b^i
\end{aligned}
$$

Binary Representation

- In base two, the digits are therefore in the set $\{0, 1\}$ and

$$
\begin{aligned}
& a_n a_{n-1} \ldots a_2 a_1 a_0 \\
=& a_n \times 2^n + a_{n-1} \times 2^{n-1} + \ldots + a_2 \times 2^2 + a_1 \times 2 + a_0 \\
=& \sum_{i=0}^{n} a_i \times 2^i
\end{aligned}
$$

- Example: Convert the following numbers from base 2 notation to base 10: 11, 0, 11101, 10101101

Binary Representation

- In base two, the digits are therefore in the set $\{0, 1\}$ and

$$
\begin{aligned}
& a_n a_{n-1} \ldots a_2 a_1 a_0 \\
=& a_n \times 2^n + a_{n-1} \times 2^{n-1} + \ldots + a_2 \times 2^2 + a_1 \times 2 + a_0 \\
=& \sum_{i=0}^{n} a_i \times 2^i
\end{aligned}
$$

- Example: Convert the following numbers from base 2 notation to base 10: 11, 0, 11101, 10101101

Conversion from base 10 to base 2

- To convert n from base 10 to base 2:

```
conv : ℕ -> List {0, 1}
conv n
  | n == 0     = [0]
  | n == 1     = [1]
  | otherwise  = conv q ++ [r]
  where
  q, r = quotient and remainder of dividing n by 2
```

- Example: Write 47 in base 2

Conversion from base 10 to base 2

- To convert n from base 10 to base 2:

```
conv : ℕ -> List {0, 1}
conv n
  | n == 0     = [0]
  | n == 1     = [1]
  | otherwise  = conv q ++ [r]
  where
  q, r = quotient and remainder of dividing n by 2
```

- Example: Write 47 in base 2

Binary Arithmetic

- Addition (carry is 0 or 1):

```
 1111 <---- carries
  10110
 +01110
 ------
 100100
```

- Multiplication

```
   1110
 x 1001
 ------
   1110
  0000
 0000
1110
-------
1111110
```

Fixed-size representation

- Traditionally, numbers are stored in *words* of fixed length.
- A word usually consists of one or more *bytes*.
- A byte is 8 *bits*
- A bit is a binary digit (0 or 1)

Example: one-byte word

- We can have $2^8 = 256$ possible values, for example

```
00000000 = 0
01010101 = 85
11111111 = 255
```

Fixed-size representation

- Traditionally, numbers are stored in *words* of fixed length.
- A word usually consists of one or more *bytes*.
- A byte is 8 *bits*
- A bit is a binary digit (0 or 1)

Example: one-byte word

- We can have $2^8 = 256$ possible values, for example

```
00000000 = 0
01010101 = 85
11111111 = 255
```

Overflow

- Operations on numbers in this range can lead to results outside this range: *overflow*
- Example:

```
   10110100
+  01110101
-----------
(1)00101001
```

Signed numbers

- A negative number $-n$ is the solution to the equation

$$n + x = 0$$

  (or, equivalently, $x + n = 0$)

- Consider binary representations of size 4. We have:
  - -1 = 1111, since 0001 + 1111 = 0000
  - -7 = 1001, since 0111 + 1001 = 0000
  - -4 = 1100, since 0100 + 1100 = 0000

Signed numbers

- A negative number $-n$ is the solution to the equation

$$n + x = 0$$

  (or, equivalently, $x + n = 0$)

- Consider binary representations of size 4. We have:
  - -1 = 1111, since 0001 + 1111 = 0000
  - -7 = 1001, since 0111 + 1001 = 0000
  - -4 = 1100, since 0100 + 1100 = 0000

Two's complement

- Should 1000 stand for 8 or for -8?
- The idea is to have as many negative as positive (or zero).
- Convention: numbers starting with 1 are interpreted as negative numbers.
- With n bits we have 2^{n-1} negative numbers and $2^{n-1}-1$ strictly positive numbers.
- Algorithm: to find the two's complement, change every digit and add 1.
- Note: two's complement of $x = a_{n-1}...a_0$ is $2^n - x$ (in binary).

Two's complement

- Should 1000 stand for 8 or for -8?
- The idea is to have as many negative as positive (or zero).
- Convention: numbers starting with 1 are interpreted as negative numbers.
- With n bits we have 2^{n-1} negative numbers and $2^{n-1}-1$ strictly positive numbers.
- Algorithm: to find the two's complement, change every digit and add 1.
- Note: two's complement of $x = a_{n-1}...a_0$ is $2^n - x$ (in binary).

Two's complement

- Should 1000 stand for 8 or for -8?
- The idea is to have as many negative as positive (or zero).
- Convention: numbers starting with 1 are interpreted as negative numbers.
- With n bits we have 2^{n-1} negative numbers and $2^{n-1}-1$ strictly positive numbers.
- Algorithm: to find the two's complement, change every digit and add 1.
- Note: two's complement of $x = a_{n-1}...a_0$ is $2^n - x$ (in binary).

Two's complement

- Should 1000 stand for 8 or for -8?
- The idea is to have as many negative as positive (or zero).
- Convention: numbers starting with 1 are interpreted as negative numbers.
- With n bits we have 2^{n-1} negative numbers and $2^{n-1}-1$ strictly positive numbers.
- Algorithm: to find the two's complement, change every digit and add 1.
- Note: two's complement of $x = a_{n-1}...a_0$ is $2^n - x$ (in binary).

Two's complement

- Should 1000 stand for 8 or for -8?
- The idea is to have as many negative as positive (or zero).
- Convention: numbers starting with 1 are interpreted as negative numbers.
- With n bits we have 2^{n-1} negative numbers and $2^{n-1}-1$ strictly positive numbers.
- Algorithm: to find the two's complement, change every digit and add 1.
- Note: two's complement of $x = a_{n-1}...a_0$ is $2^n - x$ (in binary).

Two's complement (examples)

- Example: convert -34 into 8-bit, two's complement notation

Two's complement (examples)

- Example: convert -34 into 8-bit, two's complement notation

$$
\begin{aligned}
34 &= 32 + 2 = 2^5 + 2^1 \\
&= 00100010 \\
-34 &= 11011101 + 1 \\
&= 11011110
\end{aligned}
$$

Two's complement (examples)

- Example: convert -34 into 8-bit, two's complement notation

$$
\begin{aligned}
34 &= 32 + 2 = 2^5 + 2^1 \\
&= 00100010 \\
-34 &= 11011101 + 1 \\
&= 11011110
\end{aligned}
$$

- Convert the following two's complement numbers into decimal:

  (a) 00101101, *(b)* 10110100

Two's complement (examples)

- Example: convert -34 into 8-bit, two's complement notation

$$
\begin{aligned}
34 &= 32 + 2 = 2^5 + 2^1 \\
&= 00100010 \\
-34 &= 11011101 + 1 \\
&= 11011110
\end{aligned}
$$

- Convert the following two's complement numbers into decimal:

  (a) 00101101, *(b)* 10110100

$$
00101101 = 2^5 + 2^3 + 2^2 + 2^0 = 45
$$

$$
\begin{aligned}
10110100 &= -\ (01001011 + 1) \\
&= -\ (01001100) \\
&= -\ (2^6 + 2^3 + 2^2) = -76
\end{aligned}
$$

Fractions in binary

- In the decimal notation we have

$$
\begin{aligned}
& a_n \ldots a_0 . a_{-1} a_{-2} \ldots a_{-m} \\
=& a_n \times 10^n + \ldots + a_0 \times 10^0 + a_{-1} \times 10^{-1} + a_{-2} \times 10^{-2} \ldots a_{-m} \times 10^{-m} \\
=& \sum_{i=-m}^{n} a_i \times 10^i
\end{aligned}
$$

- The same idea can be extended to any base $b > 1$, in particular binary.
- Example: What is the numerical value of 101.101?

Fractions in binary

- In the decimal notation we have

$$
\begin{aligned}
& a_n \ldots a_0 . a_{-1} a_{-2} \ldots a_{-m} \\
=& a_n \times 10^n + \ldots + a_0 \times 10^0 + a_{-1} \times 10^{-1} + a_{-2} \times 10^{-2} \ldots a_{-m} \times 10^{-m} \\
=& \sum_{i=-m}^{n} a_i \times 10^i
\end{aligned}
$$

- The same idea can be extended to any base $b > 1$, in particular binary.
- Example: What is the numerical value of 101.101?

Fractions in binary

- In the decimal notation we have

$$
\begin{aligned}
& a_n \ldots a_0 . a_{-1} a_{-2} \ldots a_{-m} \\
=& a_n \times 10^n + \ldots + a_0 \times 10^0 + a_{-1} \times 10^{-1} + a_{-2} \times 10^{-2} \ldots a_{-m} \times 10^{-m} \\
=& \sum_{i=-m}^{n} a_i \times 10^i
\end{aligned}
$$

- The same idea can be extended to any base $b > 1$, in particular binary.
- Example: What is the numerical value of 101.101?

$$
\begin{aligned}
101.101 &= 2^2 + 2^0 + 2^{-1} + 2^{-3} \\
&= 5 + \frac{1}{2} + \frac{1}{8} \\
&= 5\frac{5}{8}
\end{aligned}
$$

Operations on fractions

- Operations on fractions are carried out in the expected way.
- Example: Compute $1010.001 + 1.101$

Fixed-point representation

- The standard binary representation *could* be used to represent fractions, e.g. by using n bits for the integral part and n bits for the fractional part: *fixed-point representation*.
- This method is *not used* in implementing artithmetic, because of the limited range of numbers that can be represented.
- With n bits for the integral part we can represent integers between -2^{n-1} and 2^{n-1}. E.g., for $n = 32$ we can represent only integers between -2147483648 and 2147483648.
- If $m = 32$, then between each representable integers we can represent $2^{32} - 1 = 4294967295$ fractions.

Floating-point representation

- Floating-point representation is used to represent a larger range of values (**note:** not a larger *number* of values!).
- Idea: use something *similar* to scientific notation, e.g.

$$\begin{aligned} & -13E-2 \\ = & \\ & -13 \times 10^{-2} \\ = & \\ & -0.13 \end{aligned}$$

- The components are: sign, exponent, mantissa:

-	-2	13
Sign	Exponent	Mantissa

Floating-point representation

- The same elements (sign, exponent, matissa) are used in the floating-point representation.
- In the following, we shall use an 8-bit representation where the individual bits are allocated as follows:

1 bit	3 bits	4 bits
Sign	Exponent	Mantissa

- The mantissa is non-negative (since we have the separate sign bit for the result), but the exponent must be a signed number.
- The IEEE standard uses for the exponent a different representation from two's complement: excess notation.

Excess notation

- Excess notation is an alternative to two's complement for representing signed numbers.
- Let n be the (fixed) length of the representation. Then:
  - A number of the form $1a_1 \ldots a_{n-1}$ represents the number $a_1 \ldots a_{n-1}$ in standard binary notation.
    - Remark: therefore 100...0 is used to represent the number 0.
  - A number of the form $0a_1 \ldots a_{n-1}$ represents the same value as the n-bits two's complement of $1a_1 \ldots a_{n-1}$.
- The value of a number x is therefore $x - 2^{n-1}$. This justifies the terminology *excess* 2^{n-1} notation.
- Example: What numbers do 1010 and 0001 represent in excess 8 notation?

Two's complement vs. excess notation

Bit pattern	3-bit 2's complement	excess 4
000	0	-4
001	1	-3
010	2	-2
011	3	-1
100	-4	0
101	-3	1
110	-2	2
111	-1	3

- Excess notation simplifies comparisons.

Two's complement vs. excess notation

Bit pattern	3-bit 2's complement	excess 4
000	0	-4
001	1	-3
010	2	-2
011	3	-1
100	-4	0
101	-3	1
110	-2	2
111	-1	3

- Excess notation simplifies comparisons.

Floating-point representation, simplified version

- In a first approximation, floating-point numbers are represented as

1 bit	3 bits	4 bits
Sign (1 for '-')	Exponent (excess 4)	Mantissa (unsigned)

  with the exponent telling us how to shift the mantissa (left or right and to how many places).

- Example: What number is encoded by 11011100?

Floating-point representation, simplified version

- In a first approximation, floating-point numbers are represented as

1 bit	3 bits	4 bits
Sign (1 for '-')	Exponent (excess 4)	Mantissa (unsigned)

  with the exponent telling us how to shift the mantissa (left or right and to how many places).

- Example: What number is encoded by 11011100?

1	101	1100
Sign -	1	.1100

  - The exponent tells us to shift the mantissa one place to the left, so we have `1.100`, which is `1.5`
  - The sign is `-`, so the end result is `-1.5`

Floating-point representation, simplified version

- Example: What number is encoded by 01011001?

Floating-point representation, simplified version

- Example: What number is encoded by 01011001?

0	101	1001
Sign +	1	.1001

  - Exponent shifts mantissa one place to the left: `1.001`
  - Sign is +, therefore the end result is `1.001` = $1\frac{1}{8}$
- Example: What is the floating-point encoding of $-3\frac{3}{4}$?

Floating-point representation, simplified version

- Example: What number is encoded by 01011001?

0	101	1001
Sign +	1	.1001

  - Exponent shifts mantissa one place to the left: `1.001`
  - Sign is +, therefore the end result is `1.001` = $1\frac{1}{8}$
- Example: What is the floating-point encoding of $-3\frac{3}{4}$?
  - The sign is '-', so the first bit is `1`
  - The standard representation of $3\frac{3}{4}$ is 11.11, which is `.1111` shifted twice to the left.
    - The mantissa is `1111`
    - The exponent is 2 in excess 4, i.e. `110`.
  - Putting it all together: `11101111`

Disadvantages of the simplified version

- Using the simplified version we can represent a range between $-7\frac{1}{2}$ and $7\frac{1}{2}$, which is not an improvement over the standard representation using 4+4 bits.
- The smallest non-zero value is $\frac{1}{32}$ (better than with 4+4 standard).
- The representation is *not unique*. For example $1\frac{1}{2}$ can be represented as 01011100 *or* as 01110110.
- Therefore, the simplified representation is *inefficient*: we cannot represent 2^8 different real numbers (much worse than the standard representation!)

Normalization

- To eliminate the redundancy, we adopt the following convention: the first bit of the mantissa will always be 1.
- Therefore, the correct representation of $1\frac{1}{2}$ is 01011100.
- Problem: how do we represent 0?
  - Convention: a bit pattern of all 0s.

Efficient representation

- We do not need to explicitly represent the starting 1 of the mantissa, since it's always there.
- Therefore, $1\frac{1}{2}$ would actually be represented as 01011000.
- This extends the range of the representation.

Efficient representation examples

The previous examples become:

- `11011100` = $-1\frac{3}{4}$ (instead of -1.5)
- `01011001` = $1\frac{9}{16}$ (instead of $1\frac{1}{8}$)
- $-3\frac{3}{4}$ = `11101110` (instead of `11101111`)

Efficient representation examples

The previous examples become:

- `11011100` = $-1\frac{3}{4}$ (instead of -1.5)
- `01011001` = $1\frac{9}{16}$ (instead of $1\frac{1}{8}$)
- $-3\frac{3}{4}$ = `11101110` (instead of `11101111`)

Efficient representation examples

The previous examples become:

- `11011100` = $-1\frac{3}{4}$ (instead of -1.5)
- `01011001` = $1\frac{9}{16}$ (instead of $1\frac{1}{8}$)
- $-3\frac{3}{4}$ = `11101110` (instead of `11101111`)

Efficient representation examples

The previous examples become:

- `11011100` = $-1\frac{3}{4}$ (instead of -1.5)
- `01011001` = $1\frac{9}{16}$ (instead of $1\frac{1}{8}$)
- $-3\frac{3}{4}$ = `11101110` (instead of `11101111`)

Efficient representation examples

The previous examples become:

- `11011100` = $-1\frac{3}{4}$ (instead of -1.5)
- `01011001` = $1\frac{9}{16}$ (instead of $1\frac{1}{8}$)
- $-3\frac{3}{4}$ = `11101110` (instead of `11101111`)

Efficient representation examples

The previous examples become:

- `11011100` = $-1\frac{3}{4}$ (instead of -1.5)
- `01011001` = $1\frac{9}{16}$ (instead of $1\frac{1}{8}$)
- $-3\frac{3}{4}$ = `11101110` (instead of `11101111`)

Floating-point

Representation	Sign	Exponent	Mantissa
Toy 8-bit	1	3 bits	4 bits
Single Precision	1	8 bits	23 bits
Double Precision	1	11 bits	52 bits

Truncation errors

- What is the 8-bit floating point representation of $3\frac{9}{16}$?

- The difference between the original number and the translation of its floating-point representation is called **truncation** or **round-off error**.

Truncation errors

- What is the 8-bit floating point representation of $3\frac{9}{16}$?
  - $3\frac{9}{16} = (11.1001)_2$ and we have (in standard base 2):

$$11.1001 = .111001 \times 10^{10}$$

  - dropping the leading 1 from the mantissa still leaves five digits: 11001
  - the 8 bit representation is therefore 01101100
  - converting back to standard base 10 we obtain $3\frac{1}{2}$, an error of $\frac{1}{16}$.
- The difference between the original number and the translation of its floating-point representation is called **truncation** or **round-off error**.

Truncation errors

- What is the 8-bit floating point representation of $3\frac{9}{16}$?
  - $3\frac{9}{16} = (11.1001)_2$ and we have (in standard base 2):

$$11.1001 = .111001 \times 10^{10}$$

    - dropping the leading 1 from the mantissa still leaves five digits: 11001
    - the 8 bit representation is therefore 01101100
    - converting back to standard base 10 we obtain $3\frac{1}{2}$, an error of $\frac{1}{16}$.
- The difference between the original number and the translation of its floating-point representation is called **truncation** or **round-off error**.

Non-terminating representations

- In the decimal system, a fraction such as $\frac{1}{3}$ has an infinite periodic representation: $0.33333\ldots$.
- Which rational numbers have infinite representations depends on the base. For example:
  - In base 3, $\frac{1}{3}$ is 0.1
  - In base 2, $\frac{1}{10}$ has an infinite periodic representation
- Many numbers which have finite representations in base 10 have infinite representations in base 2. Any number that has an infinite representation in base 10 also has an infinite one in base 2.
- Irrational numbers, such as $\sqrt{2}$, have infinite, non-periodic representations in any (natural) base.

Non-terminating representations

- In the decimal system, a fraction such as $\frac{1}{3}$ has an infinite periodic representation: $0.33333\ldots$.
- Which rational numbers have infinite representations depends on the base. For example:
  - In base 3, $\frac{1}{3}$ is 0.1
  - In base 2, $\frac{1}{10}$ has an infinite periodic representation
- Many numbers which have finite representations in base 10 have infinite representations in base 2. Any number that has an infinite representation in base 10 also has an infinite one in base 2.
- Irrational numbers, such as $\sqrt{2}$, have infinite, non-periodic representations in any (natural) base.

Non-terminating representations

- In the decimal system, a fraction such as $\frac{1}{3}$ has an infinite periodic representation: $0.33333\ldots$.
- Which rational numbers have infinite representations depends on the base. For example:
  - In base 3, $\frac{1}{3}$ is 0.1
  - In base 2, $\frac{1}{10}$ has an infinite periodic representation
- Many numbers which have finite representations in base 10 have infinite representations in base 2. Any number that has an infinite representation in base 10 also has an infinite one in base 2.
- Irrational numbers, such as $\sqrt{2}$, have infinite, non-periodic representations in any (natural) base.

Non-terminating representations

- In the decimal system, a fraction such as $\frac{1}{3}$ has an infinite periodic representation: $0.33333\ldots$.
- Which rational numbers have infinite representations depends on the base. For example:
  - In base 3, $\frac{1}{3}$ is 0.1
  - In base 2, $\frac{1}{10}$ has an infinite periodic representation
- Many numbers which have finite representations in base 10 have infinite representations in base 2. Any number that has an infinite representation in base 10 also has an infinite one in base 2.
- Irrational numbers, such as $\sqrt{2}$, have infinite, non-periodic representations in any (natural) base.

Arithmetic on floating-point numbers

- The arithmetical operations follow from the representation of floating-point numbers
  - for example,

$$\sigma_1 m_1 \times 2^{e_1} \times \sigma_2 m_2 \times 2^{e_2} = (\sigma_1 + \sigma_2)(m_1 \times m_2) \times 2^{e_1+e_2}$$

    with the respective operations performed within the constraints of the representation (in particular, for $\sigma_1 = \sigma_2 = 1$, we have $\sigma_1 + \sigma_2 = 0$).

- Together with truncation errors, this leads to a loss of the usual properties of arithmetical operations.

Associativity

- Example: What is $3\frac{1}{2} + \frac{1}{16} + \frac{1}{16}$?

- This example shows that addition on floating-point numbers is *not associative*.
- Similarly, multiplication is not associative either.

Associativity

- Example: What is $3\frac{1}{2} + \frac{1}{16} + \frac{1}{16}$?
- Version 1: $(3\frac{1}{2} + \frac{1}{16}) + \frac{1}{16} = 3\frac{1}{2} + \frac{1}{16} = 3\frac{1}{2}$
- Version 2: $3\frac{1}{2} + (\frac{1}{16} + \frac{1}{16}) = 3\frac{1}{2} + \frac{1}{8} = 3\frac{5}{8}$
- This example shows that addition on floating-point numbers is *not associative*.
- Similarly, multiplication is not associative either.

Associativity

- Example: What is $3\frac{1}{2} + \frac{1}{16} + \frac{1}{16}$?
- Version 1: $(3\frac{1}{2} + \frac{1}{16}) + \frac{1}{16} = 3\frac{1}{2} + \frac{1}{16} = 3\frac{1}{2}$
- Version 2: $3\frac{1}{2} + (\frac{1}{16} + \frac{1}{16}) = 3\frac{1}{2} + \frac{1}{8} = 3\frac{5}{8}$
- This example shows that addition on floating-point numbers is *not associative*.
- Similarly, multiplication is not associative either.

Floating-point summary

- The common floating-point representations use the following allocations of bits:

Representation	Sign	Exponent	Mantissa
Single Precision	1	8 bits	23 bits
Double Precision	1	11 bits	52 bits

- The exponent is encoded using excess representation.
- The mantissa has an implicit first bit 1.
- The IEEE standard specifies four *special* or *non-numerical values*:
  - '00...0' is used to represent the value 0
  - '10...0' is used for *NaN*
  - '01...1' is used for $+\infty$
  - '11...1' is used for $-\infty$
  - Example: $1/0 = +\infty$, $\sqrt{-1} = NaN$
- Truncation errors lead to potentially confusing results.

Floating-point summary

- The common floating-point representations use the following allocations of bits:

Representation	Sign	Exponent	Mantissa
Single Precision	1	8 bits	23 bits
Double Precision	1	11 bits	52 bits

- The exponent is encoded using excess representation.
- The mantissa has an implicit first bit 1.
- The IEEE standard specifies four *special* or *non-numerical values*:
  - '00...0' is used to represent the value 0
  - '10...0' is used for *NaN*
  - '01...1' is used for $+\infty$
  - '11...1' is used for $-\infty$
  - Example: $1/0 = +\infty$, $\sqrt{-1} = NaN$
- Truncation errors lead to potentially confusing results.

Floating-point summary

- The common floating-point representations use the following allocations of bits:

Representation	Sign	Exponent	Mantissa
Single Precision	1	8 bits	23 bits
Double Precision	1	11 bits	52 bits

- The exponent is encoded using excess representation.
- The mantissa has an implicit first bit 1.
- The IEEE standard specifies four *special* or *non-numerical values*:
  - '00...0' is used to represent the value 0
  - '10...0' is used for *NaN*
  - '01...1' is used for $+\infty$
  - '11...1' is used for $-\infty$
  - Example: $1/0 = +\infty$, $\sqrt{-1} = NaN$
- Truncation errors lead to potentially confusing results.

Floating-point summary

- The common floating-point representations use the following allocations of bits:

Representation	Sign	Exponent	Mantissa
Single Precision	1	8 bits	23 bits
Double Precision	1	11 bits	52 bits

- The exponent is encoded using excess representation.
- The mantissa has an implicit first bit 1.
- The IEEE standard specifies four *special* or *non-numerical values*:
  - '00...0' is used to represent the value 0
  - '10...0' is used for *NaN*
  - '01...1' is used for $+\infty$
  - '11...1' is used for $-\infty$
  - Example: $1/0 = +\infty$, $\sqrt{-1} = NaN$
- Truncation errors lead to potentially confusing results.

More information

- The situation is actually more complicated than this! For more information, have a look at

  What Every Computer Scientist Should Know About Floating-Point Arithmetic, David Goldberg 1991