Proposal to Improve Data Format Conversions for a Hybrid Number System Processor

LUCIAN JURCA, AUREL GONTEAN, FLORIN ALEXA, DANIEL-IOAN CURIAC
Department of Applied Electronics, Department of Automation and Applied Informatics
"Politehnica" University of Timisoara
B-dul Vasile Parvan nr.2, Timisoara
ROMANIA

Abstract: - In this paper we show how the multiplication or division of two single-precision floating-point operands can be solved faster and more accurately than in previous works that used logarithmic arithmetic. The logarithm calculation and ALU operation are fused in order to perform one single non-redundant addition in the critical path for finding the logarithm of the result. A second non-redundant addition is used to produce the result in floating-point format. Using Matlab analysis, the conversion error was also diminished by using correction values in the look-up tables content.

Keywords: floating point, logarithmic number system, partial product, 4:2 compressor, redundant adder, ALU.

1 Introduction

From the beginning, the floating-point (FP) units offered sufficient advantages for being significantly developed and widespread in time, and thus their performance has been continuously improved. However, compared to fixed-point arithmetic, the FP operations are more complex and imply more stages.

The increase of integration density has permitted the development, as an alternative, of the logarithmic number system (LNS) processors out of which we mention [4], [5] and [8], but in these the main difficulty is to implement the addition and subtraction operations.

Avoiding these disadvantages and at the same time keeping the qualities of both FP and LNS can be achieved through the design of a hybrid unit which combines the attributes of the FP processor with logarithmic arithmetic. Very interesting and attractive solutions in this direction were offered by Lai in [1], [2] and [3], where addition and subtraction were performed in FP and multiplication, division, square root and all the other operations in LNS. For the format conversions, a linear-interpolation algorithm was implemented by using multipliers and non-redundant adders. This algorithm will be presented in section 2 of the paper.

However the redundant adders are useful when a series of additions occur in sequence, as happens in this case. The method of redundant summation of partial products with other inputs has already been used in [5] and [8] to implement the LNS addition and subtraction. Applying this method, one single non-redundant adder is required at the end of the interpolation. But this idea was never exploited to improve the data format conversions FP-LNS and LNS-FP in which special non-monotonic functions must be interpolated.

Thus, in section 3 we present how we can obtain the logarithms of the two operands in carry-save form and the way in which we proceed in the case in which the second term of the linear interpolation is subtracted.

In section 4 we present a new ALU organization which supposes the using of one single non-redundant addition in the critical path instead of three as in [1], [2] and [3] for finding the logarithm of the result of multiplication or division.

In section 5 we describe a method for reducing the format conversion error to half.

Section 6 will conclude the paper.

2 Data format conversion algorithms

A binary number A in FP system, in single-precision format is written:

$$A = (-1)^S (1 + 0.M) \cdot 2^{E-127} \quad (1)$$

where S represents the sign bit, M represents the normalized significand with 23 bits and E represents the biased exponent with 8 bits.

In the LNS a binary number z is represented:

$$z = (-1)^{S_z} \cdot 2^{N_z}, \quad (2)$$

where S_Z is the sign bit and N_Z is a fixed-point number having n bits, out of which i bits (i=8) for the integer part I_Z, and f bits (f=23) for the fractional part F_Z. We have:

$$n = i + f \quad \text{and} \quad N_Z = I_Z + F_Z \tag{3}$$

Considering the normalized significand (1+0.M, including the hidden bit) in the domain [1,2), the integer part of the logarithm of the number is given by the value of the unbiased exponent and the fractional part by the logarithm of the significand.

In [1], [2] and [3] the calculation of logarithm and anti-logarithm used the partition of the argument and the memorizing of only certain values for reducing the amount of memory and, in addition, it applied a correction method based on the memorization in the same points of the values of the function derivative, after which the linear interpolation was performed. Thus, it was noted y = 0.M and the significand y was partitioned in two parts: y_1, containing the most significant 11 bits and y_2, containing the least significant 12 bits. The values of the function log(1+y)-y in these $2^{11} = 2048$ points were memorized in internal ROM (ROMA) as correction values E_y provided through the application of the address y1.

Thus the following approximation was obtained:

$$\log(1+y) \cong y + E_y \pm \Delta E_y \times y_2 \tag{4}$$

A second look-up table (ROMA') was needed for the memorizing of the values of the derivative function ΔE_y. Adopting for ΔE_y a 12-bit representation, the complete conversion between the two formats was made through a reading in the look-up tables, a 12×12 bit multiplication and two 23-bit additions.

The calculation of the anti-logarithm was made in the same way. Considering C the result of finding the anti-logarithm, then:

$$C = 2^{E+127+0.M} = 2^{E+127} \cdot 2^{y}, \tag{5}$$

where E represents the integer part in LNS format and M represents the fractional part.

Y is partitioned in the same way and a ROM (ROMC) was used for memorizing the conversion error E_y in 2048 points, as well as the difference ΔE_y (ROMC'). The final result of the conversion was:

$$2^{y} \cong (1+y) - E_y \pm \Delta E_y \times y_2 . \tag{6}$$

The correction values E_y, for both log(1+y)-y and $(1+y)-2^y$ are represented in Fig.1.

In equation (4) the product $\Delta E_y \times y_2$ must be added in the cases that correspond to the ascending portion of the representation log(1+y)-y and subtracted in the cases that correspond to the descending portion of the curve, while in equation (6) this product must be subtracted in the cases that correspond to the ascending portion of the representation $(1+y)-2^y$ and added in the cases that correspond to the descending portion of this curve.

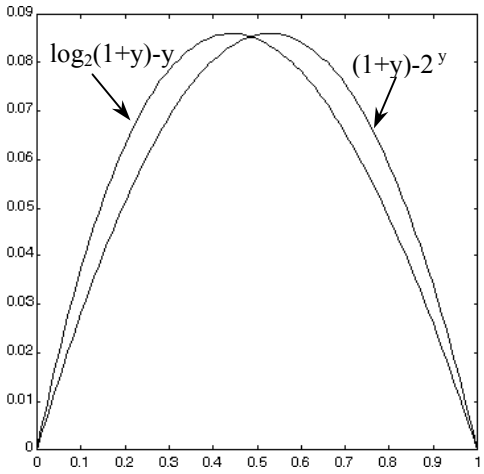

Fig.1 The conversion errors between log(1+y) and y and respectively (1+y) and 2^y.

The circuits for computing logarithms and anti-logarithms allowed the performing of the multiplication and division operations of two operands A and B by means of addition and subtraction operations:

$$A \times B = \text{anti}\log(\log A + \log B), \tag{7}$$

$$A / B = \text{anti}\log(\log A - \log B). \tag{8}$$

Implementing the equations (4), (6), (7) and (8) led to a 6-stage pipeline structure [1], [2], which allowed a 100 MHz clock frequency, in 0.8 μm CMOS technology. Of course, the signal propagation speed through this structure depended on this process too, but, in our paper, we will refer only to the length of the critical path for the carry propagation. The critical stages were, on the one hand, those where the products $\Delta E_y \times y_2$ from (4) and (6) were computed and, on the other hand, the stage where the final addition/subtraction from (4) and ALU operation - addition/subtraction from equations (7)/(8) - were performed. This happened because the speed advantage resulted from the vertical carry propagation in the multiplication area was diminished by the horizontal carry propagation in three non-redundant adders. Furthermore, ALU operated with data of any polarity, which complicated its control logic and led to a further delay.

Later on, the same author presented a new architecture in which the product $E_y \times \Delta E_y$ was calculated not with binary multipliers but with PLA circuits [3], which permitted a saving of area on the chip, maintaining however the same computation speed. In all variants the conversion error was maintained at 3×10^{-7} while the LSB in single-precision format had a weight of 1.19×10^{-7}.

3 New logarithmic unit organization

In order to eliminate the disadvantages mentioned above we propose a new organization of the logarithmic unit that keeps in carry-save form the logarithms of operands A and B and thus, they could be memorized in the latch of a pipelined structure.

Through this approach, the terms $y_{A,}$, $Ey_{A,}$ and y_B, Ey_B which are added to the products $\Delta Ey_A \times y_{2A}$ and $\Delta Ey_B \times y_{2B}$ (see equation 4) will be introduced in the Wallace tree besides the 12 initial partial products of each product.

The problem which is still to be solved is that of the situation where in equation (4), the term $\Delta E_y \times y_2$ is negative and its two's complement conversion, i.e. of all the 12 partial products, would be necessary. This happens starting from the address 907 to 2047 of ROMA and ROMA', a situation which corresponds to the negative slope on the diagram of the function log(1+y)-y shown in Fig.1. We managed to avoid this shortcoming through an artifice, which allows the total elimination of the cases in which the product $\Delta E_y \times y_2$ must be subtracted. As shown in Fig.2, we can write the following equation:

$$Ey_{(n)} - \Delta Ey_{(n)} \times y_2 = Ey_{(n+1)} + \Delta Ey_{(n)} \times (\overline{y_2} + 1) = \\ = Ey_{(n+1)} + \Delta Ey_{(n)} \times \overline{y_2} + \Delta Ey_{(n)}. \quad (9)$$

The implementation of this equation leads to an arrangement of the partial products as they are presented in Fig.3. A generic presentation, with "q_y" for the complemented "p_y" bits of y_2, respectively with "p_d" for the bits of ΔE_Y, was used.

We can obtain the same result of the logarithm computation if we implement the right part of equation (9). Starting from the memory location corresponding to the address 907 of ROMA, instead of memorizing the value $Ey_{(n)}$, $Ey_{(n+1)}$ is memorized, i.e. exactly what should have been found at the next address. In each location a supplementary bit will be memorized, called the control bit, which takes the value 0 for addresses 0…906, and 1 for addresses 907…2047. If this bit is 1, then the generation of partial products will be done with y_2 having the bits reversed, and another pseudo-partial product with a size equal to that of the least significant partial product, having the value $\Delta Ey_{(n)}$, will be added. If the control bit is 0, then the generation of partial products will be done with y_2 unreversed, and the bits "p_d" of the first pseudo-partial product from Fig.3 will all be 0.

As we can see in Fig.4 the Wallace tree for one operand will have as inputs 15 initial pseudo-partial products and it will provide two data words: "sum" and "carry". If in this stage we did the non-redundant addition of these, we would obtain the value of the logarithm of the significand of each operand applied to the input of the two logarithm computation circuits working in parallel. Further on, the two fractional numbers obtained would be concatenated to the exponents of the two operands, in order to obtain the logarithms of the operands. Finally, the two logarithms would be applied to the ALU, in order to be added or subtracted. However, in this case too, we would have two consecutive non-redundant additions, which slows down the process of obtaining the result.

To avoid this situation, we can also consider the two pairs of data words "sum" and "carry" as pseudo-partial products, and thus they are again introduced in a new reduction block. This will provide, in the end, two data words, the final "sum" and "carry", which will be added then, with the help of a fast adder. This final reduction block of the last pseudo-partial products will be included in the ALU as it should act under a control logic to allow the implementation of both addition and subtraction.

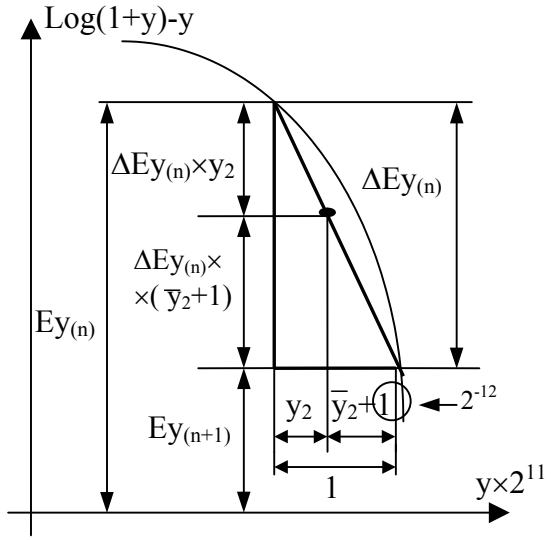

Fig.2 Negative slope segment achieved through linear interpolation between consecutive memorized values E_y.

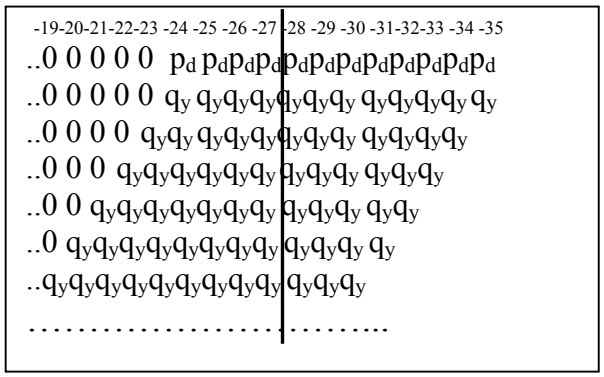

Fig.3 A section through the multiplication area after the implementation of equation (9).

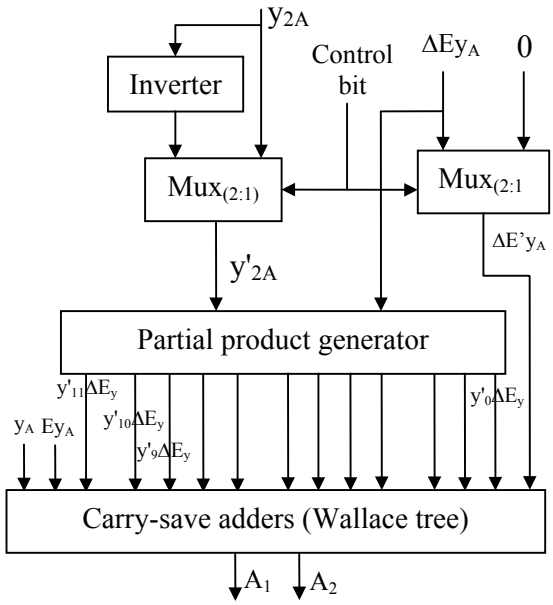

Fig.4 Hardware for computing the logarithm of the operand in carry-save form.

The implementation of this method leads to the generating of a big Wallace tree with two branches, which has 30 pseudo-partial products, 4 levels of reduction blocks and which has its lower end in the ALU. We used, as reduction blocks, 4:2 compressor blocks and one single CSA block of 3:2 full adders on the first level of each branch of the tree.

For the anti-logarithm computation circuit, the procedure applied is the same, the data words "sum" and "carry" being obtained after only 3 levels of compressors. They are then added with a fast non-redundant adder in order to obtain the significand of the final result. In this case too, we will take measures to avoid the subtraction of the term $\Delta Ey \times y_2$, but we also take into consideration the fact that in equation (6) the term Ey must be subtracted too. The product $\Delta Ey \times y_2$ is subtracted in the cases which correspond to the positive slope on the diagram of the function $1+y-2^y$, presented in Fig.1, while it is added in the other cases. As equation (9) can no longer be used, the sum of the two negative terms from equation (8) will be written as follows:

$$\begin{aligned} &-Ey_{(n)} - \Delta Ey_{(n)} \times y_2 = -\left(Ey_{(n)} + \Delta Ey_{(n)} \times y_2\right) = \\ &= -\left[Ey_{(n)} + \Delta Ey_{(n)} - \Delta Ey_{(n)} \times (\overline{y_2} + 1)\right] = \\ &= -Ey_{(n+1)} + \Delta Ey_{(n)} \times \overline{y_2} + \Delta Ey_{(n)}. \end{aligned} \qquad (10)$$

Thus, in the ROMC locations from the address 0 to the address 1082 the two's complement of the quantities $Ey_{(n+1)}$, which should have been found at the next address, as well as the value 1 for the control bit are memorized; from address 1083 to address 2047 (when $\Delta Ey \times y_2$ is positive) the two's complement of the values $Ey_{(n)}$, as well as the value 0 for the control bit will be memorized directly.

4 ALU design

In this approach we do not extract the bias value from the exponents of the two operands. For this we extend the ALU with one bit to the left, while the bias value is extracted or added to the resulting exponent, depending on the performed operation, multiplication or division. The advantage is that ALU will operate with positive numbers. Obviously, the implementation of the square root operation supposes the extraction of the bias value from input data.

When ALU performs a subtraction, the subtractor (the number which is subtracted from the other term) is represented by two reduced pseudo-partial products, whose non-redundant addition is no longer performed. This means that both terms must be converted into two's complement code. To avoid the reconversion from two's complement in sign-magnitude code of the result, in the case in which it is negative, we will use the same method as in [6], only modified for 4 operands.

We note A_1, A_2, respectively B_1, B_2 the four final reduced pseudo-partial products, and $A=A_1+A_2$ represents the subtractend, while $B=B_1+B_2$ represents the subtractor, in the case in which a subtraction is performed. Now we can write the two terms which are simultaneously computed in the adder/subtracter circuit:

$$A - B = (A_1 + A_2 + \overline{B_1} + \overline{B_2} + 1) + 1 , \qquad (11)$$

$$\overline{B - A} = (A_1 + A_2 + \overline{B_1} + \overline{B_2} + 1) + 0 . \qquad (12)$$

Equation (12) can be checked in (13) as follows**:**

$$A - B = -(B - A) = \overline{B - A} + 1 . \qquad (13)$$

When we replace the term "$\overline{B - A}$" with the value given by equation (12), we retrieve equation (11).

As we can see in Fig.5 we maintain the situation of initial carry-in 1, respectively that of initial carry-in 0 at the two adders which work in parallel, as in [6] and a carry-in equal with 1 at the last 4:2 compressor block, in the case of performing a subtraction. Obviously, this carry-in (bit line S_{OP}) will be 0 in the case of addition. The carry-in is applied at the unused input C_{in} of the least significant 4:2 compressor.

Further on, the length of the last block of the tree, included in the ALU, will be supplemented with 9 bits to the left, for the concatenation of the positive exponents (with the bias value of 127 included)

which represent the biased integer parts of the logarithms of the two operands. The concatenation of the exponents will be done at the terms A_1 and B_1, obtaining the final pseudo-partial products $\underline{A_1}$ and $\underline{B_1}$, while in the 9-bit positions of the integer part corresponding to A_2 and B_2 of fractional weight, it will be completed with zeros, obtaining $\underline{A_2}$ and $\underline{B_2}$.

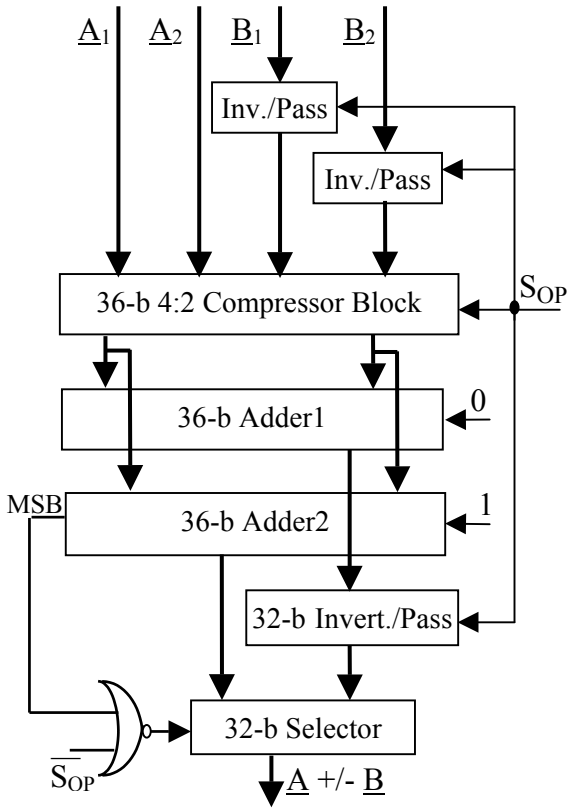

Fig.5 Block diagram of the adder/subtracter circuit.

The three blocks "Inv./Pass" will be transparent when an addition is performed and they will invert the bits of the data from the inputs when a subtraction is performed. The block Selector will select the result from Adder1 unchanged, in the case of addition, and in the case of subtraction it will select the result from Adder2 or from Adder1 inverted, depending on the MSB of Adder2.

Adder1 and Adder2 are 36-bit adders. The four least significant bits from the outputs of the two adders will be lost, and thus, at the output of the circuit, we will regain the single-precision format plus one bit in the MSB position, which avoids the overflow due to the accumulation of bias values in the case of multiplication. The justification of the 36-bit length for the adders and the compressor block will be done in the next section.

In our design and gate level simulation we used 2-level hybrid adders, with seven (1+3×2) 8-b carry look-ahead adders (CLA) plus in the most significant position two 4-b CLA on the 1st level (with input carry 0 and 1 respectively) and a carry select mechanism on the 2nd level.

Following [7] and the assumptions regarding the carry propagation time through different logical gates, we considered the carry propagation time through an inverter or transmission gate like one unit (≈FO4) and subsequently, 2 units for a 2-input NOR and NAND gate, 3.5 units for a XOR gate and 11 units for a 4:2 compressor. The propagation time through the Wallace tree which contains three levels of 4:2 compressor blocks and a pseudo-partial product generator in a branch is 40 units. The gate level simulation of the new ALU in a most unfavorable case led to a carry propagation time of 40 units also, so this pipeline stage doesn't slow down the logarithmic unit computing. In comparison with [1], [2] and [3] where an additional proper addition is included in both these stages, we can say that our variant (keeping the 6-stage pipeline structure) is at least 1.6÷1.7 times faster.

5 Error analysis and correction

Using Matlab analysis to estimate the errors introduced by implementing the algorithm described in [1] for the generation of binary logarithm and anti-logarithm, we had the confirmation of the value of 3×10^{-7} mentioned by Lai as the maximum conversion error. For the further minimization of this error, we suggest a correction of the look-up tables content, which will add correction values on certain address intervals of the ROMA and ROMC. Baring in mind that the error in floating-point single-precision format, i.e. the value of the least significant bit provided by any output of the ROMA or ROMC, is 1.19×10^{-7}, it means that to the calculated values of E_y we can add corrections of one or two LSB, after which they are directly memorized (ROMA), respectively, they are transformed in two's complement and then memorized (ROMC). The correction value "cor" that must be operated in some memory locations depends on the minimum and maximum error in each of the 2048 intervals. It is given by the matlab equation:

Cor=round((max(err)+min(err))/(2*1.19*10^-7)).

For example, in Fig.6.a and b we present the error for the first 40960 values of the logarithm, before and after the correction is done. A more extended representation of the error, for the first 3,686,400 values of the total of 8,388,608 possible ones (in logarithm domain [0, 1)), shows us that the error is kept under 1.5×10^{-7} (Fig.6.c).

As far as the computation of the product $\Delta Ey\times y_2$ is concerned or, more generally, the interpolators

using multipliers that truncate lesser-significant partial-product bits, they have received attention recently in [9]. We can notice from Fig.3, in which a section of the multiplication area is presented, that, if we perform the calculation of the truncation error in the most disadvantageous case, when all bits of a smaller or equal weight with "-28" (the bits of the right side of the vertical line) are equal to 1, we obtain the value 0.6×10^{-7}. This value represents half of the representation error in single-precision format. But, because each bit in the multiplication area represents a logical AND of two bits that can be 0 or 1 with equal probability, the weight of all these bits is statistically 0.15×10^{-7} (i.e. 1LSB/8). According to [9], the removal of all bits from this area, i.e. the elimination of the hard structures from the whole Wallace tree and ALU, can be statistically compensated by adding a 1 in the column of weight –26 next to the 11th partial product. As we observe in Fig.3, we keep 27 bits for the fractional part of the logarithm (or anti-logarithm) that leads to a 36-b structure for the final sum.

6 Conclusions

In this paper we describe a new organization of a logarithmic unit that accepts single-precision floating-point inputs/output and provides a result in 6×40=240FO4. The algorithm of the data format conversions FP-LNS and LNS-FP was improved in comparison with other related works, i.e. it becomes roughly 1.6 times faster and almost twice as accurate. In a very recent work, [10], in which a conventional floating-point approach was used, a double-precision division lasted 453FO4. We can say that our proposal is comparable in terms of speed with this last one but implies less hardware and latency.

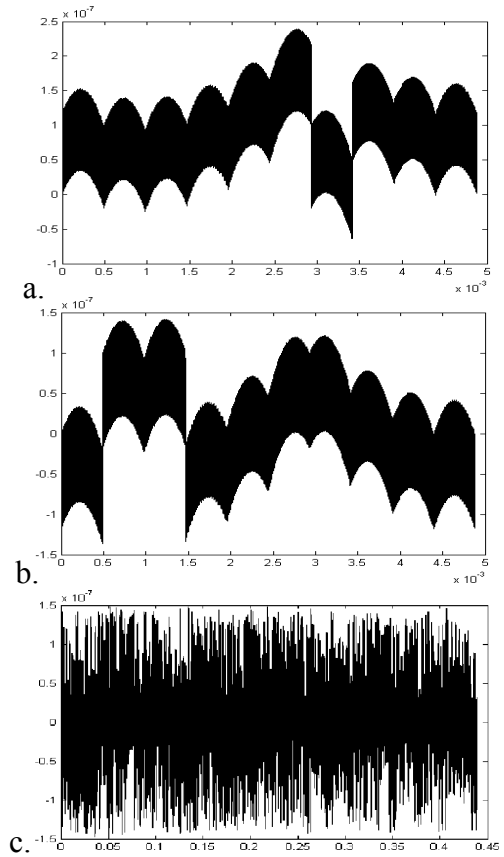

Fig.6 The error for the first 40,960 values of the logarithm (a) before, (b) after the correction and (c) the error for the first 3,686,400 values after the correction.

References:

[1] F. Lai, A 10-ns Hybrid Number System Data Execution Unit for Digital Signal Processing Systems, *IEEE Journal of Solid-State Circuits*, Vol. 26, No. 4, pp. 590-599, Apr. 1991.

[2] F. Lai, C.F.E. Wu, A Hybrid Number System Processor with Geometric and Complex Arithmetic Capabilities, *IEEE Trans. on Comput.* Vol. 40, No.8, pp. 952-961, Aug.1991.

[3] F. Lai, The Efficient Implementation and Analysis of a Hybrid Number System Processor, *IEEE Transactions on Circuits and Systems*, Vol. 46, No. 6 ICSPE5, pp 382-392, June 1993.

[4] D.M. Lewis, 114 MFLOPS LNS Arithmetic Unit for DSP Applications, *IEEE Journal of Solid-State Circ*, vol.30,No.12,pp.1547-1553,Dec.1995.

[5] J.N. Coleman, E.Chester, C. Softley and J.Kadlec Arithmetic on the European Logarithmic Micro-processor, *IEEE Transactions on Computers,* Special Edition on Computer Arithmetic, Vol. 49, No. 7, pp.702-715, July 2000.

[6] H. Fuji & all, A Floating-Point Cell Library and a 100-MFLOPS Image Signal Processor, *IEEE Journal of Solid-State Circuits*, Vol.27, No.7, pp.1080-1088, July 1992.

[7] J.Mori & all, A 10-ns 54×54-b Parallel Structured Full Array Multiplier with 0.5-μm CMOS Tehnology, *IEEE Journal of Solid-State Circuits*, Vol.26, No.4, pp. 600-605, April 1991.

[8] M. Arnold, A Pipelined LNS ALU*, Workshop on VLSI, Orlando, FL*, pp. 155-161, April 19-20, 2001.

[9] E. G. Walters III, M.J. Schulte, Efficient Function Approximation Using Truncated Multipliers and Squarers, *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, pp. 232-239, 2005.

[10] H. Nikmehr, B. Phillips, C.C.Limm, A Fast Radix-4 Floating-Point Divider with Quotient Digit Selection by Comparison Multiples, *The Computer Journal,* Vol. 50 Issue 1, pp.81-92, Jan.2007, Oxford University Press.