Journal of Computer Science 3 (2): 104-106, 2007
ISSN 1549-3636

Computer Arithmetic Algorithms for Mega-Digit Floating Point Numbers' Precision

Musbah J. Aqel and Mohammed H. Saleh
Faculty of Electrical and Computer Engineering, Applied Science University, Amman, 11931, Jordan

Abstract: IEEE standard 754 floating point is the most common representation used for floating point numbers, and many computer arithmetic algorithms are developed for basic operations on this standard. In this study, new computer algorithms are proposed to increase the precision range and to solve some problems that are available while using these algorithms. However, these algorithms provide an optional range of required accuracy (Mega-Digit precision) to meet new computer's applications.

Key words: IEEE 754 standard, floating-point numbers, computer arithmetic, mega-digit precision

INTRODUCTION

There are several ways to represent real numbers on computers. Fixed point places a radix point somewhere in the middle of the digits, and is equivalent to using integers that represent portions of some unit[1]. Floating point representation is the most important representation, which is defined in IEEE 754 standard[2]. 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 virtually on all contemporary processors and arithmetic coprocessors.

The IEEE standard defines both a 32-bit single and 64-bit double format, with 8-bit and 11-bit exponent respectively. . The implied base is 2. In addition, the standard defines two extended formats, single and double, whose exact format is implementation dependent. The extended formats include additional bits in the exponent (extended range) and in the significand (extended precision)[2].

There are many computer algorithms that have been developed to perform the basic operations for floating-point arithmetic[3]. However, IEEE 754 has gone beyond the simple definition of a format to lay down specific practices and procedures so that floating-point arithmetic produces uniform, predictable results independent of the hardware platform. However, some problems and difficulties may arise as a result of performing these operations[4]. Some of these problems can be summarized as follows:

* Exponent overflow
* Exponent underflow
* Significand underflow
* Significand overflow

In order to increase the precision ranges while performing arithmetic operations and to make the precision range a user dependent (i.e. unlimited and optional) according to the application at hand. This is called mega –digit precision. So, a set of computer arithmetic algorithms have been proposed to increase the precision range to mega-digit and also to solve the above mentioned problems.

The algorithms are developed to solve most arithmetic operations including some most common mathematical functions, which are necessary (scientific and general) to solve scientific and general-purpose problems. These algorithms are developed and implemented by C++ language to increase the accuracy and extendibility of this language.

Floating point proposed algorithms: To facilitate the algorithms' performance, there are some points, which are considered and assumed for algorithms development:

* All the numbers are entered as a string format except for the data within the standard range of the machine accuracy.
* If the intended number is huge in precision and difficult to be entered by the user, then it should be stored in a file and, by using the overloaded operator; the file will be loaded (file name).However, the output is also can be stored in a file for further use.
* The entered number (float point number) should be checked for its format. If it is written incorrectly, the software will give error massage.
* Whatever the size of accuracy required, the last two digits in the floating -point number will be zero.
* Whenever the number is entered, it will be mapped into an array with a free size that may equal to data segment reserved. While mapping, the number will be represented in the array as follows:

  A: decimal point by (-1).

Corresponding Author: Dr. Musbah, J. Aqel, Faculty of Electrical and Computer Engineering, Applied Science University, Amman, 11931, Jordan

B: The plus sign of a positive signed number or unsigned number will be represented by (-2).

C: The minus sign of a negative signed number is represented by (-3).

* Due to mapping, the float point numbers that have many useless zeros, either to the left or to the right of the decimal point, will be truncated and not mapped.
* The accuracy size is a machine dependent in the sense of the memory size (i.e. array size) that reserved by the machine, where the array size allowed for the number is machine dependent and differs from machine to machine

Floating point addition algorithm for extendible accuracy

* The two numbers will be mapped into two arrays including their sign and decimal point.
* The numbers will be normalized and shifted to left or right and place zeros at the normalized positions so that the length of the two numbers becomes equal and then represented in the arrays.
* The two numbers will be checked out after mapping, if any has a minus sign.
* For example, (-3), then the subtraction procedure will be called otherwise it proceeds in the addition procedure
* Normal addition will be carried out.
* Whenever a carry out is taken place, it will be stored in the position of the sign bit for the two numbers.

Example: Assume that two small numbers are taken, X = 53.0320, Y = 324.689053210, then X + Y is required. After checking these two numbers and performing normalization then these two numbers are now mapped into two arrays as follows:

-2	0	0	5	3	-1	0	3	2	0	0	0	0	0	0

-2	7	3	2	4	-1	6	8	9	0	5	3	2	1	0

Then after carrying out normal addition between the contents of the two arrays the result will be as follows:

-2	7	3	7	7	-1	7	3	1	0	5	3	2	1	0

Floating point subtraction algorithm for extendible accuracy

* Map the two numbers in two different arrays including their sign and decimal point
* Carry out the 9's complement of the subtrahend number
* Carry out normal addition procedure.
* Then, follow the rules of subtraction with 9's complement regarding to the result.

Example: Assume X=30.25, Y= 30131.256, then carry out X – Y.

After performing normalization or the two numbers, then X=00030.250

And Y=30131.256.

Take the 9'complement of subtrahend, then X=69868.743

Perform normal addition as explained in previous algorithm.

Floating point multiplication algorithm for extendible accuracy

* The two numbers will be kept in two arrays and double size of the largest of the two numbers array will be reserved.
* The decimal point will be removed from its position from the two arrays and will be placed back after completing multiplication process.
* The multiplication will proceed as normal multiplication procedure with writing the result on the same double sized array.
* The decimal point will be placed back after completing the multiplication process as usual in the double sized array. It will be placed in the position equal to its original digits position in the two numbers (i.e.: number of digits in the first number plus number of digits in the second number. Then, a shift left will be performed.

Example: Assume that *X= 57.321* and *Y=1.123456*, then *X, Y* is required.

The size of the two numbers will be determined after mapping in arrays, and the largest in size will determine the size of the output array that, the result will be stored in it. This output array will be double in size, in this example it is equal to double of size of Y (14).

The decimal point will be shifted to the rightmost of the two numbers.

5	7	3	2	1	-1	0

1	1	2	3	4	5	6	-1	0

Normal multiplication will be carried out and stored in the double size array as follow:

The final result will be determined by calculating the number of digits after decimal in both numbers (X =3, Y =6), which is equal here (9). Then, by placing the decimal point 9 digits to the left. The final result will be (64. 397621385).

-2	6	4	-1	3	9	7	6	2	1	3	8	5	0

Floating point division algorithm for extendible accuracy: The dividend and the divisor will be compared. If the dividend is less than the divisor then go to step (4). Otherwise do the following:

* Subtract the divisor from the dividend and count how many times this value can be subtracted. The number of times is considered as a result of the division and it represents the value that will be placed as an integer part to the left of the decimal point of the result.
* Take the result of subtraction and shift right the decimal point of the original number one position to the right and then carry out the subtraction as in step (I) except that the result of count is placed to the right of the decimal point.
* Carry out the result of true subtraction as in step (II) till the required precision is reached.
* Whenever the dividend is less than the divisor, the dividend will be normalized to the left of the decimal point with one digit only.
* Carry out subtraction as in step (1) but the result will be placed to the right of the decimal point, where a number of zeros will be placed directly after the decimal point equal to the shifted digits that carried out by the dividend.

Example: Assume that *X= 9.83*, *Y=7.0* then find X/Y.

Since dividend is greater than divisor, then the value (7) will be checked how many times can be subtracted from (9). The result is (1) and the result of true subtraction is (2). The dividend number now is (2.83).

he dividend will be shifted right one position so that the dividend becomes (28.3).

Carrying out the same previous procedure, the divisor can be subtracted (4) times from the dividend[21.3,14.3,7.3,0.3].

The result of these steps (1.4) for one digit precision only. This procedure can be repeated up to the required precision size.

Floating point exponential algorithm for extendible accuracy: Given that e^n: where, *N* is a float number with any accuracy

Assume that $A^N = e^{x1}$ (1)

Find *Ln* of both sides

$Ln\ A = e^{x1}$

$N\ Ln\ A = x1$ (2)

Find *Ln A*

Assume that $A = e^{x2}$

$Where\ A = 1 + x2 + x_2^2 / 2! + ... + x_2^m / m!$

Assume that m is the accuracy required

Then, m = (accuracy required + k)

This k number is used to prevent any approximation error and $k \geq 15$

$Ln\ A = Ln\ (e^{x2})$

$Where\ A = 1 + x2 + x_2^2 / 2! + ... + x_2^m / m!$

Here, bisection method is used.

Find x_2 using halving procedure where the values of x_2 are [*0, A*]

The value of x2 which is equal to *Ln A*

$X_2 = Ln\ A$ from (2)

Substitute the value of x_2 in step 2

$N\ x_2 = x_1$

Then $A^N = e^{x1}$

$x_2^N = e^{x1}$ from (2)

Direct substitution of x_1 in the series

$e^{x2} = 1 + x_2 + x_2^2 / 2! + ... + x_2^m / m!$

CONCLUSION

Computer arithmetic algorithms are developed and proposed to solve the limited precision range defined by IEEE 754 standard. These algorithms are implemented by C++ language and could successfully perform all the basic and exponential operation with unlimited (i.e. Mega-digit) precision range. This will be helpful to newly introduced computer's applications that require more precision range than IEEE 754 standard.

REFERENCES

1. Mano, M., 2000. Computer System Architecture. Prentice-Hall.
2. IEEE Computer Society, 1985. IEEE standard for binary floating-point arithmetic. IEEE Standard 754.
3. Koren, I., 1993. Computer Arithmetic Algorithms. Englewood Cliffs, NJ, Prentice-Hall.
4. Hayes, J.P., 1998. Computer Architecture and Organization. McGraw-Hill, Third Edn.