# Computer Arithmetic module 2

By
Soumya Das
Asst prof. dept. of cse
GCE Kalahandi

# Number Representation

**Sign Magnitude**

Sign magnitude is a very simple representation of negative numbers. In sign magnitude the first bit is dedicated to represent the sign and hence it is called sign bit.

Sign bit ‘1’ represents negative sign.

Sign bit ‘0’ represents positive sign.

In sign magnitude representation of a n – bit number, the first bit will represent sign and rest n-1 bits represent magnitude of number.

For example,

+25 = 011001

Where 11001 = 25

And 0 for ‘+’

-25 = 111001

Where 11001 = 25

And 1 for ‘-‘.

**Range of number represented by sign magnitude method** = $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$ (for n bit number)

# 2's complement method

To represent a negative number in this form, first we need to take the 1's complement of the number represented in simple positive binary form and then add 1 to it.

For example:

$(-8)_{10} = (1000)_2$

1's complement of 1000 = 0111

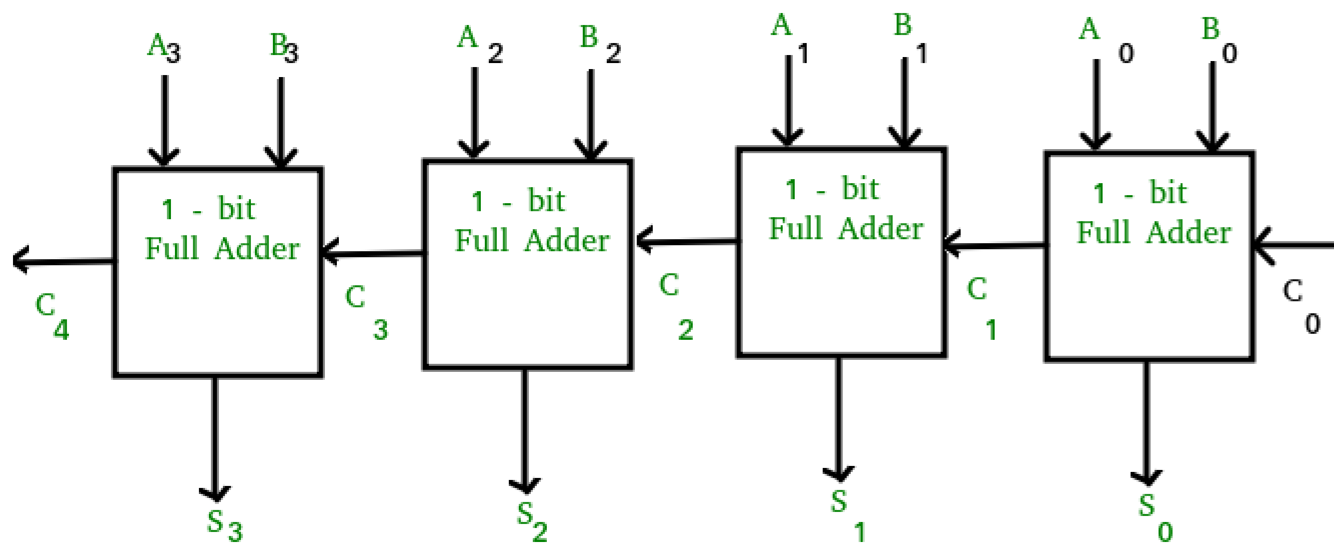
Adding 1 to it, 0111 + 1 = 1000

So, $(-8)_{10} = (1000)_2$

**Range of number represented by 2's complement =** ($-2^{n-1}$ to $2^{n-1} - 1$)

# Ripple carry adders

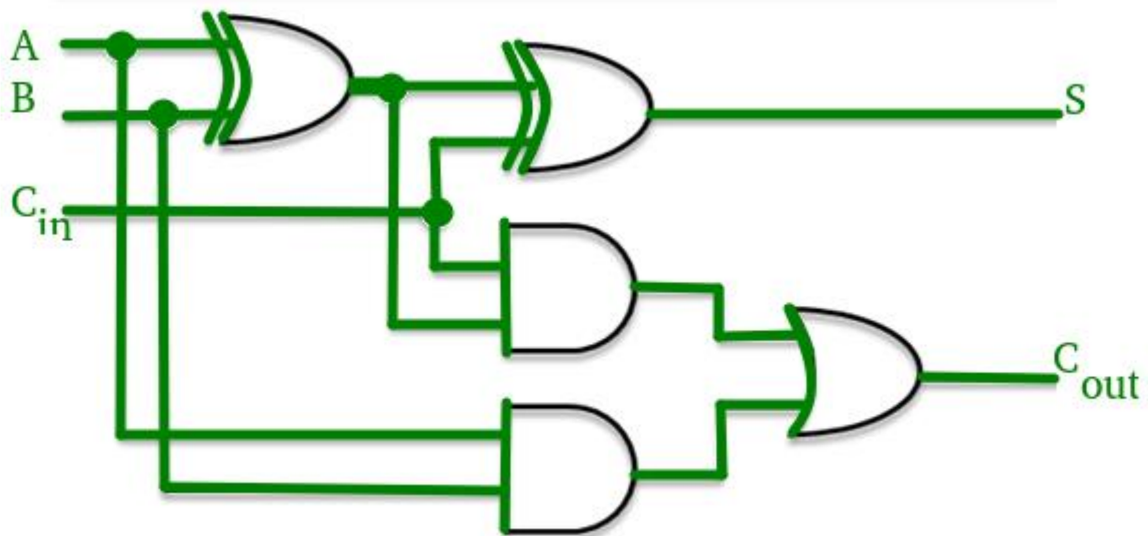
for each adder block, the two bits that are to be added are available instantly. However, each adder block waits for the carry to arrive from its previous block. So, it is not possible to generate the sum and carry of any block until the input carry is known. The block waits for the block to produce its carry. So there will be a considerable time delay which is carry propagation delay.



A3
B3
A 2
B 2
A 1
B 1
A 0
B 0
1 - bit
Full Adder
1 - bit
Full Adder
1 - bit
Full Adder
1 - bit
Full Adder
C 4
C 3
C 2
C 1
C 0
S 3
S 2
S 1
S 0

# Carry Look-ahead Adder

A carry look-ahead adder reduces the propagation delay by introducing more complex hardware. In this design, the ripple carry design is suitably transformed such that the carry logic over fixed groups of bits of the adder is reduced to two-level logic.


A
B
C
in
S
C
out

# Advantages and Disadvantages of Carry Look-Ahead Adder :

**Advantages –**

The propagation delay is reduced.

It provides the fastest addition logic.

**Disadvantages –**

The Carry Look-ahead adder circuit gets complicated as the number of variables increase.

The circuit is costlier as it involves more number of hardware.

# Booth algorithm

- Booth algorithm gives a procedure for **multiplying binary integers** in signed 2’s complement representation **in efficient way**, i.e., less number of additions/subtractions required. It operates on the fact that strings of 0’s in the multiplier require no addition but just shifting and a string of 1’s in the multiplier from bit weight 2^k to weight 2^m can be treated as 2^(k+1 ) to 2^m.
- As in all multiplication schemes, booth algorithm requires examination **of the 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:
- The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1’s in the multiplier
- The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous ‘1’) in a string of 0’s in the multiplier.
- The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

# Booth algorithm contd.

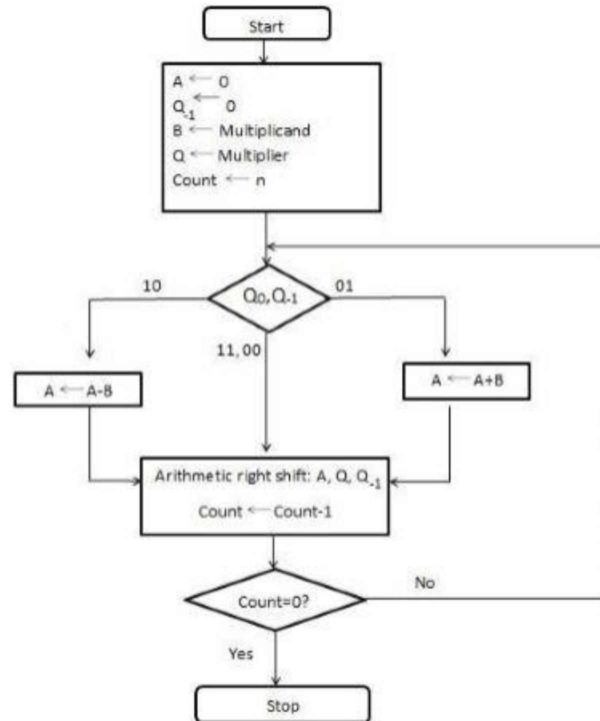



Figure 1

- **Example –** A numerical example of booth’s algorithm is shown below for n = 4. It shows the step by step multiplication of -5 and -7.
- MD = -5 = 1011, MD = 1011, MD'+1 = 0101 MR = -7 = 1001

# The explanation of first step is as follows:

| OPERATION | AC | MR | QN+1 | SC |
|---|---|---|---|---|
| | 0000 | 1001 | 0 | 4 |
| AC + MD’ + 1 | 0101 | 1001 | 0 | |
| ASHR | 0010 | 1100 | 1 | 3 |
| AC + MR | 1101 | 1100 | 1 | |
| ASHR | 1110 | 1110 | 0 | 2 |
| ASHR | 1111 | 0111 | 0 | 1 |
| AC + MD’ + 1 | 0010 | 0011 | 1 | 0 |

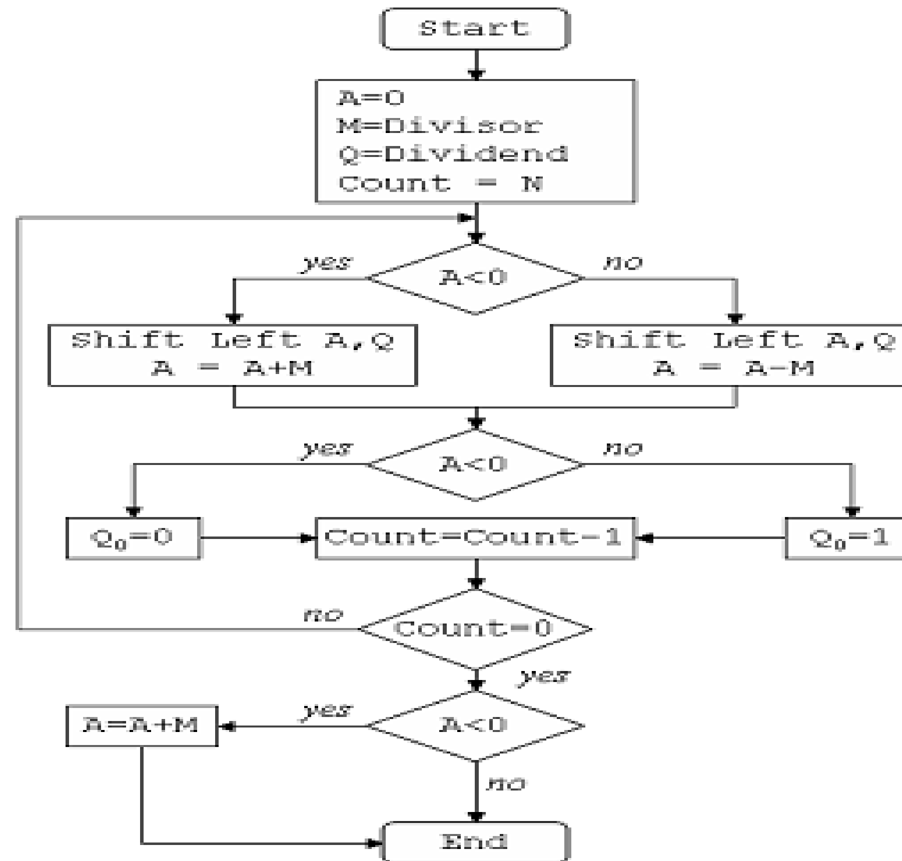
Product is calculated as follows:

Product = AC MR Product = 0010 0011=35

# Division algorithm(Restoring)

- A division algorithm provides a quotient and a remainder when we divide two number. They are generally of two type **slow algorithm and fast algorithm**. Slow division algorithm are restoring, non-restoring, non-performing restoring, SRT algorithm and under fast comes Newton–Raphson and Goldschmidt. Restoring term is due to fact that value of register A is restored after each iteration.

# Division algorithm flow chart



**Examples:**

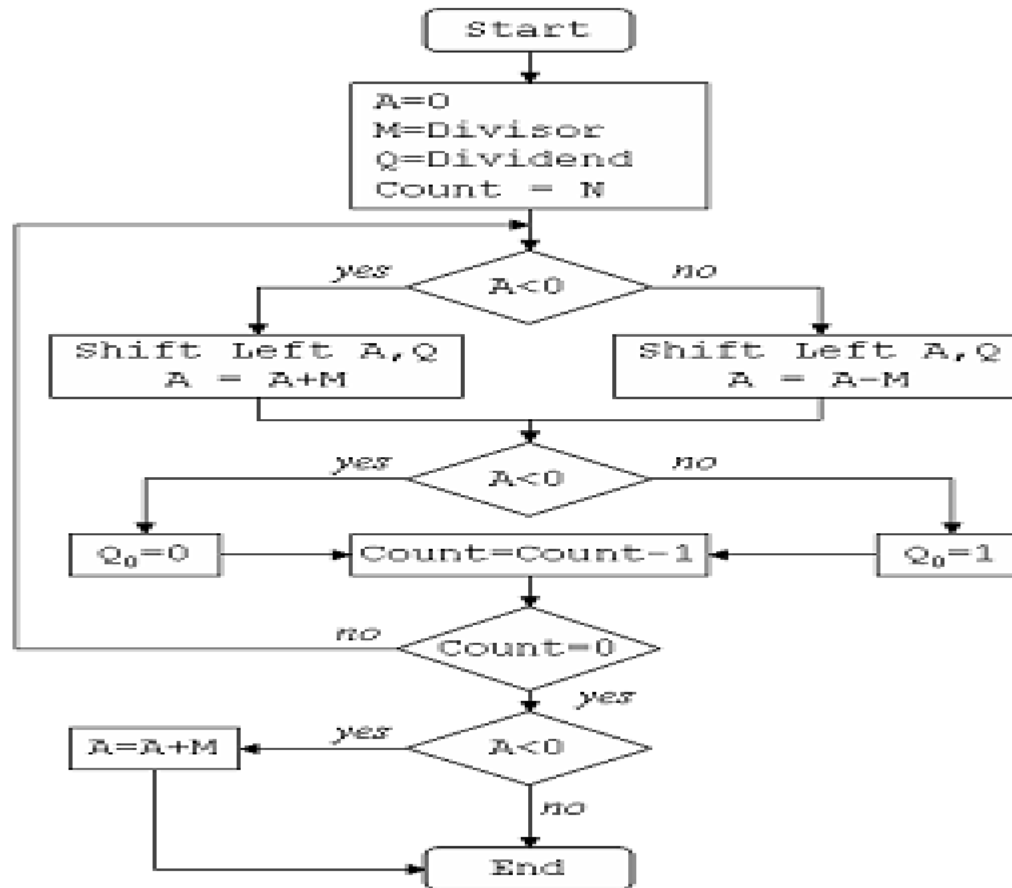
Perform Division Restoring Algorithm Dividend = 11 Divisor = 3

| N | M | A | Q | OPERATION |
|---|---|---|---|---|
| 4 | 00011 | 00000 | 1011 | initialize |
| | 00011 | 00001 | 011_ | shift left AQ |
| | 00011 | 11110 | 011_ | A=A-M |
| | 00011 | 00001 | 0110 | Q[0]=0 And restore A |
| 3 | 00011 | 00010 | 110_ | shift left AQ |
| | 00011 | 11111 | 110_ | A=A-M |
| | 00011 | 00010 | 1100 | Q[0]=0 |
| 2 | 00011 | 00101 | 100_ | shift left AQ |
| | 00011 | 00010 | 100_ | A=A-M |
| | 00011 | 00010 | 1001 | Q[0]=1 |
| 1 | 00011 | 00101 | 001_ | shift left AQ |
| | 00011 | 00010 | 001_ | A=A-M |
| | 00011 | 00010 | 0011 | Q[0]=1 |

# Non-Restoring division

- Non-Restoring division is less complex than the restoring one because simpler operation are involved i.e. addition and subtraction, also now restoring step is performed. In the method, rely on the sign bit of the register which initially contain zero named as A.

# Division algorithm(Non Restoring)



**Examples:** Perform Non_Restoring Division for Unsigned Integer Dividend =11 Divisor =3 -M =11101

| N | M | A | Q | ACTION |
|---|---|---|---|---|
| 4 | 00011 | 00000 | 1011 | Start |
| | | 00001 | 011_ | Left shift AQ |
| | | 11110 | 011_ | A=A-M |
| 3 | | 11110 | 0110 | Q[0]=0 |
| | | 11100 | 110_ | Left shift AQ |
| | | 11111 | 110_ | A=A+M |
| 2 | | 11111 | 1100 | Q[0]=0 |
| | | 11111 | 100_ | Left Shift AQ |
| | | 00010 | 100_ | A=A+M |
| 1 | | 00010 | 1001 | Q[0]=1 |
| | | 00101 | 001_ | Left Shift AQ |
| | | 00010 | 001_ | A=A-M |
| 0 | | 00010 | 0011 | Q[0]=1 |

Quotient = 3 (Q) Remainder = 2 (A)

# Floating point representation of numbers

- **32-bit representation floating point numbers IEEE standard**

  **Normalization**
- Floating point numbers are usually normalized
- Exponent is adjusted so that leading bit (MSB) of mantissa is 1
- Since it is always 1 there is no need to store it
- Scientific notation where numbers are normalized to give a single digit before the decimal point like in decimal system e.g. $3.123 \times 10^3$

# Floating point arithmatic

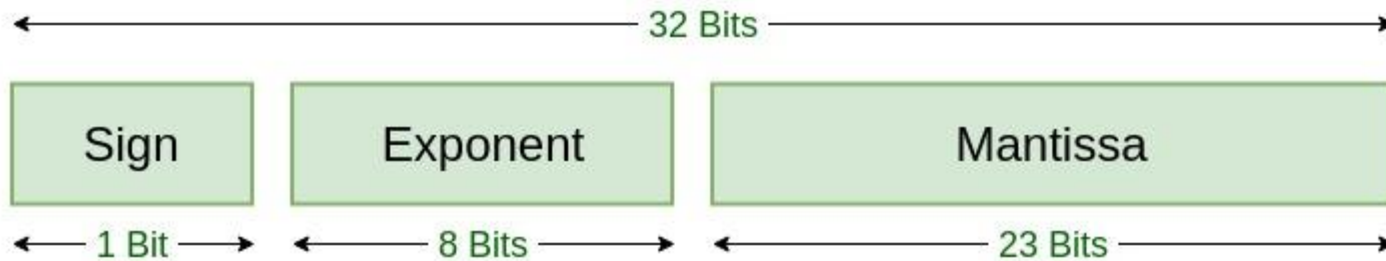
- A finite number can also represented by four integers components, a sign (s), a base (b), a significand (m), and an exponent (e). Then the numerical value of the number is evaluated as
- $(-1)^s \times m \times b^e$ ———— Where $m < |b|$
- Depending on base and the number of bits used to encode various components, the IEEE 754 standard defines five basic formats. Among the five formats, the binary32 and the binary64 formats are single precision and double precision formats respectively in which the base is 2.

# IEEE Standard 754 Floating Point Numbers

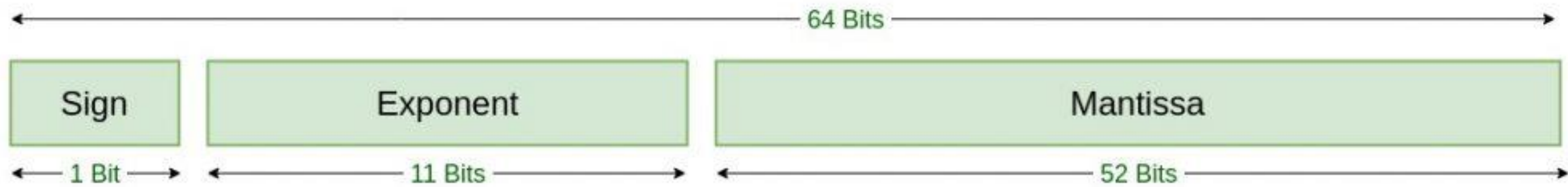
- The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation which was established in 1985 by the **Institute of Electrical and Electronics Engineers (IEEE)**. The standard addressed many problems found in the diverse floating point implementations that made them difficult to use reliably and reduced their portability. IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macs, and most Unix platforms.
- There are several ways to represent floating point number but IEEE 754 is the most efficient in most cases. IEEE 754 has 3 basic components:
- **The Sign of Mantissa –**
  This is as simple as the name. 0 represents a positive number while 1 represents a negative number.
- **The Biased exponent –**
  The exponent field needs to represent both positive and negative exponents. A bias is added to the actual exponent in order to get the stored exponent.
- **The Normalised Mantissa –**
  The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Here we have only 2 digits, i.e. O and 1. So a normalised mantissa is one with only one 1 to the left of the decimal.
- **IEEE 754 numbers are divided into two based on the above three components: single precision and double precision.**

# Table – 1 Precision Representation

| Precision | Base | Sign | Exponent | Significand |
|---|---|---|---|---|
| Single precision | 2 | 1 | 8 | 23+1 |
| Double precision | 2 | 1 | 11 | 52+1 |




Single Precision
IEEE 754 Floating-Point Standard




Double Precision
IEEE 754 Floating-Point Standard

# Example

85.125 85 = 1010101 0.125 = 001 85.125 = 1010101.001 =1.010101001 x 2^6 sign = 0

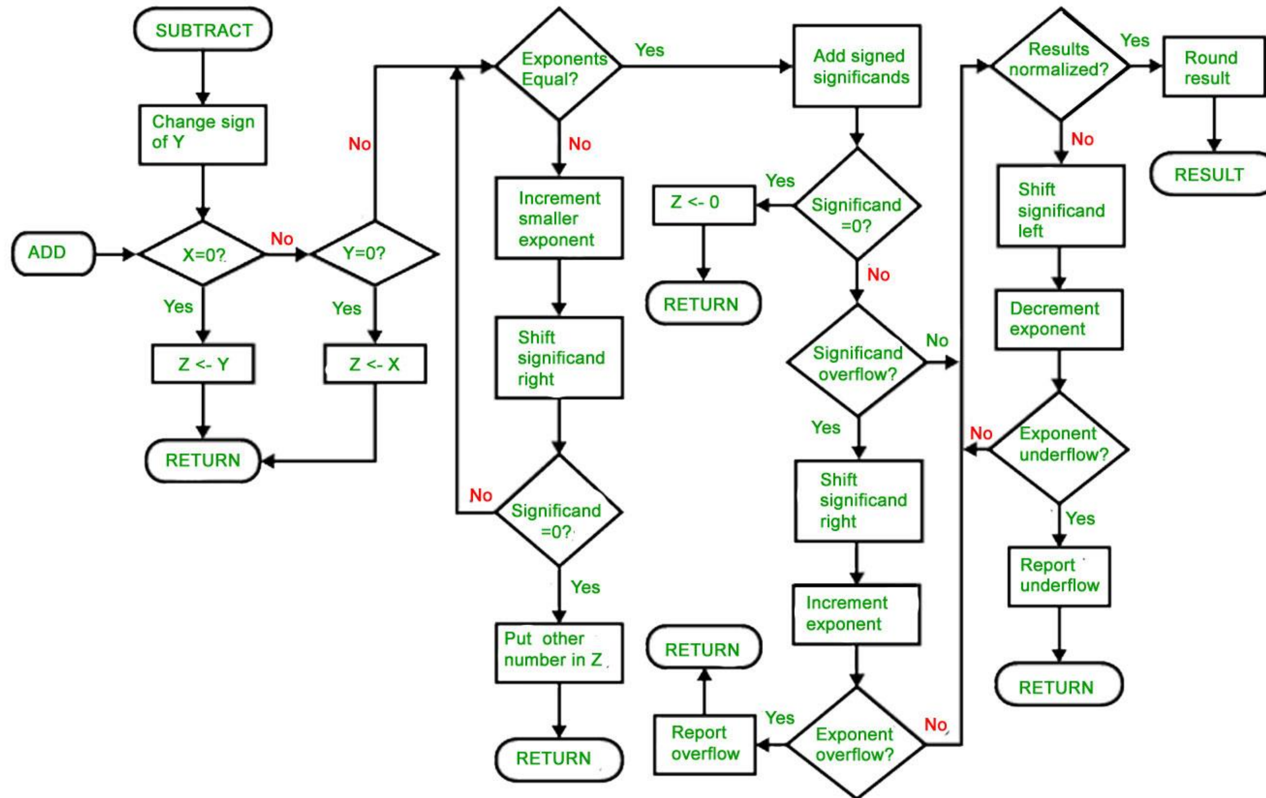
1. Single precision: biased exponent 127+6=133 133 = 10000101 Normalised mantisa = 010101001 we will add 0's to complete the 23 bits The IEEE 754 Single precision is: = 0 10000101 01010100100000000000000 This can be written in hexadecimal form **42AA4000**
2. Double precision: biased exponent 1023+6=1029 1029 = 10000000101 Normalised mantisa = 010101001 we will add 0's to complete the 52 bits The IEEE 754 Double precision is: = 0 10000000101 010101001000000000000000000000000000000000000000 0000 This can be written in hexadecimal form **4055480000000000**

- **Special Values:** IEEE has reserved some values that can ambiguity.
- **Zero –**
  Zero is a special value denoted with an exponent and mantissa of 0. -0 and +0 are distinct values, though they both are equal.
- **Denormalised –**
  If the exponent is all zeros, but the mantissa is not then the value is a denormalized number. This means this number does not have an assumed leading one before the binary point.
- **Infinity –**
  The values +infinity and -infinity are denoted with an exponent of all ones and a mantissa of all zeros. The sign bit distinguishes between negative infinity and positive infinity. Operations with infinite values are well defined in IEEE.
- **Not A Number (NAN) –**
  The value NAN is used to represent a value that is an error. This is represented when exponent field is all ones with a zero sign bit or a mantissa that it not 1 followed by zeros. This is a special value that might be used to denote a variable that doesn’t yet hold a value.

| EXPONENT | MANTISA | VALUE |
|---|---|---|
| 0 | 0 | exact 0 |
| 255 | 0 | Infinity |
| 0 | not 0 | denormalised |
| 255 | not 0 | Not a number (NAN) |

| DENORMALIZED | NORMALIZED | APPROXIMATE DECIMAL | |
|---|---|---|---|
| Single Precision | $\pm 2^{-149}$ to $(1 - 2^{-23}) \times 2^{-126}$ | $\pm 2^{-126}$ to $(2 - 2^{-23}) \times 2^{127}$ | ± approximately $10^{-44.85}$ to approximately $10^{38.53}$ |
| Double Precision | $\pm 2^{-1074}$ to $(1 - 2^{-52}) \times 2^{-1022}$ | $\pm 2^{-1022}$ to $(2 - 2^{-52}) \times 2^{1023}$ | ± approximately $10^{-323.3}$ to approximately $10^{308.}$ |

# Floating point addition and subtraction




Floating-Point Addition and Subtraction ($Z \leftarrow X \pm Y$)

# floating point addition

**For example,** we have to add $\mathbf{1.1 * 10^3}$ and **50.**

We cannot add these numbers directly. First, we need to align the exponent and then, we can add significand.

After aligning exponent, we get $\mathbf{50 = 0.05 * 10^3}$

Now adding significand, **0.05 + 1.1 = 1.15**

So, finally we get $\mathbf{(1.1 * 10^3 + 50) = 1.15 * 10^3}$

Here, notice that we shifted **50** and made it **0.05** to add these numbers.

**Now let us take example of floating point number addition**

We follow these steps to add two numbers:

1. Align the significand
2. Add the significands
3. Normalize the result

**Let the two numbers be**

x = 9.75 y = 0.5625

Converting them into 32-bit floating point representation,

**9.75**’s representation in 32-bit format = **0 10000010 00111000000000000000000**

**0.5625**’s representation in 32-bit format = **0 01111110 00100000000000000000000**

Now we get the difference of exponents to know how much shifting is required.

(**10000010 – 01111110**)2 = (**4**)10

Now, we shift the mantissa of lesser number right side by 4 units.

Mantissa of **0.5625 = 1.00100000000000000000000**

(note that 1 before decimal point is understood in 32-bit representation)

Shifting right by **4** units, we get **0.00010010000000000000000**

Mantissa of **9.75** = **1. 00111000000000000000000**

Adding mantissa of both

**0. 00010010000000000000000**

**+ 1. 00111000000000000000000**

**1. 01001010000000000000000**

In final answer, we take exponent of bigger number

So, final answer consist of :

Sign bit = **0**

Exponent of bigger number = **10000010**

Mantissa = **01001010000000000000000**

32 bit representation of answer = **x + y = 0 10000010 01001010000000000000000**

# FLOATING POINT SUBTRACTION

Subtraction is similar to addition with some differences like we subtract mantissa unlike addition and in sign bit we put the sign of greater number.

**Let the two numbers be**

x = 9.75
y = – 0.5625

Converting them into 32-bit floating point representation

**9.75**’s representation in 32-bit format = **0 10000010 00111000000000000000000**

**– 0.5625**’s representation in 32-bit format = **1 01111110 00100000000000000000000**

Now, we find the difference of exponents to know how much shifting is required.

(**10000010 – 01111110**)2 = (**4**)10

Now, we shift the mantissa of lesser number right side by 4 units.

Mantissa of **– 0.5625 = 1.00100000000000000000000**

(note that 1 before decimal point is understood in 32-bit representation)

Shifting right by **4** units, **0.00010010000000000000000**

Mantissa of **9.75**= **1. 00111000000000000000000**

Subtracting mantissa of both

**0. 00010010000000000000000**

**– 1. 00111000000000000000000**

**1. 00100110000000000000000**

Sign bit of bigger number = **0**

So, finally the answer = **x – y = 0 10000010 00100110000000000000000**