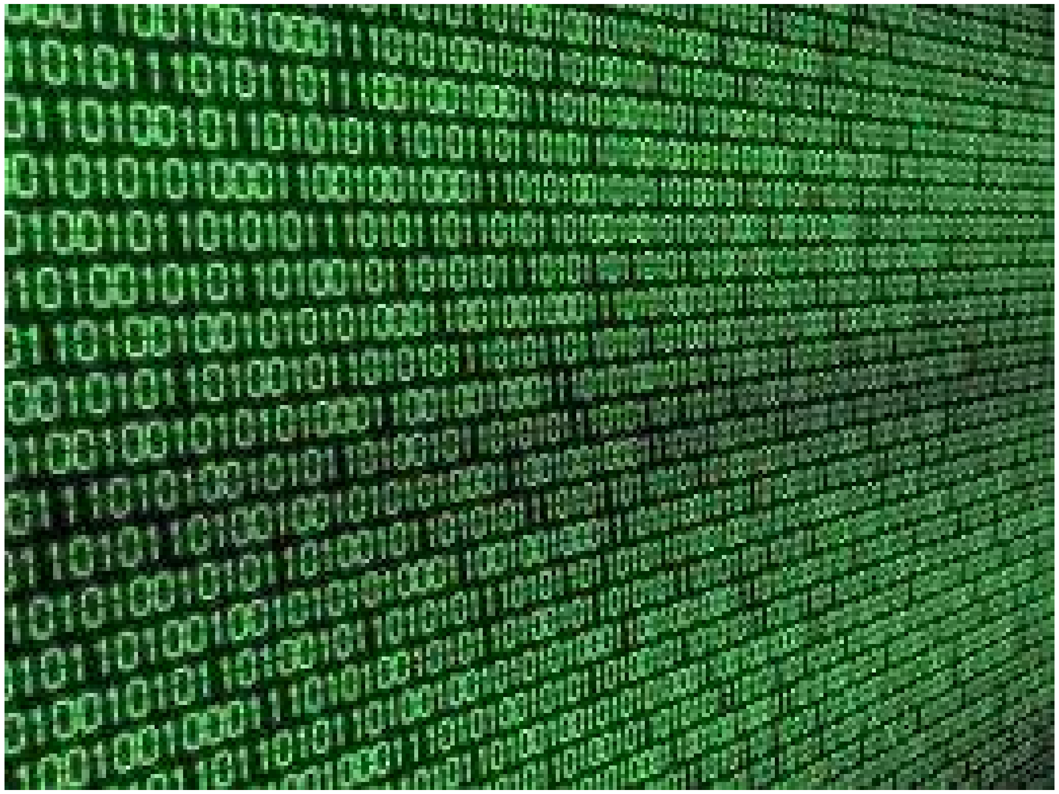
SCS1205 / IS1102
Computer Systems

Computer Arithmetic

Dr. Ajantha Atukorale
aja@ucsc.cmb.ac.lk

Integer Representation



Binary Addition

- 0 + 0 = **0**
- 0 + 1 = **1**
- 1 + 0 = **1**
- 1 + 1 = **10** (carry: **1**)

- E.g.

```
      1  1  1  1  1     (carry)
         0  1  1  0  1
 +       1  0  1  1  1
   ___________________
 =     1 0  0  1  0  0
   ___________________
```

Binary Subtraction

- 0 - 0 = **0**
- 0 - 1 = **1** **(* with borrow)**
- 1 - 0 = **1**
- 1 - 1 = **0**

- E.g.

```
     *     *  *        (borrow)
     1  0  1  1  0  1
 -   0  1  0  1  1  1
    -----------------
 =   0  1  0  1  1  0
    -----------------
```

Binary Multiplication

To perform a **binary multiplication** problem, we need to understand how addition works with binary numbers and follow the same process of multiplication and addition we would use with decimal numbers.

```
          1 0 1 1     Multiplicand
        x 1 0 1 0     Multiplier
      -----------
          0 0 0 0
+       1 0 1 1
+     0 0 0 0
+   1 0 1 1
  ---------------
=   1 1 0 1 1 1 0
  ---------------
```

Binary Division

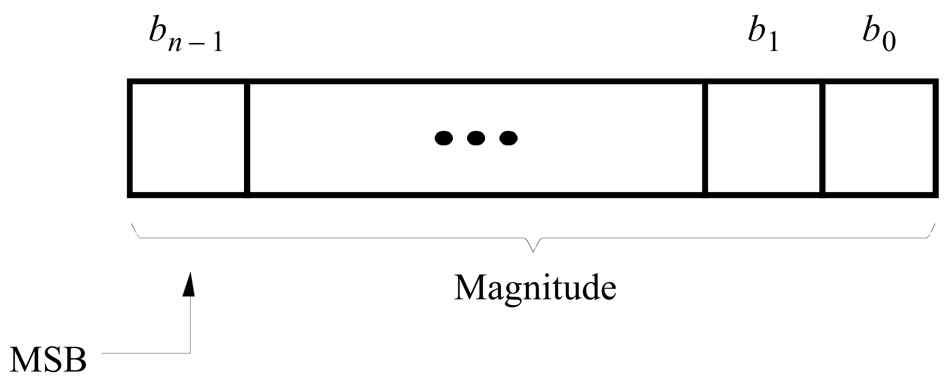
To perform a binary division, we need to follow the same process as we do for dividing regular numbers but, in this case, we only need to decide if it's going to be a 1 or a 0.

```
                 1 0 1
             ___________
 1 0 1      | 1 1 0 1 1
            - 1 0 1
              -----
                0 1 1
              - 0 0 0
                -----
                  1 1 1
                - 1 0 1
                  -----
                    1 0
```

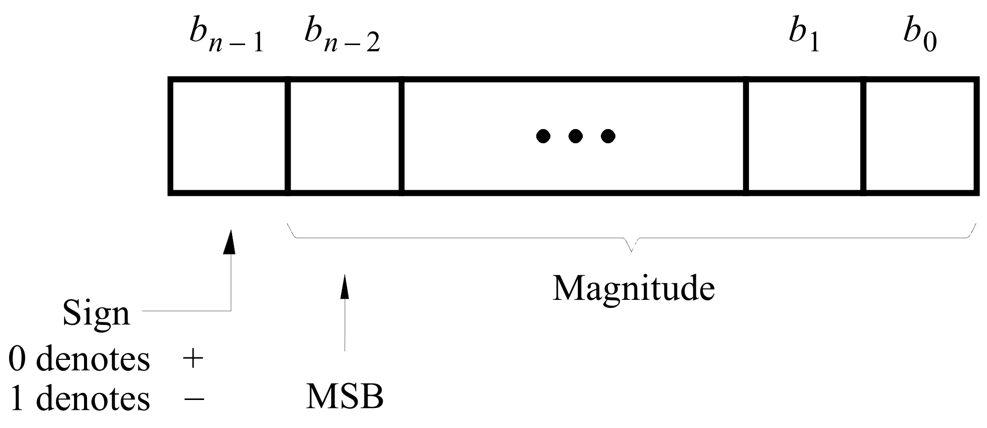
Representing Numbers

- Problems of number representation
  - Positive and negative
  - Radix point
  - Range of representation
- Different ways to represent numbers
  - Unsigned representation: non-negative integers
  - Signed representation: integers
  - Floating-point representation: fractions

Formats for representation of integers



(a) Unsigned number



(b) Signed number

Unsigned and Signed Numbers

- Unsigned binary numbers
  - Have **0** and **1** to represent numbers
  - **Only positive** numbers stored in binary
  - The **Smallest** binary number would be …

    0 0 0 0 0 0 0 0 which equals to **0**

  - The **largest** binary number would be …

    1 1 1 1 1 1 1 1 which equals ….

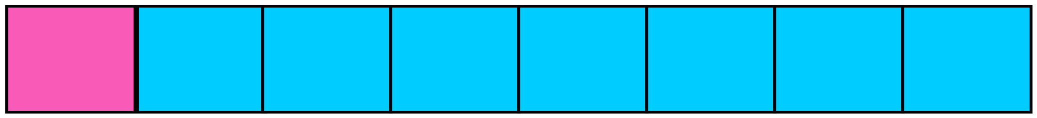
    128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = **255 = 2^8-1**
  - Therefore the range is 0 - 255 (256 numbers)

    $0 \leftrightarrow 2^n$-1

Unsigned and Signed Numbers

- Signed binary numbers
  - Have **0** and **1** to represent numbers
  - The leftmost bit is a sign bit
    - **0** for positive
    - **1** for negative

Sign bit



- Compare & Contrast: **ASCII & Extended ASCII**

Unsigned and Signed Numbers

- Signed binary numbers
  - The **Smallest** positive binary number is

    0 0 0 0 0 0 0 0 which equals to **0**

  - The **largest** positive binary number is

    0 1 1 1 1 1 1 1 which equals ….

    64 + 32 + 16 + 8 + 4 + 2 + 1 = **127 = 2^7- 1**

Therefore the **range** for positive numbers is **0 - 127** (**128** numbers)

$$0 \leftrightarrow 2^{n-1}-1$$

Negative Numbers in Binary

- Problems with simple signed representation
  - **Two** representation of **zero**: **+ 0** and **– 0**

    0 0 0 0 0 0 0 0 and **1 0 0 0 0 0 0 0**

    Which one is correct?
  - Need to consider **both sign** and **magnitude** in arithmetic
    - E.g. Consider two 8 bit numbers: **5 – 3**

      = **5 + (-3)**

      = **0 0 0 0 0 1 0 1 + 1 0 0 0 0 0 1 1**

      = **1 0 0 0 1 0 0 0**

      = **-8 ???**

Negative Numbers in Binary…

- Problems with simple signed representation
  - Need to consider both sign and magnitude in arithmetic
    - E.g. = **18 + (-18)**

      = **0 0 0 1 0 0 1 0 + 1 0 0 1 0 0 1 0**

      = **1 0 1 0 0 1 0 0**

      = **-36 ???**

Negative Numbers in Binary…

- **One's Complement**
  - A method which we can use to represent **negative binary numbers** in a **signed** binary number system.
  - In one's complement, **positive numbers** (also known as **non-complements**) remain **unchanged** as before with the sign-magnitude numbers.
  - **Negative numbers** however, are represented by taking the **one's complement** (inversion, negation) of the **unsigned positive number**.
  - Since positive numbers always start with a "**0**", the complement will always start with a "**1**" to indicate a negative number.

Negative Numbers in Binary…

- One's Complement
  - To take the one's complement of a binary number, all we need to do is **change each bit** in turn.
  - One's complement of $(10010100)_2$ ➔$(01101011)_2$
  - The easiest way to find the one's complement of a signed binary number when building **digital arithmetic** or logic decoder circuits is to use **Inverters**.
  - A **4-bit** representation in the one's complement format can be used to represent decimal numbers in the range from **-7 to +7** with **two** representations of zero.
  - For an n-bit signed binary number: (One's Complement)

$$\mathbf{-(2^{n-1} - 1) <= D <= +(2^{n-1} - 1)}$$

Negative Numbers in Binary…

- One's Complement
  - Binary subtraction using One's Complement of negative numbers → The end-around carry of the sum **is added** to the **least significant bit**.
  - Consider two 8 bit numbers: 5 – 3

```
=    5 + (-3)
=    0 0 0 0 0 1 0 1 + 1 1 1 1 1 1 0 0
=  1 0 0 0 0 0 0 0 1 (one carry bit as the 9th bit)
=    0 0 0 0 0 0 0 1
   +               1
   -----------------
=    0 0 0 0 0 0 1 0
=   +2
```

Negative Numbers in Binary…

- **Two’s Complement**
  - Two’s Complement or 2’s Complement as it is also termed, is another method like the previous sign-magnitude and one’s complement form, which we can use to represent negative binary numbers in a signed binary number system.
  - In two’s complement, the positive numbers are exactly the same as before for unsigned binary numbers.
  - A negative number, however, is represented by a binary number, which when added to its corresponding positive equivalent results in zero.

Negative Numbers in Binary…

- Two’s Complement
  - In two’s complement form, a negative number is the 2’s complement of its positive number with the subtraction of two numbers being A – B = A + (2’s complement of B) using much the same process as before as basically**, two’s complement is one’s complement + 1**.
  - The main advantage of two’s complement over the previous one’s complement is that there is **no double-zero problem** plus it is a **lot easier to generate** the two’s complement of a signed binary number.

Two’s Compliment (8 bit pattern)

0 1 1 1 1 1 1 1	=	+127
.		
.		
0 0 0 0 0 0 1 1	=	+3
0 0 0 0 0 0 1 0	=	+2
0 0 0 0 0 0 0 1	=	+1
0 0 0 0 0 0 0 0	=	0
1 1 1 1 1 1 1 1	=	-1
1 1 1 1 1 1 1 0	=	-2
1 1 1 1 1 1 0 1	=	-3
.		
1 0 0 0 0 0 0 1	=	-127
1 0 0 0 0 0 0 0	=	-128

Negative Numbers in Binary…

- Two's Complement
  - **Complementation** is an alternative way of representing negative binary numbers. This alternative coding system allows for the **subtraction of negative** numbers by using simple **addition**.
  - For an n-bit signed binary number: (Two's Complement)

    $$-(2^{n-1}) <= D <= +(2^{n-1} - 1)$$

  - Consider two 8 bit numbers: 5 – 3

    = **5 + (-3)**

    = **0 0 0 0 0 1 0 1 + 1 1 1 1 1 1 0 1**

    = **1 0 0 0 0 0 0 1 0** = **+2**

  - the 9th overflow bit is **disregarded** as we are only interested in the first 8-bits.

Negative Numbers in Binary…

- The representation of a negative integer (**Two's Complement**) is established by:
  - Start from the **signed binary representation** of its **positive** value
  - Copy the bit pattern from **right** to **left** until a **1** has been copied
  - Complement the remaining bits: all the **1's** with **0's**, and all the **0's** with **1's**
  - **An exception: 1 0 0 0 0 0 0 0 = -128**

Two’s Compliment benefits

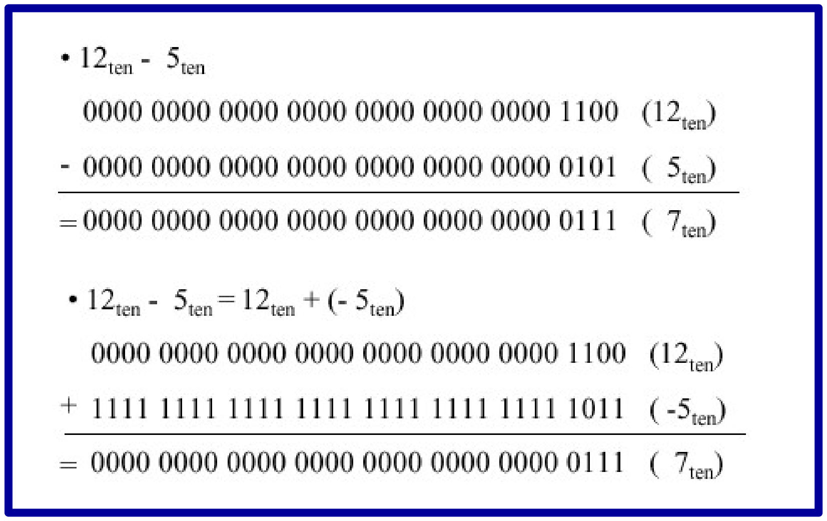
- One representation of zero
- Arithmetic works easily
- Negating is fairly easy

Ranges of Integer Representation

- 8-bit unsigned binary representation
  - Largest number:
  - Smallest number:
- 8-bit two’s complement representation
  - Largest number:
  - Smallest number:

Subtraction Vs Addition

- When subtracting two numbers, **two alternatives** present themselves.
- First, one can formulate a **subtraction** algorithm, which is distinct from addition.
- Second, one can **negate the subtrahend [**in (a – b), the subtrahend is b] then perform **addition**.
- Since we already know how to perform addition as well as twos complement negation, the second alternative is **more practical**.



• $12_{ten} - 5_{ten}$

$$\begin{array}{rr} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1100 \quad (12_{ten}) \\ - & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101 \quad (\ 5_{ten}) \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111 \quad (\ 7_{ten}) \end{array}$$

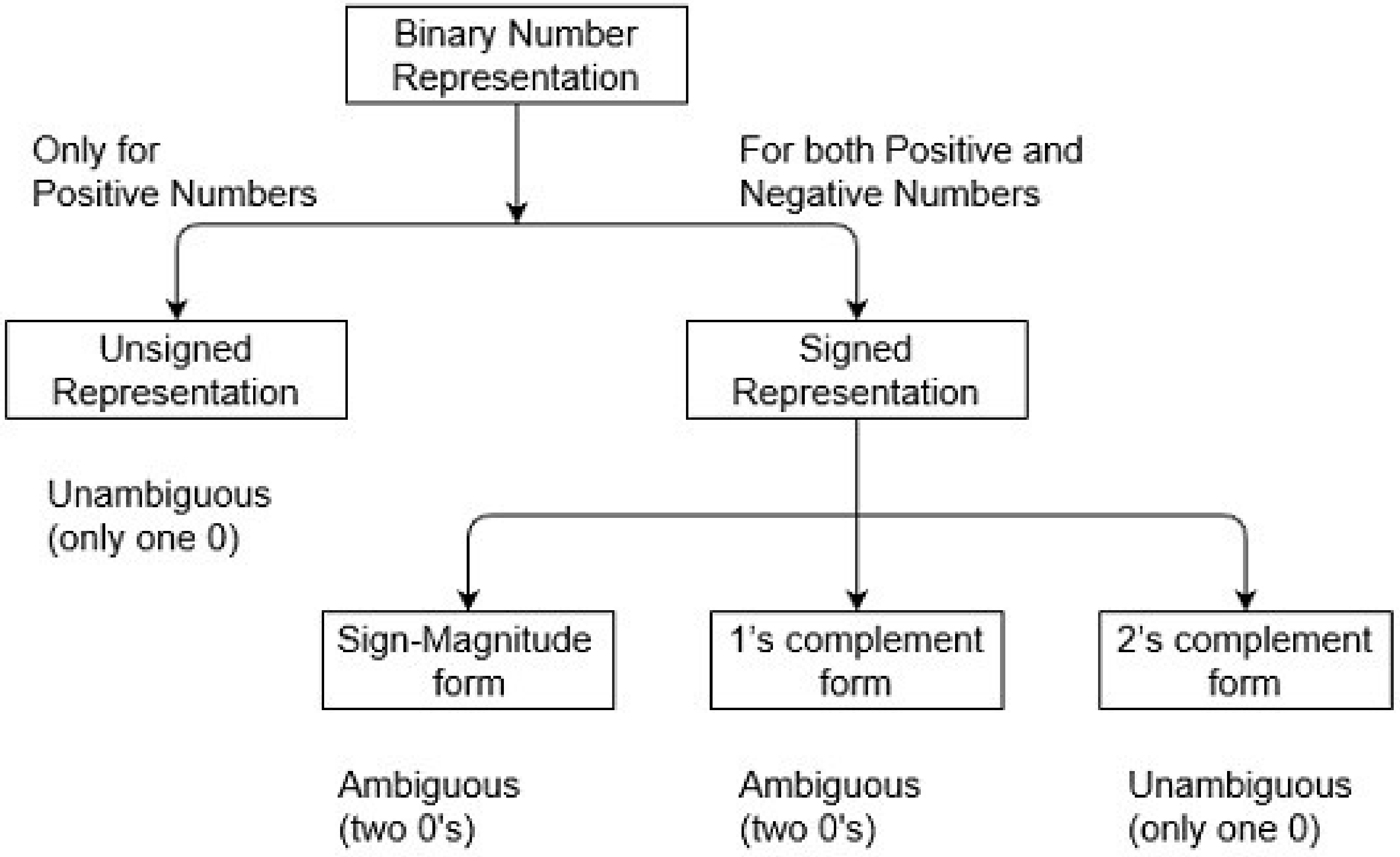
• $12_{ten} - 5_{ten} = 12_{ten} + (-5_{ten})$

$$\begin{array}{rr} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1100 \quad (12_{ten}) \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011 \quad (-5_{ten}) \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111 \quad (\ 7_{ten}) \end{array}$$

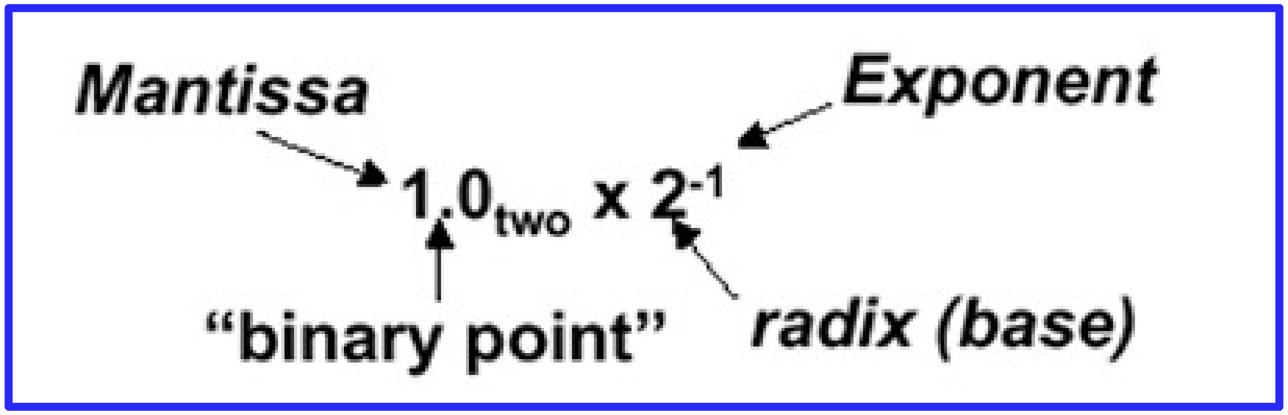
Binary Integers

- Questions:
  - Can you represent **+128** in 8-bit two’s complement?
  - Represent -100 and - (-100) in 8-bit two’s complement?
  - What are the standard implementation of Two’s–Complement arithmetic for different data representations (short, int, long, float, double) for C, C++, Java and Python.
  - Does a computer know the difference between signed and unsigned binary numbers?
  - Calculate the results of $(109)_{10}$ + $(49)_{10}$ as 8-bit signed numbers. Any problem? What is that?

Representation of Binary Numbers



Non-Integer (Fractions) Representation



Fractions in Decimal

16.357 = the SUM of ...

$7 * 10^{-3} = {}^{7}/_{1000}$

$5 * 10^{-2} = {}^{5}/_{100}$

$3 * 10^{-1} = {}^{3}/_{10}$

$6 * 10^{0} = 6$

$1 * 10^{1} = 10$

${}^{7}/_{1000} + {}^{5}/_{100} + {}^{3}/_{10} + 6 + 10 = \mathbf{16\ {}^{357}/_{1000}}$

Fractions in Binary

10.011 = the SUM of ...

$1 * 2^{-3} = {}^{1}/_{8}$

$1 * 2^{-2} = {}^{1}/_{4}$

$0 * 2^{-1} = 0$

$0 * 2^{0} = 0$

$1 * 2^{1} = 2$

${}^{1}/_{8} + {}^{1}/_{4} + 2 =$ **$2\ {}^{3}/_{8}$**

i.e. $10.011 = 2\ {}^{3}/_{8}$ in Decimal (Base 10)

Now it’s your turn …

UCSC

What is

011.0101

in Base 10?

Fractions in Binary

011.0101 = the SUM of ...

$1 * 2^{-4} = {}^{1}/_{16}$

$0 * 2^{-3} = 0$

$1 * 2^{-2} = {}^{1}/_{4}$

$0 * 2^{-1} = 0$

$1 * 2^{0} = 1$

$1 * 2^{1} = 2$

$0 * 2^{2} = 0$

${}^{1}/_{16} + {}^{1}/_{4} + 1 + 2 =$ **$3\ {}^{5}/_{16}$**

Decimal Scientific Notation

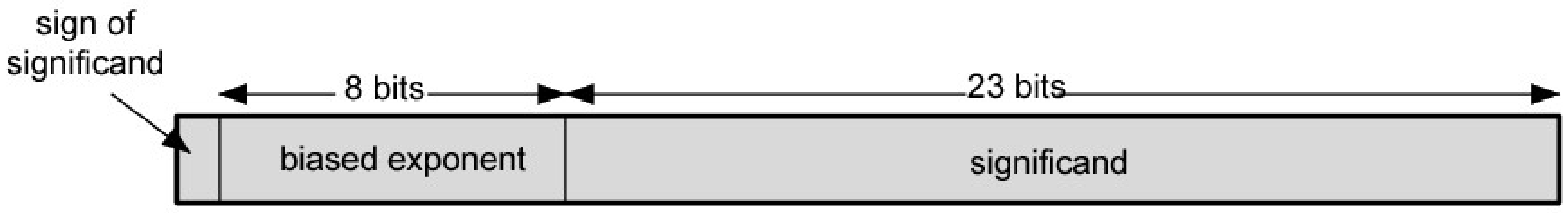
- Consider the following representation in decimal number …
  - **135.26** = **1.3526** x 10^{2}
  - **13526000** = **1.3526** x 10^{7}
  - **0.0000002452** = **2.452** x 10^{-7}
- **1.3526** x 10^{7} has the following components:
  - a Mantissa (Significand) = **3526**
  - an Exponent = **7**
  - a Base = **10**

Binary Scientific Notation

- Scientific notation for binary. Examples …
  - 0.000001001
  - 0.00001001 X 2^{-1}
  - 0.0001001 X 2^{-2}
  - 0.001001 X 2^{-3}
  - 0.01001 X 2^{-4}
  - 0.1001 X 2^{-5}
  - 1.001 X 2^{-6}

Binary Floating Point Representation

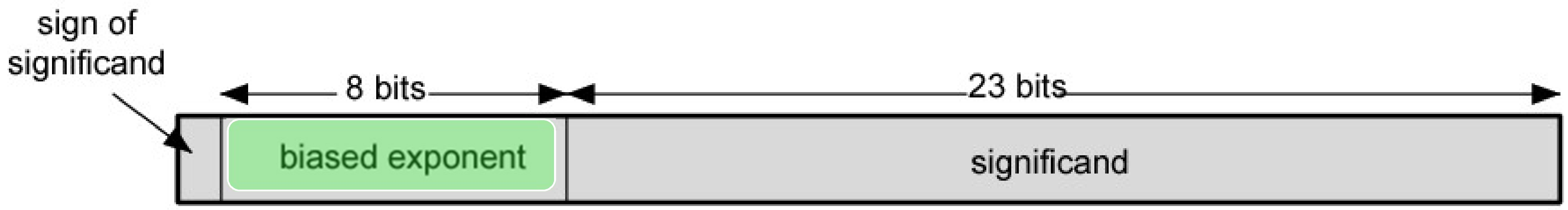
- We can represent a binary number in the form (32-bit Representation):



Binary Floating Point Number $= \boxed{\pm S \times B^{\pm E}}$

Significand = Mantissa = Coefficient

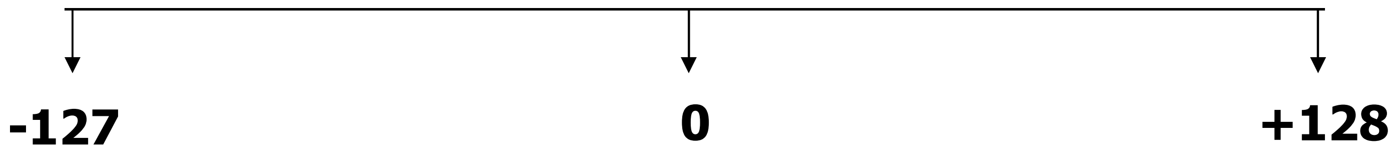
Binary Floating Point Representation



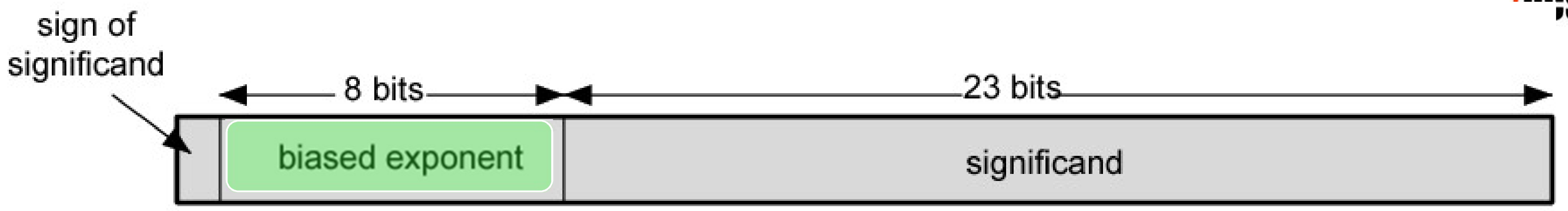
- The **exponent** value is stored in the **8 bits**.
- The representation used is known as a **biased representation**. A fixed value, called the **bias**, is **subtracted** from the field to get the **true exponent value**.
- Typically, the bias equals ($2^{k-1} - 1$), where k is the number of bits in the binary exponent.
- With 8-bit field a bias of 127 (2^7 - 1), the true exponent values are in the range **-127** to **+128** (instead of 0 – 255).

Biased Exponents (8 bit exponent)

8	7	6	5	4	3	2	1	BE
0	0	0	0	0	0	0	0	-127
0	0	0	0	0	0	0	1	-126
0	0	0	0	0	0	1	0	-125
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
1	1	1	1	1	1	0	1	+126
1	1	1	1	1	1	1	0	+127
1	1	1	1	1	1	1	1	+128



Binary Floating Point Representation

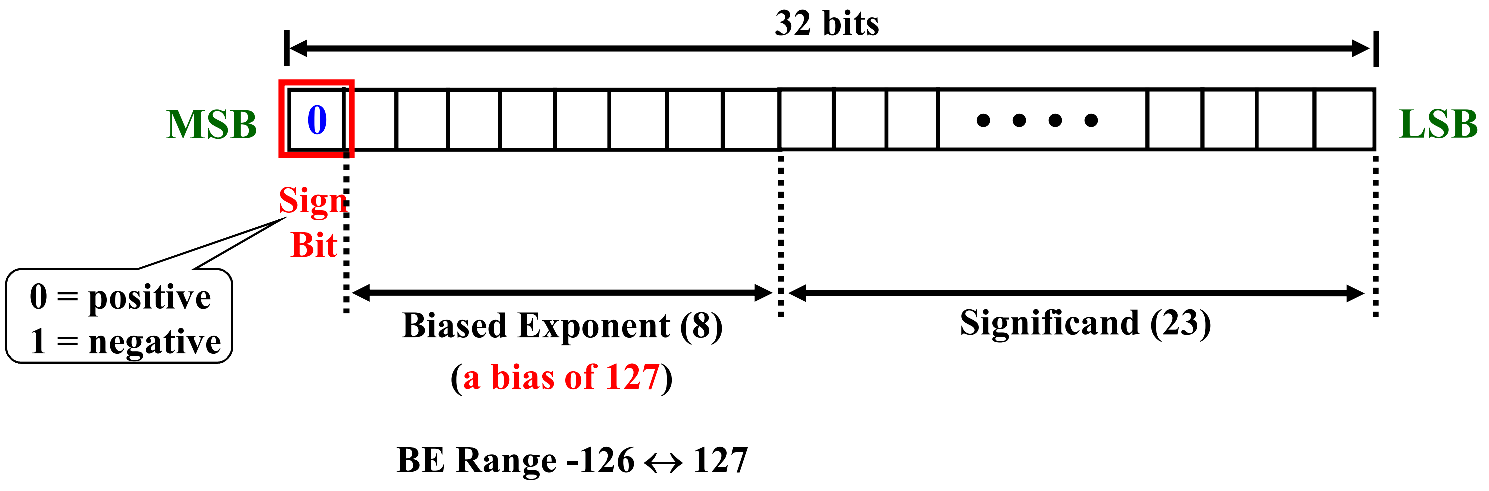


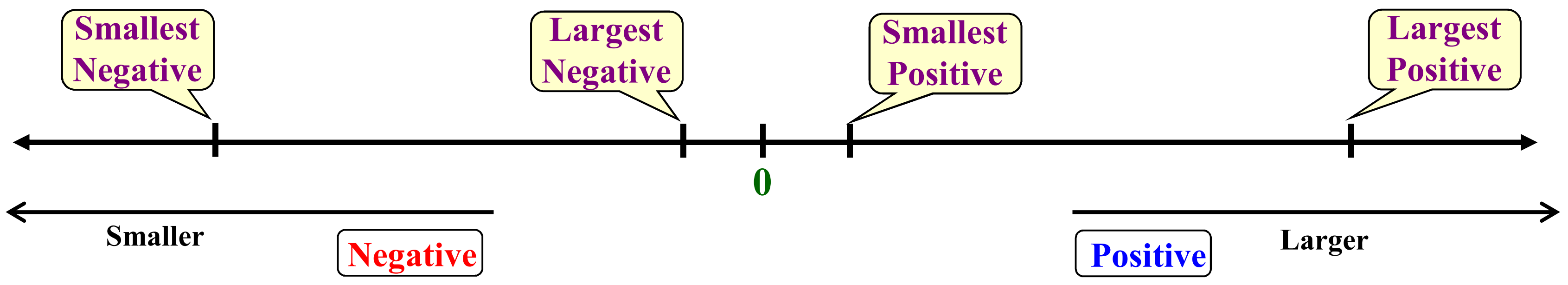
- In biasing the exponent, one does not use **2's complement** or a **sign bit**.
- Instead one **adds a bias** (equal to the magnitude of the **most negative number**) to the **exponents** and represents the **result** of that addition.
- Exponents of −127 (**all 0s**) and +128 (**all 1s**) are **reserved for special values**.
- –126 ≤ **exponent** ≤ +127

Special Value	**Exponent**	**Significand**
+/- 0	0000 0000	0
Denormalized number	0000 0000	Nonzero
NaN	1111 1111	Nonzero
+/- infinity	1111 1111	0

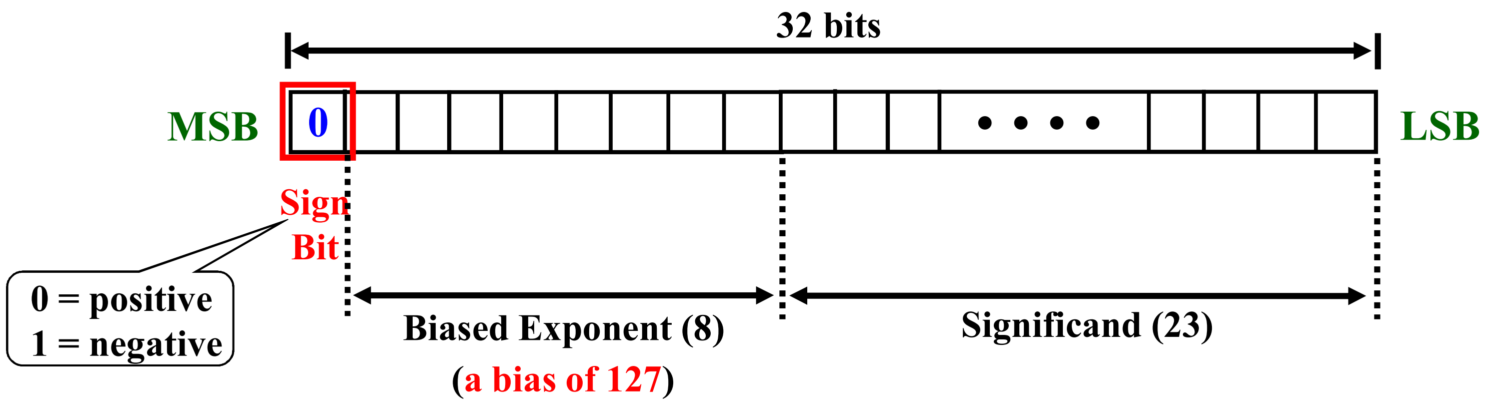
Binary Floating Point Range (32-bit)

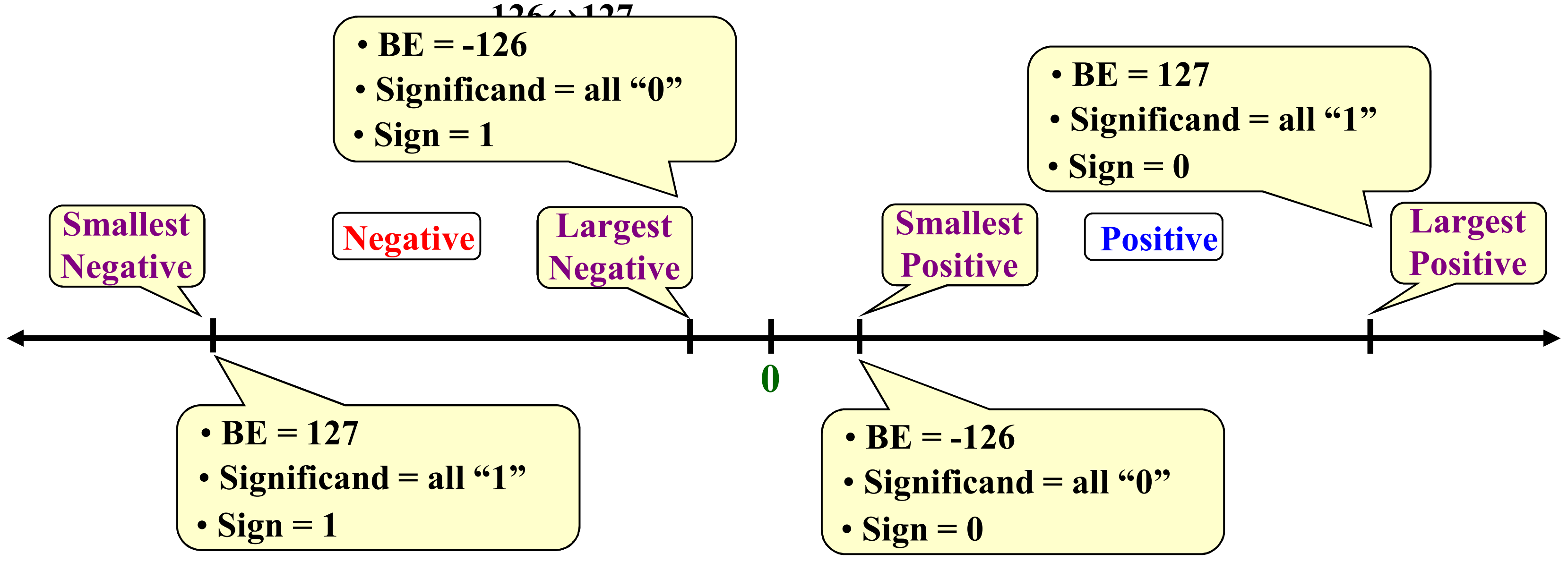
Format of typical floating-point notations





Binary Floating Point Range (32 bit)





Binary Floating Point Representation

- 1/0 = ∞
- log (0) = -∞
- sqrt (-1) = NaN

Special Value	Exponent	Significand
+/- 0	0000 0000	0
Denormalized number	0000 0000	Nonzero
NaN	1111 1111	Nonzero
+/- infinity	1111 1111	0

- Smaller numbers are possible, but those numbers must be represented with a zero exponent and a mantissa whose leading bit(s) are zero, which leads to a loss of precision. These numbers are called **denormalized numbers** or **denormals** (newer specifications refer to these as **subnormal** numbers)

Binary Floating Point Representation

- A **normal number** is one in which the **most significant** digit of the significand is **nonzero**.
- A normal nonzero number is one in the form:

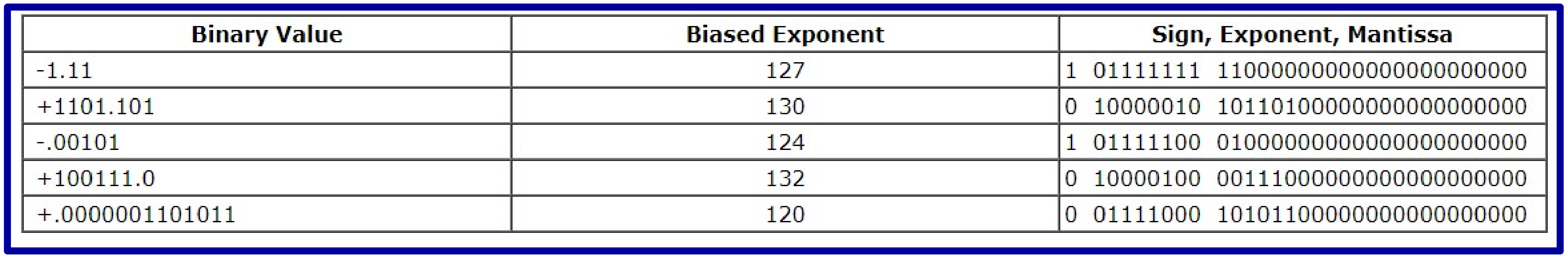
$$\boxed{\boldsymbol{Number} = \pm 1.\boldsymbol{b_1 b_2 b_3 b_4} \ldots \boldsymbol{bn} \times 2^{\pm E}}$$

$$where\ \boldsymbol{b_i}\ is\ a\ binary\ digit$$

- Because the most significant bit is always **one**, it is unnecessary **to store this bit**; rather, it is **implicit bit**.
- Given a number that is not normal, the number may be **normalized by shifting** the radix point to the right of the leftmost 1 bit and adjusting the exponent accordingly.

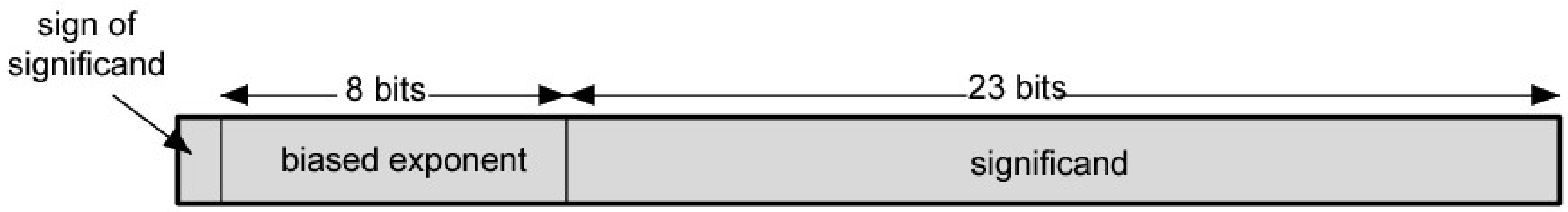
Binary Floating Point Representation

- You may have noticed that in a **normalized mantissa**,
- the **digit 1** always appears to the **left of the binary point**.
- In fact, the **leading 1 is omitted** from the **mantissa's actual storage** because it is redundant.



Binary Value	Biased Exponent	Sign, Exponent, Mantissa
-1.11	127	1 01111111 11000000000000000000000
+1101.101	130	0 10000010 10110100000000000000000
-.00101	124	1 01111100 01000000000000000000000
+100111.0	132	0 10000100 00111000000000000000000
+.0000001101011	120	0 01111000 10101100000000000000000

Binary Floating Point Representation (32 bit)



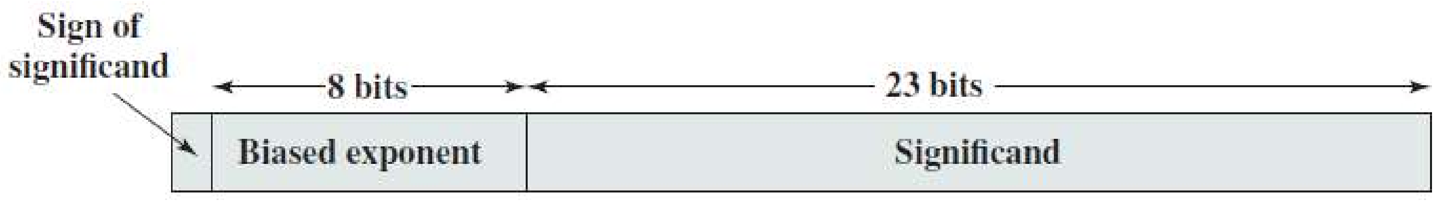
- The sign is stored in the first bit of the word.
- The first bit of the **true significand** is always **1** and need **not be stored** in the significand field.
- The value **127** is **added** to the true exponent to be stored in the exponent field.
- The base is **2**.

For 32 bit Floating Point Representation:

Real Exponent + 127 → Biased Exponent

Binary Floating Point Representation (32 bit)

- A **normal number** is one in which the **most significant** digit of the significand is **nonzero**.
- This is called **Normalization**.
- A normal nonzero number is one in the form:



(a) Format

$$\begin{array}{rcl}
1.1010001 \times 2^{10100} & = 0\ 10010011\ 10100010000000000000000 = & 1.6328125 \times 2^{20} \\
-1.1010001 \times 2^{10100} & = 1\ 10010011\ 10100010000000000000000 = & -1.6328125 \times 2^{20} \\
1.1010001 \times 2^{-10100} & = 0\ 01101011\ 10100010000000000000000 = & 1.6328125 \times 2^{-20} \\
-1.1010001 \times 2^{-10100} & = 1\ 01101011\ 10100010000000000000000 = & -1.6328125 \times 2^{-20}
\end{array}$$

(b) Examples

Figure 10.18 Typical 32-Bit Floating-Point Format

Computer Organization and Architecture (10th Edition) by William Stallings

Binary Floating Point Representation (32 bit)

E.g. Represent 263.3_{10} in 32-bit binary floating point number?

263 ➔ 100000111

0.3 ➔ .0100110011001101

263.3 ➔ 100000111.0100110011001101

➔1.000001110100110011001101x2^{8} (Normalized No.)

Exponent ➔ 8 + 127 = 135_{10} ➔ 10000111

S	Exponent (8)	Significand (23)
0	1000 0111	00000111010011001100110

Binary Floating Point Representation (32 bit)

0	0110 1000	101 0101 0100 0011 0100 0010

- Sign: 0 => positive
- Exponent:
  - $0110\ 1000_2 = 104_{10}$
  - Bias adjustment: 104 - 127 = -23
- Significand:

  $\mathbf{= 1} + 1\text{x}2^{-1} + 0\text{x}2^{-2} + 1\text{x}2^{-3} + 0\text{x}2^{-4} + 1\text{x}2^{-5} + \ldots$

  $=1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$

  $= 1.0 + 0.666115$
- Represents: $\mathbf{1.666115 \text{ x } 2^{-23}}$

Binary Floating Point Representation (32 bit)

0	0110 1000	101 0101 0100 0011 0100 0010

$$\boldsymbol{Value} = (-1)^{sign} x\, 2^{(E-127)} x\, (1 + \sum_{i=1}^{23} b_i\, x\, 2^{-i})$$

E= 104_{10} and **Significand** = 0.666115_{10}

$$\textbf{Value} = (-1)^0 \times 2^{104-127} \times (1 + 0.666115)$$
$$= 1.666115 \times 2^{-23}$$

IEEE Standard for Binary Floating-Point

- The most important floating-point representation is defined in **IEEE Standard 754**, adopted in **1985** and revised in **2008**.
- This standard was developed to facilitate the **portability** of programs from **one processor to another** and to encourage the development of **sophisticated**, numerically oriented programs.
- The standard has been widely adopted and is used on **virtually all contemporary processors** and arithmetic coprocessors.
- **IEEE 754-2008** covers both **binary** and **decimal** floating-point representations.

IEEE Standard for Binary Floating-Point

- The IEEE Standard for **Floating-Point Arithmetic** (IEEE 754) uses various sizes of exponent, but also uses **offset notation** for the format of each precision.
- It uses excess $\mathbf{2^{n-1} - 1}$ (i.e. excess-**15**, excess-**127**, excess-**1023**, excess-**16383**).
- This **biased representation**, is a digital coding scheme where **all-zero** corresponds to the **minimal negative** value and **all-one** to the **maximal positive** value.

IEEE Standard for Binary Floating-Point

- The three basic binary formats have bit lengths of **32**, **64**, and **128** bits, with exponents of **8**, **11**, and **15** bits, respectively.

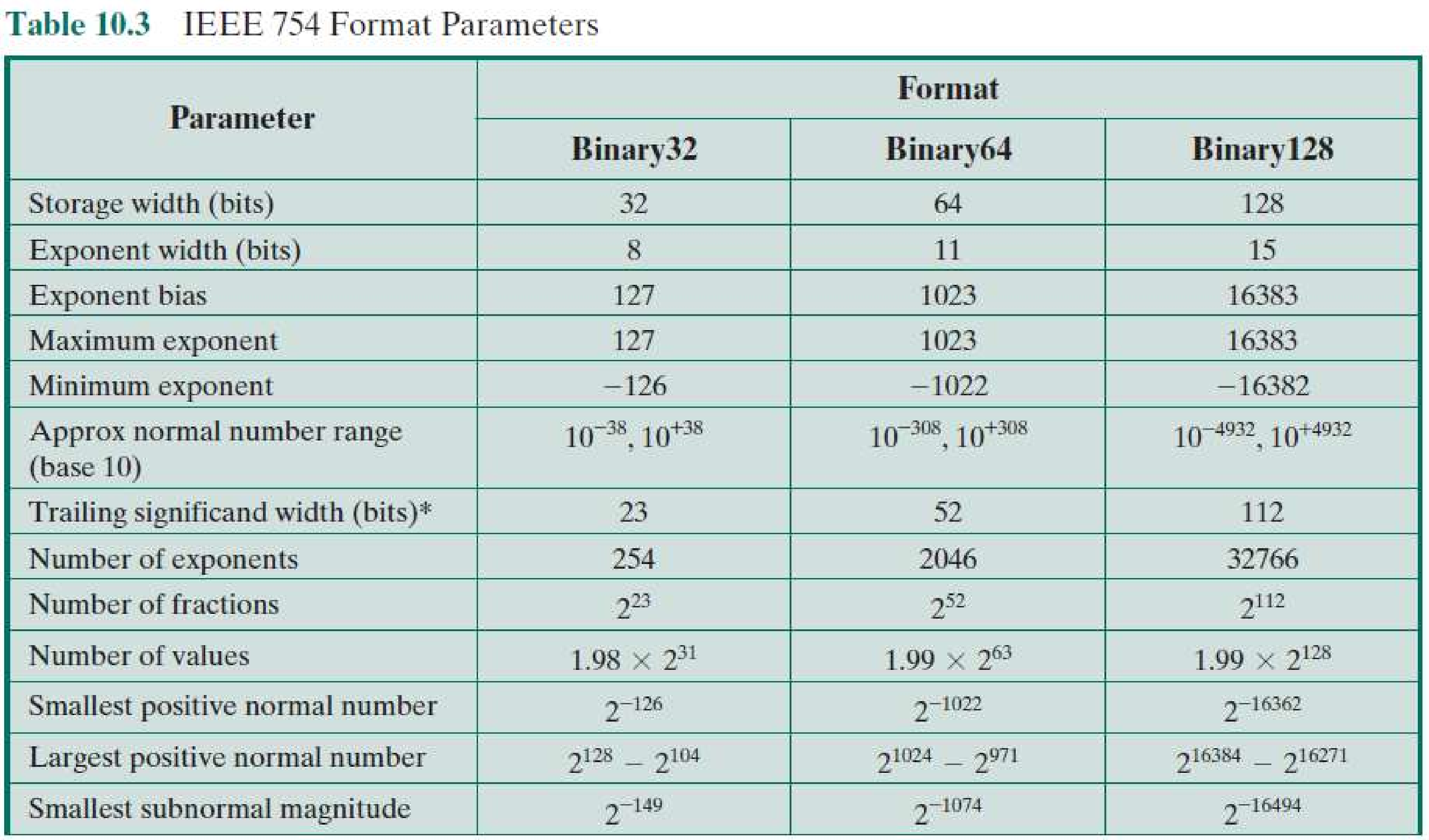
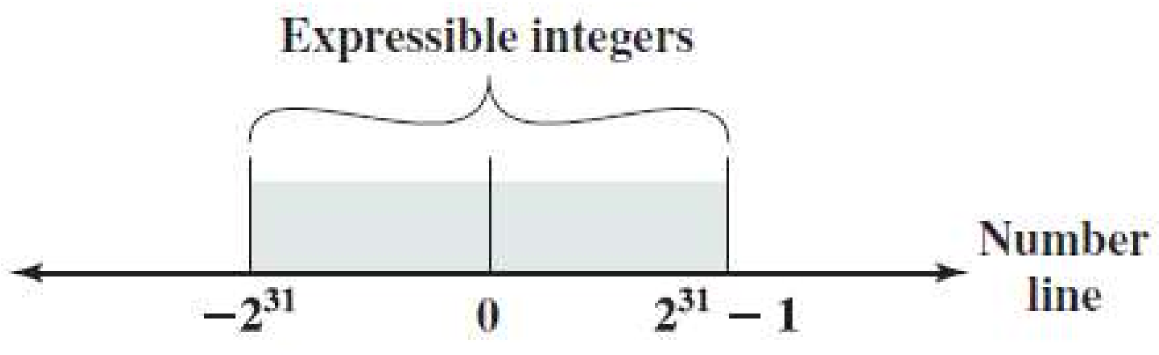


Table 10.3 IEEE 754 Format Parameters

Parameter	Format		
	Binary32	Binary64	Binary128
Storage width (bits)	32	64	128
Exponent width (bits)	8	11	15
Exponent bias	127	1023	16383
Maximum exponent	127	1023	16383
Minimum exponent	-126	-1022	-16382
Approx normal number range (base 10)	$10^{-38}, 10^{+38}$	$10^{-308}, 10^{+308}$	$10^{-4932}, 10^{+4932}$
Trailing significand width (bits)*	23	52	112
Number of exponents	254	2046	32766
Number of fractions	2^{23}	2^{52}	2^{112}
Number of values	1.98×2^{31}	1.99×2^{63}	1.99×2^{128}
Smallest positive normal number	2^{-126}	2^{-1022}	2^{-16362}
Largest positive normal number	$2^{128} - 2^{104}$	$2^{1024} - 2^{971}$	$2^{16384} - 2^{16271}$
Smallest subnormal magnitude	2^{-149}	2^{-1074}	2^{-16494}

Binary Floating Point Range

- The range of numbers that can be represented in a **32-bit** word. Using **Two's complement integer** representation, all of the integers from (-2^{31}) to ($2^{31} – 1$) can be represented, for a total of **2^{32}** different numbers.



(a) Twos complement integers

Binary Floating Point Range

- A 32-bit **floating-point format** can represent:

  Negative numbers between $-(2 - 2^{-23}) * 2^{128}$ and $- 2^{-127}$

  Positive numbers between 2^{-127} and $(2 - 2^{-23}) * 2^{128}$

- IEEE 754 32-bit base-2 floating-point variable has a maximum value of $(2 − 2^{−23}) × 2^{127} ≈ 3.4028235 × 10^{38}$.

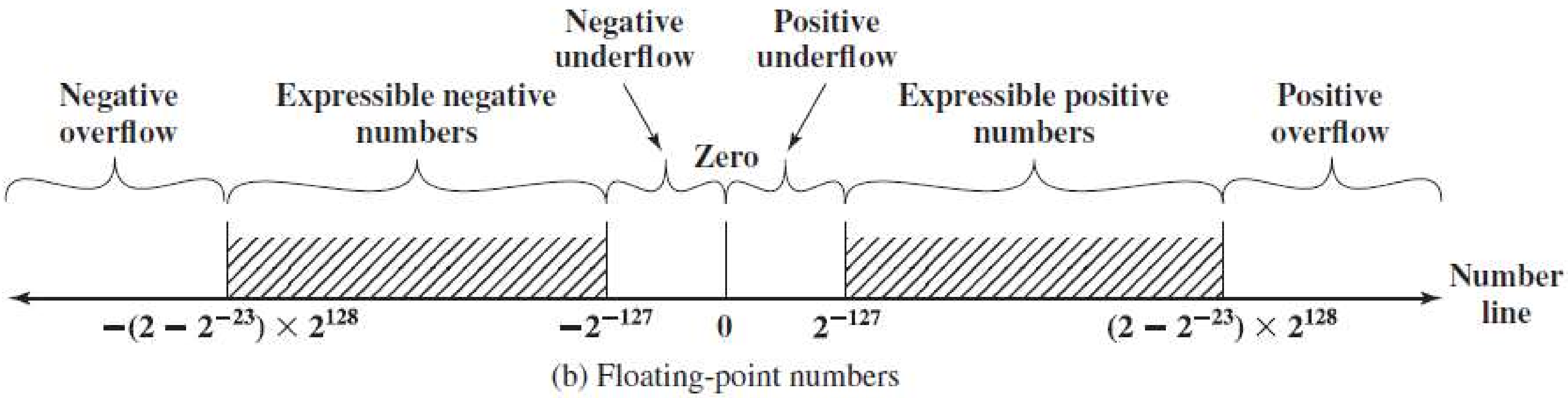
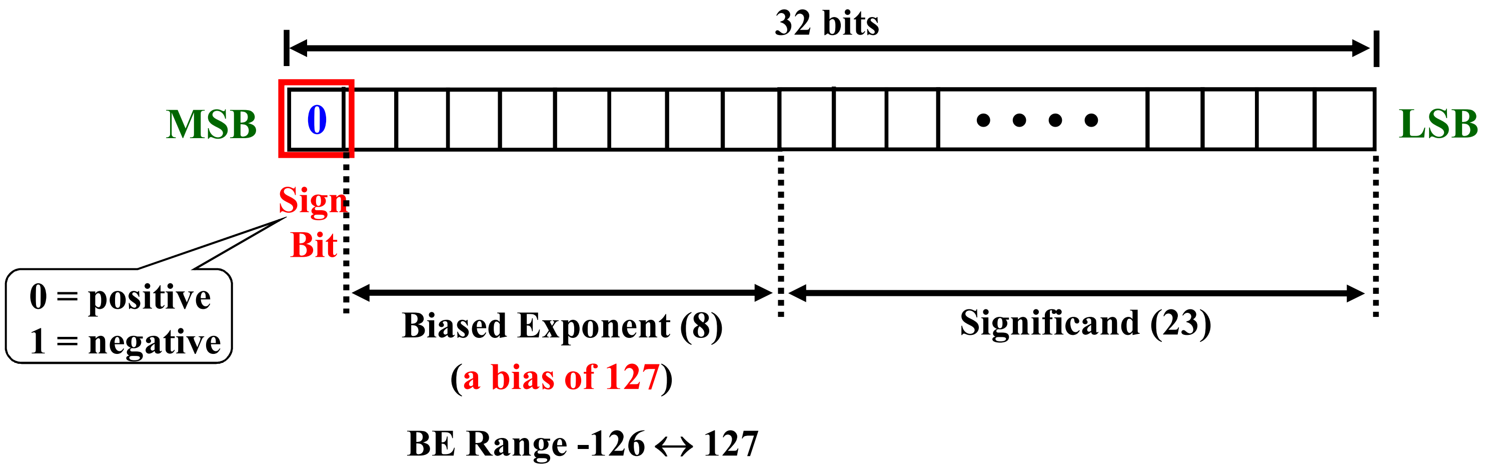
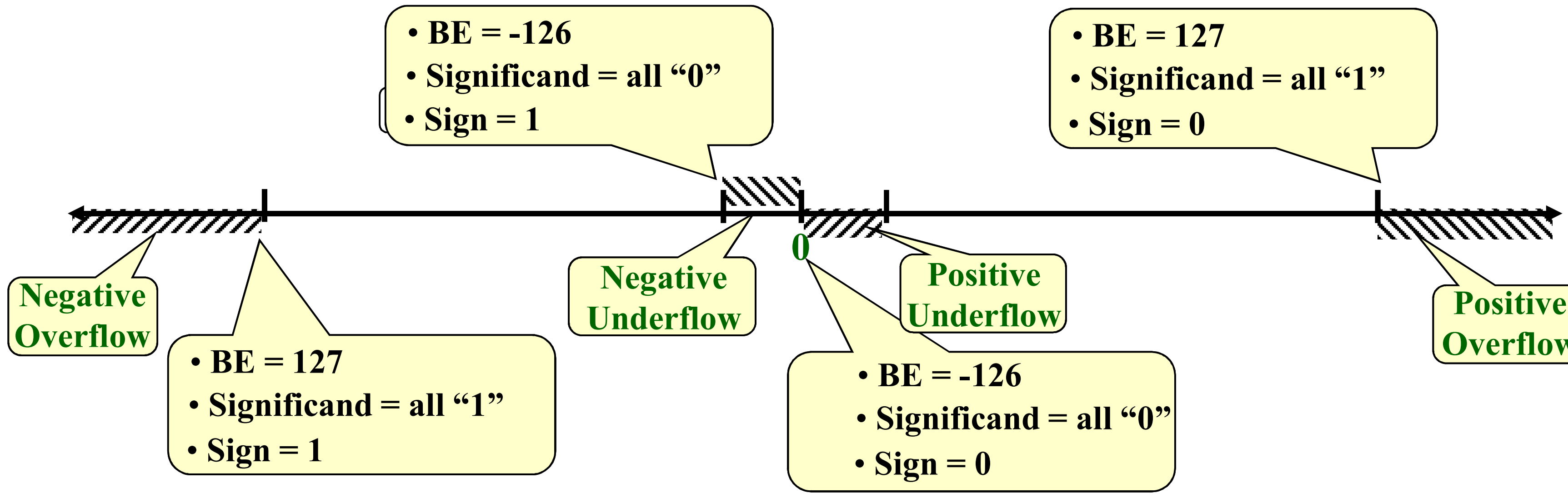

Negative overflow
Expressible negative numbers
Negative underflow
Zero
Positive underflow
Expressible positive numbers
Positive overflow
Number line
$-(2 - 2^{-23}) \times 2^{128}$
-2^{-127}
0
2^{-127}
$(2 - 2^{-23}) \times 2^{128}$
(b) Floating-point numbers

Figure 10.19 Expressible Numbers in Typical 32-Bit Formats

Binary Floating Point Range (32 bit)

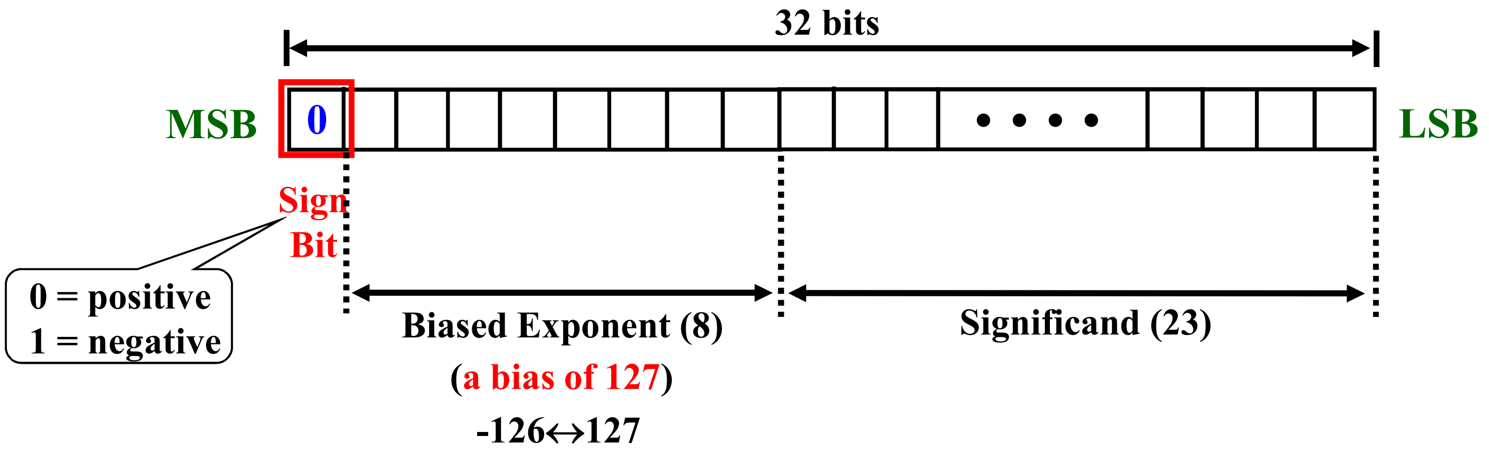
Format of typical floating-point notations

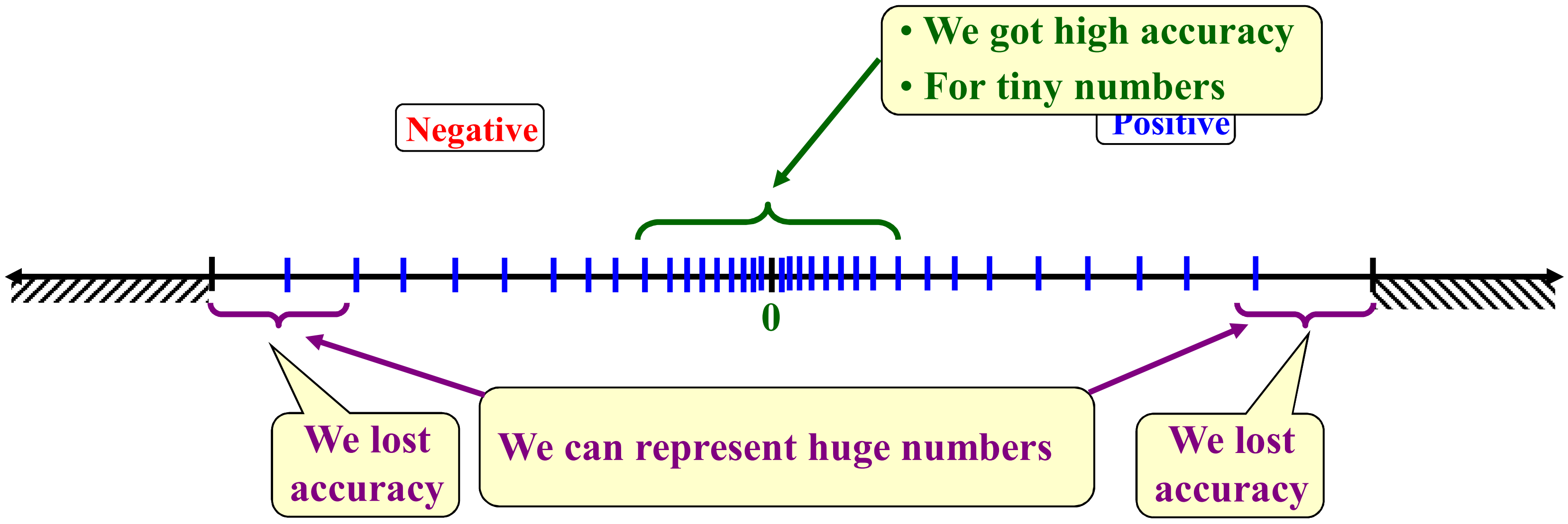




Binary Floating Point Range (32 bit)

Format of typical floating-point notations

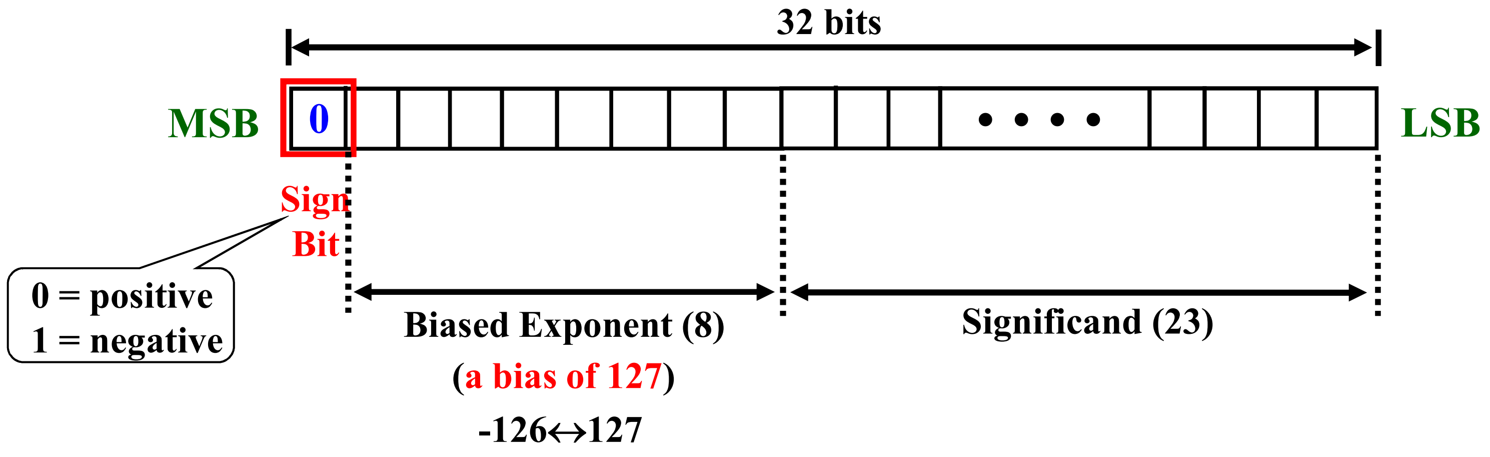




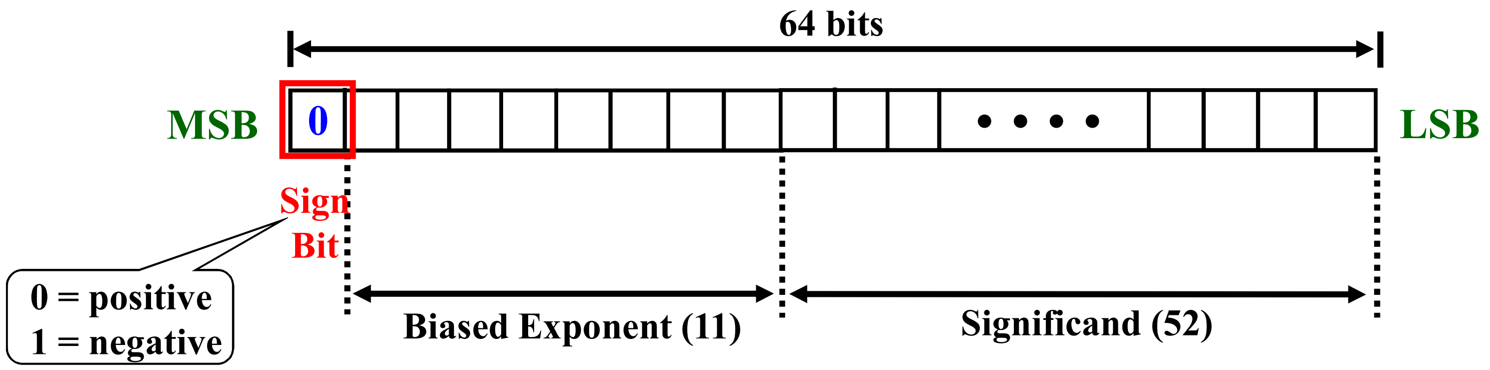
Binary Floating Point Range

IEEE-754 Floating-Point Notations

① **Single-Float** (single-precision floating-point) – "*float*" in C/C++



② **Double-Float** (double-precision floating-point) – "*double*" in C/C+



Floating Point Overflow

- Floating point representations can overflow, e.g.,

$$\begin{array}{r} 1.111111 \times 2^{127} \\ +\ 1.111111 \times 2^{127} \\ \hline 11.111110 \times 2^{127} \end{array}$$

$$\boxed{1.1111110 \times 2^{128} \approx \infty}$$

Floating Point Underflow

- Floating point numbers can also get too *small*, e.g.,

$$\begin{array}{r} 10.010000 \times 2^{-126} \\ \div\ 11.000000 \times 2^{0} \\ \hline 0.110000 \times 2^{-126} \end{array}$$

$$\boxed{1.100000 \times 2^{-127} \approx 0}$$

Limitations

- Floating-point representations **only approximate** real numbers
- Using a greater number of bits in a representation can reduce errors but can **never eliminate** them
- Floating point **errors**
  - **Overflow/underflow** can cause programs to crash
  - Can lead to erroneous results / hard to detect
- When the **smaller** floating-point number is shifted to make the exponents equal, some of the **less significant bits are lost**.
- This loss of information (precision) is known as **rounding**.

Round-off Error

- A roundoff error (rounding error) is the **difference between** the result produced by a given algorithm using **exact arithmetic** and the result produced by the same algorithm using **finite-precision**, **rounded arithmetic**.
- Rounding errors are due to **inexactness in the representation** of real numbers and the arithmetic operations done with them.
- This is a form of **quantization** error.

Round-off Error

- When using approximation equations or algorithms, especially when using **finitely many digits to represent real numbers**, one of the goals of numerical analysis is to **estimate computation errors**.
- **Computation errors**, also called **numerical errors**, include both **truncation** errors and **roundoff** errors. (**truncation** means to chop off the fractional portion of a number or **chopping** errors)

Suppose we want to represent π:
3.14159265358979323846264338327950.....

Represent it as 6 digits (decimal) in mantissa:

3.141592 (if we truncate)
3.141593 (if we round)

Round-off Error

- There are **two** common rounding rules, **round-by-chop** and **round-to-nearest**.
- The **IEEE** standard uses **round-to-nearest**.
- **Round-by-chop** – If the left over is **less than 0.5** then the floating point number is **truncated/chopped** after the mantissa's maximum digit position.
- **Round-to-nearest** – If the left over is **greater than 0.5** then **add 1 to LSB** of the mantissa (carry as necessary) of the floating point number.
  - If the left over is **0.5** and LSB of mantissa is one (1) then **add 1 to LSB** else do **not add 1 to LSB** .

Round-off Error

What is the 16-bit floating point representation for decimal 1008.8125?

Use 16-bit representation ➔ **1**-bit sign, **5**-bit exponent and **10**-bit mantissa.

1008 = 1111110000

0.8125 = 0.1101

1008.8125 = 1111110000.1101

$= 1.1111100001101 \times 2^9$

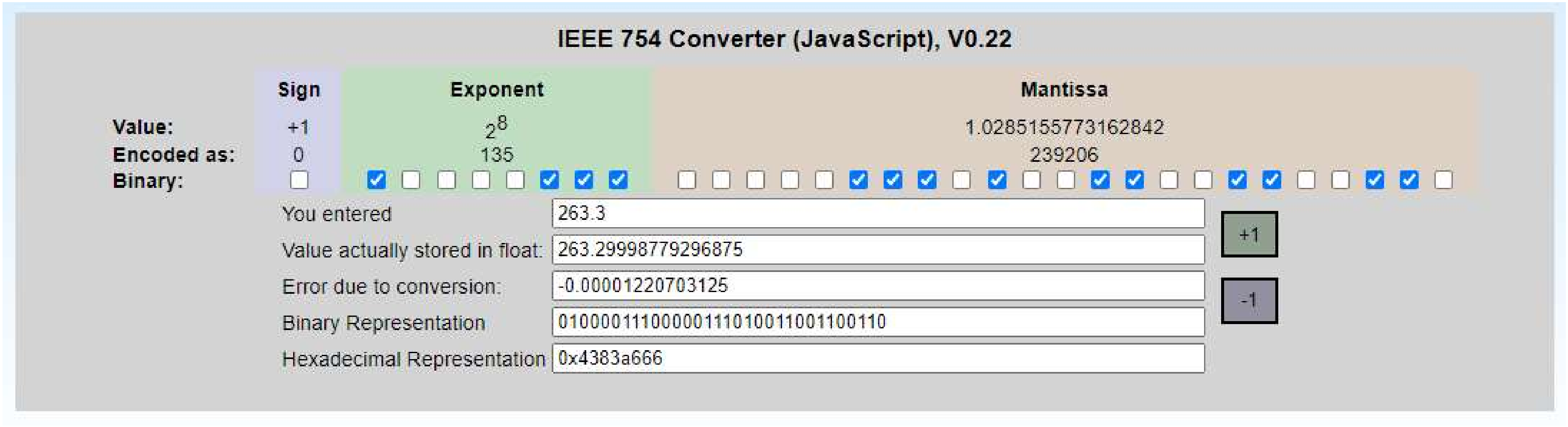
Biased Exponent = 9 +15 =24 = $(11000)_2$

0 11000 1111100001101

After rounding the mantissa: (add 1)

0 11000 1111100010

Binary Floating Point Representation (32 bit)



IEEE 754 Converter (JavaScript), V0.22

	Sign	Exponent	Mantissa
Value:	+1	2^8	1.0285155773162842
Encoded as:	0	135	239206
Binary:	☐	✓ ☐ ☐ ☐ ☐ ✓ ✓ ✓	☐ ☐ ☐ ☐ ☐ ✓ ✓ ✓ ☐ ✓ ☐ ☐ ✓ ✓ ☐ ☐ ✓ ✓ ☐ ☐ ✓ ✓ ☐

You entered	263.3
Value actually stored in float:	263.29998779296875
Error due to conversion:	-0.00001220703125
Binary Representation	01000011100000111010011001100110
Hexadecimal Representation	0x4383a666

https://www.h-schmidt.net/FloatConverter/IEEE754.html

22/7 (rational number) 3.142857142857142857142857142857...

Pi (irrational number) 3.14159265358979323846264338327950...

Difference 0.00126448926734961868021375957764

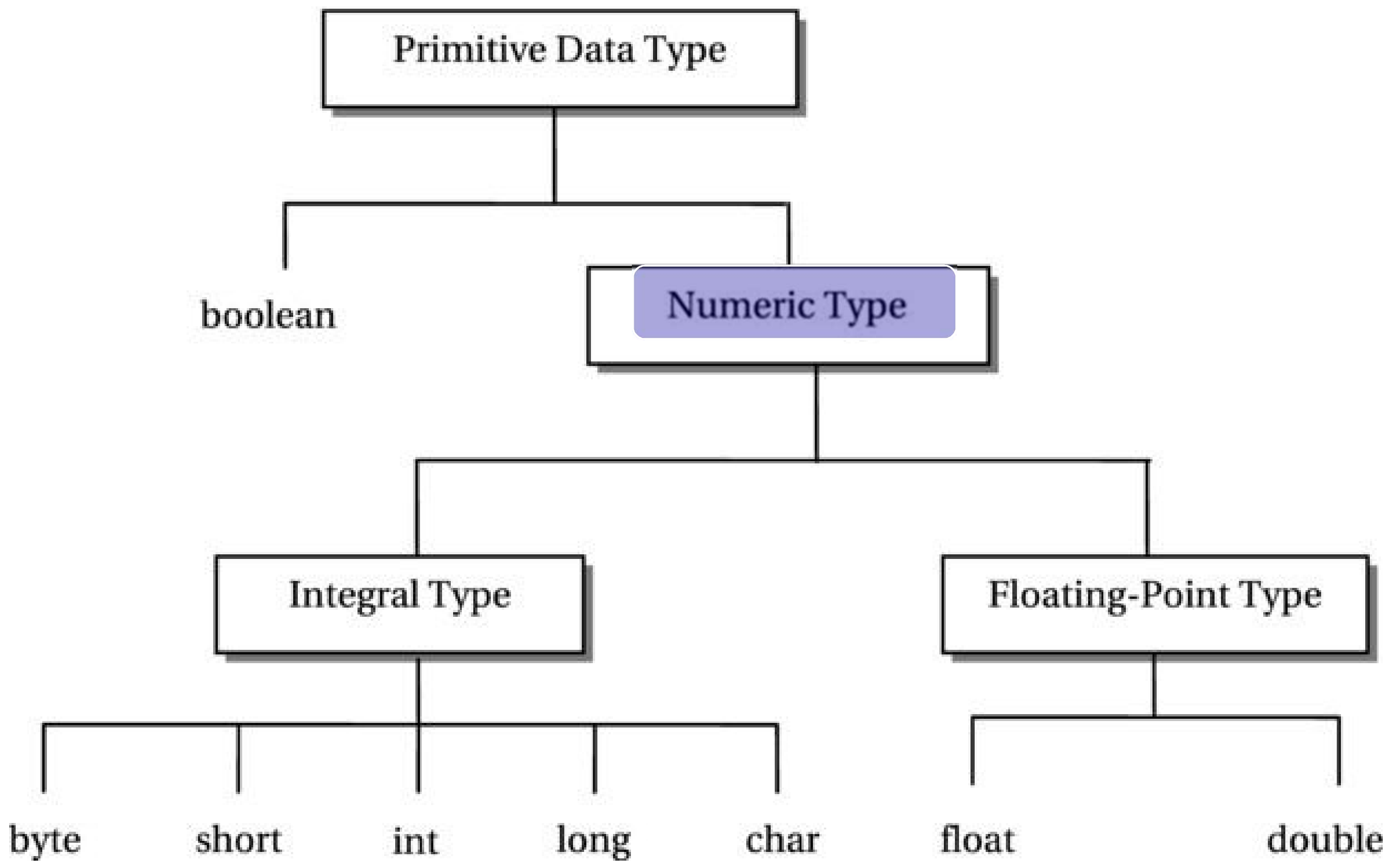
Real world example

Patriot missile failure due to magnification of roundoff error

On 25 February 1991, during the Gulf War, an American Patriot missile battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi Scud missile.

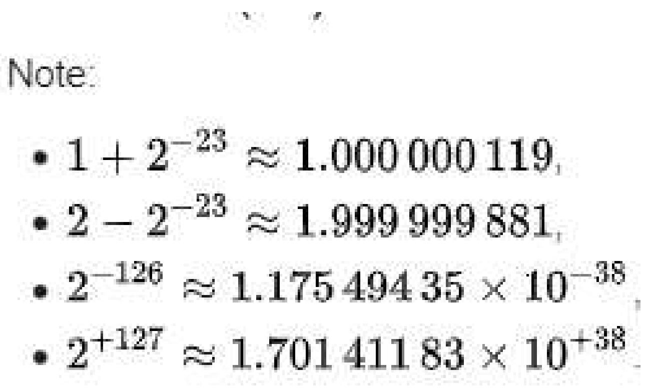
A report of the then-General Accounting Office entitled "Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia" reported on the cause of the failure: an inaccurate calculation of the time since boot due to computer arithmetic errors.

Summary


Primitive Data Type
boolean
Numeric Type
Integral Type
Floating-Point Type
byte
short
int
long
char
float
double

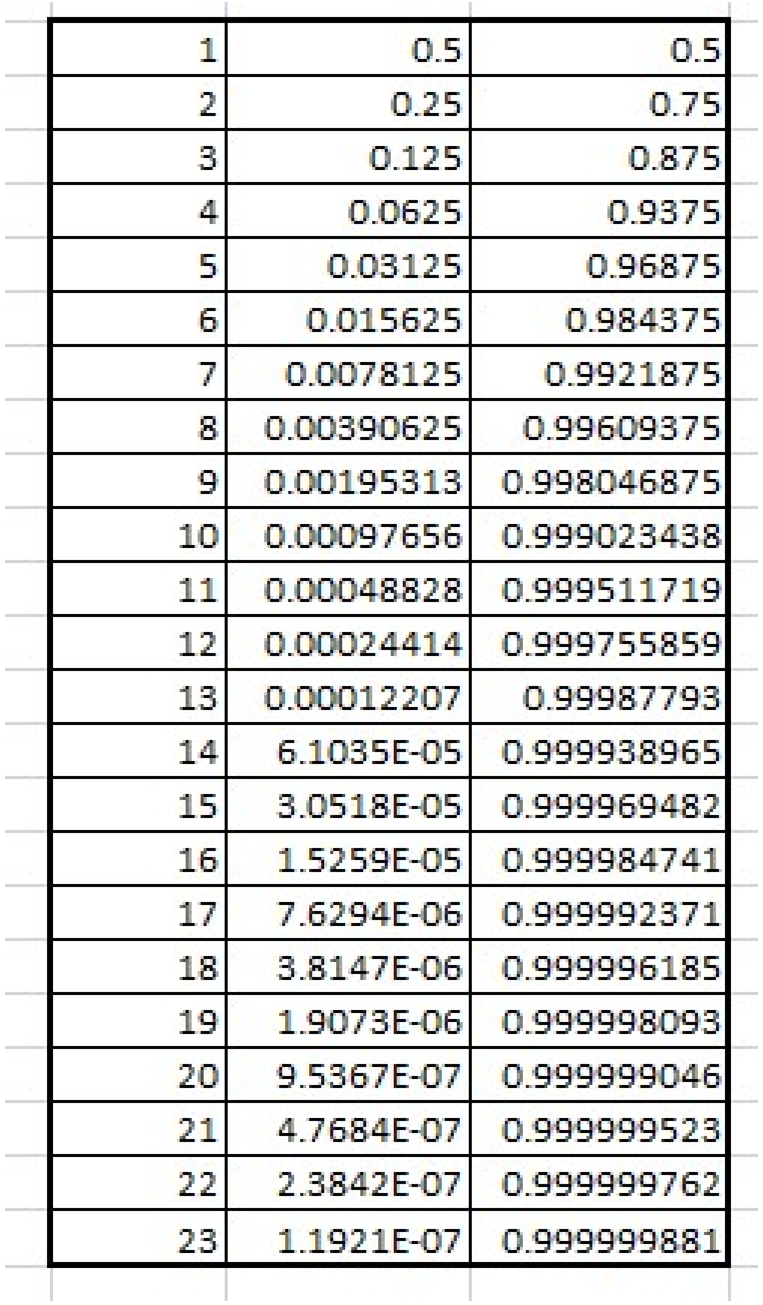
End of the Lecture …

Components of 2^{-i}



Note:

- $1 + 2^{-23} \approx 1.000\,000\,119$,
- $2 - 2^{-23} \approx 1.999\,999\,881$,
- $2^{-126} \approx 1.175\,494\,35 \times 10^{-38}$,
- $2^{+127} \approx 1.701\,411\,83 \times 10^{+38}$.



1	0.5	0.5
2	0.25	0.75
3	0.125	0.875
4	0.0625	0.9375
5	0.03125	0.96875
6	0.015625	0.984375
7	0.0078125	0.9921875
8	0.00390625	0.99609375
9	0.00195313	0.998046875
10	0.00097656	0.999023438
11	0.00048828	0.999511719
12	0.00024414	0.999755859
13	0.00012207	0.99987793
14	6.1035E-05	0.999938965
15	3.0518E-05	0.999969482
16	1.5259E-05	0.999984741
17	7.6294E-06	0.999992371
18	3.8147E-06	0.999996185
19	1.9073E-06	0.999998093
20	9.5367E-07	0.999999046
21	4.7684E-07	0.999999523
22	2.3842E-07	0.999999762
23	1.1921E-07	0.999999881