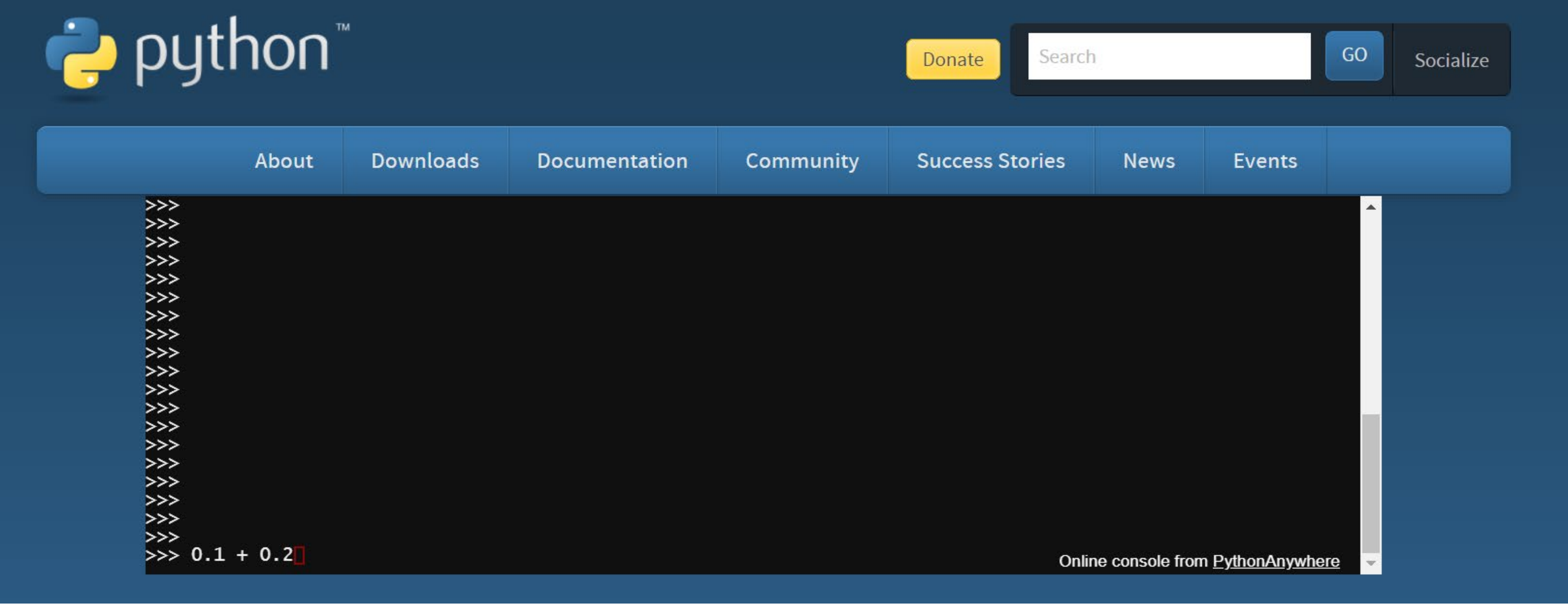
Unit 3

Finite Representation of Numbers

Albert Sung



In python, 0.1 + 0.2 = ?

a) 0.29999999999999996
b) 0.29999999999999999
c) 0.3
d) 0.30000000000000001
e) 0.30000000000000004

Outline of Unit 3

- 3.1 Unsigned Integers
- 3.2 Signed Integers
- 3.3 Fixed-Point Representation
- 3.4 Floating-Point Representation

Unit 3.1

Unsigned Integers

Unsigned Integers

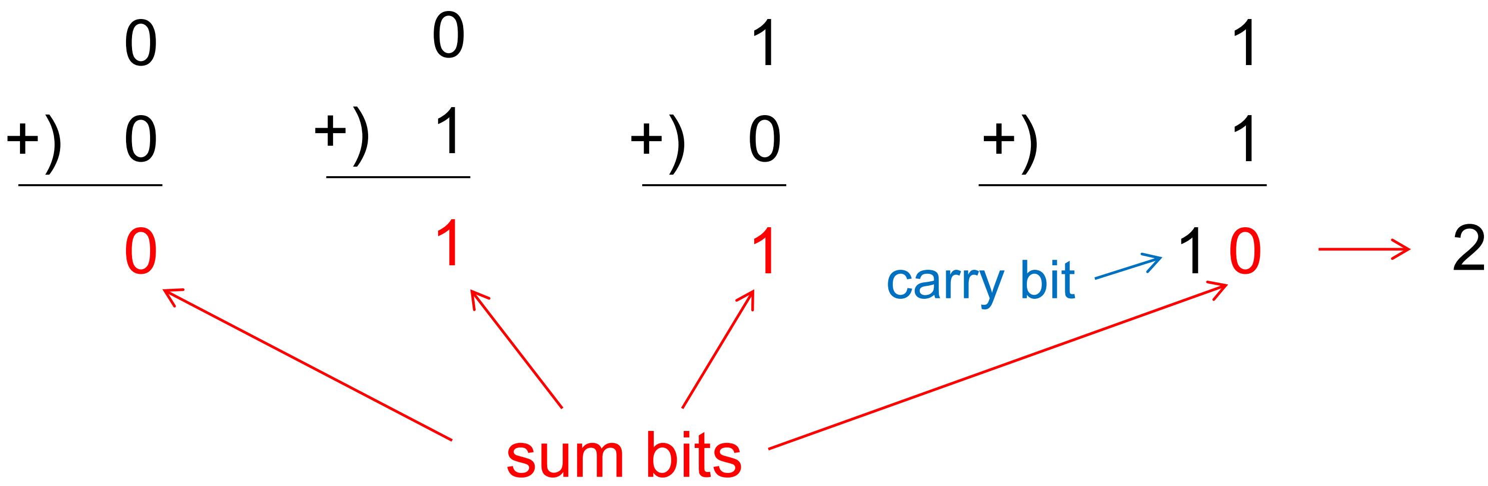
- Mathematics
  - Range is from 0 to ∞
- Computer
  - Range limited by computer's word size (n bits)
  - Range is from 0 to $2^n - 1$
    - $n = 4$, range is from 0 to 15
    - $n = 8$, range is from 0 to 256
    - $n = 16$, range is from 0 to 65535
    - $n = 32$, range is from 0 to 4294967295

4-bit word	Unsigned Integers
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

C/C++: `unsigned int`

- C/C++ data types:
  - **`int`**
    - can represent all integers
  - **`unsigned int:`**
    - can represent non-negative integers
- In general, the use of unsigned integers should be avoided because
  - Negative numbers cannot be represented.
  - Unexpected behavior can result when you mix signed and unsigned integers.
- Unsigned integers are useful in some situations:
  - Bit manipulation
  - Specific algorithms, e.g., encryption or random number generation.

Rules of 1-bit Addition



Unsigned Addition

- Find the sum of $(0110)_2$ and $(0111)_2$.

Beware of overflow.

Carries	0	1 0	1 1 0	1 1 0
	0 1 1 0	0 1 1 0	0 1 1 0	0 1 1 0
	+) 0 1 1 1	+) 0 1 1 1	+) 0 1 1 1	+) 0 1 1 1
Sum	1	0 1	1 0 1	1 1 0 1

Sum the LSBs

$0 + 1 = 1$

Sum the 2nd LSBs and the carry bit

$0 + 1 + 1 = 10$

The result is

$(1101)_2 = (13)_{10}$

Unit 3.2

Signed Integers

Signed Integers

- Representing negative numbers as well as positive numbers
- Three different representations:

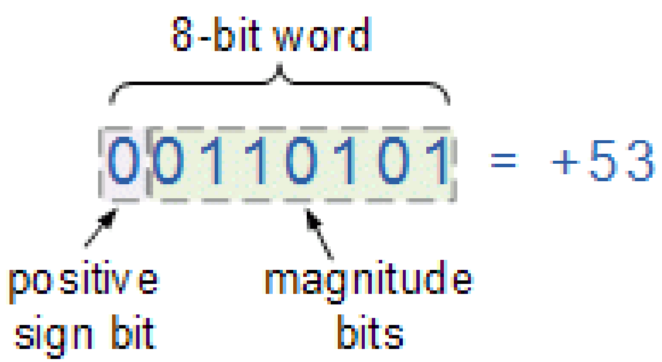
a) Sign-magnitude form

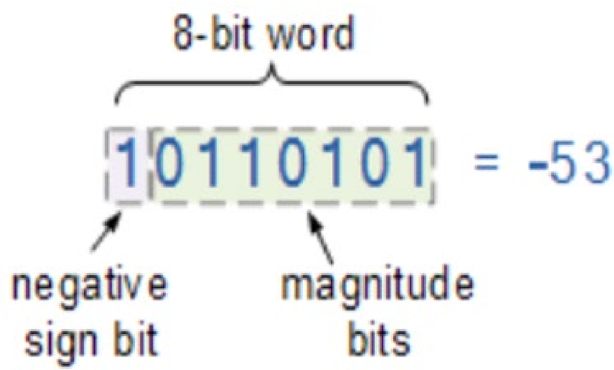
b) One's complement

c) Two's complement

Sign-Magnitude Form

- The MSB indicates the sign (known as sign bit).
- If this bit is set to 1, the number is negative else it is positive.
- The other $n-1$ bits represent the magnitude of the number.
- Range is from $-(2^{n-1}-1)$ to $2^{n-1}-1$.





Sign-Magnitude Form

Sign	4	2	1	
1	1	1	1	-7
1	1	1	0	-6
1	1	0	1	-5
1	1	0	0	-4
1	0	1	1	-3
1	0	1	0	-2
1	0	0	1	-1
1	0	0	0	-0
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7

Difficult to perform addition/subtraction

Two 0's Problem:

- There are two different representations of 0.

One's Complement

- The MSB is the sign bit.
- A negative number $-x$ is obtained by inverting (or flipping) each bit of the corresponding positive number x.
- The same range as sign-magnitude form, i.e., from $-(2^{n-1} - 1)$ to $2^{n-1} - 1$.
- Also has the two 0's problem.

One's Complement

8 4 2 1	
1 0 0 0	-7
1 0 0 1	-6
1 0 1 0	-5
1 0 1 1	-4
1 1 0 0	-3
1 1 0 1	-2
1 1 1 0	-1
1 1 1 1	-0
0 0 0 0	0
0 0 0 1	1
0 0 1 0	2
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 0	6
0 1 1 1	7

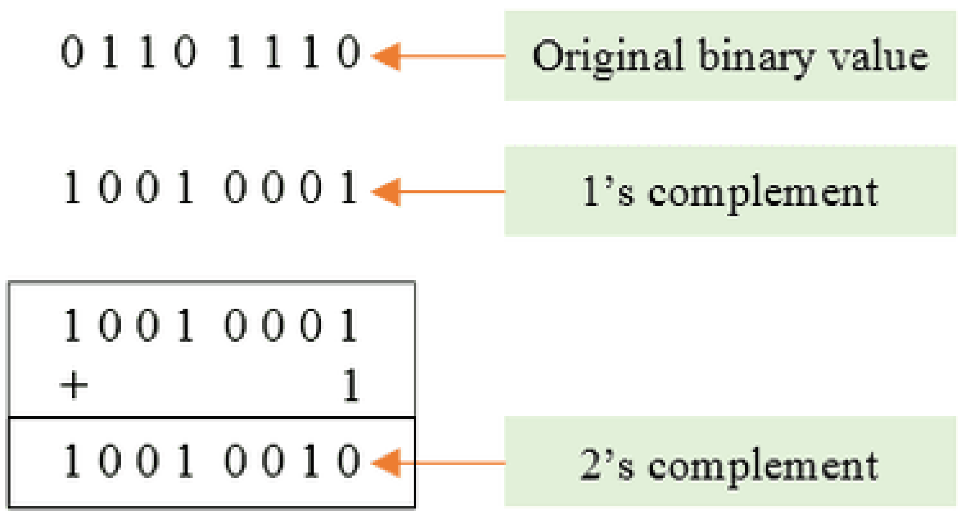
- Addition is straightforward.
- If the carry extends past the end of the word it is said to have "wrapped around", and the bit must be added back to the LSB.

```
  1010   -5        1011     -4
+0101   +5      + 0100     +4
  1111   -0        1111     -0
```

```
Wrap-around
  0101   5
  1100  -3
========
 10001   (carry added back to LSB)
  0010   2
```

Two's Complement

- The MSB is the sign bit.
- A negative number $-x$ is obtained by inverting (or flipping) each bit of the corresponding positive number x and then adding 1 to its LSB.
- The range is from $-(2^{n-1} - 1)$ to 2^{n-1}.



Conversion from $+x$ to $-x$

- Example 1: 0100 (representing +4 in the 4-bit case)
  - Change all 0's to 1 and all 1's to 0, so the 1's complement of the number is 1011
  - Add 1 to the LSB of this number
    - $(1011) + 1 = 1100$ (-4)
- Example 2: 0110 (representing +6 in the 4-bit case)
  - Change all 0's to 1 and all 1's to 0, so the 1's complement of the number is 1001
  - Add 1 to the LSB of this number
    - $(1001) + 1 = 1010$ (-6)

Two's Complement

8 4 2 1	
1 0 0 1	-7
1 0 1 0	-6
1 0 1 1	-5
1 1 0 0	-4
1 1 0 1	-3
1 1 1 0	-2
1 1 1 1	-1
0 0 0 0	0
0 0 0 1	1
0 0 1 0	2
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 0	6
0 1 1 1	7

MSB has weight $-2^{n-1} = -8$.

$$\begin{aligned}(1011) &= (1 \times -8) + (0 \times 4) + (1 \times 2) + (1 \times 1) \\ &= -8 + 2 + 1 \\ &= -5\end{aligned}$$

Without two 0's Problem:

- There is a unique representations of 0.

Widely used in computer systems because addition/subtraction is easy.

```
 0101  +5
 1101  -3
10010  +2   (simply drop the carry)
```

Overflow

- Overflow is a computational error.
- It occurs when a computational result cannot be correctly represented due to insufficient number of bits.
- Adding two numbers in a 4-bit two's complement system
  - The range of sum is $-8 \leq sum \leq 7$
  - Overflow occurs when $sum < -8$, or $sum > 7$

Addition Examples

- Compute $A + B$ in a 4-bit system ($-8 \leq sum \leq 7$)

  i) $A = 5, B = 4;\quad A + B = 9$ (too large)

$$
\begin{array}{r|l}
 & 0\,1\,0\,1 \\
+) & 0\,1\,0\,0 \\
\hline
 & 1\,0\,0\,1
\end{array}
$$

$1001 = -7_{10}$ (incorrect, overflow!)

  ii) $A = -5, B = -4;\quad A + B = -9$ (too small)

$$
\begin{array}{r|l}
 & 1\,0\,1\,1 \\
+) & 1\,1\,0\,0 \\
\hline
(1) & 0\,1\,1\,1
\end{array}
$$

$0111 = 7_{10}$ (incorrect, overflow!)

Subtraction Examples

- Compute $A - B$ in a 4-bit system $(-8 \leq sum \leq 7)$

  i) $A = 7, B = 5;$ $A - B = 2$ (within range)

$$
\begin{array}{r|l}
 & 0\,1\,1\,1 \\
+) & 1\,0\,1\,1 \\
\hline
(1) & 0\,0\,1\,0
\end{array}
$$

$0010 = 2_{10}$ (correct!)

  ii) $A = 7, B = -5;$ $A - B = 12$ (too large)

$$
\begin{array}{r|l}
 & 0\,1\,1\,1 \\
+) & 0\,1\,0\,1 \\
\hline
 & 1\,1\,0\,0
\end{array}
$$

$1100 = -4_{10}$ (incorrect, overflow!)

Unit 3.3

Fixed-Point Representation

Fixed-Point Representation

Unsigned fixed point		Integer	Fraction
Signed fixed point	Sign	Integer	Fraction

- We can represent these numbers using:
  - Sign-magnitude form
  - One's complement
  - Two's complement
- Two's complement is preferred in computer systems because of
  - Unambiguous property (no two-0 problem)
  - Easier for arithmetic operations

An Example

- Assume 8-bit two's complement format with
  - 1 bit for the sign,
  - 4 bits for the integer part, and
  - 3 bits for the fractional part.

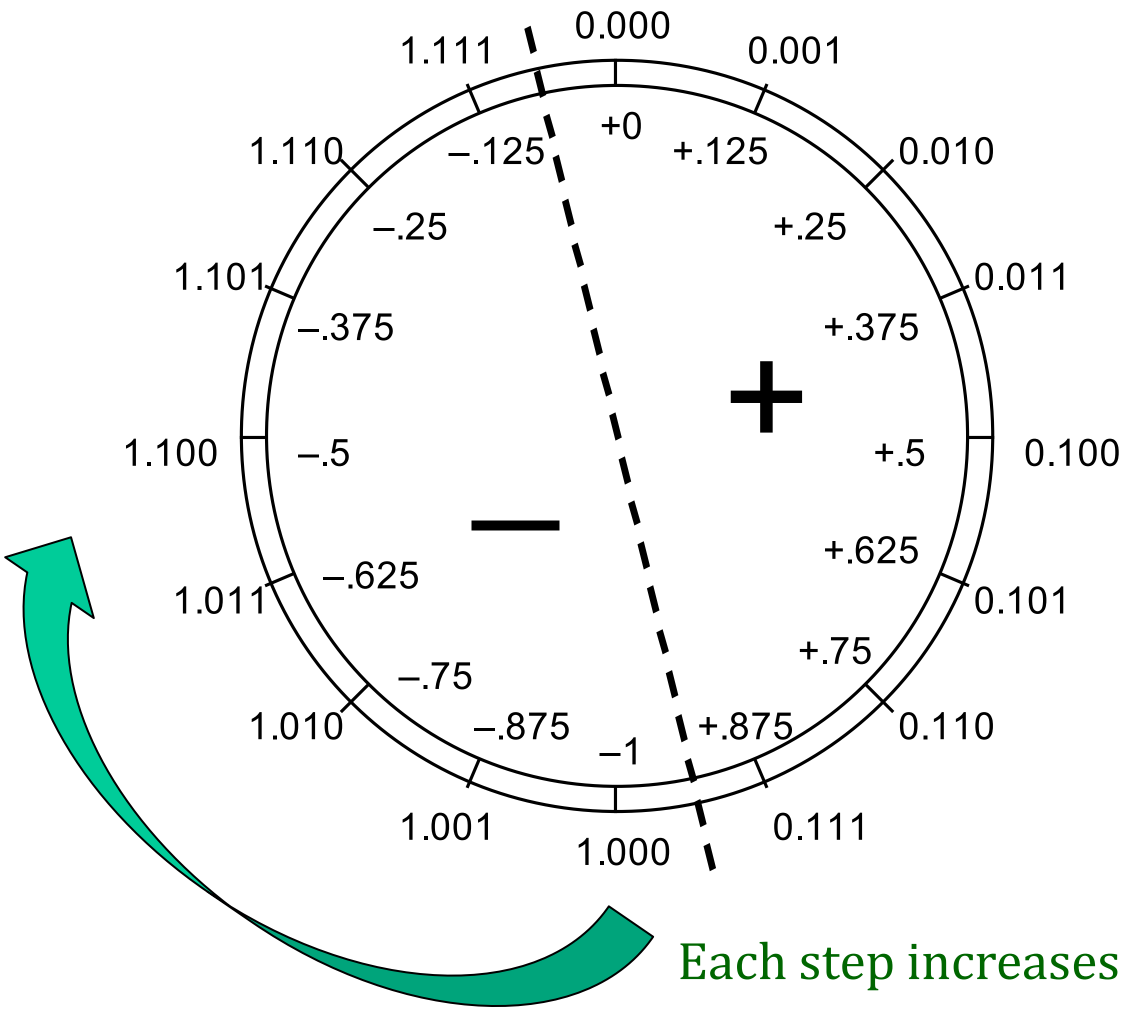
  Example:
  - $(10010.110)_2 = (-1 \times 2^4 + 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-2})$

    $= -16 + 2 + 0.5 + 0.25$

    $= -13.25$

- Disadvantage: range is relatively limited.
  - usually inadequate for numerical analysis as it does not allow enough numbers and accuracy.

Fixed-Point 2's-Complement Numbers



4-bit format (1 bit for sign and 3 bits for fraction)

Range: from -1 to $+\frac{7}{8}$

Each step increases by $+\frac{1}{8} = +0.125$

Limited Range and Precision

- Consider 32-bit (1 sign bit, 15 bits for integer, 16 bits for fraction)

Smallest	0	000000000000000	0000000000000001
	Sign bit	Integer part	Fractional part
Largest	0	111111111111111	1111111111111111
	Sign bit	Integer part	Fractional part

- The smallest positive number is $2^{-16} \approx 0.000015$.
- The largest positive number is

$$(2^{15} - 1) + (1 - 2^{-16}) = (32768 \ - 1) + (0.9999847412 \ldots)$$
$$= 32767.9999847\ldots$$

Pros and Cons

- Arithmetic calculations as efficient as integers
  - Integer representation can be considered a special case of fixed-point representation (with no fractional part)
  - Simple and power, still being used in many game and DSP applications.
- Insufficient range and precision
  - Solution: Floating point representation
    - The radix point can be moved to either left or right to increase accuracy or range (next section).

Unit 3.4

Floating-Point Representation

Floating Point Representation

- The decimal point (or radix) is not set in a fixed position.
- Like scientific notation
  - 234.1058 is written as 2.341058×10^2.
- In binary,
  - $\pm 1.ssssss \times 2^{eeeeee}$
- Use "significand" (also called "mantissa") and "exponent" to represent a number with the radix point being float around.

IEEE 754 Standard

- Developed in response to divergence of representations in 1985
- Now almost universally adopted
- Two representations
  - Single precision (32 bits)
    - C/C++ data type: **`float`**
  - Double precision (64 bits)
    - C/C++ data type: **`double`**

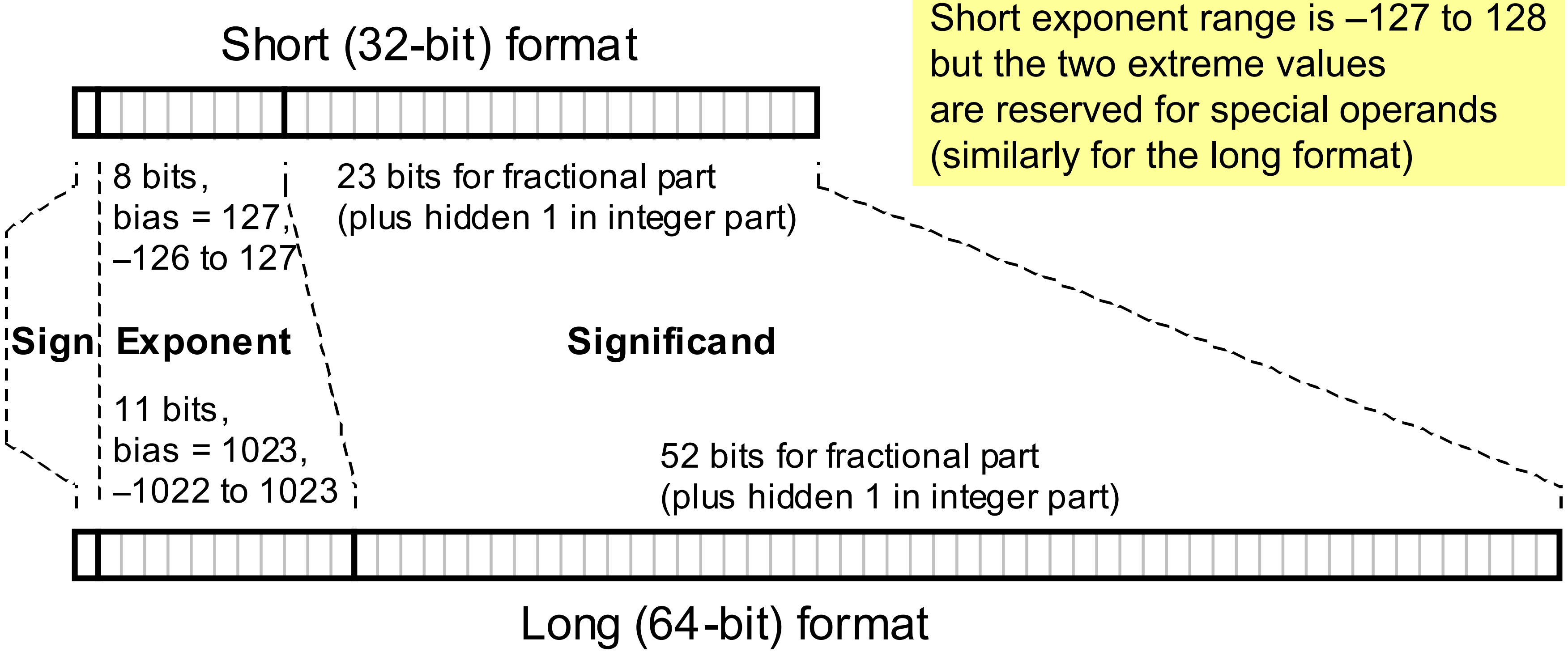
IEEE 754 Format

sign	*exponent* (single: 8 bits, double: 11 bits)	*fraction* (single: 23 bits, double: 52 bits)

$$x = (-1)^{sign} \times (1 + fraction) \times 2^{exponent - bias}$$

- *sign*: 0: positive, 1: negative
- significand = 1 + *fraction*
  - Hidden 1: Always has a leading 1 before the radix point, so no need to represent it explicitly.
  - $1.0 \le |signficand| < 2.0$
- actual exponent = $exponent - bias$
  - *exponent* is unsigned.
    - Single precision: $bias = 127$
    - double precision: $bias = 1203$

IEEE 754 Format: Alternative View



Short exponent range is –127 to 128
but the two extreme values
are reserved for special operands
(similarly for the long format)

Single-Precision Range

- Exponents 00000000 and 11111111 are reserved.
- Smallest value
  - *exponent* = 00000001
    $\Rightarrow$ actual exponent = $1 - 127 = -126$
  - *fraction* = 000…0 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - *exponent* = 11111110
    $\Rightarrow$ actual exponent = $254 - 127 = 127$
  - *fraction* = 111…1 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{127} \approx \pm 3.4 \times 10^{38}$

Floating-Point Precision

- Relative Precision
  - all fraction bits are significant
- Single-precision: approximately 2^{-23}
  - equivalent to $23 \times \log_{10} 2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
- Double-precision: approximately 2^{-52}
  - equivalent to $52 \times \log_{10} 2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Example: Decimal to Float

Represent -0.75 in IEEE 754 single-precision.

Solution:

- $-0.75_{10} = -0.11_2 = (-1)^1 \times 1.1_2 \times 2^{-1}$
- $sign = 1$
- $fraction = 1000 \ldots 00$ (single: 23 bits)
- $exponent$ = actual exponent + $bias$ $= -1 + 127 = 126$

  $= 01111110_2$
- Single-precision: $101111110 1000 \ldots 00$

Example: Float to Decimal

What number is represented by the single-precision float 11000000101000000…00 ?

Solution:

- $sign = 1$
- $fraction = 01000 \ldots 00$ (single: 23 bits)

  $\Rightarrow$ significant $= 1 + 01_2 = 1.25_{10}$
- $exponent = 10000001_2 = 129_{10}$

  $\Rightarrow$ actual exponent $= exponent - bias$

  $= 129 - 127 = 2$
- Number $= (-1)^1 \times 1.25 \times 2^2 = -5$

Special Values

$e = 0$ and $f = 0$	denotes the number zero (which can not be normalized) Note that there is a +0 and -0.
$e = 0$ and $f \neq 0$	denotes a *denormalized number*. (ignored in our discussion)
$e = \mathrm{FF}$ and $f = 0$	denotes infinity (∞). There are both positive and negative infinities.
$e = \mathrm{FF}$ and $f \neq 0$	denotes an undefined result, known as *NaN* (Not a Number).

- An infinity is produced by an overflow or by division by zero.
- An undefined result is produced by an invalid operation such as trying to find the square root of a negative number, adding two infinities, etc.

Addition and Subtraction

- Consider $x + y$ or $x - y$.
- The exponents of x and y must be of the same value.
  - If not, move the lower exponent to the same value of the higher exponent.
- Just work on the fraction or significand part
- Normalize it back after addition
- Subtraction uses two's complement and carry the same process as addition

Example (addition with equal exponent):

X: 0 1101 0111 111 0011 1010 0000 1100 0011

Y: 0 1101 0111 000 1110 0101 1111 0001 1100

Step 1: Important step, are the exponent the same? Yes, in this case

Step 2: Both exponents are 1101 0111 = 215_{10}, no need to bother the bias.

Step 3: we can add the significand direct; don't forget the hidden 1

X: 1. 111 0011 1010 0000 1100 0011 $\times 2^{215-127}$

Y: 1. 000 1110 0101 1111 0001 1100

X+Y: 11. 000 0001 1111 1111 1101 1111 $\times 2^{215-127}$

Step 4: Because "11", move the radix point one place to the right to align with the format

X+Y 1. 100 0000 1111 1111 1110 1111 1 $\times 2^{\mathbf{216}-127}$

Computer lost this last bit, lack of memory space

exponent becomes 1101 0111 +1 =1101 1000

Step 5: X+Y= [0] [1101 1000] [100 0000 1111 1111 1110 1111]

Example (addition with unequal exponent):

X: 0 1101 0111 111 0011 1010 0000 1100 0011

Y: 0 1101 0001 000 1110 0101 1111 0001 1100

exponent of X: 1101 0111 = 215_{10}

exponent of Y: 1101 0001 = 209_{10}

Step 1: Thus shift the radix point of Y to the right by 6 places; hidden 1

1. 000 1110 0101 1111 0001 1100 becomes

0. 000 0010 0011 1001 0111 1100 011100

(lost these 6 bits in the computer, no space)

Step 2: Add X to Y's significand

$$
\begin{array}{rll}
 & 1.111\ 0011\ 1010\ 0000\ 1100\ 0011 & \\
+ & 0.000\ 0010\ 0011\ 1001\ 0111\ 1100 & \times (2^{215-127}) \\
 & 1.111\ 0101\ 1101\ 1010\ 0011\ 1111 & \times (2^{215-127})
\end{array}
$$

Step 3: The form the computer will save is

[0] [1101 0111] [111 0101 1101 1010 0011 1111]

Example (subtraction with unequal exponent):

16-7=9, X-Y

(16) X: [0] [1000 0011] [000 0000 0000 0000 0000 0000] actual exp=4, 2^4

(7) Y: [0] [1000 0001] [110 0000 0000 0000 0000 0000] actual exp=2, 2^2

Y has a lower exponent than X

Step 1: shift Y's exponent by 2 places; hidden 1

Y: 1. 110 0000 0000 0000 0000 0000 $\times (2^2)$

Y: 0. 011 1000 0000 0000 0000 0000 $\times (2^4)$

Step 2: use 2's complement to get –Y and then add it to X

```
        0. 011  1000  0000  0000  0000  0000
        1. 100  0111  1111  1111  1111  1111
  +                                        1
  ------------------------------------------
-Y      1. 100  1000  0000  0000  0000  0000
+X      1. 000  0000  0000  0000  0000  0000
  ------------------------------------------
X-Y=   10. 100  1000  0000  0000  0000  0000
```

simply drop the carry 1; $\times (2^4)$

Step 3: shift radix point to the right by 1 place

1. 001 0000 0000 0000 0000 0000 $\times (2^3)$

IEEE 754 format: [0] [1000 0010] [001 0000 0000 0000 0000 0000]

Check back in decimal: X-Y = $(1+0.125)(2^{130-127}) = 9$

Roundoff Error

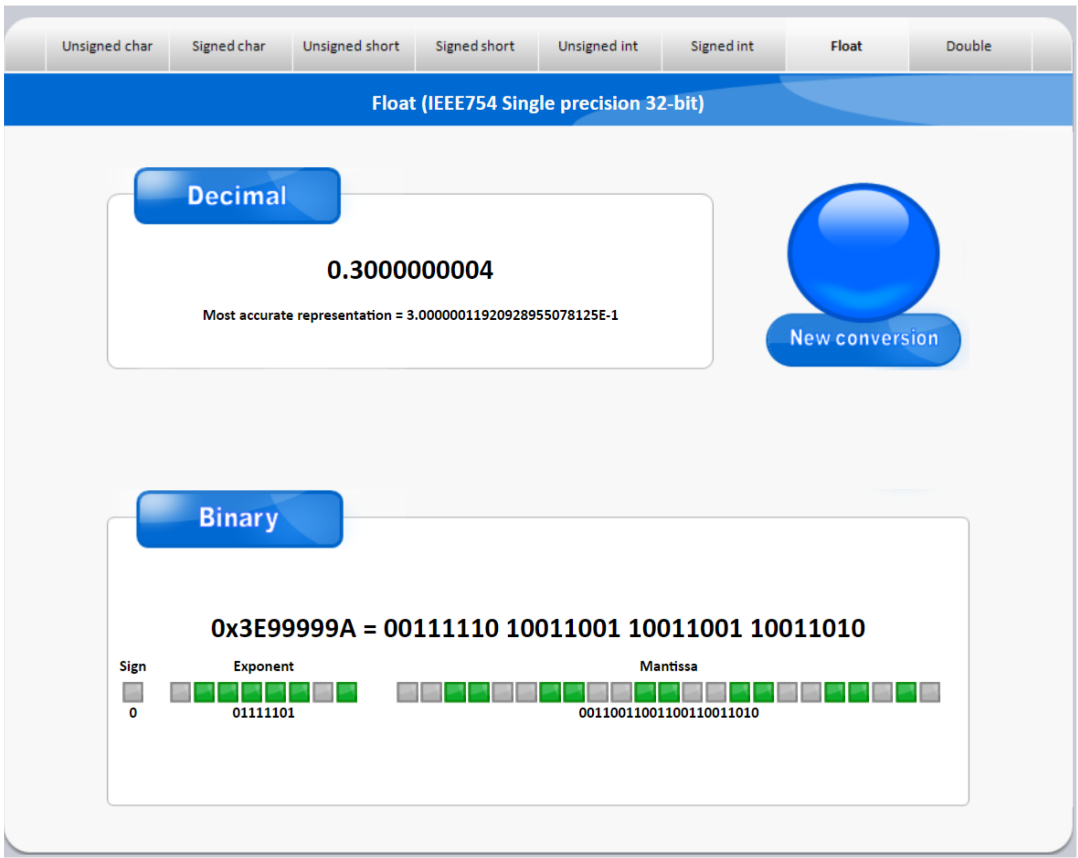
- $0.2_{10} = 0.00\overline{1100}_2$
  - Single precision:
    00111110 01001100 11001100 11001101 $\times (2^{124-127})$
- $0.1_{10} = 0.000\overline{1100}_2$
  - Single precision:
    00111101 11001100 11001100 11001101 $\times (2^{123-127})$

$$
\begin{array}{rl}
1.10011001100110011001101 & \times (2^{124-127}) \\
+\ 0.110011001100110011001101 & \text{shift right by 1} \\
10.011001100110011001100111 & \\
1.0011001100110011001100111 & \text{shift right by 1, } \times (2^{125-127}) \\
1.00110011001100110011010 & \text{rounding to the nearest}
\end{array}
$$

Roundoff Error

- The true sum should be $0.3_{10} = 0.0\overline{1001}_2$
- Now we have 1.00110011001100110011010
  - Single precision:

    $00111110\ 10011001\ 10011001\ 10011010 \times (2^{125-127})$



The computed sum is greater than 0.3.

Converter:

- https://www.binaryconvert.com//convert_float.html