**Addition and Subtraction:-**

Arithmetic instructions in digital computers work on data to produce the results necessary for solving computational problems. These instructions are responsible for processing data on a computer. There are four basic arithmetic operations, addition, subtraction, multiplication, and division.

We consider addition and subtraction for the following types of data:

* Fixed-point binary data in signed-magnitude representation
* Fixed-point binary data in signed-2's complement representation

Computers employ a signed-magnitude approach to implement floating-point operations. Most computers utilize signed-2's complement approach for arithmetic operations on integers. The leftmost bit in the number is utilized to represent the sign in this technique; 0 denotes a positive integer, while 1 indicates a negative integer. The magnitude of the number was supported by the remaining bits in the number.

**Addition Algorithm**

The addition algorithm specifies that:

* If the signs of P and Q are the same, add both the magnitudes and put the sign of P to the result, as shown in the table below.
* Compare both the magnitudes and subtract the small number from the greater number when the signs of P and Q disagree.
* In cases where P > Q, the output signs must be equal to P, or the complement of P's sign in cases where P < Q.
* Subtract Q from P and change the sign of the output to positive when the two magnitudes are equal.

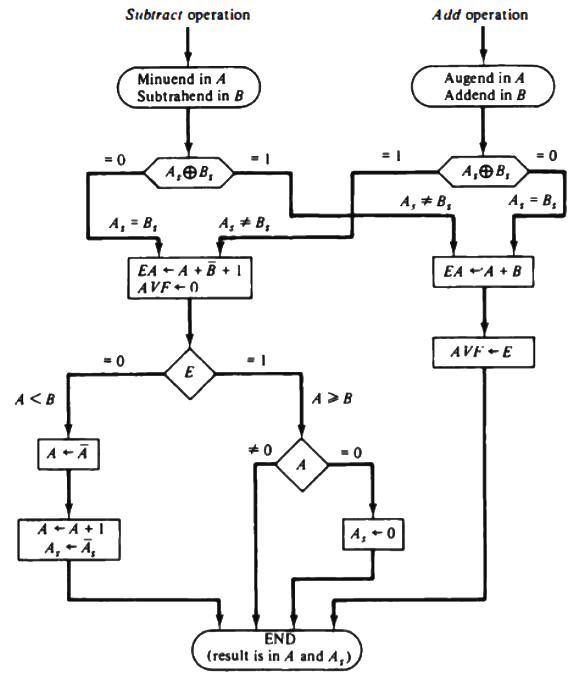
**Subtraction Algorithm**

The subtraction algorithm states that:

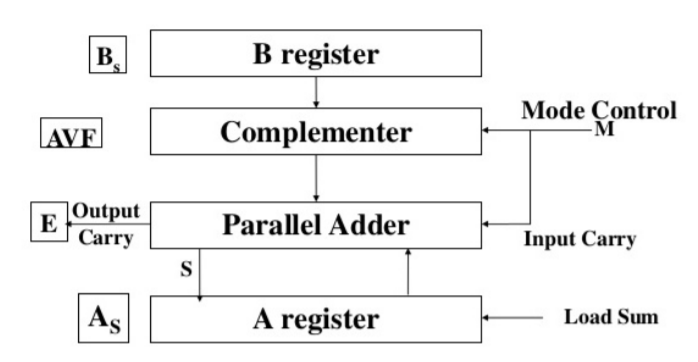
* When the signs of P and Q differ, the subtraction method says to add both the magnitudes and put the sign of P to the result.
* Compare both the magnitudes and subtract the smaller number from the greater number when the signs of P and Q are the same.
* In cases where P > Q, the output signs must be equal to P, or the complement of P's sign in cases where P < Q.
* Subtract Q from P and change the sign of the output to positive when the two magnitudes are equal.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operations** | **Addition of Magnitudes** | **Subtraction of Magnitudes** | | |
|  |  | **P>Q** | **P<Q** | **P=Q** |
| **(+P) + (+Q)** | +(P+Q) |  |  |  |
| **(+P) + (-Q)** |  | +(P-Q) | -(Q-P) | +(P-Q) |
| **(-P) + (+Q)** |  | -(P-Q) | +(Q-P) | +(P-Q) |
| **(-P) + (-Q)** | -(P+Q) |  |  |  |
| **(+P) - (+Q)** |  | +(P-Q) | -(Q-P) | +(P-Q) |
| **(+P) - (-Q)** | +(P+Q) |  |  |  |
| **(-P) - (+Q)** | -(P+Q) |  |  |  |
| **(-P) - (-Q)** |  | -(P-Q) | +(Q-P) | +(P-Q) |

### Flowchart



**Hardware Implementation**



* A and B be two registers that hold the magnitudes of No
* As and Bs be two flip-flops that hold the corresponding signs
* The Result is transferred into A and As.
* Parallel adder is needed to perform the micro operation A + B.
* Parallel sub tractor are needed to perform A – B or B – A.

– Can be accomplished by means of compliment and add

* Comparator circuit is needed to establish if A > B, A = B or A < B

– Comparison can be determine from the end carry after the subtraction

* The sign relationship can be determine from an exclusive-OR gate with As and Bs as inputs
* Output carry are transferred to E flip-flop – Where it can be checked to determine the relative magnitude of the Nos.
* Add overflow flip-flop (AVF) holds the overflow bit when A and B are added.

**Multiplication and Division:-**

We will discuss multiplication and division arithmetic algorithms and show the procedure for digital hardware implementation.

Multiplication of fixed-point binary numbers in signed-magnitude representation is done by successive shift and adds operations. For example, multiplication of numbers 10111(23) and 10011(19).

**23        10111**

**19      x 10011**

**10111**

**10111x**

**00000xx      + (adding all)**

**00000xxx**

**110110101     Product 437**

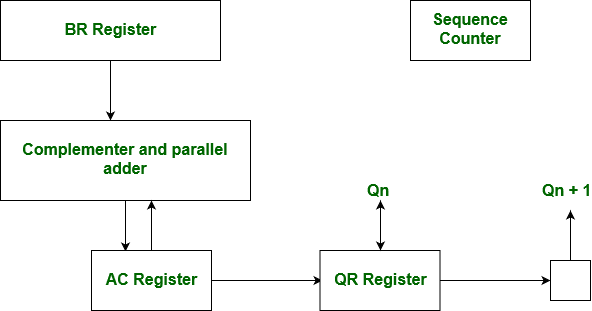
The process consists of looking at successive Multiplier, least significant bit first. If the Multiplier is 1, the multiplicand is copied down; otherwise, zero is copied. And like we do in standard multiplication, the numbers copied down in successive lines are shifted one position to the left. Finally, all binaries are added, and the total sum is the result. The sign of the product (result) is determined from the signs of multiplicand and Multiplier. If they are alike, the final product sign is positive. If they are unlike, the sign of the product is negative.

# Booth’s 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:

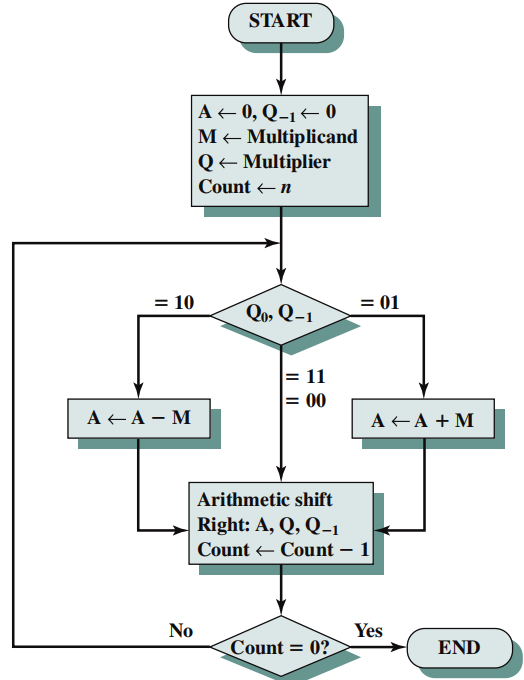
* The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1’s in the multiplier
* The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous ‘1’) in a string of 0’s in the multiplier.
* The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

**Hardware Implementation of Booths Algorithm –** The hardware implementation of the booth algorithm requires the register configuration shown in the figure below.



**Booth’s Algorithm Flowchart –**

We name the register as A, and Q, A, M and Q respectively. Q0 designates the least significant bit of multiplier in the register Q. An extra flip-flop Q-1is appended to Q to facilitate a double inspection of the multiplier. The flowchart for the booth algorithm is shown below.



A and the appended bit Q **-1** are initially cleared to 0 and the sequence n is set to a number n equal to the number of bits in the multiplier. The two bits of the multiplier in Q0 and Q-1are inspected. If the two bits are equal to 10, it means that the first 1 in a string has been encountered. This requires subtraction of the multiplicand from the partial product in A. If the 2 bits are equal to 01, it means that the first 0 in a string of 0’s has been encountered. This requires the addition of the multiplicand to the partial product in A. When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. As a consequence, the 2 numbers that are added always have a opposite signs, a condition that excludes an overflow. The next step is to shift right the partial product and the multiplier (including Q-1). This is an arithmetic shift right (ashr) operation which A and Q ti the right and leaves the sign bit in A unchanged. The sequence counter is decremented and the computational loop is repeated n times. Product of negative numbers is important, while multiplying negative numbers we need to find 2’s complement of the number to change its sign, because it’s easier to add instead of performing binary subtraction. product of two negative number is demonstrated below along with 2’s complement.

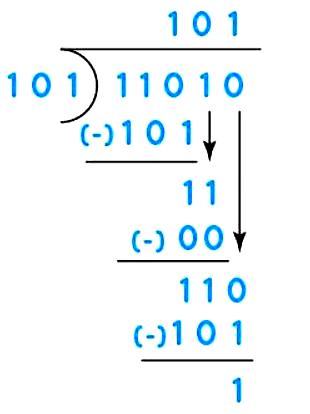
**Division Algorithm:-**

Division to two fixed-point binary numbers in signed-magnitude representation is done by the process of successive compare, shift, and subtract operations. The binary division is simpler than decimal because the quotient is either 0 or 1. There is no need to calculate how many times the dividend or partial remainder fires into the divisor. You can follow the following steps for binary division.

Step 1: Compare the divisor with the dividend; if the divisor is greater, place 0 as the quotient, then bring down the second bit of the dividend. If the divisor is smaller, multiply it by one and the result must be subtracted. Then, subtract the result from the above to get the remainder.

Step 2: Bring down the next number bit from the dividend and perform step1.

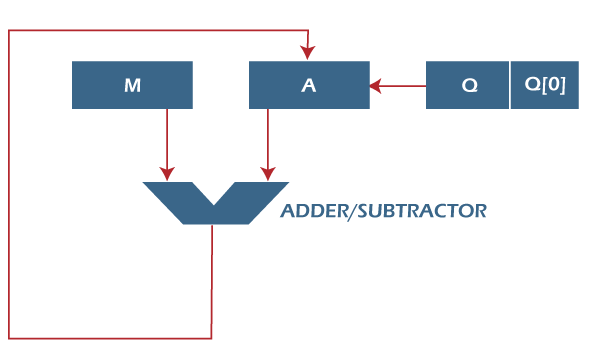
Step 3: Repeat the whole process until the remainder becomes 0, or the whole dividend is divided.



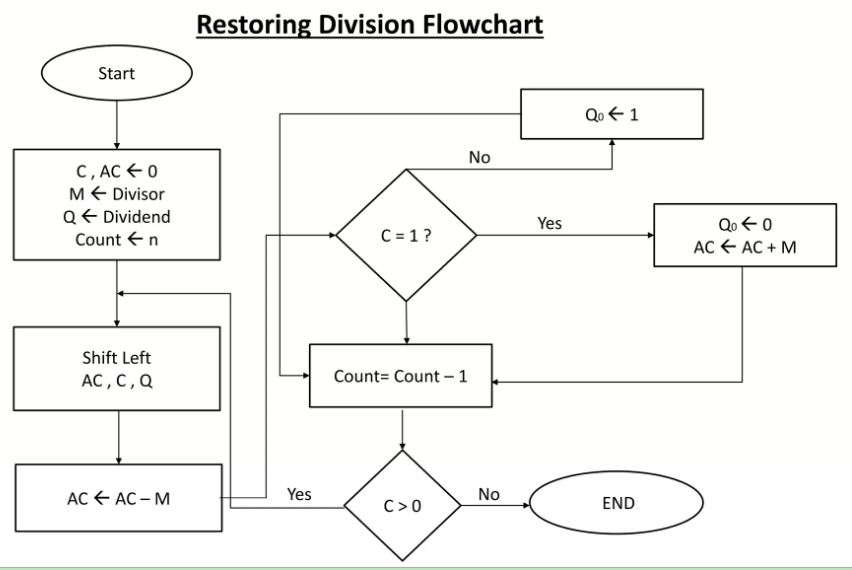
**Division Algorithm**

Division is usually performed on the fixed point fractional numbers. When we perform division operations on two numbers, the division algorithm will give us two things, i.e., quotient and remainder. This algorithm is based on the assumption that 0 < D < N. With the help of digit set {0, 1}, the quotient digit q will be formed in the restoring division algorithm. The division algorithm is generally of two types, i.e., fast algorithm and slow algorithm. Goldschmidt and Newton-Rap son are the types of fast division algorithm, and STR algorithm, restoring algorithm, non-performing algorithm, and the non-restoring algorithm are the types of slow division algorithm.

In this section, we are going to perform restoring algorithm with the help of an unsigned integer. We are using restoring term because we know that the value of register A will be restored after each iteration. We will also try to solve this problem using the flow chart and apply bit operations.



Here, register Q is used to contain the quotient, and register A is used to contain the remainder. Here, the divisor will be loaded into the register M, and n-bit divided will be loaded into the register Q. 0 is the starting value of a register. The values of these types of registers are restored at the time of iteration. That's why it is known as restoring.



Now we will learn some steps of restoring division algorithm, which is described as follows

**Step 1:** In this step, the corresponding value will be initialized to the registers, i.e., register A will contain value 0, register M will contain Divisor, register Q will contain Dividend, and N is used to specify the number of bits in dividend.

**Step 2:** In this step, register A and register Q will be treated as a single unit, and the value of both the registers will be shifted left.

**Step 3:** After that, the value of register M will be subtracted from register A. The result of subtraction will be stored in register A.

**Step 4:** Now, check the most significant bit of register A. If this bit of register A is 0, then the least significant bit of register Q will be set with a value 1. If the most significant bit of A is 1, then the least significant bit of register Q will be set to with value 0, and restore the value of A that means it will restore the value of register A before subtraction with M.

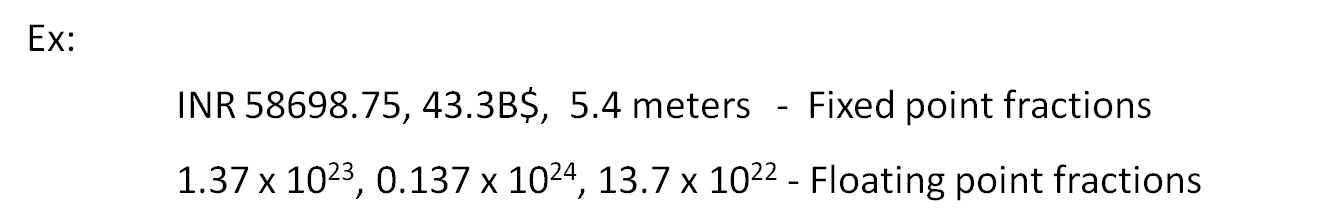
**Step 5:** After that, the value of N will be decremented. Here n is used as a counter.

**Step 6:** Now, if the value of N is 0, we will break the loop. Otherwise, we have to again go to step 2.

**Step 7:** This is the last step. In this step, the quotient is contained in the register Q, and the remainder is contained in register A.

**Floating Point Arithmetic**

A floating-point (FP) number is a kind of fraction where the radix point is allowed to move. If the radix point is fixed, then those fractional numbers are called fixed-point numbers. The best examples of fixed-point numbers are those represented in commerce, finance while that of floating-point is the scientific constants and values.



The disadvantage of fixed-point is that not all numbers are easily representable. For example, continuous fractions are difficult to be represented in fixed-point form. Also, very small and very large fractions are almost impossible to be fitted with efficiency.

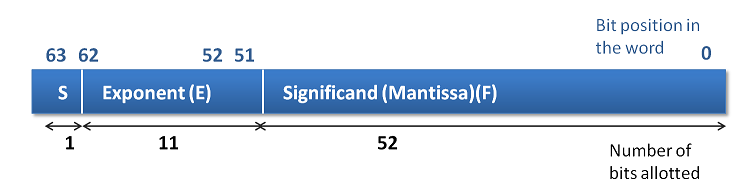
A floating-point number representation is standardized by IEEE with three components namely the sign bit, Mantissa and the exponent. The number is derived as:

**F = (-1) s M x 2E**

IEEE-754 standard prescribes single precision and double precision representation as in figure 10.1.



Single Precision



Double Precision

**Floating point normalization**

The mantissa part is adjusted in such a way that the value always starts with a leading binary '1' i.e. it starts with a non zero number. This form is called a normalized form. In doing so, the '1' is assumed to be the default and not stored and hence the mantissa 23 or 52 bits get extra space for representing the value. However, during calculations, the '1' is brought in by the hardware.

**Floating point operations and hardware**

Programming languages allow data type declaration called real to represent FP numbers. Thus it is a conscious choice by the programmer to use FP. When declared real the computations associated with such variables utilize FP hardware with FP instructions.

Add Float, Sub Float, Multiply Float and Divide Float is the likely FP instructions that are associated and used by the compiler. FP arithmetic operations are not only more complicated than the fixed-point operations but also require special hardware and take more execution time. For this reason, the programmer is advised to use real declaration judiciously.

Table: - suggests how the FP arithmetic is done.

|  |  |  |
| --- | --- | --- |
| Addition | X+Y | (adjusted Xm + Ym) 2Ye where Xe ≤ Ye |
| Subtraction | X-Y | (adjusted Xm - Ym) 2Ye where Xe ≤ Ye |
| Multiplication | X \* Y | (adjusted Xm x Ym) 2Xe+Ye |
| Division | X/Y | (adjusted Xm / Ym) 2Xe-Ye |

**Addition and Subtraction**

FP addition and subtraction are similar and use the same hardware and hence we discuss them together. Let us say, the X and Y are to be added.

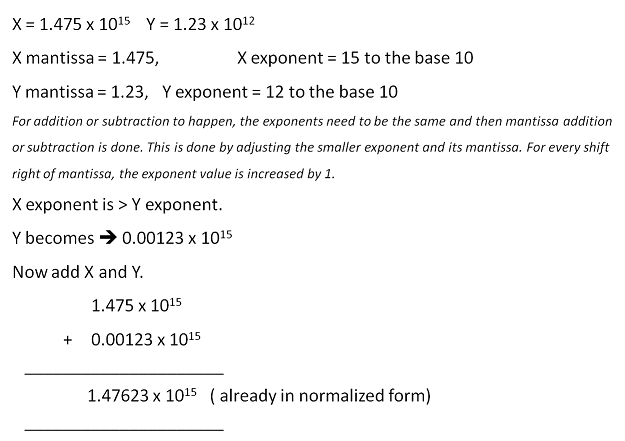
**Algorithm for FP Addition/Subtraction**

Let X and Y be the FP numbers involved in addition/subtraction, where Ye > Xe.

**Basic steps:**

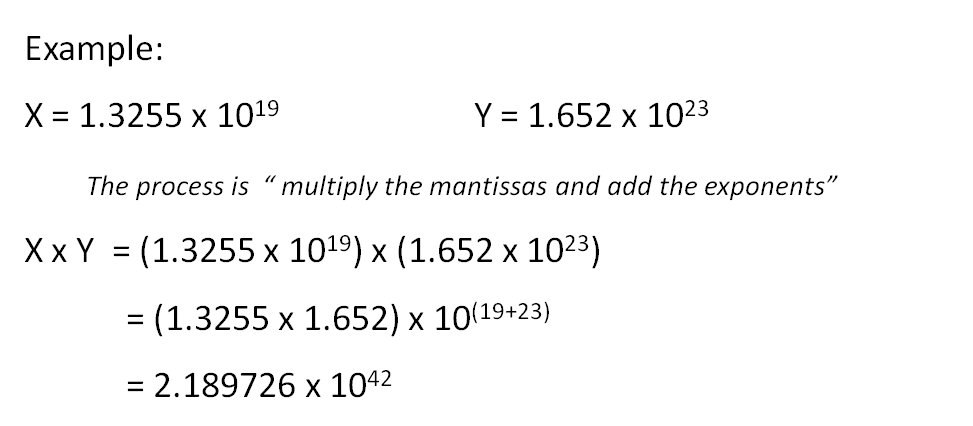
* Compute Ye - Xe, a fixed point subtraction
* Shift the mantissa of Xm by (Ye - Xe) steps to the right to form Xm2Ye-Xe if Xe is smaller than Ye else the mantissa of Ym will have to be adjusted.
* Compute Xm2Ye-Xe ± Ym
* Determine the sign of the result
* Normalize the resulting value, if necessary

We explain this with an example and as below:



**Multiplication and Division**

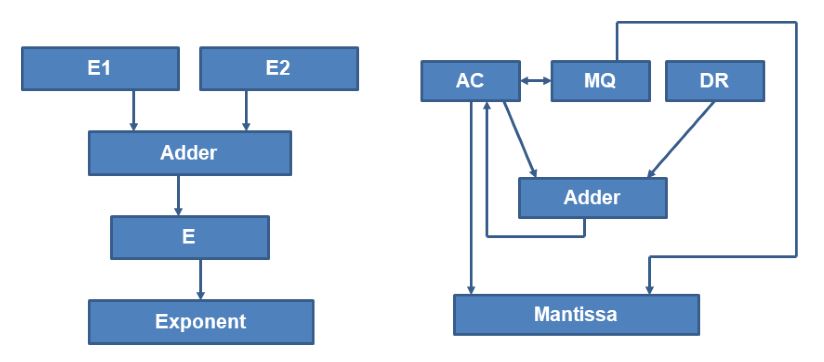
Multiplication and division are simple because the mantissa and exponents can be processed independently. FP multiplication requires fixed point multiplication of mantissa and fixed-point addition of exponents. As discussed in chapter 3 (Data representation) the exponents are stored in the biased form. The bias is +127 for IEEE single precision and +1023 for double precision. During multiplication, when both the exponents are added it results in excess 127. Hence the bias is to be adjusted by subtracting 127 or 1023 from the resulting exponent.



Floating Point division requires fixed-point division of mantissa and fixed point subtraction of exponents. The bias adjustment is done by adding +127 to the resulting mantissa. Normalization of the result is necessary in both the cases of multiplication and division. Thus FP division and subtraction are not much complicated to implement.

All the examples are in base10 (decimal) to enhance the understanding. Doing in binary is similar.

**Floating Point Hardware: -** The floating-point arithmetic unit is implemented by two loosely coupled fixed point data path units, one for the exponent and the other for the mantissa. One such basic implementation is shown in figure 10.2.



Typical Floating Point Hardware

Along with early CPUs, Coprocessors were used for doing FP arithmetic as FP arithmetic takes at least 4 times more time than the fixed point arithmetic operation. These coprocessors are VLSI CPUs and are closely coupled with the main CPU. 8086 processor had 8087 as coprocessor; 80x86 processors had 80x87 as coprocessors and 68xx0 had 68881 as a coprocessor. Pipelined implementation is another method to speed up the FP operations. Pipelining has functional units which can do the part of the execution independently.

**Thank You**