ARE IEEE 754 32-BIT AND 64-BIT BINARY FLOATING-POINT ACCURATE ENOUGH?

Bernaridho Hutabarat[1,2*)], I Ketut Eddy Purnama[1], Mochamad Hariadi[1], and Mauridhi Hery Purnomo[1]

1. Department of Electrical Engineering, Faculty of Industrial Engineering, Institut Teknologi Sepuluh Nopember, Sukolilo, Surabaya 60111, Indonesia
2. Department of Informatics Engineering, Ma Chung University, Malang 65151, Indonesia

*)*E-mail: bernaridho.hutabarat@machung.ac.id*

Abstract

This paper describes a research toward the accuracy of floating-point values, and effort to reveal the real accuracy. The methods used in this research paper are assignment of values, assignment of value of arithmetic expressions, and output the values using floating-point value format that helps reveal the accuracy. The programming-tool used are Visual C# 9, Visual C++ 9, Java 5, and Visual BASIC 9. These tools run on top of Intel 80 x 86 hardware. The results show that $1*10^{-x}$ cannot be accurately represented, and the approximate accuracy ranges only from 7 to 16 decimal digits.

Keywords: accuracy, binary, floating-point, IEEE 754

1. Introduction

IEEE 754 [1] is a standard for floating-point processed inside computer hardware. It defines a set of floating-point types. The two most common types are 32-bit and 64-bit binary floating-point, called single-precision and double-precision respectively.

The standard does not elaborate the method of converting the binary floating-point value into the decimal ones, nor does it explain how to convert decimal fraction (or decimal floating-point) into binary floating-point. For example: the value of π (3.14159) represented in binary fraction as 40490FD0 and 400921F9F01B866E, in 32-bit and 64-bit binary floating-point, respectively. IEEE standard [1] does not elaborate how to obtain the result.

Users or programmers never enter or write binary floating-point in the source-code. They will write in terms of (perceived) decimal floating-point.

While the conversion of binary floating-point into decimal fraction are almost always accurate and easy, the conversion from the other direction will almost always be inaccurate and difficult. The initial example will prove this claim, when we display the value 0.1.

Many users, programmers, and lecturers are not aware of the inaccuracy. While the inaccuracy is harmless in 'toy programs', the inaccuracy can cause severe problems and even catastrophic consequences in critical application programs like the explosion of Ariane 5 rocket in 1996 [1].

As Goldberg [2] puts it, there are only few books on the subject. Floating-point is touched very briefly in textbooks like [3-4]. Recently, some literatures are written to fill this gap [5-7].

It is to raise awareness and understanding of the IEEE 754 32-bit and 64-bit floating-point in accuracy (lacking in manuals like [8-9], and in standards like [10-11] that this paper is written. In addition, this paper shows simple ways to prove the inaccuracy.

This paper contribution is twofold: 1) Algorithm to convert binary fraction into decimal fraction; 2) Method to determine upper limit of accuracy within a binary floating-point format.

Binary floating-point cannot accurately represent decimal fraction. More specifically we hypothesize that: 1) No guarantee of accuracy for integral values $>2^{24}$ for single-precision, and integral values $>2^{53}$ for double-precision; 2) Single-precision accuracy $>10^{-38}$ and $<10^{+38}$; 3) Double-precision accuracy $>10^{-307}$ and $<10^{+307}$. All those hypotheses will be put to the test.

Our hypothesis further state that the factors determining the accuracy of binary floating points are: 1) The width of fraction digits; 2) The presence and usage of large

unsigned integral registers; 3) The sophistication of conversion algorithm.

IEEE binary floating-point format specifies three parts: sign-bit, power-bits, and fraction-bits (IEEE 754-1985). The sign-bit always occupies 1 bit only. The number of power-bits and fraction-bits vary. Single-precision has 8 power bits, double-precision has 11. Single-precision has 23 fraction bits, double-precision has 52 fraction bits [12-15] (Figure 1).

Single precision binary floating-point format uses 32-bit data format, while the double-precision uses 64-bit data format. The value of an object of binary floating-point is computed as follows (Formula 1):

$$(-1)^{sign} \times 2^{Power} \times \text{Fraction} \quad (1)$$

IEEE 754-2008 calls the fraction as *significand*. It is 'significand' because it is significant in determining the accuracy for both integral values and floating-point values. Programming books (like [8-9]) often write '7-8 *significand* digits' and '15-16' *significand* digits'. The former refers to 32-bit, while the latter refers to 64-bit binary floating-point. We still use the term fraction to ease the discussion about accuracy of decimal fraction.

The value on the rightmost column can be derived from *signicand* * $log_{10}(2)$. It is the significant (decimal) digits that the programmers must rely on stating the accuracy of their programs.

2. Methods

There are two methods to test the several hypothesis. The first is to check the assigned value (no arithmetic). The second is to check the result of computation, using arithmetic operators.

The first methodology consists of these steps: 1) Assign some perceived decimal fraction value into floating-point variables; 2) Output the value using various formats; 3) Check the accuracy of output values.

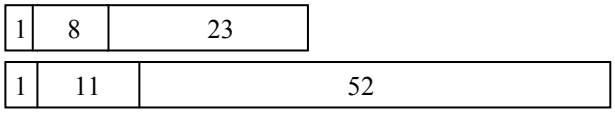

Figure 1. Single-precision (upper) and Double-precision (Lower)

Table 1. *Significand* and Significant

	Significand (binary digit)	Significant (decimal digit)
32-bit	23	7-8
64-bit	52	15-16

The second methodology consists of these steps: 1) Assign some perceived decimal fraction value into two floating-point variables; 2) Operate the two floating-point variables; 3) Assign the returned-value of the operator into another variable; 4) Note the value using various formats; 5) Assign the integral version of the input values into two integral variables; 6) Operate the two integral variables; 7) Note the output value; 8) Compare the output values of floating-point version with the output values of integral version.

We choose values that represent some patterns to make the cases interesting.

Representative languages and tools. In this paper we choose several programming-languages to write examples: BASIC, C#, C++, and Java. We use Visual C# 9 (2008), Visual C++ 9 (2008), Visual BASIC 9 (2008), and NetBeans 6.

Those programming languages were chosen due to their popularity. Besides the popularity, the free-license software equivalent are available. MONO free software organization and community has made code-translator for C# and BASIC available on top of Linux Operating System. This availability means that the researchers interrested in proving the result of this research can perform the tests in cost effective way.

Hypothesized register and conversion of binary floating-point. Converting a binary floating-point value to string value is a complex process. The hardware does not literally represent 0.5_{10} or 0.25_{10}. Neither does the hardware literally represent 0.1_2 or 0.01_2. Hardware cannot literally represent the values like in Figure 2.

Binary floating-point with x fraction bits require integralregisters capable of 10^x [7,16]. Value 0.1_2 requires 10^1 to represent it accurately, i.e., 0.5_{10}. Value 0.01_2 requires 10^2 to represent it accurately, i.e., 0.25_{10}. In the latter example, if integral registers cannot represent 10^2, then we cannot represent 0.25_{10} (and thus, cannot represent 0.01_2) accurately.

The hardware first compute 10^x. Then the hardware computes the x fraction bits as if they are integral values [2,14,17-18]. The decimal fraction is gained by integral division, with the former divided by the latter. The result of integral division is stored in long unsigned integral register (see Formula 1). Formula 2 exemplifies the case for $x = 3$.

$$\sum_{i=x}^{1} b_i \frac{10^i}{2^i} \quad (2)$$

$$b_3 \frac{10^3}{2^3} + b_2 \frac{10^2}{2^2} + b_1 \frac{10^1}{2^1} \quad (3)$$

Using the example value, 0.001_2 is now processed as 1000_2, but is not evaluated to 8. It is evaluated as $1*(10^3/2^3) + 0*(10^2/2^2) + 0*(10^1/2^1)$, result in 125.

Finally that integral value is divided by 10^x [18]. In our example, the 125 is divided by 10^3, the final result is 0.125_{10}. The last bit (2^0) is never computed. Thus, the decimal fraction as the result of simplified conversion is computed using the formula depicted in Formula 3.

$$\frac{\sum_{i=x}^{1} b_i \frac{10^i}{2^i}}{10^x} \quad (4)$$

Limit of conversion The size of unsigned integral registers determines the limit of conversion. The Pre-Pentium mainboard of Intel 80 x 86 provide 80-bit registers, which limit the conversion to 10^{24} ($\approx 2^{80}$). The first step of conversion with those regiters is formulated in Formula 3.

$$b^{24} \frac{10^{24}}{2^{24}} + b_2 \frac{10^2}{2^2} + b_1 \frac{10^1}{2^1} \quad (5)$$

The Pentium main boards and later version provide 128-bit registers, which raises the conversion limit to 10^{38} ($\approx 2^{128}$). With these registers, the first step of conversion is depicted in Formula 5.

$$b^{38} \frac{10^{38}}{2^{38}} + b_2 \frac{10^2}{2^2} + b_1 \frac{10^1}{2^1} \quad (6)$$

All the limits related to both formulas assume the absence of sophisticated algorithms conversion. The series show the importance of width of unsigned integral registers to gain accuracy of decimal fraction produced from binary floating-point. On the next section the experiments and their results are detailed.

First set: boundary integral values. In the first set of experiments, we want to prove theory that accuracy is not guaranteed: 1) For single-precision in representing integral value $>2^{24}$ ($\geq$16777217); 2) For single-precision in representing integral value $>2^{53}$ ($\geq$9007199254740993).

Second set: value of 10 power –n. In the second set of experiments, we want to prove the theory that $1 * 10^{-n}$ with $n>0$ cannot be represented accurately by single-precision or double-precision object. Note that values like 0.5, 0.4 or anything else are not qualified, since they are not $1 * 10^{-n}$. These tests are intended to prove the inequality depicted in Formula 7.

$$\boxed{0.1_2} \quad \boxed{0.01_2} \quad \boxed{0.001_2}$$

Figure 2. Fraction Values cannot be Literally Represented

$$\sum_{i:=1}^{n} 2^{-n} \neq 10^{-x} \quad (7)$$

Third set: multiply the first pair of floating-point values. On this experiment, we multiply two single-precision values with designed accuracy of 8 decimal fraction (0.12345678 and 0.87654321), and assign the result into a single-precision floating-point object. These tests are to prove that single-precision object cannot accurately represent decimal fraction down to 10^{-38} as implicitly suggested by programming books (e.g., [8-9]).

Fourth set: multiply the second pair of floating-point values. These experiments are similar to the previous set, but the input output are of double-precision. These tests are to prove that while using double-precision object may improve accuracy, a claim that double-precision object can hold decimal fraction down to 10^{-308} is a false one.

Fifth set: multiply the third pair of floating-point values. These experiments are similar to the previous set but the multiplicand and multiplier are multiplied by 0.01. We want to observe the effect of shifting a value by decimal fraction (10^{-x}, $x>0$). We choose the case where $x = 2$.

Coloring. To help the readers focusing on important things within the result of experiment, we use coloring. Inaccurate values or results are yellow-colored (highlighted with low-intensity color). Desired or accurate values are green-colored (highlighted with high-intensity color).

The sample values are chosen in such a way that their integral equivalent (e.g., 12345678 * 87654321) can be computed using common Intel 80x86 hardware.

3. Results and Discussion

Theoretical upper bound. All the research papers consulted in this research [5,19-21]. [3-4,14-15,18,21-24,28-29] did not provide direct relationship between the width of fraction and the width of available unsigned registers in determining the upper bound of accuracy of decimal fraction computed within binary floating-point. We establish the direct relation-ships between y fraction bits and z bit of unsigned integral registers in the statement (8) and (9) below:

y fraction-bits requires [11.03521 bits to accurately represent the value in decimal fraction (8)

z bits unsigned integral registers can accurately represent $\lfloor \frac{z}{11.035} \rfloor$ decimal fractions without sophisticated algorithm (9)

Accurate representation of up to x decimal fraction (10^{-x}, x >0) requires $\lceil 11.03521$ bits (10) unsigned integral registers

Following (3), 2^{-y} fraction requires unsigned integral registers capable of representing unsigned integral value of $10^{y-2}{}_{\log} = 10^{y * 3.321928095}$, or $10^{3.3219 * y}$, approximately. In turn, the width of unsigned integral value is $\log_2 (10^{3.3219 * y})$. Since 10 is $2^{3.3219}$ we can rewrite $\log_2 (10^{3.3219 * y})$ as $\log_2 (2^{3.3219 * 3.3219 * y})$. That brings us to the conclusion that to represent 2^{y}fraction accurately as decimal fraction requires $3.32192^2 * y$bit, or 11.03521 y.

On the reverse direction z bit unsigned integral register will be capable of representing $\lfloor \frac{z}{11.035}$ decimal-digits fraction accurately without sophisticated algorithm. The statement (9) is the logical consequence of statement (8). For example, the 128-bit unsigned integral registers will be capable of accurately representing 11 decimal fraction.

The requirements can still be perceived from another direction: the required decimal accuracy. Suppose we want x decimal fraction accuracy. The size of required unsigned integral register is $\lceil x * 11.0352$. If we want 12 decimal fraction accuracy we need 133 bits unsigned integral registers. This is only possible if the 128-bit unsigned integral registers are present and they are equipped with 5 extra/guard bits [3].

The measures in statements (8) and (9) justify the inaccuracy the result of sample computations in this paper. For example, to accurately representing 16 decimal fraction we need 177 bits according to statement (10). Since the tested hardware does not have 177 bits, normally the result will be inaccurate.

The measure written in the statements (8)-(10) are the yardstick in measuring the effectiveness of algorithms to improve the decimal fraction accuracy. Those measure are absent in all research papers consulted in this research.

The methods in this paper are simple enough to carry out to prove that the accuracy of binary floating-point does not meet the claims in the programming books [8-9] and standards [10-11]. We recommend the authors and standard committees to correct the claim of floating-point accuracy in the programming books and standard.

First set: boundary integral values. The results in Table 2 show that all programming tools produce inaccurrate values for any integral value $>2^{24}$. The highlighted values confirm our hypothesis that the integral value representation of single-precision is no longer guaranteed to be accurate when the value $>2^{24}$.

Table 2. Results and Inaccuracies of Single-precision Floating-point

Language	Assigned value	Displayed value
BASIC	16777216.0	16777220.0
BASIC	16777217.0	16777220.0
C#	16777216.0	16777220.0
C#	16777217.0	16777220.0
C++	16777216.0	16777216.0
C++	16777217.0	16777216.0
Java	16777216.0	16777216.0
Java	16777217.0	16777216.0

Table 3. Results and Inaccuracies of Double-precision Floating-point

Language	Assigned value	Displayed value
BASIC	9007199254740992.0	9007199254740990.0
BASIC	9007199254740993.0	9007199254740990.0
C#	9007199254740992.0	9007199254740990.0
C#	9007199254740993.0	9007199254740990.0
C++	9007199254740992.0	9007199254740992.0
C++	9007199254740993.0	9007199254740992.0
Java	9007199254740992.0	9007199254740992.0
Java	9007199254740993.0	9007199254740992.0

Table 4. Value from Simple Assignment: Single-precision

Language	0.1	0.01
Visual BASIC	0.100000000000	0.0100000000
Visual C#	0.100000000000	0.0100000000
Visual C++	0.100000000015	0.0099999998
Java	0.100000000015	0.0099999998

Table 3 show that all programming tools produce inaccurrate results for any integral value $>2^{53}$ using double-precision floating-point. Inaccurate values or results are yellow-coloured (highlighted with low-intensity colour). Desired or accurate values are green coloured (highlighted with high-intensity colour).

Second set: assigning 0.1 and 0.01. On these experiments, values 0.1 and 0.01 are assigned to single-precision object. The values are then displayed using 10 decimal-fraction format.

The results in Table 4 show Java and Visual C++ display inaccurate values, with error rate 1.5^{-10} (1.5^{-11}/ 10^{-1}) for displaying 0.1_{10}. Java and Visual C++ display inaccurate values with higher error rate (2^{-10}) in displaying the decimal value 0.01_{10}.

Third set: assigning 0.1 and 0.01. On this experiment, we multiply two single-precision values (0.12345678 and 0.87654321), and assign the result into a single-precision floating-point object. The desired (accurate)

value is written in the last row of Table 5. Table 5 show that all programming tools display inaccurate values, with C++ and Java producing the values that are nearest to accurate one.

Fourth set: multiply the second pair of floting-point values. On these experiments we multiply two double-precision values (0.12345678 and 0.87654321), and assign the result into a double-precision floating-point object.

Compared with Table 5, Table 6 show that using double-precision does improve the accuracy. But binary floating-point still introduce difficulty for programmers that created C++ and Java programming tools, since both tools produce inaccurate values.

Fifth set: multiply the third pair of floating-point values. In these experiments the multiplicands and multipliersof the previous experiments are multiplied by 0.01. For the unaware programmers, the previous experiment may bring hope that the accuracy of double-precision can reach 10^{-307} (since accuracy down to 10^{-16} seem to be achievable).

Table 7 lists the result of multiplying 1.2345678E-3 with 8.7654321E-3. Only Visual C++ produces result that is still accurate. The error percentage for Visual BASIC and C# is small:1.84816916537407E-13%.

Theoretically, the expected accuracy is mostly 10^{-16}. But all programming tools exceed the theoretical limit.

Table 5. Value from Multiplication: Single-precision

BASIC:	0.108215205000000000
C#	0.108215205000000000
C++	0.108215205371379900
Java:	0.108215205371379900
Actual:	0.108215202237463800

Table 6. Value from Multiplication: Double-precision

BASIC:	0.1082152022374640
C#	0.1082152070974710
C++	0.1082152022374638
Java:	0.1082152022374638
Actual:	0.1082152022374638

Table 7. Result of the Final Experiment

BASIC:	0.0000001082152022374640
C#	0.0000001082152022374640
C++	0.0000001082152022374638
Java:	0.0000001082152022374600
Actual:	0.0000001082152022374638

Retrospection. All programming tools fail to produce accurate integral values when the integral value $>2^{n+1}$ with n represents (width of) *significand*. Thus, the accurate integral values representable by binary floating are not very large.

Some programming tools (Visual C# and Visual BASIC) seem to be able to represent 10^{-x} accurately. Theoretically this is impossible (see Formula 11).

$$\sum_{i:=1}^{n} 2^{-n} \neq 10^{-x} \quad (11)$$

A possible explanation for the seemingly accurate values of 10^{-x} shown by Visual BASIC and Visual C# are the sophistication of conversion algorithm. The code-translator takes the input string-value, performs the conversion (with truncation and rounding), and knows how to give the perception to the users.

The explanation can be used to explain the phoneomenon in the third experiment. In dealing with more complicated values resulting from the expression containing artihmetic operator-call (like 0.12345678 * 0.87654321), BASIC and C# produce inaccurate values. If the accuracy of $10^{-x}$is real, the result of multiplication should be accurate down to x decimal fraction.

The inaccuracy is also evident at the fourth experiement. Both Visual BASIC and Visual C# – that previously give the impression of capable of accurately representing decimal fraction $1 * 10^{-x}$ – fail to produce accurate values, compared to other programming tools. On the other hand, Java and Visual C++ prove that the theoretical limit of 10^{-16} (for double-precision) is achievable. Both can display the result accurately up to 16 decimal fraction.

The results as in Table 7 are worth explaining. Theoretically the maximum precision is up to 16 decimal digits, but all programming tools surpass that limit. The possible factor is the presence of large unsigned registers. If the 80-bit or 128-bit unsigned integral registers are unavailable, the results in Table 7 cannot be achieved.

Large unsigned integral registers, however, is not the only factor. The width of *significand* also determines the accuracy. When the result of multiplying two single-precision binary floating point objects (values 0.12345678 and 0.87654321) is assigned to single-precision object, the results from all sample programming tools are not accurate, even when 128-bit unsigned integral registers are available. This inaccuracy is due to the limit of 32-bit floating-point accuracy to 2^{-24} (and therefore 10^{-8}).

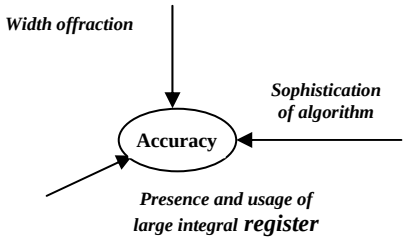

Figure 3. Factors Determining the Real Decimal Accuracy

Finally, Table 7 shows shifting the decimal value to 10^{-x} (x>0) decreases the accuracy. Because 10^{-1} cannot be accurately represented, multiplying a floating-point value by 10^{-x} decreases the accuracy. The factors determining the accuracy is pictured in Figure 3.

4. Conclusion

The single-precision 32-bit and double-precision 64-bit IEEE 754 binary floating-point cannot accurately represent decimal fraction 10^{-x} since no binary fraction is exactly equivalent to 10^{-x}. The accuracy of integral value for single-precision is limited to–(2^{24}) .+(2^{24}), while for the double-precision it is limited to –(2^{53}).+ (2^{53}).Converting binary floating-point to decimal fraction requires large integral registers (≥ 80 bit), with the basics of conversion algorithm for *significand* is depicted in this paper. The perceived accurate values of 10^{-x} are due to the conversion algorithm. The contribution of this research is an algorithm to convert binary fraction into decimal fraction, and the method to determine upper limit of accuracy of a binary floating-point format given the width of unsigned integral register. This relationship is not found in all the works listed in the references of this paper. Setting aside the factor of algorithm's sophistication, the width of fraction is the primary factor, followed by the size of used large unsigned integral register. This research is limited to the conversion of binary fraction into decimal fraction, the inaccuracy of integral value representation, and the inaccuracy of result of simple arithmetic. The tested hardware is limited to Intel processor, and the software is limited to C#, C++, Java, and BASIC programming tool. Further research may look into the accuracy of computation using natural and decimal logarithm. Natural logarithms play vital role for many scientific computations.

References

[1] W. Kahan, Intel and Floating Point, 2008, in *www.intel.com/standards/floatingpoint.pdf*.
[2] D. Goldberg, ACM Computing Surveys 23/1 (1991) 5.
[3] W. Stalling, Computer Organization and Architecture, 7th ed, Pearson Education, 2006, p.750.
[4] A.S. Tanenbaum, Structured Computer Organization, 5th ed, Pearson Prentice Hall, Boston, 2006, p.777.
[5] W.D. Clinger, Conference on Programming Language Design and Implementation, Utah, USA, 2003, p.360.
[6] M.F. Cowlishaw, Proceedings 16th IEEE Symp. Comput. Arithmetic. Washington DC, USA, 2003, p.104.
[7] J.J. Fernández, I. Garc´ıa, E.M. Garzón, Future Generation Computer Systems, Elsevier, 19/8 (2003) 1321.
[8] H.M. Deitel, P.J. Deitel, Visual BASIC 2008 How To Program, Pearson Education, New Jersey, 2009, p.1452.
[9] P.J. Deitel, H.M. Deitel, Java: How to Program, 5th Edition, Pearson, New Jersey, 2006, p.1546.
[10] ECMA International, Standard ECMA-334 C# Language Specification, 4th ed, ECMA International, Geneva, 2006, p.553.
[11] ECMA International, Standard ECMA-335 C# Common Language Infrastructure, 4th ed, ECMA International, Geneva, 2006, p.558.
[12] M. Cornea, 19th IEEE International Symposium on Computer Arithmetic, Portland, Oregon, USA, 2009, p.225.
[13] IEEE, IEEE 754 Standard for Floating-Point Arithmetic, IEEE Xplore Digital Library, *http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4610935*, 2008.
[14] K. Karuri, R. Leupers, G. Ascheid, H. Meyr, M. Kedia, Proceeding of Conference on Design, European Design and Automation Association 3001 Leuven, Belgium, Belgium, 2006, p.221.
[15] A.P. Stamatakis, H. Meier, T. Ludwig, Proceedings of 18th IEEE/ACM International Parallel/Distributed Processing, New York, USA, 2004, p.456.
[16] M.A. Erle, M.J. Schulte, B.J. Hickmann, IEEE Symposium on Computer Arithmetic, Montpellier, France, 2007, p.46-55.
[17] M. Cornea, C. Anderson, J. Harrison, P. Tang, E. Schneider, E. Gvozdev, C. Tsen. Proceedings of the 18th IEEE Symposium on Computer Arithmetic, Montpellier, France, 2007, p.29.
[18] T. Lang, A. Nannarelli, IEEE Trans. Comput. 56/6 (2007) 727.
[19] M. Burtscher, P. Ratanaworabhan. IEEE Trans. Comput. 58/1 (2009) 18.
[20] S. Graillat, V. Ménissier-Morain, Proceedings of the 21st International Symposium on Nonlinear Theory and its Applications, Vancouver, Canada, 2007, p.341.
[21] M. Hiromoto, H. Ochi, Y. Nakamura, IPSJ Transactions on System LSI Design Methodology, Information and Media Technologies 4/2 (2009) 250.
[22] A. Kaivani, A.Z. Alhosseini, S. Gorgin, M. Fazlali,

Proceeding of the 9th International Conference on Information Technology, IEEE Computer Society, Washington DC, USA, 2006, p.273.
[23] G.R. Morris, V.K. Prasanna, Proceedings of the 9th Annual High Performance Embedded Computing Workshop, Lexington, USA, 2005, p.420.
[24] S. Oishi, K. Tanabe, T. Ogita, S.M. Rump, J. Comput. Appl. Math. 205/1 (2007) 533.
[25] S.M. Rump, P. Zimmermann, S. Boldo, G. Melquiond, BIT Numerical Mathematics 49/2 (2009) 419.
[26] H.B. Shah, C. Gorg, M.J. Harrold, IEEE Transactions on Software Engineering, IEEE Computer Society Digital Library, 2010, p.150. DOI:*http://doi.ieeecomputersociety.org/10.1109/TSE.2010.7*.
[27] C. Tsen, M.J. Schulte, S.G. Navarro, Proceedings of the IEEE International Conference on Application-Specific Systems, Architecture, and Processors, Montreal, Canada, 2007, p.115.
[28] C. Tsen, M.J. Schulte, S.G. Navarro, Proceedings of the 25th IEEE International Conference on Computer Design, Lake Tahoe, CA, 2007, p.288.
[29] C. Tsen, S.G. Navarro, M. Schulte, B. Hickmann, K. Compton, Proceedings 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors, Boston, USA, 2009, p.8.