Hardware Implementation of Continued Logarithm Arithmetic

Tomáš Brabec
Faculty of Electrical Eng. at Czech Technical University
Dept. of Computer Science and Engineering
Karlovo nám. 13, 121 35 Prague 2, Czech Republic
brabect1@fel.cvut.cz

Abstract

This paper gives details on architecture of an arithmetic unit built on principles of continued logarithms and provides its sample characterization using FPGA technology. Continued logarithms can be used for exact arithmetic, but similarly to other exact/reliable methods they face the instant problem of poor performance. Their naturally binary character, however, gives them a good potential to be realized directly in hardware. We prove feasibility of this approach by constructing a continued logarithm unit and quantifying its possible performance.

1. Introduction

Continued logarithms form a number system capable of representing real numbers. As it relies on error-free rational arithmetic, it is useful for exact arithmetic, which in other words means that computing with continued logarithms produces accurate results with arbitrary precision [7]. This paradigm was first introduced in [5] as a derivative of regular continued fractions and has a better potential for binary arithmetic and hardware implementation. Likewise continued fractions, continued logarithms feature many advantageous properties such as reversibility and incrementality of computation, implicit error bound tracking, good rational approximation and etc. [6].

Unfortunately, continued logarithms also keep the major problem of continued fraction arithmetic, that is the high complexity of arithmetic algorithms. These algorithms generally compute bilinear fractional transformations operating with eight-tuples of big-integer coefficients [5, 9]. This introduces significant computational overhead, which is especially notable for sequential software implementation and which can practically prevail over most advantages of continued representations.

There is, however, a chance to minimize this overhead by exploiting parallelism inside the (bi)linear fractional transformations and use dedicated hardware support. This possibility is further supported by regularity and uniformity of arithmetic algorithms, but its final success relies on efficient use of hardware resources. Even though continued fractions in their regular form are not generally useful in this sense, there were some past attempts for their hardware implementation [9] using a factorization into binary digits. More recently a similar idea appeared in [10].

Our intention here is to investigate the use of continued logarithms as a possible alternative and show its promise for practical realization. For this reason, we cover the most important aspects of continued logarithms in Section 2 – the representation, principles of arithmetic, and some important properties. We also compare continued logarithms with regular continued fractions, showing on some practical experiments that there is no major difference between them. We try to keep that section compact but still complete so as to provide sufficient material for understanding the remaining parts.

The idea of constructing a continued logarithm unit was already introduced in [2], but its description was rather general and we showed only preliminary results. Here we give a more detailed and complete overview of the unit's architecture (Sections 3.1 and 3.2) and present final results of its implementation (Section 3.3). We also provide an analysis in Section 3.4 that quantifies a performance potential of the unit, showing a significant improvement over the software implementation.

There is, however, a limitation behind the continued logarithm representation, which affects applicability of the proposed unit. This problem and its possible solution is finally discussed in Section 4.

2. Continued Logarithm Arithmetic

2.1. Basics

Continued logarithm representation of a real number x is defined by a recursive formula (1), which produces one

Table 1. BCL "Cubic" transformations of fractional coefficients during processing/generating a digit of individual variables.

Digit	x-transformation	y-transformation	z-transformation
–	$\begin{pmatrix} -A & B & -C & D \\ -E & F & -G & H \end{pmatrix}$	$\begin{pmatrix} -A & -B & C & D \\ -E & -F & G & H \end{pmatrix}$	$\begin{pmatrix} -A & -B & -C & -D \\ E & F & G & H \end{pmatrix}$
/	$\begin{pmatrix} B & A & D & C \\ F & E & H & G \end{pmatrix}$	$\begin{pmatrix} C & D & A & B \\ G & H & E & F \end{pmatrix}$	$\begin{pmatrix} E & F & G & H \\ A & B & C & D \end{pmatrix}$
0	$\begin{pmatrix} A+B & A & C+D & C \\ E+F & E & G+H & G \end{pmatrix}$	$\begin{pmatrix} A+C & B+D & A & B \\ E+G & F+H & E & F \end{pmatrix}$	$\begin{pmatrix} E & F & G & H \\ A-E & B-F & C-G & D-H \end{pmatrix}$
1	$\begin{pmatrix} 2A & B & 2C & D \\ 2E & F & 2G & H \end{pmatrix}$	$\begin{pmatrix} 2A & 2B & C & D \\ 2E & 2F & G & H \end{pmatrix}$	$\begin{pmatrix} A & B & C & D \\ 2E & 2F & 2G & 2H \end{pmatrix}$

digit per iteration starting from the most significant one. As (1) consists of four definition cases, there are in total four digits ('$-$', '/', '0' and '1') respectively assigned to individual cases. Thus starting from the top, digit '$-$' corresponds to $-x_i$, '/' to $1/x_i$ and etc. It happens from the nature of (1) that the digits '$-$' and '/' have "sign" character as they may appear only at the beginning of a number representation. This is why we denote this representation a *binary continued logarithm* (BCL), even though there are actually four digits.

$$x_{i+1} = \begin{cases} -x_i & \text{for } x_i \in [-\infty, 0), \\ 1/x_i & \text{for } x_i \in [0, 1), \\ 1/(x_i - 1) & \text{for } x_i \in [1, 2), \\ x_i/2 & \text{for } x_i \in [2, +\infty], \end{cases} \tag{1}$$
$$i = 0, 1, \ldots \text{ and } x_0 = x \in \mathbb{R}.$$

Let us show the use of (1) to find a BCL representation of a rational number $x = -\frac{2}{3}$. We start from $i = 0$ by letting $x_0 = x$ and iteratively continue to higher indices as shown below, finding out that $-\frac{2}{3} = (-/010)_{\text{BCL}}$.

i	x_i	digit	condition	x_{i+1}
0	$-\frac{2}{3}$	'–'	$[-\infty, 0)$	$-(-\frac{2}{3})$
1	$\frac{2}{3}$	'/'	$[0, 1)$	$1/\frac{2}{3}$
2	$\frac{3}{2}$	'0'	$[1, 2)$	$1/(\frac{3}{2} - 1)$
3	2	'1'	$[2, +\infty]$	$2/2$
4	1	'0'	$[1, 2)$	$1/(1 - 1)$
5	∞			

Although it is not explicitly defined, the recursive process may terminate once the value of x_i reaches ∞. By terminating the digit sequence for such i, we avoid an infinite suffix of trailing ones caused by equivalence $\frac{\infty}{2} = \infty$.

Recursive nature of (1) and the employed rational functions imply that the BCL representation has a character of composed linear fractional transformations (LFTs). This character is typical for many similar representations (e.g. [6, 10, 11]), all of which base their arithmetic on a bilinear fractional transformation (BLFT). BLFT is a rational function of the following form

$$\begin{aligned} z(x, y) &= \frac{Axy + By + Cx + D}{Exy + Fy + Gx + H} \\ &= \begin{pmatrix} A & B & C & D \\ E & F & G & H \end{pmatrix} (x, y), \end{aligned} \tag{2}$$

where A through H are integer coefficients and x, y are real-valued arguments, in our case having a form of continued logarithms.

Notice that the BLFT function is general enough to compute all basic arithmetic operations, being the only sufficient operator to constitute whole continued logarithm arithmetic. As explained in [4], BLFT is closed to composition of LFTs and so it naturally matches the character of continued logarithms. This closure actually means that the bilinear form of BLFT stays preserved, no matter what input (x, y) and output (z) digits are being processed.

It thus follows that after processing k digits from x and l from y and after generating i digits of a result z, the state of (2) changes from $z(x, y) = z_0(x_0, y_0)$ to $z_i(x_k, y_l)$, defined by

$$z_i(x_k, y_l) = \begin{pmatrix} A_i^{k,l} & B_i^{k,l} & C_i^{k,l} & D_i^{k,l} \\ E_i^{k,l} & F_i^{k,l} & G_i^{k,l} & E_i^{k,l} \end{pmatrix} (x_k, y_l). \tag{3}$$

This new state is solely characterized by actual values of fractional coefficients and indices i, k, l. Notice that each fractional coefficient has associated all three indices, as its value changes with processing of every digit, input or output. The way these coefficients change depends on a variable and a digit being processed.

The four digits and the three variables account together to twelve possible transformations, which are summarized in Tab. 1. The table shows a new state of a BLFT (2) transformed correspondingly to a given variable and a processed

digit. One can derive these transforms easily [2] by substituting for a given variable from the BCL definition formula (1) into (2). A partial example of their use is shown below, but the particular details are left to Section 3.1.

The last substantial aspect of the continued logarithm arithmetic is its understanding as a second-level arithmetic with the underlying variable-length integer arithmetic being the first level. It means that, despite of x, y, z having BCL representation, the coefficients A to H of (2) are integers represented in a normal, positional number system. Therefore, all operations with these coefficients found in Tab. 1 use conventional integer arithmetic. This understanding is very important and we will return to it in the following sections, where we inspect properties of continued logarithms.

Example: We illustrate the continued logarithm arithmetic on computing a function $z = xy$, where we put for simplicity $x = y = 2 = (10)_{\mathrm{BCL}}$. We use here a special notation where the symbol $\overset{d}{=}_v$ denotes an equivalence attained by processing a digit d from an input variable v. Likewise, the symbol $\overset{d}{\rightarrow}_z$ represents an output z-transformation, during which an output digit d is emitted.

$$z_0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} (2,2) \overset{\text{'1'}}{=}_x \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} (1,2)$$
$$\overset{\text{'1'}}{=}_y \begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} (1,1) \geq 4 \quad \overset{\text{'1'}}{\rightarrow}_z z_1$$
$$z_1 = \begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} (1,1) \geq 2 \quad \overset{\text{'1'}}{\rightarrow}_z z_2$$
$$z_2 = \begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix} (1,1) \overset{\text{'0'}}{=}_x \begin{pmatrix} 4 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \end{pmatrix} (\infty,1)$$
$$\overset{\text{'0'}}{=}_y \begin{pmatrix} 4 & 4 & 4 & 4 \\ 4 & 0 & 0 & 0 \end{pmatrix} (\infty,\infty) = 1 \quad \overset{\text{'0'}}{\rightarrow}_z z_3$$
$$z_3 = \begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 4 & 4 \end{pmatrix} (\infty,\infty) = \infty$$

We can see that the produced sequence '1', '1', '0' corresponds to a BCL representation of a number $z = 4$.

2.2. Properties

The BCL definition formula does not directly evoke that continued logarithms are really continued fractions, but this fact is immediate from formulas (4) and (5) introducing a *canonical continued logarithm* (CCL). Both BCL and CCL representations are equal and they are tied together through powers of two so that every k_i within CCL represents the number of '1' digits between i-th and $(i-1)$-th '0' digit within BCL (see [5] for details).

$$x = s\,(\chi)^e\,, \tag{4}$$

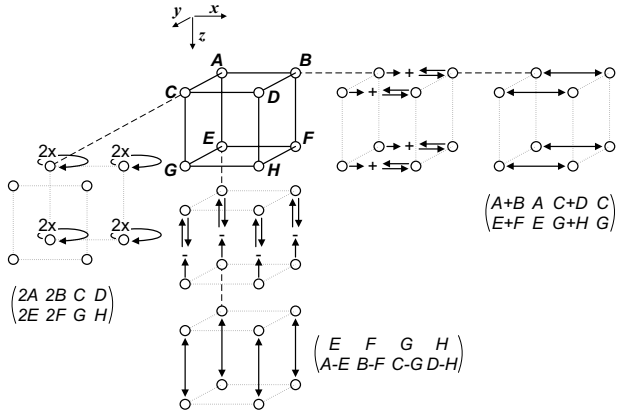


Figure 1. Graphical view of some BCL cubic transformations (see Tab. 1) after mapping fractional coefficients onto a cube.

where s, e and χ are defined as follows:

$$s = \begin{cases} -1 & \text{for } x < 0, \\ +1 & \text{otherwise,} \end{cases}$$
$$e = \begin{cases} -1 & \text{for } |x| < 1, \\ +1 & \text{otherwise,} \end{cases}$$
$$\chi = 2^{k_0} + \cfrac{2^{k_0}}{2^{k_1} + \cfrac{2^{k_1}}{\ddots + \cfrac{2^{k_{n-1}}}{2^{k_n} + \ldots}}} \geq 1. \tag{5}$$

The canonical form is especially useful for studying theoretical properties of continued logarithms, because it is similar to the form of regular continued fractions. The difference is that regular continued fractions use a "linear" approximation of a real number x by an integer $\lfloor x \rfloor$, while continued logarithms use a "logarithmic" approximation by a close power of two, namely $2^{\lfloor \log_2 x \rfloor}$. Similarity of this principle suggests analogy in properties of continued logarithms and regular continued fractions. Let us mention for instance uniqueness of representation, built-in error analysis, reversibility of computation, etc. [6]. Proofs of these properties known for regular continued fractions [8] can be easily adapted to continued logarithms. The logarithmic nature, on the other hand, makes some properties distinct. A particular example is approximation convergence, which is much slower for continued logarithms.

Other interesting properties are due to theoretical principles of continued arithmetic and they are thus common to both considered continued representations. It is especially the uniformity and regularity of BLFT transformations, which is nicely illustrated by mapping the fractional coefficients of (2) onto a cube (see Fig. 1). Individual cube

dimensions correspond to the three variables of a bilinear transformation and at the same time they define directions, in which coefficient transformations happen. For details refer to [9]. The graphical view also reveals high degree of parallelism among individual transformations.

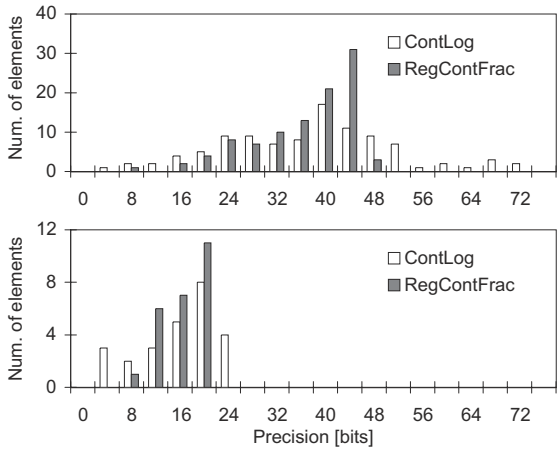
A unique feature of continued logarithms is their use of only powers of two (see (5)), which significantly reduces the complexity of underlying binary arithmetic, i.e. operations performed with fractional coefficients A to H. Where regular continued fractions multiply and divide by general integers, CCL use integer powers of two. That is, continued logarithms reduce all multiplications and divisions to simple shifts while keeping the complexity of addition and subtraction unchanged. In case of the BCL representation, all transformations then use only 1-bit shifts, add/sub and exchange operations (see Tab. 1). Notice that it is namely this property which gives continued logarithms their better potential for direct hardware realization.

2.3. Practical Evaluation

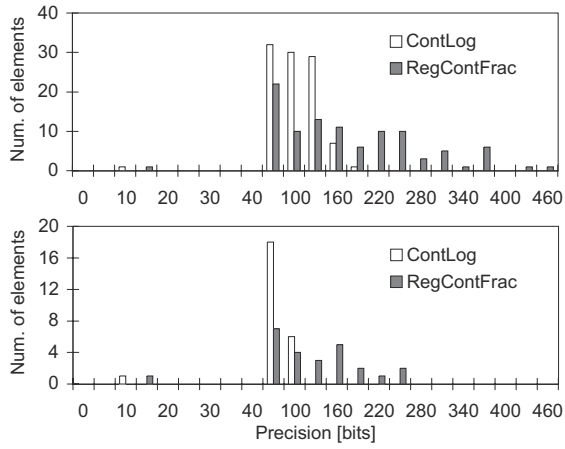
Properties and principles of continued arithmetic makes it promising for implementation of exact arithmetic. Unfortunately, there is a general problem with relatively high complexity of algorithms computing the (bi)linear fractional transformations. This is partially caused by the complexity of arithmetic operations with fractional coefficients, such as multiplication and notably division. As continued logarithms replace these critical operations with relatively simple shifts, performance of their algorithms should increase. On the other hand, continued logarithms have worse approximation convergence than regular continued fractions and it will take them longer to reach a result with a required precision. It is hard to predict, which of these conflicting properties will have greater impact on the overall performance.

To evaluate the effect of continued logarithm properties and to see if they are rather advantageous or not, we compared continued logarithms with regular continued fractions on few practical examples. There were two parameters we observed – performance and precision. Performance is considered here naturally as time to complete the computation and its importance is immediate. The precision parameter represents a peak precision (i.e. bit-length) required for fractional coefficients during the computation. We observed this parameter to get an idea of hardware resource requirements, i.e. register size and data path width, and thus to indicate which of the considered representations would be more effective from hardware design perspective.

For the evaluation purposes, we used Java as an implementation basis and run several simple benchmarks, including random-generated BLFT evaluation, polynomial evaluation and matrix inversion. BLFTs were generated with



(a) Hilbert matrix: 10×10 (top) and 5×5 (bottom)



(b) Random matrix: 10×10 (top) and 5×5 (bottom)

Figure 2. Histogram of fractional coefficients' precision across the computed elements of an inverted matrix.

8-bit signed coefficients and with floating-point arguments in range $[0.0, 1.0]$. Rump's polynomial [12] served as an example of precision sensitive calculation and it was also used to verify accuracy of our implementation. Computing a matrix inverse was finally the most complex benchmark we also used to measure performance. Particular implementations we compared were BCL representation introduced in preceding sections and generic regular continued fractions described in [5].

A complete list of results with detailed comments may be found in [1] and we thus limit here to rephrase only important conclusions. We can generally say that none of both continued representations has an evident advantage over the other one. In the test of random BLFTs, the results were almost equal. Regular continued fractions had average preci-

Table 2. Peak precision of BLFT fractional coefficients detected during computing a result of Rump's polynomial with its precision specified in number of decimal digits.

Result prec. [dec. digits]	1	2	3	4	5	6	7	8	9	10	11	12	13	≥14
Coef. prec. (reg. cont. frac.) [bit]	126	126	126	126	126	126	126	126[1)]	126	126	126	126	126	126
Coef. prec. (BCL) [bit]	170	170	170	170	171	171	171	171	178	178	178	178	345	345[1)]

[1)] At this point an exact solution of a result in a corresponding continued representation has been reached. Note that the solution of Rump's polynomial is a rational number and as such it has a finite representation in both reg. cont. fractions and BCL. To convert this solution into a decimal number system with indicated precision, the continued representation needed to be processed completely (i.e. evaluated to the exact value).

sion per fractional coefficient 51 bits, while continued logarithms required 56 bits. Significant difference showed results of Rump's polynomial evaluation (see Tab. 2). We computed a solution of the polynomial with required precision and observed precision of fractional coefficients during intermediate BLFT calculations. As shown, an approximate solution of the polynomial up to few decimal digits required roughly comparable results (126 and 170 bits per fractional coefficient). However, exact solution in a continued logarithm arithmetic would require 2.5 times more bits per fractional coefficient than regular continued fractions. Finally, the most interesting were results of matrix inversion, partially also because this benchmark was the most complex one. In case of ill-conditioned Hilbert and Pascal matrices, the results came up better for regular continued fractions both in terms of performance and precision of fractional coefficients. On the other hand, inverse of random-generated matrices favored continued logarithms, where performance and especially the precision was remarkably lower. Fig. 2 illustrates these results and displays histograms of fractional coefficient's precision gathered from all elements of an inverse matrix. For instance, we can see that a peak precision for a 10×10 Hilbert matrix was 46 bits for regular continued fractions and 72 bits for continued logarithms. Conversely, a random matrix of the same dimension would require 450 bits and 161 bits, respectively. For comparison of performance refer to [1].

Even though our evaluation had a limited scope, it does not seem so far that there is a major difference between continued logarithms and regular continued fractions. However, continued logarithms theoretically keep their advantage of significantly easier implementation in hardware. With this observation, we had enough justification why to use continued logarithms instead of commonly preferred continued fractions.

3. Arithmetic Unit

In [2] we presented preliminary results of designing an arithmetic unit using continued logarithm representation. In this paper we are about to describe the details of unit's architecture and present the final results of its implementation. We also evaluate performance improvement that such unit could eventually offer.

3.1. Principles

The process of computing BLFT (2) is based on principles of digit-serial on-line computation and interval-like function evaluation. The former principle defines the way of generating output digits serially by having only a partial knowledge about exact values of input arguments. Whenever this partial knowledge is insufficient for producing another output digit, there is possibility to refine it by processing more input digits from one or both arguments. The partial knowledge is specified by intervals restricting the set of possible values that input arguments may gain. With these considerations, it is an immediate consequence that at a given state of the computation we can only determine a range of BLFT functional values and not just a single point. However, this is acceptable as long as the range interval can meet output conditions (1) for continued logarithm digits. Since the exact value of the result is a part of that range, it will certainly meet the output conditions as well.

It follows that the whole process is demand-driven – it takes the actual state (3) of the computation and determines the range of z_i using information on domains of input variables x_k and y_l. Let us denote $\mathbb{Z}_i$ the range of z_i and similarly $\mathbb{X}_k$ and $\mathbb{Y}_l$ the domains of x_k and y_l. If $\mathbb{Z}_i$ cannot satisfy any output condition, input domains are refined by processing more input digits and taking input transformations on fractional coefficients. Thanks to the character of BLFT transformation, $\mathbb{Z}_i$ gets refined too. Once $\mathbb{Z}_i$ is tight enough for output, it is possible to do a z-transformation and produce a corresponding digit. Further digits for $\mathbb{Z}_{i+1}$, $\mathbb{Z}_{i+2}$ and etc. are produced as long as it is possible; after then another input digits are requested and the whole process repeats. This is formally sketched in the algorithm at the end of the paper.

Yet a bit unclear remains determining the range $\mathbb{Z}_i$. In general case, it is a hard and complex task, but in case of well-defined BLFT function, it is possible to take advantage of restrictions on domains $\mathbb{X}_k$ and $\mathbb{Y}_l$. By "well-definedness" we mean that the function is continuous and

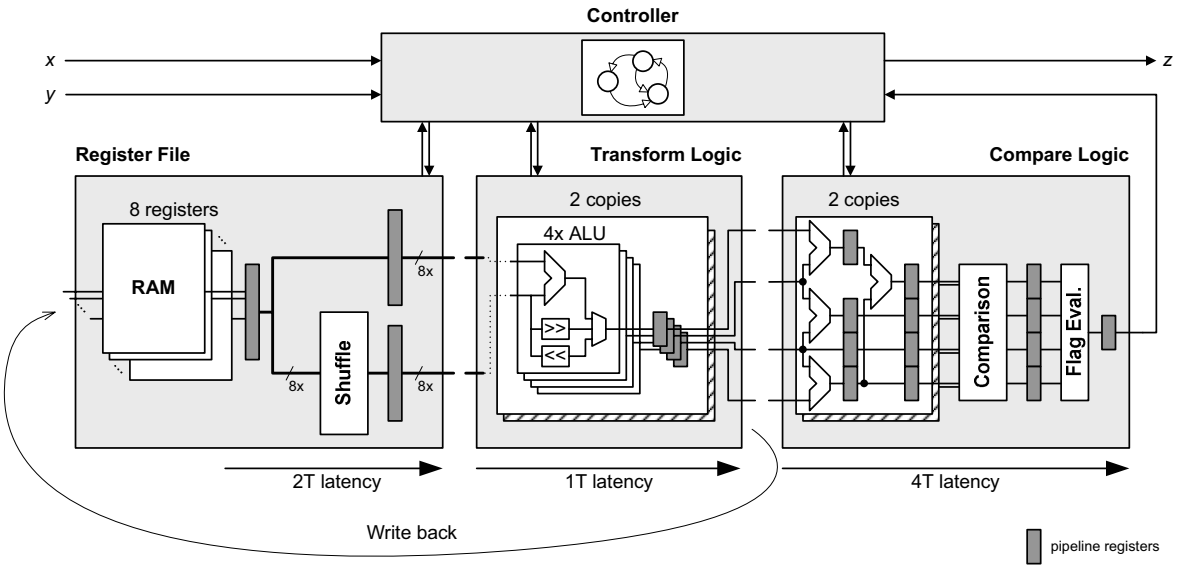


Figure 3. Simplified architecture of the continued logarithm unit.

monotone on its assumed domain. It follows from the definition of BCL (1) that this domain is implicitly restricted to $\mathbb{X}_k \times \mathbb{Y}_l \subseteq [1,\infty]^2$ as it holds $x_k, y_l \geq 1$ for $k, l \geq 2$. Let us define the following *partial sums* among fractional coefficients of BLFT:

$$\begin{aligned} N_0 &= A, & N_2 &= A + C, \\ N_1 &= A + B, & N_3 &= A + B + C + D, \\ D_0 &= E, & D_2 &= E + G, \\ D_1 &= E + F, & D_3 &= E + F + G + H. \end{aligned} \tag{6}$$

One can easily verify that using the implicit restriction to $[1,\infty]^2$ the following conditions are sufficient to make a BLFT well-defined:

$$\begin{aligned} &\operatorname{sgn}(N_0) = \operatorname{sgn}(N_1) = \operatorname{sgn}(N_2) = \operatorname{sgn}(N_3), \\ &\operatorname{sgn}(D_0) = \operatorname{sgn}(D_1) = \operatorname{sgn}(D_2) = \operatorname{sgn}(D_3) \neq 0. \end{aligned} \tag{7}$$

The partial sums actually represent the value of z_i's numerator (N_j) and denominator (D_j) in one of the corners of $[1,\infty]^2$ – e.g. $z_i(\infty, 1) = (A + C)/(E + G) = N_2/D_2$. Therefore, if the "well-definedness" conditions (7) are met, we also check what output conditions (1) the fractions N_j/D_j, $j = 0, \ldots, 3$, satisfy. If they all satisfy the same condition, then z_i's range certainly does too because its BLFT is well-defined. Output conditions may be checked using these equivalences:

$$\begin{aligned} \mathbb{Z}_i \subset [-\infty, 0) &\quad\Leftrightarrow\quad \operatorname{sgn}(N_0) = -\operatorname{sgn}(D_0), \\ \mathbb{Z}_i \subset [0, 1) &\quad\Leftrightarrow\quad 0 \leq N_j < D_j \text{ for } \forall j, \\ \mathbb{Z}_i \subset [1, 2) &\quad\Leftrightarrow\quad D_j \leq N_j < 2D_j \text{ for } \forall j, \\ \mathbb{Z}_i \subset [2, +\infty] &\quad\Leftrightarrow\quad D_j \leq N_j \text{ for } \forall j. \end{aligned} \tag{8}$$

Let us finally note that the introduced principle is deterministic, but it is finite only if input arguments are rational numbers. It is because rational numbers have finite continued logarithm representation and can be thus processed in a finite time, and because their BLFT will produce a rational number too. In general case of real-valued arguments, problems of non-computability may arise because of computing with infinite continued logarithms of irrational numbers. However, we leave this well-known limitation [9] for discussion in Section 4.

3.2. Architecture

The unit's operation follows the process of computing BLFT discussed previously and its architecture thus naturally reflects this character (see Fig. 3). The parts depicted in the figure correspond to individual steps of the computing process, except for added controller that manages the overall operation.

The *controller* takes digits from input arguments and emits computed digits of the result. The core functionality, i.e. the data intensive processing of fractional coefficients, is left to the rest of the unit representing the data path. The controller coordinates the data processing and decides on proceeding of the computation e.g. by following the algorithm discussed in the previous section. For this purpose it relies on status signals of data path modules, and especially those of comparison logic, which provides information on possibility of output.

Values of fractional coefficients are stored in the *register file*, which is composed of eight dual-port RAMs, each one

for storing a single fractional coefficient. The capacity of a RAM module thus defines the maximum precision of a fractional coefficient. The register file also provides the functionality to prepare its output data for further processing, so that computing of cubic transformations can be handled more easily. This is the purpose of the *shuffle block* that can permute register file outputs with respect to a "dimension" (x, y, z) of a selected transformation. If we call the output of the shuffle block *permuted*, then e.g. for y-transformation there will be an eight-tuple (A, B, C, D, E, F, G, H) on the direct output of the register file and the corresponding permuted output will be (B, A, D, C, F, E, H, G).

Both direct and permuted outputs of the register file are further processed by the *transform logic* module, which computes the individual cubic transformations according to a selected dimension. The module is composed of eight ALUs so that i-th ALU takes i-th direct and i-th permuted output. ALUs can be grouped by four, one group for numerator and one for denominator of a computed BLFT. The ALU itself is quite simple and its functionality covers all operations occurring within the cubic transformations (see Tab. 1). These operations include 1-bit shift left/right, addition, subtraction and negation. Important is also the ability of "no operation" used only to propagate permuted outputs so as to realize coefficients exchange such as in case of y-transformation for '/' digit.

New values of fractional coefficients produced by the transform logic are stored back to the register file. At the same time, these values are passed to the module of *compare logic*, which computes the partial sums N_j, D_j defined in (6) and evaluates well-definedness (7) and the output conditions (8). Final decision on taking the z-transformation and producing an output digit is left to the controller.

Although the arithmetic unit operates equally to its sequential software equivalent, the main difference is in employing algorithm-level and "operation"-level parallelism. The algorithm-level parallelism corresponds to parallel computing of the cubic transformations with independent fractional coefficients (Fig. 1). The "operation"-level parallelism means that thanks to the pipelining it is possible to overlap individual steps of computation, such as computing new fractional coefficients together with their partial sums and output evaluation.

Advantage of pipelining and operation overlapping is further supported by unit's multi-cycle character, which means that fractional coefficients are processed by parts (e.g. 32-bit words) rather than at once with full precision. Despite taking more clock cycles to complete a single transformation, the multi-cycle character generally offers better resource utilization and by proper choice of a word size one can leverage different speed/area trade-offs. E.g. a 32-bit unit must iterate sixteen times to compute a transformation of 512-bit fractional coefficients, while a 64-bit unit needs only eight cycles – but the 32-bit unit will most likely be smaller (see Section 3.3 for actual figures). Nonetheless, the primary motivation for introducing the multi-cycle character was to handle extreme precision of thousands bits, which can possibly occur for some ill-conditioned problems, but is rather occasional.

Table 3. Experimental results for different unit's configurations.

Word Size N	**Word Addr.** k	**Complete Precision**	**Freq.**[1)]	**Area**[1)]	**Area**[1,2)]
[bit]	[bit]	[bit]	[MHz]	[Slices]	[%]
8	4	128	218	538	3
16	4	256	205	893	6
32	4	512	180	1 676	12
64	4	1 024	130	3 303	24
32	5	1 024	175	2 113	15
16	6	1 024	200	1 313	9
8	7	1 024	205	1 063	7

[1)] Implementation results obtained from Xilinx ISE 8.1.3 tool.

[2)] The area percentage corresponds to utilization of Xilinx XC2VP30 FPGA device with 13 696 slices.

3.3. Implementation Details

The architecture was described in VHDL hardware description language, with a possibility to configure the word size N (i.e. data path width) and the number of cycles per cubic transformation $M = 2^k$. These parameters account for a total unit's precision of $N \times 2^k$ bits (per fractional coefficient) and their choice also affects area utilization and timing parameters. Such configurability is thus important for evaluating the area/performance trade-offs of a physically realized design.

For implementation we used FPGA (Field Programmable Gate Array) technology because it goes hand-in-hand with the required design-time configurability and gives possibility for future run-time reconfiguration. Important advantage of this technology is immediate availability of silicon devices and thus its potential for rapid prototyping. For purposes of our evaluation, we used Xilinx FPGA devices and in particular Virtex-II Pro family.

Implementation results for few typical configurations of the arithmetic unit are listed in Tab. 3. These results exhibit linear increase of occupied area with respect to data-path width N or address width k, which is a sign of good scalability. Simultaneously with the increasing word size, the frequency partially dropped down as a result of lengthening the critical path going through adders and relating logic. Nonetheless, all configurations were still able to operate at high frequencies. Notice that despite the drop down, using

higher word size still pays off if the associated area overhead is acceptable. Just for convenience we also provide implementation results (Tab. 4) of some highly optimized floating-point units so that one has a fair comparison between the size of a floating-point and the continued logarithm unit.

3.4. Performance Evaluation

Using the presented implementation results, one can estimate[1] the acceleration potential that the proposed arithmetic unit offers in comparison with its software equivalent. Although we base our assessment on very simplified and raw estimates of ideal SW and HW performance, its results will give us some idea of what we may expect when integrating the arithmetic unit as an accelerator in a computer system. For this evaluation we compare a 32-bit variant of the unit with a hypothetical 32-bit RISC-like processor. This means that the fractional coefficients are processed by 32-bit words in both cases.

In case of software implementation, we omit any control overhead and restrict the whole data processing only to major operations within the cubic transformations and $\mathbb{Z}_i$ range evaluation. After analysis of these major operations, we found out that any cubic transformation requires at least four operations and $\mathbb{Z}_i$ range evaluation at least sixteen operations [1]. If we assume each operation to take a single cycle, which is quite realistic, then a minimum time for complete computation:

$$\begin{aligned} T_{\text{min SW}} &= M(K_x + K_y)T_{x/y\text{-transf}} \\ &\quad + MK_z(T_{z\text{-transf}} + T_{\mathbb{Z}\text{ eval}}) \\ &\geq \big(4M(K_x + K_y) + MK_z(4 + 16)\big)T \\ &= \big(4M(K_x + K_y) + 20MK_z\big)T, \end{aligned}$$

where M is the number of words per fractional coefficient and K_x, K_y, K_z are the numbers of digits per continued logarithm representation of individual variables. T is a clock period.

Performance of the arithmetic unit is easy to estimate from its architecture on Fig.3. We see that the latency of individual parts is $L_{\text{Reg.File}} = 2T$, $L_{\text{Transf.Log.}} = T$ and $L_{\text{Cmp.Log.}} = 4T$. If we assume input symbols from both arguments to be instantly available, the arithmetic unit can operate at full bandwidth alternating x- and y-transformations, interleaved by a z-transformation whenever possible. Since the evaluation of output possibility is overlapped with cubic transformations, its overhead projects into only a small constant additive to the complete latency of the arithmetic unit. Performance of the unit is thus:

$$\begin{aligned} T_{\text{HW}} &= (K_x + K_y + K_z)(M + L_{\text{Reg.File}} + L_{\text{Transf.Log.}} \\ &\quad + L_{\text{Cmp.Log.}}) \\ &= (K_x + K_y + K_z)(M + 2 + 1 + 4)T \\ &= (K_x + K_y + K_z)(M + 7)T \end{aligned}$$

Advantage of the hardware implementation is immediate from $T_{\text{min SW}}$ versus T_{HW}. Let us put for simplicity $K_x = K_y = K_z = K$, and we get $T_{\text{min SW}} = 28MK$ for software and $T_{\text{HW}} = 3MK + 21K$ for hardware. Letting $M = 16$ for the total precision of 512 bits, we end up with

$$T_{\text{min SW}} = 448K \quad \text{and} \quad T_{\text{HW}} = 69K,$$

i.e. the arithmetic unit can at least 6.5 times outperform a processor at the same frequency. According to Tab. 3, we may equivalently asset the unit to perform as well as a processor running at 6.5×180 MHz, i.e. above GHz border.

Table 4. Implementation results for some commercially available floating-point units.

Vendor	Architecture	Area [1)] [Slices]	Freq. [MHz]
Gaisler Research [3]	Virtex-II	8 000 (DP)	65
Xilinx [13]	Virtex 4	1 200 (SP)	137.5 ÷ 170

[1)] DP/SP stands for double/single precision.

4. Conclusions

We showed that continued logarithms represent a feasible alternative to commonly preferred regular continued fractions and that principles of their arithmetic can be borrowed for efficient hardware realization. The analysis of the presented results demonstrated that the specifically designed arithmetic unit offers considerably higher performance, i.e. 6.5 times higher, than optimal sequential software implementation. Further improvement is possible by running more such units in parallel. The multi-unit architecture is certainly possible because of fine area utilization, which is in average comparable to that of conventional floating-point units.

There, however, remains the problem of real number computability, which we identified in Section 3.1 and which makes the proposed unit useful only for exact rational arithmetic. Consider a case of an irrational square root with an integer square – continued logarithms fail to compute this square as they cannot produce a finite representation of the integer from an infinite representation of an irrational number. This limitation is a result of the on-line arithmetic character combined with the representation uniqueness [9].

[1]We use this estimation as we did not have real-time performance results at the time of writing.

Nonetheless, there is a chance to find a redundant extension and solve this problem. One possibility is a detection of non-computable cases and their speculative resolution, which can take advantage of the rational arithmetic unit proposed here. The unit would just require some minor extensions of the data path and a change of control mechanism to take care of the speculation. With a relatively low additional overhead, we could finally introduce a continued logarithm unit capable of exact real arithmetic and having implementation parameters similar to those presented here.

Acknowledgment

This work was supported by Czech Technical University under the grant no. CTU0609213. The author would also like to thank anonymous referees for their helpful comments.

References

[1] T. Brabec. On Exact Real Hardware with Specialization to Continued Methods. PhD Proposal, 2006. http://service.felk.cvut.cz/anc/brabect1/pub/phdprop06.pdf.

[2] T. Brabec and R. Lórencz. Arithmetic Unit Based on Continued Fractions. In *Proceedings of the 7th International Scientific Conference on Electronic Computers and Informatics ECI 2006*. http://service.felk.cvut.cz/anc/brabect1/pub/eci06.pdf.

[3] E. Catovic. GRFPU - High Performance IEEE-754 Floating-Point Unit, 2004. http://www.gaisler.com/doc/grfpu_wp.pdf.

[4] P. Flajolet, B. Vallée, and I. Vardi. Continued Fractions from Euclid to The Present Day, 2000.

[5] R. W. Gosper. Continued Fraction Arithmetic. Unpublished manuscript, 1977.

[6] R. W. Gosper, M. Beeler, and R. Schroeppel. HAKMEM. Technical report, Cambridge, MA, USA, 1972. Item 101.

[7] P. Gowland and D. Lester. A Survey of Exact Arithmetic Implementations. In J. Blanck, V. Brattka, and P. Hertling, editors, *CCA*, volume 2064 of *Lecture Notes in Computer Science*, pages 30–47. Springer, 2000.

[8] A. Y. Khinchin. *Continued Fractions, 3rd ed.* The University of Chicago Press, 1964.

[9] P. Kornerup and D. W. Matula. An Algorithm for Redundant Binary Bit-Pipelined Rational Arithmetic. *IEEE Trans. Computers*, 39(8):1106–1115, 1990.

[10] M. Niqui. Exact Arithmetic on the Stern-Brocot Tree. Technical Report NIII-R0325, Nijmeegs Instituut voor Informatica en Informateikunde, 2003.

[11] P. J. Potts. *Exact Real Arithmetic using Möbius Transformations*. PhD Thesis, Imperial College, London, 1998.

[12] S. M. Rump. Algorithms for Verified Inclusions – Theory and Practice. *Reliability in Computing: The Role of Interval Methods in Scientific Computing*, pages 109–126, 1988.

[13] Xilinx, Inc. APU Floating-Point Unit v2.1, 2006.

Algorithm: Computing BCL of BLFT

input:	`coeffs` ... coefficients of BLFT, see (2)
	`x` ... *null*-terminated BCL of x variable
	`y` ... *null*-terminated BCL of y variable
output:	`z` ... *null*-terminated BCL of z variable
	`i` ... number of produced BCL digits

```
i = k = l = 0;
x_k = x;    y_l = y;    z_i = null;
Z_i = ∅;    X_k = Y_l = [−∞, +∞];

while (Z_i ≠ {+∞}) {
   Z_i = rangeBLFT(coeffs, X_k, Y_l);
         /* see Eq. (8) and relating */

   switch (Z_i) {
      case Z_i ⊆ [2, +∞] :  digit = '1'; break;
      case Z_i ⊆ [1, 2) :   digit = '0'; break;
      case Z_i ⊆ [0, 1) :   digit = '/'; break;
      case Z_i ⊆ [−∞, 0) :  digit = ' − '; break;
      default :  digit = null;
   }

   if (digit ≠ null) {
      coeffs = transform(coeffs, DIRECTION_z, digit);
         /* see Tab. 1 */
      z_i = digit ∘ z_{i+1};    /* concat.  */
      i++;
      z_i = null;
      continue;
   }

   /* Function select() selects an input arg.
    * for input transformation.  Selection may
    * be random or deterministic - e.g.
    * strict alternation */
   if (select(x_k, y_l) == x_k) {
      digit = getMSD(x_k);
      if (digit == null) continue;
      x_{k+1} = discardMSD(x_k);
         /* rest of BCL repres.  except
          * most significant digit (MSD) */
      coeffs = transform(coeffs, DIRECTION_x, digit);
         /* see Tab. 1 */
      k++;
      X_k = rangeBCL(x_k);
         /* uses implicit restriction to
          * subset of [1,∞]      */
   } else {
      digit = getMSD(y_l);
      if (digit == null) continue;
      y_{l+1} = discardMSD(y_l);
         /* rest of BCL repres.  except MSD */
      coeffs = transform(coeffs, DIRECTION_y, digit);
         /* see Tab. 1 */
      l++;
      Y_l = rangeBCL(y_l);
         /* implicit restriction (see X_k) */
   }
}

z = z_0;
```