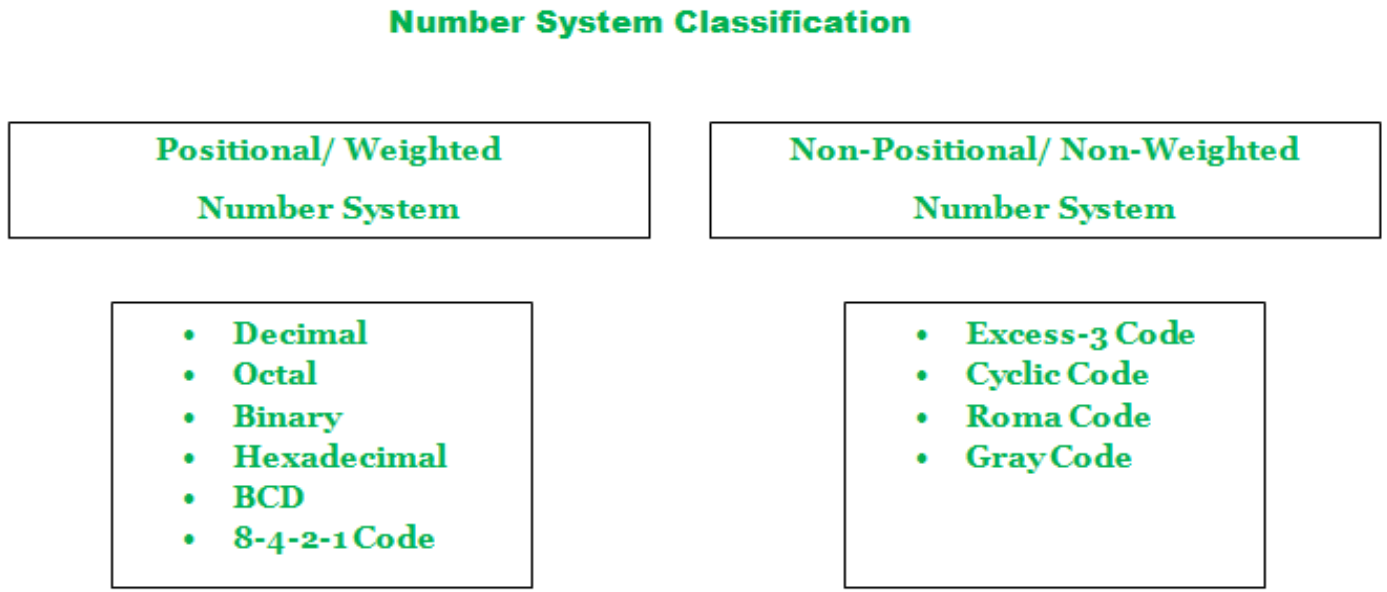
Number System

A **number** is a method used for representing an arithmetic value, measure, or count, of a physical quantity. A number system is defined as a method of naming and representing numbers. The concept of **number system** helps in defining the rules associated with the numbers and different operations on numbers.

A number system is determined with the help of its radix or base. The **radix** or **base** of a number system is nothing but the total number of symbols used in the number system for representing the different numbers.

The number systems are broadly classified into two types, namely −

1. Positional Number System
2. Non-Positional Number System



Positional Number System

Positional number system is the type of number system in which the weight or value of the digit (or symbol) depends upon its position in the number. The positional number system is also known as **weighted number system.** This is because, in the positional number system, there is a weight associated with the position in the number.
Therefore, in the positional number system, each digit of the number is weighted according to its position of occurrence in the number. When we travel toward left along the number, the weights increase by a constant factor that is equivalent to the base of the number system. Also, in the positional number system, a **radix point (.)** is used to differentiate the positions corresponding to integral weights from the positions corresponding to the fractional weights.

Types of Positional Number Systems
There are four very popular positional number systems, which are:
1. Decimal Number System 2. Binary Number System 3. Octal Number System 4. Hexadecimal Number System

1. Decimal Number Systems The number system is having digit 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; this number system is known as a decimal number system because total ten digits are involved. The base of the decimal number system is 10.

2. Binary Number Systems The modern computers do not process decimal number; they work with another number system known as a binary number system which uses only two digits 0 and1.The base of binary number system is 2 because it has only two digit 0 and 1.

3. Octal Numbers The base of a number system is equal to the number of digits used, i.e., for decimal number system the base is ten while for the binary system the base is two. The octal system has the base of eight as it uses eight digits 0, 1, 2, 3, 4, 5, 6, 7.

4. Hexadecimal Numbers These numbers are used extensively in microprocessor work.The hexadecimal number system has a base of 16, and hence it consists of the following sixteen number of digits.
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Non-Positional (or Non-weighted) Number System

Non-positional number system is also known as non-weighted number system. Digit value is independent of its position. Non-positional number system is used for shift position encodes and error detecting purpose.
Few examples of non-weighted number system are gray code, roman code, excess-3 code, etc.

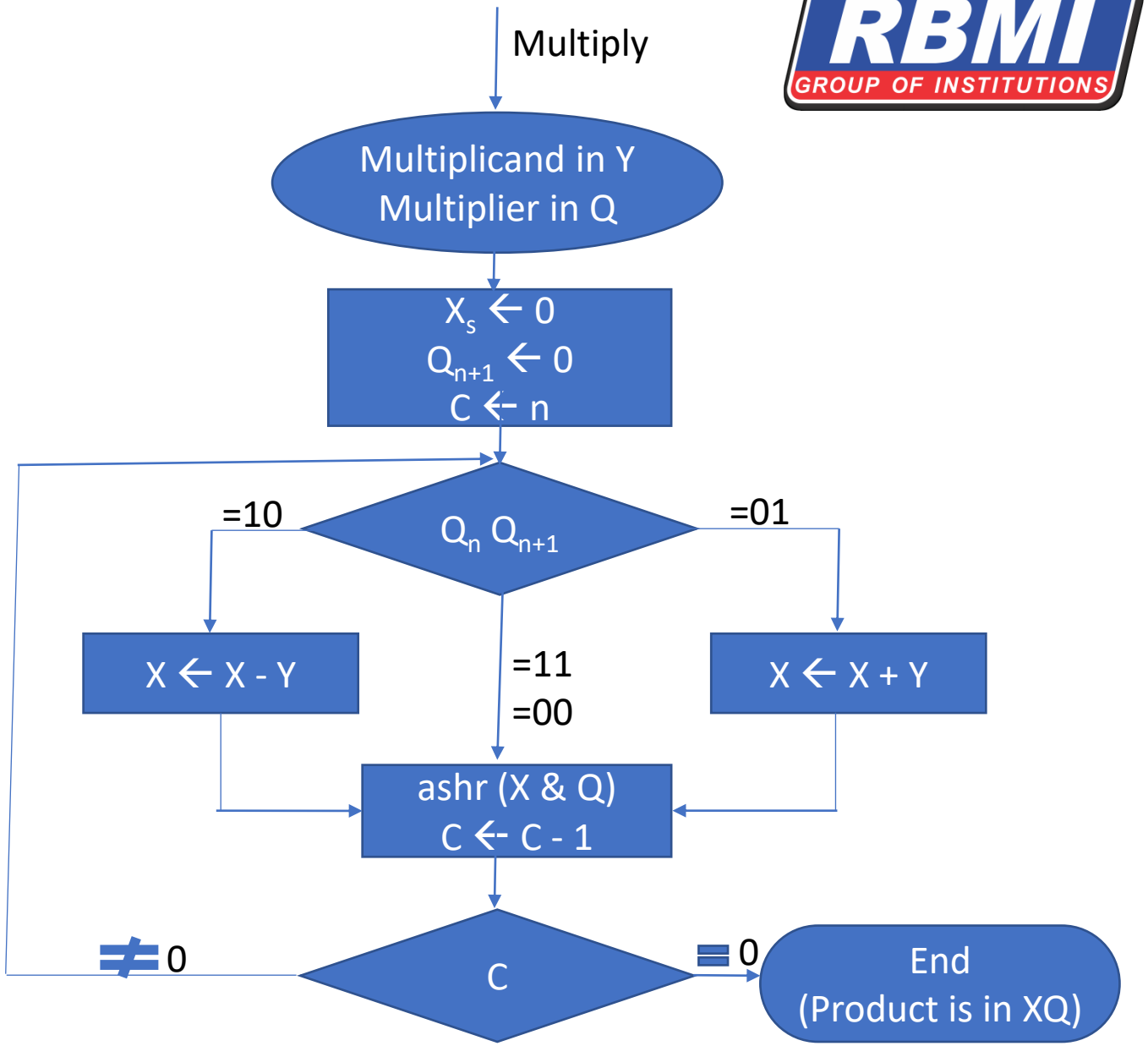
Booth Multiplication Algorithm

Booth's algorithm gives a procedure to perform multiplication of binary numbers, where numbers are in signed 2's complement representation. This algorithm requires examination of multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product or left unchanged according to following rules:

(a) The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.

(b) The multiplicand is added to partial product upon the encountering the first 0 (provided that there was a previous 1) in a string of 0's in the multiplier.

(c) The partial product is left unchanged when the multiplier bit is identical to the previous multiplier bit.



The flowchart for booth algorithm is basically contains registers X, Y and Q. Register Y contains the multiplicand and multiplier is in register Q. C is a sequence counter whose initial value is set to a number n, equal to the number of bits in the multiplier.

In the flowchart X and Q_{n+1} are cleared to 0 and the sequence counter C is set to number n equal to the number of bits of the multiplier. The two bits of the multiplier are inspected. If two bits Q_n and Q_{n+1} are 10, means that the first 1 in a string of 1's has been encountered and a subtraction of multiplicand from the partial product in X is required. But if two bits Q_n and Q_{n+1} are 01, means that the first 0 in a string of 0's has been encountered and addition of the multiplicand to the partial product in X is required. When the two bits are 00 or 11 i.e. equal, the partial product does not change. The next step is to shift right the partial product and the multiplier. This is an arithmetic shift right (ashr) operation which shifts X and Q to the right and leaves the sign bit in X unchanged. The sequence counter C is decremented and the loop continues until C becomes 0.

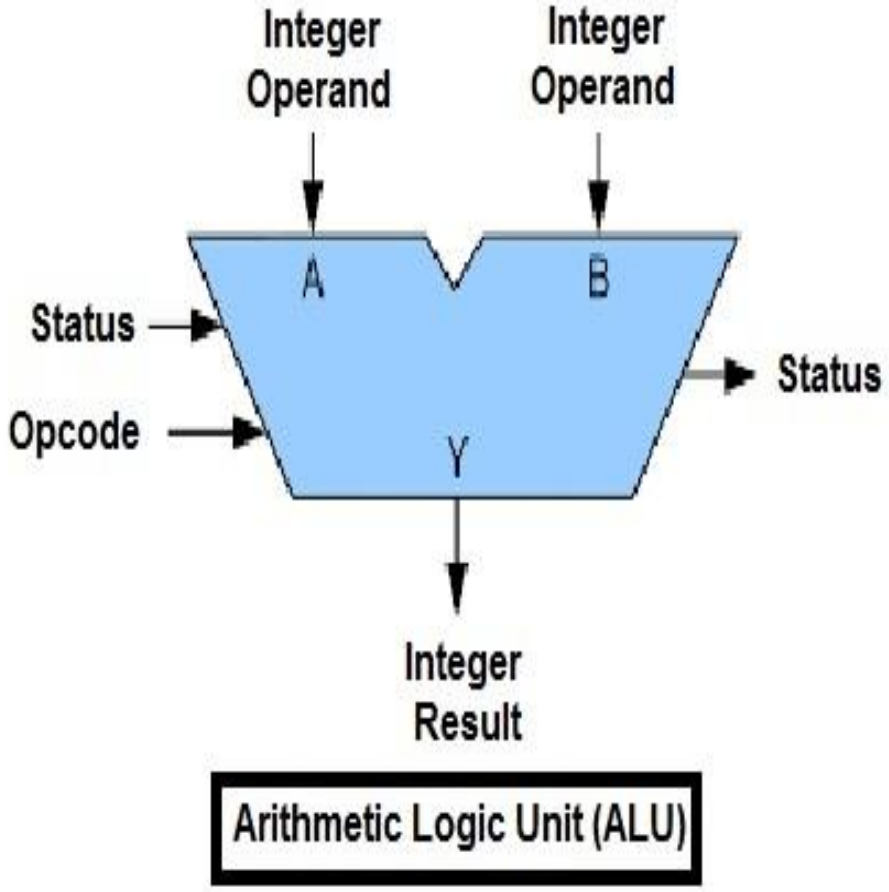
Format for Multiplication

Qn	Qn+1	Y and Y=Y -1	X	Q	Qn+1	C

Arithmetic Logic Unit (ALU)

Inside a computer, there is an Arithmetic Logic Unit (ALU), which is capable of performing logical operations (e.g. AND, OR, Ex-OR, Invert etc.) in addition to the arithmetic operations (e.g. Addition, Subtraction etc.). The control unit supplies the data required by the ALU from memory, or from input devices, and directs the ALU to perform a specific operation based on the instruction fetched from the memory. ALU is the “calculator” portion of the computer.

An arithmetic logic unit(ALU) is a major component of the central processing unit of the a computer system. It does all processes related to arithmetic and logic operations that need to be done on instruction words. In some microprocessor architectures, the ALU is divided into the arithmetic unit (AU) and the logic unit (LU).



An ALU can be designed by engineers to calculate many different operations. When the operations become more and more complex, then the ALU will also become more and more expensive and also takes up more space in the CPU and dissipates more heat. That is why engineers make the ALU powerful enough to ensure that the CPU is also powerful and fast, but not so complex as to become prohibitive in terms of cost and other disadvantages.

ALU is also known as an Integer Unit (IU). The arithmetic logic unit is that part of the CPU that handles all the calculations the CPU may need. Most of these operations are logical in nature. Depending on how the ALU is designed, it can make the CPU more powerful, but it also consumes more energy and creates more heat. Therefore, there must be a balance between how powerful and complex the ALU is and how expensive the whole unit becomes. This is why faster CPUs are more expensive, consume more power and dissipate more heat.

Different operation as carried out by ALU can be categorized as follows –

- **logical operations** − These include operations like AND, OR, NOT, XOR, NOR, NAND, etc.
- **Bit-Shifting Operations** − This pertains to shifting the positions of the bits by a certain number of places either towards the right or left, which is considered a multiplication or division operations.
- **Arithmetic operations** − This refers to bit addition and subtraction. Although multiplication and division are sometimes used, these operations are more expensive to make. Multiplication and subtraction can also be done by repetitive additions and subtractions respectively.

Floating Point Representation

- Numbers too large for standard integer representations or that have fractional components are usually represented in scientific notation, a form used commonly by scientists and engineers.
- Examples:

4.25×10^{1} 10^{-3}

-3.35×3 -1.42×10^{2}

Normalized Floating Point Numbers

- We are most interested in normalized floating-point numbers, a format which includes:
  - sign
  - significand $(1.0 \leq \text{Significand} < \text{Radix})$
  - integer power of the radix

Examples of Normalized Floating Point Numbers

These are normalized:

- $+1.23456789 \times 10^{1}$
- $-9.987654321 \times 10^{12}$
- $+5.0 \times 10^{0}$

These are ***not*** normalized:

- $+11.3 \times 10^{3}$ *significand > radix*
- -0.0002×10^{7} *significand < 1.0*
- -4.0×10^{½} *exponent not integer*

Normalizing Floating Point Data

Floating point data is normalized so that there is the significand is always one:

$100001.101_2 = 1.00001101 \times 2^5$

$1100.0101_2 = 1.1000101 \times 2^3$

Since the most significant bit is always 1, we can assume that it is implied and that we do not actually have to represent it.

Biased Exponents

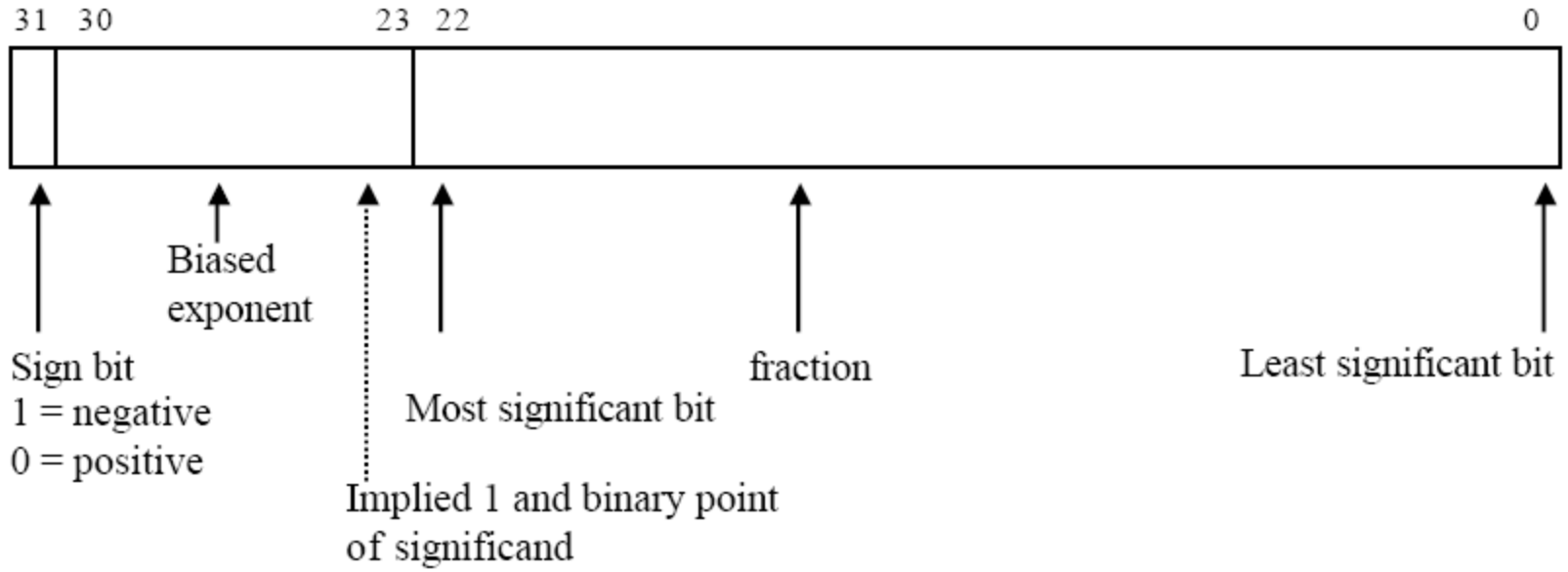
- Short floating point numbers uses 8-bits for the exponents, which we want to range from -128 to +127.
- A biased exponent uses some value other than 0 as the baseline, which must be subtracted to get the actual exponent value.
- Example (in short floating point):
  - exponent 135 = 135 - 127 = 2^{8}
  - exponent 120 = 120-127 = 2^{-7}

Representing Floating Point Values In Memory

There are three standard formats for representing floating-point numbers:

- 32-bit format (*single-precision*)
- 64-bit format (*double-precision*)
- 80-bit format (*extended precision*)

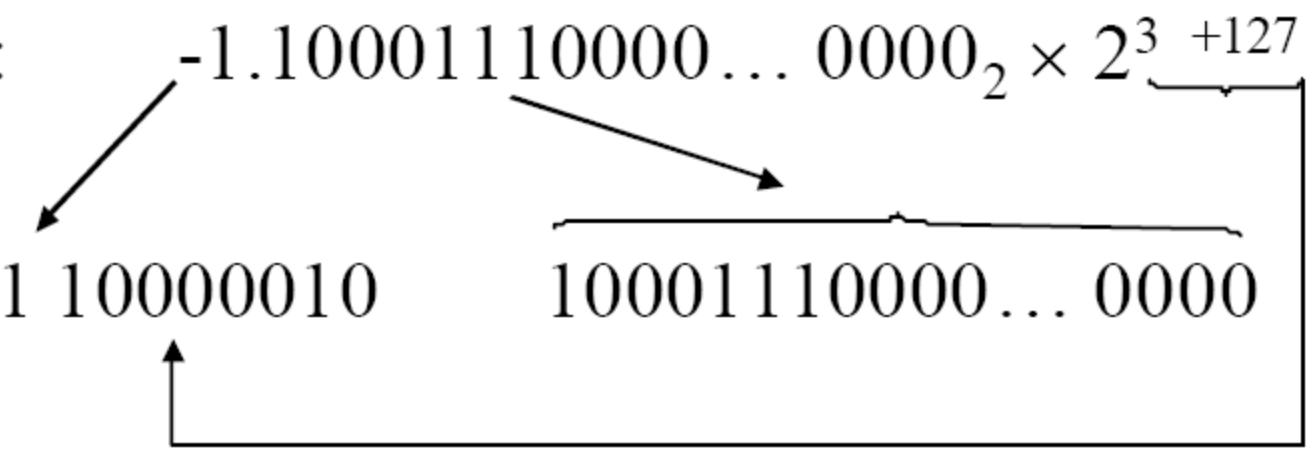
Short Floating Point Numbers



Representing Values

$-12.4375_{10} = -1100.0111_2$

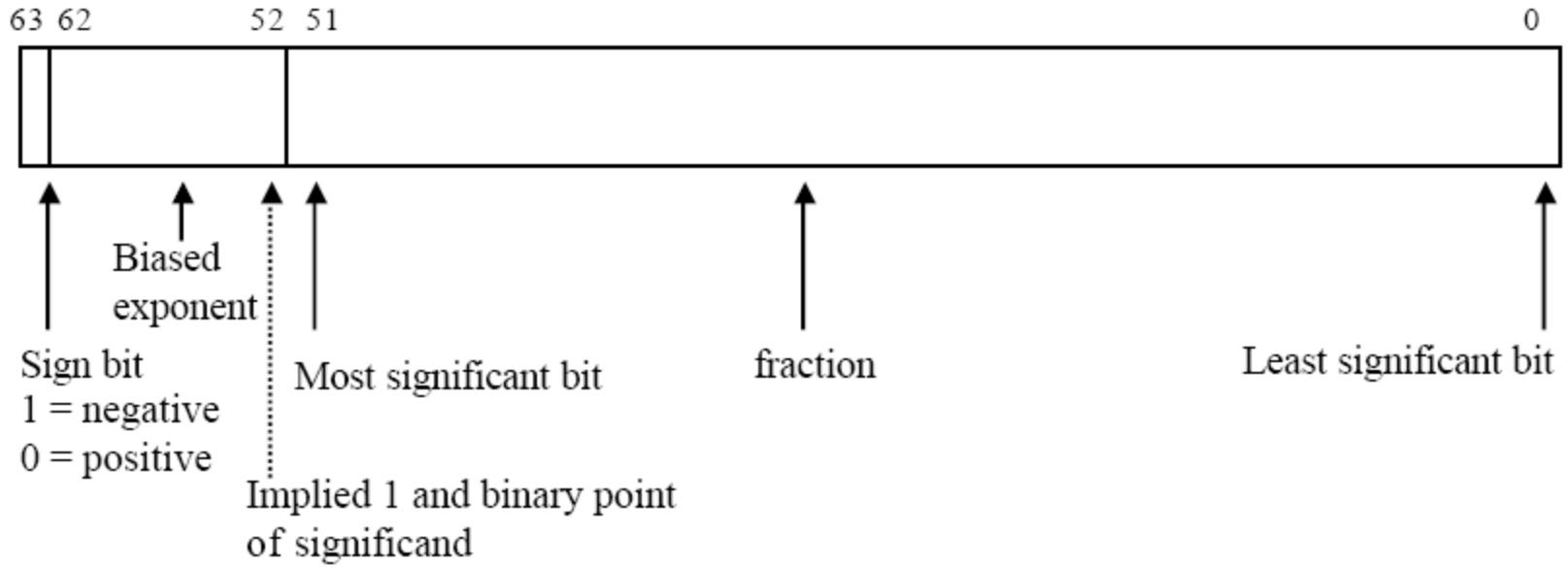
Short: $-1.10001110000\ldots\ 0000_2 \times 2^3$ +127



$1100\ 0001\ 0100\ 0111\ 0000\ \ldots\ 0000_2$

$= C1470000h$

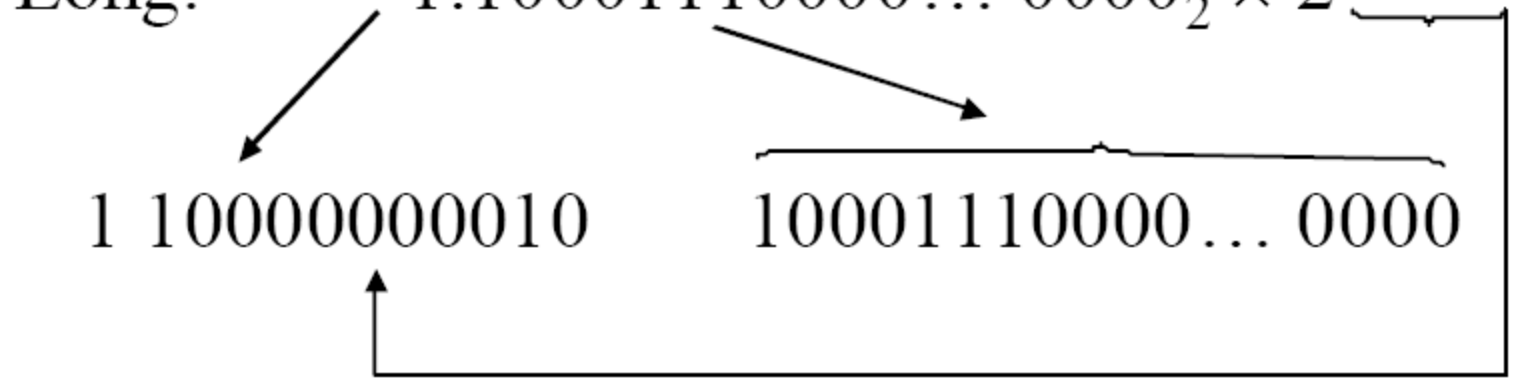
Long Floating Point Numbers



Representing Values

$-12.4375_{10} = -1100.0111_2$

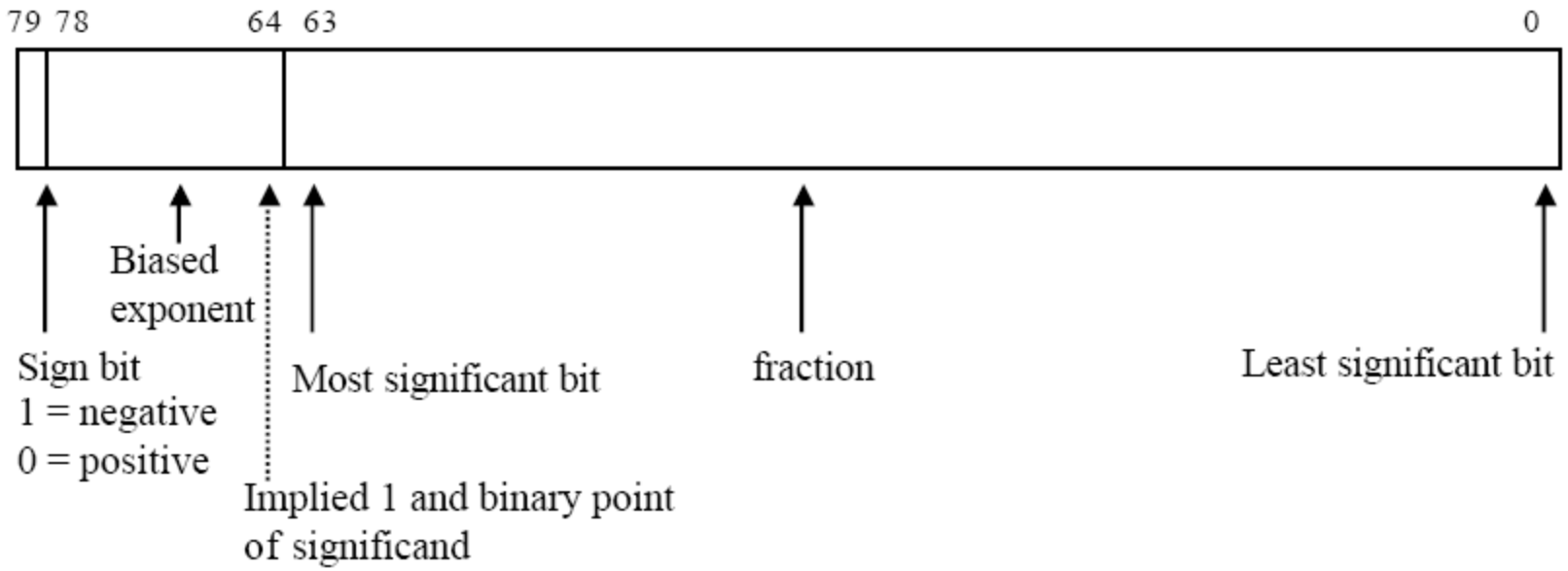
Long: $-1.100011110000\ldots\ 0000_2 \times 2^{3\ +1023}$



$1100\ 0000\ 0010\ 1000\ 1110\ 0000\ \ldots\ 0000_2$

$= C028E00000000000h$

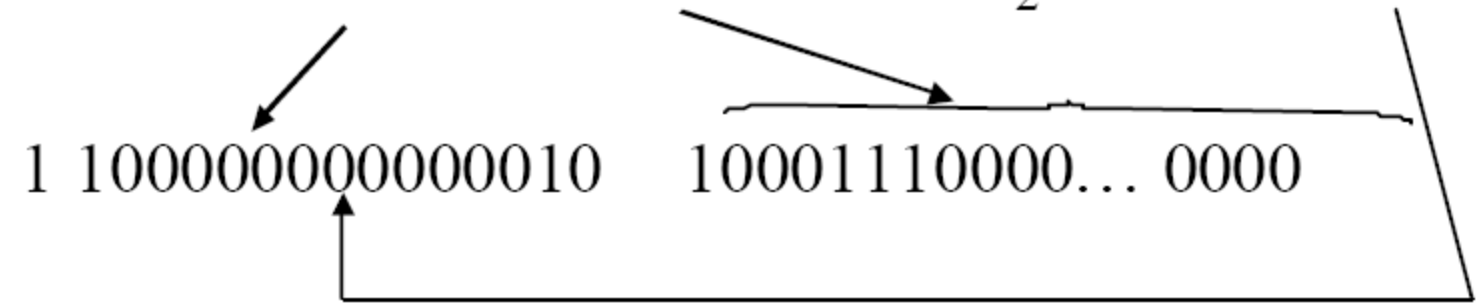
Extended Floating Point Numbers



Representing Values

$-12.4375_{10} = -1100.0111_2$

Extended: $-1.10001110000\ldots\ 0000_2 \times 2^3$ $+16383$



$1100\ 0000\ 0000\ 0010\ 1100\ 0111\ \ 0000\ \ldots\ 0000_2$

$= C002C700000000000000h$

Floating Point Addition

- To add two floating point values, they have to be aligned so that they have the same exponent.
- After addition, the sum may need to be normalized.
- Potential errors include overflow, underflow and inexact results.
- Examples:

$$\begin{array}{rr} 2.34 \times 10^3 & 6.22 \times 10^8 \\ + \quad 0.88 \times 10^3 & + 3.93 \times 10^8 \\ \hline 3.22 \times 10^3 & 10.15 \times 10^8 = 1.015 \times 10^8 \end{array}$$

Floating Point Subtraction

- Subtracting floating point values also requires re-alignment so that they have the same exponent.
- After subtraction, the difference may need to be normalized.
- Potential errors include overflow, underflow and inexact results, and the difference may have one signficant bit less than the operands..
- Examples:

$$\begin{array}{r} 2.34 \times 10^3 \\ \underline{-0.88 \times 10^3} \\ 1.46 \times 10^3 \end{array} \qquad \begin{array}{r} 6.44 \times 10^4 \\ \underline{-\ 6.23 \times 10^4} \\ 0.21 \times 10^4 = 2.1 \times 10^3 \end{array}$$

Floating Point Multiplication

- Multiplying floating point values does not requires re-alignment - realigning may lead to loss of significance.
- After multiplication, the product may need to be normalized.
- Potential errors include overflow, underflow and inexact results.
- Examples:

$$\begin{array}{r} 2.4 \times 10^{-3} \\ \times\ 6.3 \times 10^{2} \\ \hline 15.12 \times 10^{1} = 1.512 \times 10^{2} \end{array}$$

Floating Point Division

- Dividing floating point values does not requires re-alignment.
- After division, the (floating point) quotient may need to be normalized – there is no remainder
- Potential errors include overflow, underflow, inexact results and attempts to divide by zero.
- Examples:

$$1.86 \times 10^{13} \div 7.44 \times 10^{5} = 0.25 \times 10^{8}$$

$$2.5 \times 10^{7}$$