Chapter – 4

Computer Arithmetic

Data Representation:

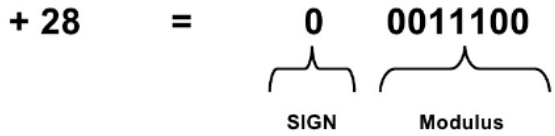
The data representation may be decimal, binary, octal, qunery, hexadecimal, etc. The data representation should have:

1. Provision For Decimal Position:

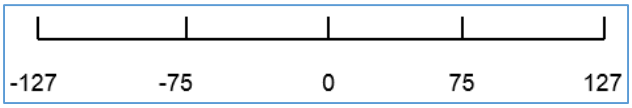
- **Fix Point Representation:** The decimal position is either at beginning or at the end of number. For example: 3.2 = 0.32*10
- **Floating Point Representation:** Here, second register is used to determine the position of decimal.

2. Provision For Sign Representation:

- **Signed Magnitude Representation:**
  This method uses the far left-hand bit which is often referred to as the sign bit. The general rule is that a positive number starts with 0 and a negative number starts with a 1.



- **One's Complement Representation:**
  It is the opposite of something. Because computer do not prefer to subtract, this method finds the complement of a positive number and then addition can take place.



Example

Find the one's complement of +100

1. Convert +100 to binary.
2. Swap all the bits.
3. Check answer by adding 127 to result.

Example

$+100 = 01100100_2$

$01100100_2 = 10011011_2$ (bit interchange)

$10011011_2 = -27_{10}$

$127_{10} + -27_{10} = 100_{10}$

- **Two's Complement Representation:**
  The only difference between the two processes is that the left most bit is -128_{10} rather than -127_{10}. The process for two's complement is exactly the same as one's complement, however we required to add 1 to the results.

  Example:
  Find the two's complement of $+15_{10}$
  1. Convert $+15_{10}$ to binary.
  2. Swap all the bits.
  3. Add 1 to the result.

  Example

  $+15 = 00001111_2$
  $00001111_2 = 11110000_2$ (Interchange)
  $11110000_2 + 1_2 = 11110001_2$
  $-128 + 64 + 32 + 16 + 1 = -15_{10}$

INTEGER ARITHMETIC:

1. NEGATION:

In sign-magnitude (1's Complement) representation, the rule for forming the negation of an integer is simple, just invert the sign bit. In 2's Complement notation, the negation of an integer can be formed with the following rules:

a. Take the Boolean complement of each bit of the integer (including the sign bit). That is, set each 1 to 0 and each 0 to 1.
b. Treating the result as an unsigned binary integer, add 1.

Among these two representation 2's Complement representation is the most widely used method.

2. ADDITION/SUBTRACTION:

Algorithm:

Step 1: Convert all the given numbers in binary form.
Step 2: If any number is negative then find its 2's Complement.
Step 3: Perform addition, if carry occurs then discard the carry.
Step 4: If sign bit of result is 1, then the result is negative. To find its magnitude re-complement the number.

Overflow:

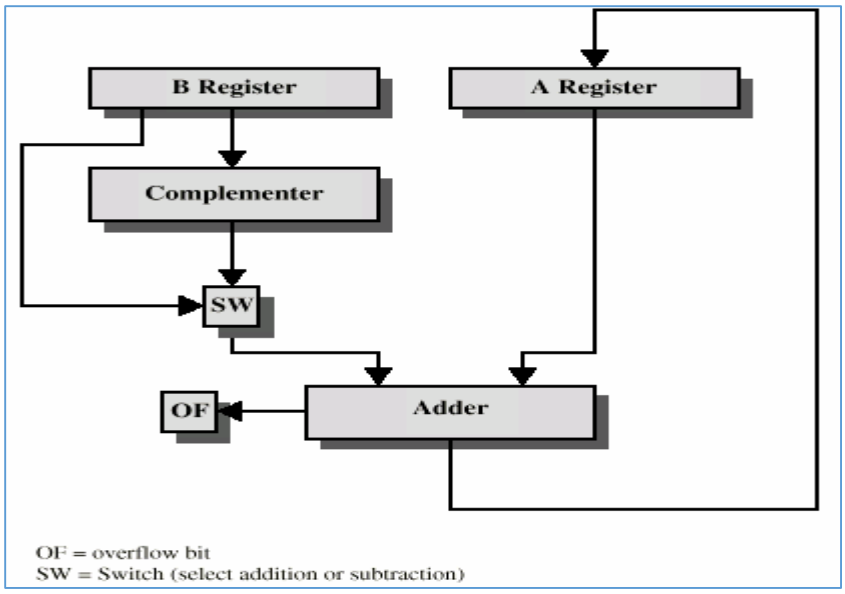
'N' bit operation produces more than N bits. This condition is called overflow condition.

Consider:

A ⟶ Register A

+ B ⟶ Register B

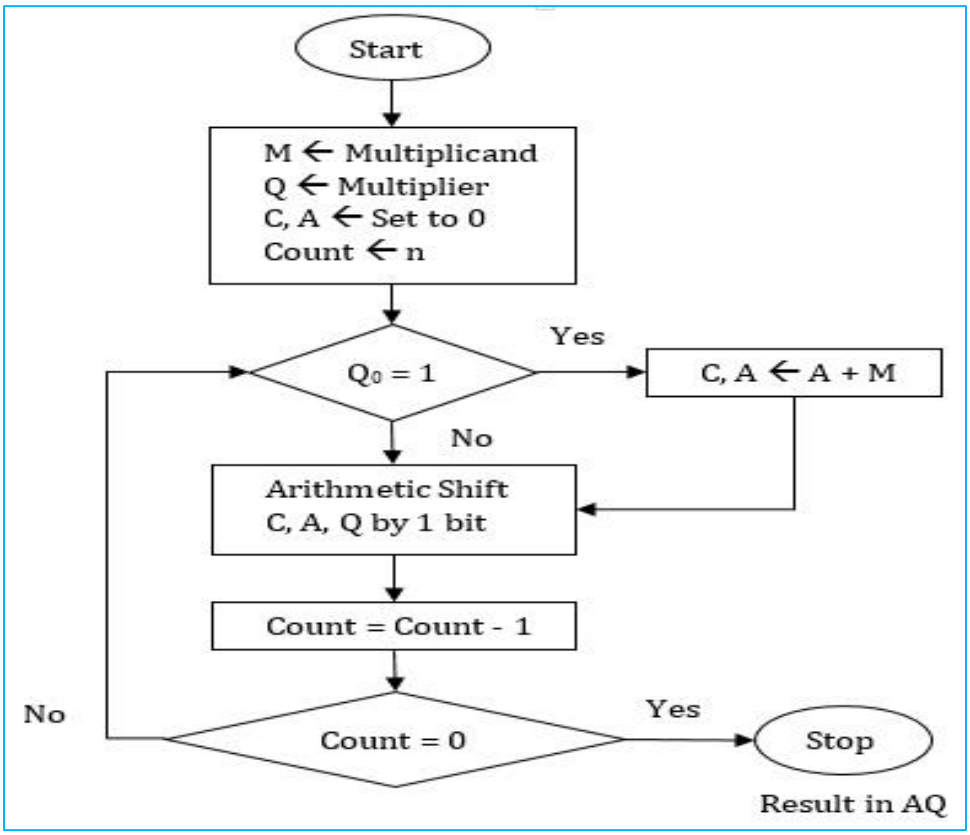
C



The central element is binary adder in which two numbers are presented for addition. The addition produces a sum and overflow condition. Two numbers are presented form two registers A and B. and finally the result is stored in accumulator. The overflow flag indicates, the overflow of data.

In above shown diagram if the number is negative then it is passed to adder after 2's Complement.

3. Unsigned Binary Multiplication Algorithm:



Perform unsigned multiplication of $(13)_{10} * (11)_{10}$

Solution:

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

$(11)_{10} = (1011)_2$

C	A	Q	M	Count	Comment
0	0000	1101	1011	4	Initialize
0	1011	1101	1011	4	A ← A+M
0	0101	1110	1011	3	Right Shift C, A, Q
0	0010	1111	1011	2	Right Shift C, A, Q
0	1101	1111	1011	2	A ← A+M
0	0110	1111	1011	1	Right Shift C, A, Q
1	0001	1111	1011	1	A ← A+M
0	1000	1111	1011	0	Right Shift C, A, Q

Perform unsigned multiplication of $(8)_{10} * (3)_{10}$

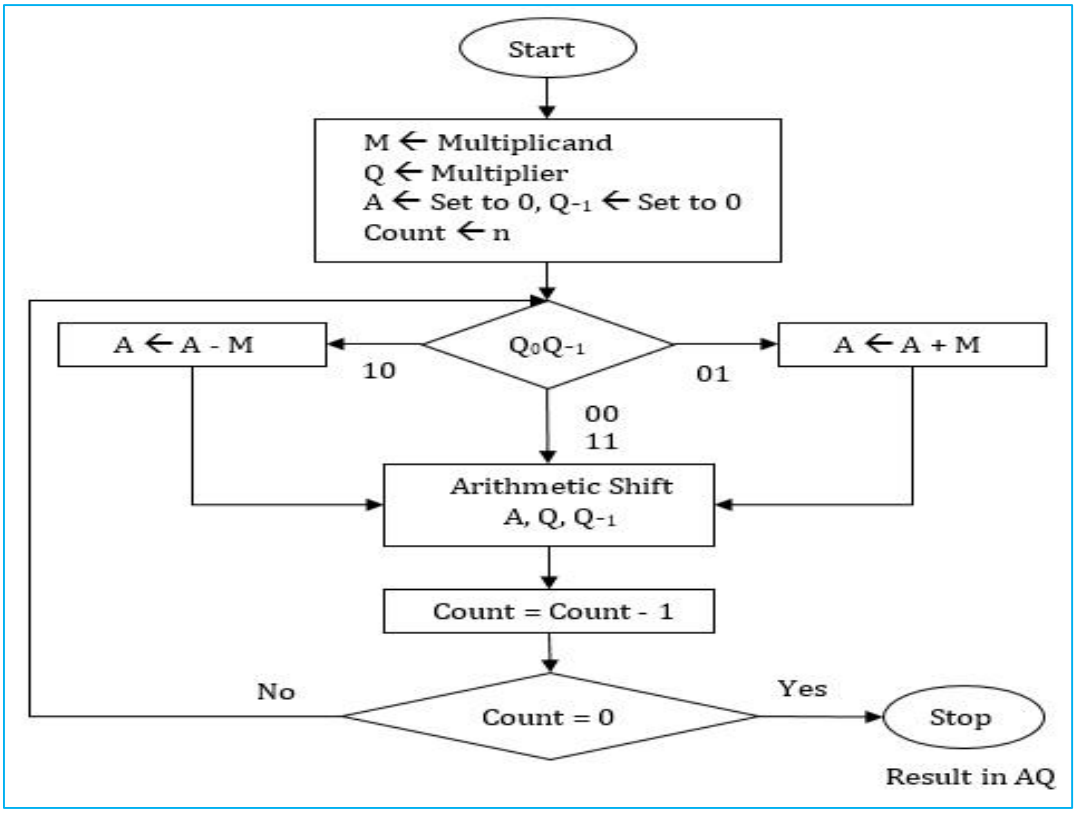
Solution:

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

$(3)_{10} = (0011)_2$

C	A	Q	M	Count	Comment
0	0000	1000	0011	4	Initialize
0	0000	0100	0011	3	Right Shift C, A, Q
0	0000	0010	0011	2	Right Shift C, A, Q
0	0000	0001	0011	1	Right Shift C, A, Q
0	0011	0001	0011	1	A ← A+M
0	0001	1000	0011	0	Right Shift C, A, Q

4. Signed Binary Multiplication (Booth's) Algorithms:



Perform multiplication of $(7)_{10}$ * $(-3)_{10}$

Solution:

$(7)_{10} = (0111)_2$

$(-3)_{10} = (1101)_2$

A	Q	Q_{-1}	M	Count	Comment
0000	0111	0	1101	4	Initialize
0011	0111	0	1101	4	A ← A-M
0001	1011	1	1101	3	Right Shift A, Q, Q_{-1}
0000	1101	1	1101	2	Right Shift A, Q, Q_{-1}
0000	0110	1	1101	1	Right Shift A, Q, Q_{-1}
1101	0110	1	1101	1	A ← A+M
1110	1011	0	1101	0	Right Shift A, Q, Q_{-1}

Perform signed multiplication of $(6)_{10}$ * $(-4)_{10}$

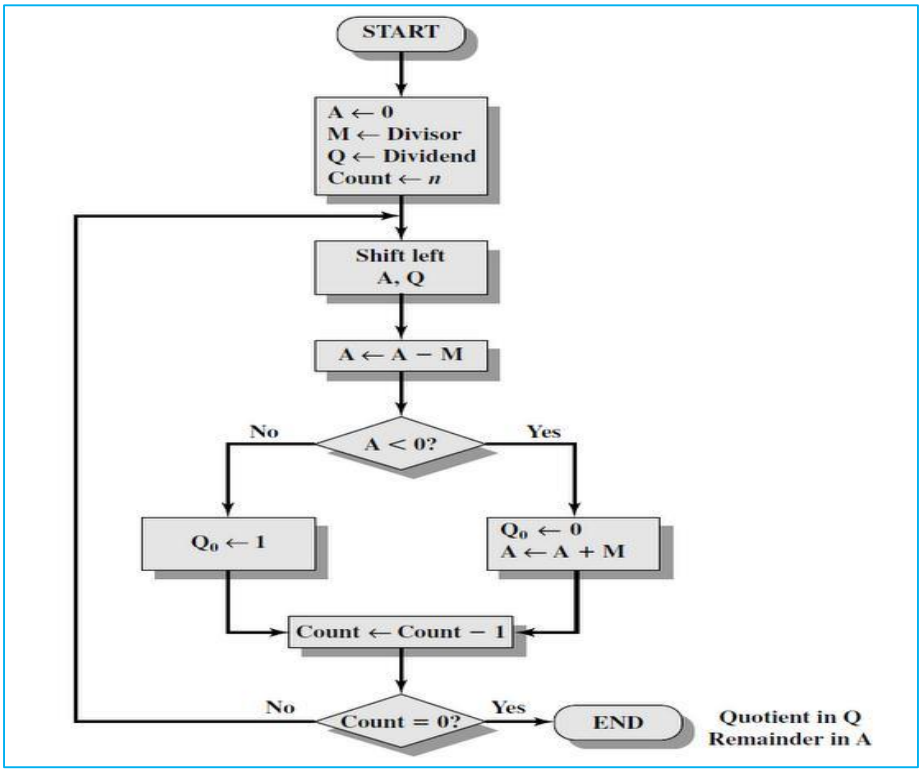
Solution:

$(6)_{10} = (0110)_2$

$(-4)_{10} = (1100)_2$

A	Q	Q_{-1}	M	Count	Comment
0000	0110	0	1100	4	Initialize
0000	0011	0	1100	3	Right Shift A, Q, Q_{-1}
0100	0011	0	1100	3	A ← A-M
0010	0001	1	1100	2	Right Shift A, Q, Q_{-1}
0001	0000	1	1100	1	Right Shift A, Q, Q_{-1}
1101	0000	1	1100	1	A ← A+M
1110	1000	0	1100	0	Right Shift A, Q, Q_{-1}

5. Unsigned Binary Division Algorithm:



Perform unsigned division of $(7)_{10}$ / $(3)_{10}$

Solution:

$(7)_{10} = (0111)_2$

$(3)_{10} = (0011)_2$

A	Q	M	Count	Comment
0000	0111	0011	4	Initialize
0000	1110	0011	4	Left Shift A, Q
1101	1110	0011	4	A ← A-M
0000	1110	0011	3	Q_0 ← 0, A ← A+M

0001	1100	0011	3	Left Shift A, Q
1110	1100	0011	3	A ← A-M
0001	1100	0011	2	Q_0 ← 0, A ← A+M
0011	1000	0011	2	Left Shift A, Q
0000	1000	0011	2	A ← A-M
0000	1001	0011	1	Q_0 ← 1
0001	0010	0011	1	Left Shift A, Q
1110	0010	0011	1	A ← A-M
0001	0010	0011	0	Q_0 ← 0, A ← A+M

Perform unsigned division of $(8)_{10}$ / $(4)_{10}$

Solution:

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

$(4)_{10} = (0100)_2$

A	Q	M	Count	Comment
0000	1000	0100	4	Initialize
0001	0000	0100	4	Left Shift A, Q
1101	0000	0100	4	A ← A-M
0001	0000	0100	3	Q_0 ← 0, A ← A+M
0010	0000	0100	3	Left Shift A, Q
1110	0000	0100	3	A ← A-M
0010	0000	0100	2	Q_0 ← 0, A ← A+M
0100	0000	0100	2	Left Shift A, Q
0000	0000	0100	2	A ← A-M
0000	0001	0100	1	Q_0 ← 1
0000	0010	0100	1	Left Shift A, Q
1100	0010	0100	1	A ← A-M
0000	0010	0100	0	Q_0 ← 0, A ← A+M

REPRESENTATION OF FRACTIONS:

Fractional numbers are numbers between 0 and 1. They are called real numbers in computing terms. Real numbers contain both an integer and fractional part. *Example:* 23.714 OR 01101.1001. Computers use two main methods to represent real numbers:

a. Fixed Point Representation:

This method assumes the decimal point is in a fixed position. Relies on using a fixed number of bits for both the integer and fractional parts i.e. 5 and 3. *Example:* 10011101 = 10011.101

Integer					Fractional		
16	8	4	2	1	0.5	0.25	0.125
1	0	0	1	1	1	0	1

Therefore,

16 + 2 + 1 = 19 (Integer)
0.5 + 0.125 = 0.625 (Fraction)
So 10011.101 = 19.625

Problems are:

We fix the numbers that we can represent i.e. we are limited to amount of numbers that we can actually represent.

B. Floating Point Representation

This is the preferred method because we can represent large numbers. This uses exponential notation which highlights two specific parts of a decimal number:

- Mantissa: Fractional part.
- Exponent: Is the power by 10 which the mantissa is multiplied.

The aim of floating point representation is to show how many numbers before or after the decimal point. The general representation of large floating point representation is, $\pm S*B^{\pm E}$

Where,

- S is significant
- B is base
- E is exponent

A computer will represent a binary number into THREE parts:

- Sign Bit
- Mantissa
- Exponent

Example

$-241.65 = -0.24165 \times 10^3$
$0.0028 = 0.28 \times 10^{-2}$
$110.11 = 0.11011 \times 2^3$

Floating Point Arithmetic:

The overflow and underflow condition may occur in significant and exponent.

Addition and Subtraction:

Algorithm:

1. Check for zero. If anyone operand is zero then result will be another non-zero operand.

2. Align the significant. The decimal position can change so that the exponent value of both operand will be equal.
3. Add or subtract the significant.
4. Normalize the result.

The floating point numbers and their arithmetic calculation can be illustrated as:

Floating Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$\left.\begin{aligned} X+Y &= \left(X_S \times B^{X_E - Y_E} + Y_S\right) \times B^{Y_E} \\ X-Y &= \left(X_S \times B^{X_E - Y_E} - Y_S\right) \times B^{Y_E} \end{aligned}\right\} X_E \le Y_E$ $X \times Y = \left(X_S \times Y_S\right) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Examples:

$X = 0.3 \times 10^2 = 30$
$Y = 0.2 \times 10^3 = 200$

$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$
$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$
$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$
$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$

BCD ARITHMETIC UNIT:

Binary coded decimal (BCD) is a system of writing numerals that assigns a four-digit binary code to each digit 0 through 9 in a decimal (base-10) numeral. The four-bit BCD code for any particular single base-10 digit is its representation in binary notation, as follows:

0 = 0000
1 = 0001
2 = 0010
3 = 0011
4 = 0100
5 = 0101
6 = 0110
7 = 0111
8 = 1000
9 = 1001

Numbers larger than 9, having two or more digits in the decimal system, are expressed digit by digit. For example, the BCD rendition of the base-10 number 1895 is 0001 1000 1001 0101. The binary equivalents of 1, 8, 9, and 5, always in a four-digit format, go from left to right.

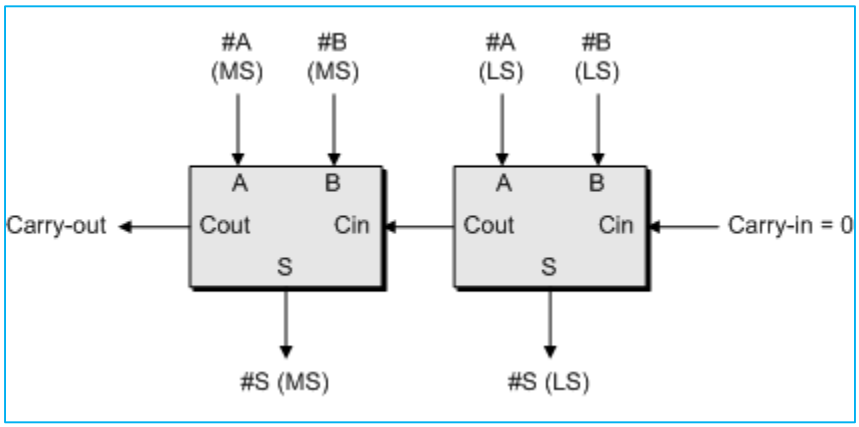
The BCD representation of a number is not the same, in general, as its simple binary representation. In binary form, for example, the decimal quantity 1895 appears as 11101100111

Other bit patterns are sometimes used in BCD format to represent special characters relevant to a particular system, such as sign (positive or negative), error condition, or overflow condition.

The BCD system offers relative ease of conversion between machine-readable and human-readable numerals. As compared to the simple binary system, however, BCD increases the circuit complexity. The BCD system is not as widely used today as it was a few decades ago, although some systems still employ BCD in financial applications.

BCD Adder:

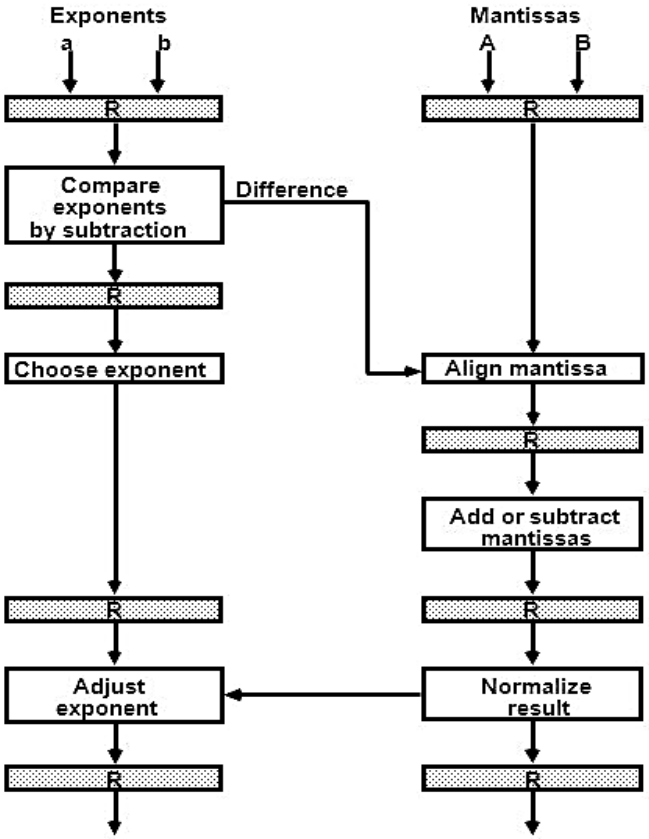
BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. A decimal parallel adder that adds n decimal digits needs n BCD adder stages with the output carry from one stage connected to the input carry of the next higher order stage.



ARITHMETIC PIPELINING:

Arithmetic pipelining is implemented to perform complex arithmetic calculations. Let us consider two variables whose values are represented in exponential form.

Let A = 2.343×10^{12} and B = 212.36×10^{10} then arithmetic addition and subtraction can be performed as follows. The operation is performed parallel.


Exponents
a
b
Mantissas
A
B
R
Compare exponents by subtraction
Difference
Choose exponent
Align mantissa
Add or subtract mantissas
Adjust exponent
Normalize result