Accuracy and Performance Trade-offs of Logarithmic Number Units in Multi-Core Clusters

Michael Schaffner*, Michael Gautschi*, Frank K. Gürkaynak*, Luca Benini*†
*ETH Zürich, 8092 Zürich, Switzerland, †Università di Bologna, Italy

***Abstract*—When compared to traditional floating point (FP) number representation, logarithmic number systems (LNS) have superior performance when evaluating complex functions, since multiplications and divisions can be calculated with ease in the logarithmic domain. However, additions and subtractions become costly nonlinear operations. Efficient LNS units (LNUs) implementing ADD/SUB operations in hardware rely on interpolation techniques to save area. Even the most advanced LNUs are still larger than standard single-precision FPUs – which renders them impractical for most general purpose processors. In this paper, we show that in a multi-core setting, when shared among several processor cores, LNUs become a very attractive solution. We present a methodology to generate LNUs with various error bounds and perform a design space exploration with different parameterizations. We show that already small precision relaxations in the order of a few units in the last place (ulp) reduce the LNU area significantly. Using examples from several signal processing domains, we demonstrate that shared approximate LNUs can outperform their standard FP counterpart on average by 2.14x in speed and 1.92x in energy-efficiency, with insignificant degradation of the output quality.**

***Keywords*-Logarithmic Number System (LNS), Shared Floating Point Unit (FPU), Approximation, Multi-core, RISC, ASIC, VLSI**

I. Introduction

The Logarithmic Number System (LNS) allows low-latency evaluation of computationally intensive non-linear functions kernels and has over the years attracted significant attention as a possible replacement of the conventional single-precision floating point number representation [1–9]. This is not only relevant for high-performance computing, but also increasingly needed for low-power, low-cost embedded applications where the demand on intensive signal-processing capabilities continues to grow on a regular basis. However, the drawback of LNS is that additions and subtractions become nonlinear operations and when implemented in hardware have to be approximated accordingly with a dedicated LNS unit (LNU).

Recent developments [9–11] have shown that these functions can be efficiently approximated using piecewise polynomial interpolation, combined with suitable function decompositions (also called *cotransformations*) in order to handle the singularity region present in LNS subtractions. LNS units that reach the numerical accuracy equivalent to single precision FP [9], [11] and double precision FP [10] have been demonstrated. But despite these developments, LNS ADD/SUB operators are still much larger than their standard FP counterparts, which makes it difficult to motivate their use as FPU replacement in general purpose processors.

When viewed in a multi-core setting, the area overhead of the LNS units change. In such a setting, one LNS can be effectively shared between several cores, as the percentage of ADD/SUB instructions in even the most intensive computations usually remain below 30%. This arrangement is more efficient than sharing standard FP units, as the MUL and DIV instructions in the logarithmic domain can be performed within the integer cores, allowing M DIV and MUL operations to be performed in parallel when only one LNU is shared among M cores.

Another remedy is to relax precision requirements since certain applications, such as image or audio processing, are error tolerant to some degree and usually do not require the equivalent accuracy of single precision FP. Using approximate computing techniques on the architecture and circuit levels [12–14], significant area and energy savings have been reported with only modest quality impact.

In this paper, we combine these ideas of LNU sharing and approximate computing in order to reduce the LNU area, latency and improve its utilization and overall energy-efficiency. Using application kernels from several signal processing domains we show that shared approximate LNUs can outperform standard single precision FPUs in several applications by an average factor of 2.14× in terms of speed and 1.92× in terms of energy-efficiency. Also, a precision relaxation of just a few units in the last place (ulp) already leads to significant reductions of the LNU area – with insignificant degradation of the output quality when applied to several image and audio processing kernels. In particular, this paper makes the following contributions:

- We develop a methodology to generate accurate and approximate LNUs capable of natively evaluating LNS ADD/SUB, typecasts (Integer to float – I2F, Float to Integer – F2I), and logarithms and exponentials with base 2 (LOG2, EXP2),
- We provide a design space exploration of LNUs in the accuracy range between half- and single precision FP,
- We integrate shared LNUs with different parameters into a multi-core RISC cluster and show comprehensive results of benchmark applications from different signal processing domains.

Sections II and III give a short introduction into LNS and related work. The architecture and generator framework is described in Section V. Core integration aspects are covered in Section VI, and the results are finally presented in Section VII.

II. Related Work

LNS has been proposed as a replacement for standard fixed-point and floating-point arithmetic [1], [2] dating back to the 1970's. As will be described in section III, the main challenge in implementing a hardware unit to perform operations in the logarithmic domain is realization of additions and subtractions which turn into non-linear operations that need to be approximated. Finding efficient methods to approximate ADD/SUB functions has driven research in the LNS domain. In early papers, implementation of LNUs with accuracy higher than 12 bits in hardware was considered infeasible due to exponentially increasing lookup-table (LUT) sizes needed for approximations. Since then, several improved implementations have been proposed. In the low-precision floating point calculation domain, with bit-widths lower than 16 bits, so-called multi-partite table [15] and high-order table based methods (HOTBM) [16] have been shown to be effective approximation methods for LNS operations [17]. LNS based operations have been used to

1063-6889/16 $31.00 © 2016 IEEE
DOI 10.1109/ARITH.2016.10

replace fixed-point operations in several applications such as QR decomposition [18], embedded model predictive control processors [19] and low power digital filtering with LNS [20]. LNS numbers have also been extended to be used for complex numbers [21] and quaternions [22].

Coleman, et al. [5] introduced the concept of a *cotransformation* to alleviate approximation difficulties related to the subtraction operation where the difference between the two operands is very small. As explained later in subsection III-D, such cotransformations are basically analytical decompositions of the problematic function to be approximated, and allow to implement the same functionality with significantly smaller coefficient table sizes. Following the example of Coleman, et al., several different cotransformation variations have been presented in [3], [4], [6–9], [11]. In a paper by [23], a solution is presented that tries to combine the advantages of both standard FP and LNS representations. The main drawback in this hybrid approach is the cost of typecasts, which are also non-linear operations, between the representations. Generators for LNS operators on FPGAs have been proposed in [10], [17], [24], [25]. Very competitive operators can be generated with the framework presented by Fu, et. al [10] which is based on the cotransformation developed by [3] and minimax polynomials [26].

Complete LNUs for ASIC processors with accuracy equivalent to IEEE single-precision FP have been presented in [8], [9], [11], [27]. Coleman, et al. [8] describe the *European Logarithmic Microprocessor* (ELM), the first microprocessor featuring an LNU. Their design combines a custom interpolation scheme with the cotransformation developed by [5], and amounts to an area of 0.906 mm^2 using a 180 nm technology with an equivalent complexity of ~97 kGE. Ismail, et al. [9] improve the ELM design and propose an LNU with lookup tables small enough to be implemented without ROMs which amounts to 0.589 mm^2 in 180 nm technology (~63 kGE). Both LNU designs are able to execute only basic LNS ADD/SUB instructions and do not have additional functionality for casts. A compact ASIC design with accuracy equivalent to single precision FP was reported by [11], [27] and has an area of 0.058 mm^2 (40 kGE) in 65 nm while at the same time supporting more operations (ADD/SUB, I2F, F2I, LOG2 and EXP2).

In this paper, we improve on above listed work on LNUs for ASIC processors by designing a compact LNU based on the cotransformation by [3] which is able to execute LNS ADD/SUB, as well as I2F, F2I, LOG2 and EXP2 instructions. A similar framework as described by Fu, et. al [10], [24] is established in order to generate exact ($\leq$ 1ulp accuracy) and approximate ($\geq$1 ulp accuracy) LNUs for different bit-widths ranging from half to single precision FP. For an accuracy equivalent to FP single precision, this design only amounts to 0.039 mm^2 (26.8 kGE) in 65 nm – which is smaller than all other state-of-the-art ASIC designs. Further, we analyze these LNUs in a shared multi-core setting and using application kernels from different signal processing domains we show significant improvements in speedup and energy-efficiency for exact and approximate designs.

III. Preliminaries

A. LNS Number Representation and Format

Standard FP number systems represent a real number a as

$$a = (-1)^s \cdot m_{\text{frac}} \cdot 2^{l_{\text{exp}}} \tag{1}$$

where s is the sign, m_{frac} the mantissa and l_{exp} the exponent. In LNS, real numbers are represented similarly, but without using a mantissa. I.e., the number is only represented by an exponent l_{exp} which now has a fractional part:

$$a = (-1)^s \cdot 2^{l_{\text{exp}}}. \tag{2}$$

The encoding used in this work is parametrized with the number of integer w_{int} and fraction w_{frac} bits in the exponent. In this case, the

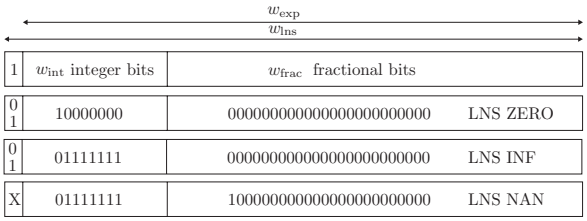

Figure 1. Encoding of the LNS numbers used in this work.

exponent is an unbiased two's complement number and its width is denoted as $w_{\text{exp}} = w_{\text{int}} + w_{\text{frac}}$. The bit-width of the complete number including the sign bit is denoted as $w_{\text{lns}} = w_{\text{exp}} + 1$. For $w_{\text{int}} = 8$ and $w_{\text{frac}} = 23$, the encoding is aligned with the IEEE 754 32-bit single-precision format. Similar to the IEEE 754 standard, special values such as zeros (ZERO), infinities (INF) and not a number (NAN) are encoded using special bit patterns. ZEROs are represented by setting the exponent to the smallest 2's complement value. INFs are represented by setting the integer part of the exponent to the maximum value and the rest of the mantissa to 0. NANs are encoded similarly, but with the highest fraction bit in the exponent set to 1 as illustrated in Figure 1 for $w_{\text{int}} = 8$ and $w_{\text{frac}} = 23$.

B. Arithmetic Operations in LNS

Certain operations can be implemented very efficiently when working with LNS. For example, multiplications, divisions, and square-roots can be calculated using a single addition, subtraction or bitshift, respectively.

$$a \cdot b = (-1)^{s_a+s_b} \cdot 2^{l_a+l_b} \tag{3}$$

$$a/b = (-1)^{s_a+s_b} \cdot 2^{l_a-l_b} \tag{4}$$

$$\sqrt{|a|} = \left(2^{l_a}\right)^{0.5} = 2^{0.5 \cdot l_a} \tag{5}$$

This is an important advantage because numbers represented in this format can be efficiently calculated by slightly modified integer ALUs and result in much shorter latencies than the equivalent FP implementations. However, these simplifications come at the cost of more complex additions and subtractions which become nonlinear operations in LNS and have to be calculated accordingly:

$$a \pm b = c, \tag{6}$$

$$l_c = \max(l_a, l_b) + \log_2(1 \pm 2^{-|l_a - l_b|}). \tag{7}$$

Using the absolute difference $r = |l_a - l_b|$, the two nonlinear functions for addition and subtraction can be defined as $\mathrm{F}^+(r) = \log_2(1+2^{-r})$ and $\mathrm{F}^-(r) = -\log_2(1-2^{-r})$. These functions are shown in Fig. 2.

C. Rounding Modes and Precision

The IEEE 754 standard defines several rounding modes that can be applied after basic arithmetic operations like multiplications

and additions. The default rounding mode is *round to nearest*, and provides average and maximum relative errors of 0.1733 and 0.5 ulp, respectively. However, due to the different spacing of the machine numbers in LNS, an ulp in FP is not equivalent to an ulp in LNS. Therefore in [6], Coleman introduced the relations

$$\frac{|\epsilon|_{\text{avg rel arith}}}{2^{w_{\text{frac}}}} = \left(2^{|\epsilon|_{\text{avg log}}} - 1\right) \quad = \left(2^{\frac{|\epsilon|_{\text{avg rel log}}}{2^{w_{\text{frac}}}}} - 1\right) \tag{8}$$

$$\frac{|\epsilon|_{\text{max rel arith}}}{2^{w_{\text{frac}}}} = \left(2^{|\epsilon|_{\text{max log}}} - 1\right) \quad = \left(2^{\frac{|\epsilon|_{\text{max rel log}}}{2^{w_{\text{frac}}}}} - 1\right) \tag{9}$$

where $|\epsilon|_{\text{avg log}}$ and $|\epsilon|_{\text{max log}}$ are the average and maximum absolute errors in the LNS domain, $|\epsilon|_{\text{avg rel log}}$ and $|\epsilon|_{\text{max rel log}}$ are the average and maximum relative errors w.r.t. to one ulp in the LNS domain, and $|\epsilon|_{\text{avg rel arith}}$ and $|\epsilon|_{\text{max rel arith}}$ are the corresponding relative errors in the FP domain. Using these relations, we can calculate that, e.g. an LNS design with $w_{\text{int}} = 8$ and $w_{\text{frac}} = 23$ should have $|\epsilon|_{\text{max rel log}} < 0.7213$ in the LNS domain in order to have equivalent precision as FP with round to nearest rounding mode ($|\epsilon|_{\text{max rel arith}} < 0.5$).

However, FP equivalent accuracy of 0.5 ulp for a certain bit-width usually comes at a high cost and is not always required. Hence, it is common to use so called *faithful* designs [10], [17] which deliver a maximum error $\leq$1 ulp. Depending on the definition, an operator is either faithful in the FP or LNS domain (we use the latter in this work). For the distinction between exact and approximate designs, we will use the following definitions. A design for a certain bit-width configuration $w_{\text{int}}.w_{\text{frac}}$ is said to be *exact* if its maximum relative error $|\epsilon|_{\text{max rel arith}} \leq 0.5$. A design is considered to be faithful if $|\epsilon|_{\text{max rel log}} \leq 1$ in the LNS domain, and *approximate* otherwise.

D. Cotransformation

While for low precision implementations with up to around 12 fractional bits $\mathrm{F}^{\pm}(r)$ can be stored in LUTs, this approach is not practical for designs requiring higher precision since the LUT storage requirements increase exponentially as the bit-width grows. To achieve higher precision, piecewise polynomial approximations have been found to work well [10], [24] – except for operations where r is small since $\mathrm{F}^{-}(\mathrm{r})$ has a singularity at zero. This region is termed the *critical region* (CR) and typically ranges from $r \in [0, 0.25)$ to $r \in [0,\ 4)$, depending on the employed interpolation scheme. In this CR, so called *cotransformations* [3–5], [7–9] are usually applied in order to decompose $\mathrm{F}^{-}(\mathrm{r})$ into sub-functions which can be approximated more efficiently.The cotransformation employed in this work was originally proposed by [3] and decomposes $\mathrm{F}^{-}(r)$ into

$$\mathrm{F}^{-}(r) = -\log_2(1 - 2^{-r}) = -\log_2\left(\frac{1 - 2^{-r}}{r}\right) + \log_2(r) \tag{10}$$

$$= \mathrm{cotrans}(r) + \log_2(r). \tag{11}$$

It has been selected since it has been successfully used to create compact LNS operators for FPGAs [10], [24]. The first term $\mathrm{cotrans}(r)$ behaves much better around 0 as shown in Figure 2, and can be readily approximated using standard minimax polynomials. The $\log_2(r)$ function still has a singularity at 0, but for finite precision arithmetic this function can be efficiently implemented with range reduction techniques of the argument [28]. I.e., the argument range can be reduced to $[1,\ 2)$ by employing a leading zero counter and a barrel shifter. The $\log_2$ function itself can be efficiently implemented on this reduced range using a minimax polynomial. The size of the CR is often set to $[0,\ 4)$ when using this decompostion as this provides the best tradeoff in terms of the overall number of polynomial segments [10], [24].

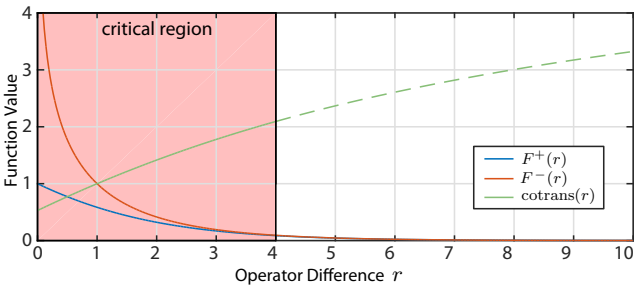

Figure 2. Plot of the $F^+(r), F^-(r)$ and the $\mathrm{cotrans}(r)$ functions - note the singularity for $r \to 0$.

IV. LNU Architecture

In this section, we will present the general architecture template of our LNU. Details on how specific parameters and LUT coefficients are obtained will be given in the next Section V. The main design goal of the LNU architecture shown in Fig. 3 is to reduce the hardware overhead and latency. For the targeted accuracy range between half precision (16 bit) and single precision (32 bit), first and second order minimax polynomial approximations of the functions $\mathrm{F}^{+}(\mathrm{r}), \mathrm{F}^{-}(\mathrm{r}), \mathrm{cotrans}(r)$, and $\log_2(r)$ have been found to be very efficient and were used throughout this evaluation. The architecture shown in Fig. 3 consists of 4 main blocks: the *Pre-* and *Postprocessing Blocks*, the *Main Interpolator Block* and the *Log/Exp Interpolator Block*. These blocks are explained in more detail below.

A. Preprocessing Block

The proposed LNU architecture uses different datapath units depending on the operation and whether or not the operation falls in the CR. The *Preprocessing Block* decodes the command, calculates the absolute operator difference $r = |l_a - l_b|$ and the operator maximum for binary operations such as ADD/SUB. At this point the block is able to determine which datapath units will be activated and generates all control signals for the LNU and performs operation dependent preparation steps on the two operators $A = [s_a, l_a]$ and $B = [s_b, l_b]$. For unary operations such as EXP/LOG and typecasts, operator B is gated to zero and A is passed through.

B. Main Interpolator Block

The *Main Interpolator Block* implements $\mathrm{F}^{+}(r)$ on the complete range $[0,\ t_{clip})$ and $\mathrm{F}^{-}(\mathrm{r})$ outside the CR $[4,\ t_{clip})$ using 1st or 2nd order piecewise polynomial approximations which have been found to provide the best latency vs. LUT area trade-off for the precision range between half and single precision. This block is also used for SUB operations in the CR $[0,\ 4)$ to evaluate $\mathrm{cotrans}(r)$, the result of which is later added to the $\log_2(r)$ value in the *Postprocessing Block*. For a given input r, the coefficients $p_i^r = p_i(r)$ for $i = \{0, ..., N\}$ (where N is the polynomial order) are selected from a set of LUTs, and the polynomial is evaluated using the Horner scheme as

$$p(r) = p_0^r + \delta_p^r \cdot \left(\ldots \left(p_{(N-1)}^r + \delta_p^r \cdot (p_N^r)\right)\right) \tag{12}$$

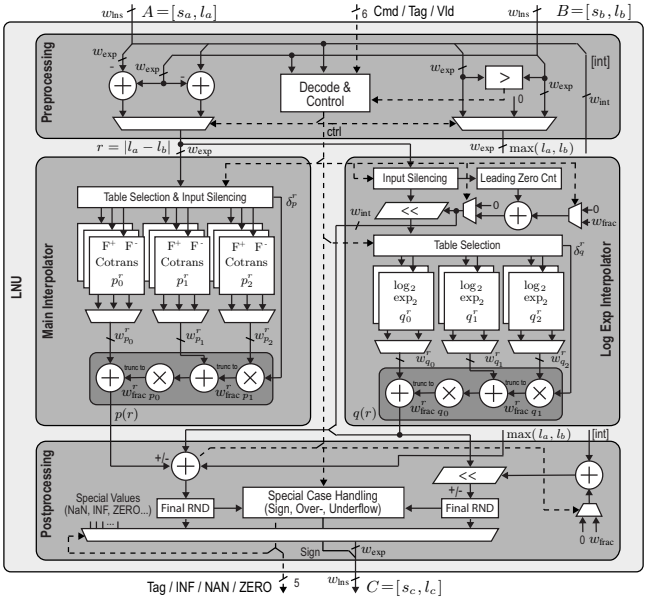

Figure 3. LNU Architecture template used in our generator (shown for $N = 2$).

where δ_p^r are the LSBs of r. Since the LNU processes only one instruction at a time, the main interpolator datapath can be shared among $\mathrm{F}^+(r)$, $\mathrm{F}^-(r)$ and $\mathrm{cotrans}(r)$. As will be explained in more detail in Section V, each LUT is subdivided into different segments, each of which contains a set of equidistantly spaced coefficient samples. The segment boundaries have been aligned to powers of two, such that the segment index can be easily determined by looking at the MSBs of the argument r. For large r, the functions values of $\mathrm{F}^+(r)$ and $\mathrm{F}^-(r)$ fall below the required precision due to their asymptotic behaviour and can be clipped to 0. This clipping threshold is denoted as t_{clip}, and amounts to $t_{clip} \approx 24.588$ for an exact single-precision design. Further, $\mathrm{F}^+(r)$ and $\mathrm{F}^-(r)$ become increasingly similar with increasing r such that one function can be replaced by the other without impact on precision. Therefore, we define a second threshold t_{repl} and reuse the $\mathrm{F}^+(r)$ tables for $\mathrm{F}^-(r)$ when $r > t_{repl}$. For an exact single-precision design, this value is $t_{repl} \approx 14$.

C. Log/Exp Block

The main objective of the *Log/Exp Interpolator Block* is to implement the $\log_2(r)$ function on the critical range $[0, 4)$ for cotransformed SUB operations. The function is implemented using a barrel shifter and leading zero counter to reduce the range of the input, and a N-th order interpolator with LUTs covering the argument range $[1, 2)$. Note that it is possible to reuse this function to also implement native typecasts from integer to LNS (I2F), and LOG2 operations in the LNS domain. For a given input r, the polynomial coefficients $q_i^r = q_i(r)$ for $i = \{0, ..., N\}$ are selected from a set of LUTs, and the approximation result $q(r)$ is again calculated using the Horner scheme as in Equation (12). In order to natively support inverse typecasts (F2I) and EXP2 operations in LNS, we add a table for the $\exp_2(r)$ function. Since this function can also be efficiently implemented using range reduction and polynomial interpolation, we can reuse the existing interpolator to calculate the function value on the range $[0, 1)$, and only have to include an additional shifter at the output. This shifter has been moved to the *Postprocessing Block* and operates in parallel to the final adder of the ADD/SUB operations such that the main ADD/SUB path suffers from no additional delay.

D. Postprocessing Block

The *Postprocessing Block* combines and/or selects the results of the two interpolation blocks. For example, SUB operations in the CR require the output $p(r)$ of the *Main Interpolation Block* and the output $q(r)$ of the *Log/Exp Interpolation Block* to be combined. A final rounding step to the output precision and special case handling such as NAN, over- and underflow detection are also performed.

V. LNU Generator

The architecture presented in the previous section serves as a parameterizable template for our LNU generator. The flow of our LNU generator is illustrated in Figure 4 and consists of three steps. In the first step (A), the generator calculates the quantized polynomial coefficients for all required functions according to the LNU specification which consists of the LNS format w_{int}, w_{frac}, polynomial order N, and error bounds $|\epsilon|$max, $|\epsilon|$avg for the relative errors in the LNS domain. In the second step (B), all coefficients are scanned and the bit-width parameters for the shared datapaths are calculated. The parameters and coefficients are then printed into the architecture template in order to form a specific LNU instance. In the third and last step (C), this instance is then exhaustively verified using RTL simulations in Mentor QuestaSim. The core generator functionality in steps A and B have been implemented in MATLAB.

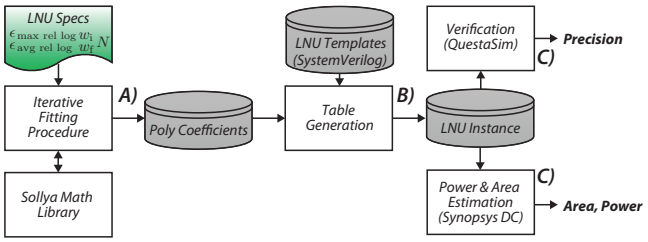

Figure 4. Overview of the LNU generation flow.

A. Polynomial Fitting

For all function approximations, we use N th order piecewise *minimax* polynomials. The coefficients are obtained using an efficient, quantization aware implementation of Remez's algorithm [26] available in the Sollya math library [29]. Since the functions used here are increasingly difficult to approximate as $r \to 0$, we subdivide their domains into logarithmically spaced segments, and within each segment a different number of piecewise polynomials is used. These segment boundaries are always aligned to multiples of powers of two, and the spacing between piecewise polynomials Δ_p^r is always a power of two. A table lookup can then conveniently be carried out by looking at the amount of leading zeros of r in order to activate a specific segment. Depending on the spacing Δ_p^r of the piecewise polynomials within that segment, the bits $[w_{\text{exp}} : w_{\text{frac}} + \log_2(\Delta_p^r)]$ of r are used to determine which set of polynomial coefficients to use, and the remaining LSBs are used as input into the Horner interpolator $\delta_p^r = r[w_{\text{frac}} + \log_2(\Delta_p^r) - 1 : 0]$.

The fitting procedure subdivides the function to be fitted into logarithmically spaced segment (e.g. $[0, 32)$ is split into $[16, 32)$, $[8, 16)$, $[4, 8)$, etc.), and on each segment it fits a set of piecewise polynomials with an initial spacing $\Delta_p^r = 0.25$. This choice has been made to implicitly limit the integer bit-widths in the shared interpolators. If the error of this piecewise polynomial is too large, the spacing is iteratively divided by 2 until the error requirements can be fulfilled. Once all segments have been processed, adjacent segments with equal spacing are grouped together in order to facilitate table lookup, and the coefficients are handed over to the table generation.

B. Error Calculation and Bitwidth Selection

To asses the overall error of a given piecewise polynomial, we exhaustively assess it at all bit combinations of r within its domain and compare the result to a double precision reference. The so calculated errors contain both the approximation error stemming from the polynomial approximation, and the quantization errors from the coefficient quantization and intermediate truncation/rounding steps. The impact of coefficient quantization is minimized by using a quantization aware minimax fitting method. Since a rounding step always incurs a carry propagation in hardware, we only perform one rounding step to the output precision w_{frac} at the end of a polynomial evaluation. Intermediate results after multiplications are always truncated to the fraction of the next coefficient it is being added to, as illustrated in Figure 3. The bit-widths of the polynomial coefficients are determined using a similar heuristic as described in [30]. I.e., since typically $\delta_p^r < \Delta_p^r < 1$, the weight of LSBs of a polynomial coefficient p_i^r are decreased by at least a factor of Δ_p^r in each multiplication with δ_p^r. Therefore, we heuristically determine the fraction width $w^r_{\text{frac}p_i}$ of higher order coefficients as

$$w^r_{\text{frac }p_i} = \max\left(0, w^r_{\text{frac }p_0} + i \cdot \log_2\left(\Delta_p^r\right)\right) \tag{13}$$

and the fraction of the 0th order coefficient is set to

$$w^r_{\text{frac }p_0} = w_{\text{frac}} + \max\left(0, n_{\text{guard}} - \lceil \log_2\left(\epsilon_{\text{max rel log}} - \epsilon_{\text{rnd}}\right)\rceil\right) \tag{14}$$

where $\epsilon_{\text{max rel log}}$ is the maximum error requirement, and the term $\epsilon_{\text{rnd}} = 0.5$ amounts for the maximum error due to the final rounding step. The amount of additional guard bits has been set to $n_{\text{guard}} = 3$, since fewer than 3 bits lead to a significant increase in table size.

Within the CR, where two polynomial results $p(r)$ and $q(r)$ are added together before the final rounding step, the error requirement for both polynomials is adjusted to $\tilde{\epsilon}_{\text{max rel log}} = (\epsilon_{\text{max rel log}} - \epsilon_{\text{rnd}})/2$ in order to account for the fact that we have twice as many error sources now. Error checking of the individual polynomials is performed without final rounding in this case to make sure that the polynomials are precise enough before being added. Also, ϵ_{rnd} is set to 0 in Equation (14), as it has already been accounted for in $\tilde{\epsilon}_{\text{max rel log}}$.

VI. Processor Integration

To evaluate the performance of the presented LNUs in a shared setting, we have designed a multi-core processor system based on a 32 bit OpenRISC core [31] using the UMC 65 nm LL technology. As shown in Figure 5, the system consists of four cores which share a single LNU and contains 32 kBytes of memory, distributed equally between an eight-bank tightly-coupled data-memory (L1), and a dense L2 memory. The four to one sharing ratio is motivated by the fact that in most FP programs, the fraction of ADD/SUB instructions rarely exceeds 0.25 (see Figure 8b for some examples). For comparison purposes, an identical system has been designed featuring 4 cores with a private IEEE 754 single precision compliant FPU that includes hardware support for additions, subtractions, multiplications and typecasts. For divisions we use software emulations as described in Section VII-B, since divisions are expensive in hardware[1] Note that this is a common approach of adding FP support to small embedded processors [33]. The implemented FPU is a shared normalizer design similar to [32] (but without divider), and when synthesized with 2 pipeline stages it has a complexity of 11 kGE – which is competitive with state-of-the-art implementations [33], [34].

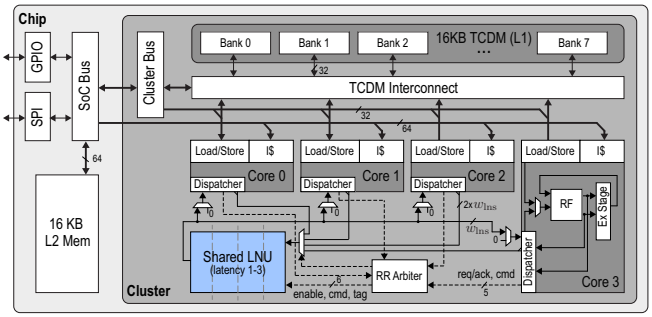

Figure 5. Integration of the shared LNU into an OpenRISC cluster.

All clusters have been designed to run at 500 MHz at 1.2V under typical case conditions. In order to meet the timing constraints for both architectures, the FPU has been pipelined once, and the LNU one, two or three times – depending on the area and latency of the specific LNU instance.

A. Modifications to the Processor Core

The LNU is shared in a completely transparent way, the programmer sees a system with as many LNUs as there are cores. A dispatcher that is tightly integrated into the datapath of each core is responsible to offload the LNU instructions, stall the cores if necessary, and silence the operator ports in case no instruction has to be offloaded such that unwanted switching activity across the interconnect is minimized. The integer ALUs of the cores have been slightly modified to be able to support the LNS sign bit and the special cases such as INF, ZERO and NAN during LNS MULT/DIV and comparison instructions. Besides the standard FP instructions defined in the OpenRISC ISA which have been mapped to the corresponding LNS instructions, we have added three special instructions allowing the cores to natively execute SQRT, EXP2 and LOG2 operations. While EXP2 and LOG2 are offloaded to the LNU, the SQRT instruction has been implemented using the shifter in the core and can be executed in 1 cycle.

B. Sharing Interconnect

The LNU interconnect contains a fair round-robin arbiter which handles requests from the processor cores. Whenever more than one core wants to access the LNU, all cores but one have to stall their pipeline and wait for an idle cycle of the LNU. Our OpenRISC architecture contains two write back port to the register file since several instructions update multiple registers concurrently. While

[1] As shown in [32], an FPU design with 12 cycle iterative division consumes ~26 kGE (transistor count divided by four).

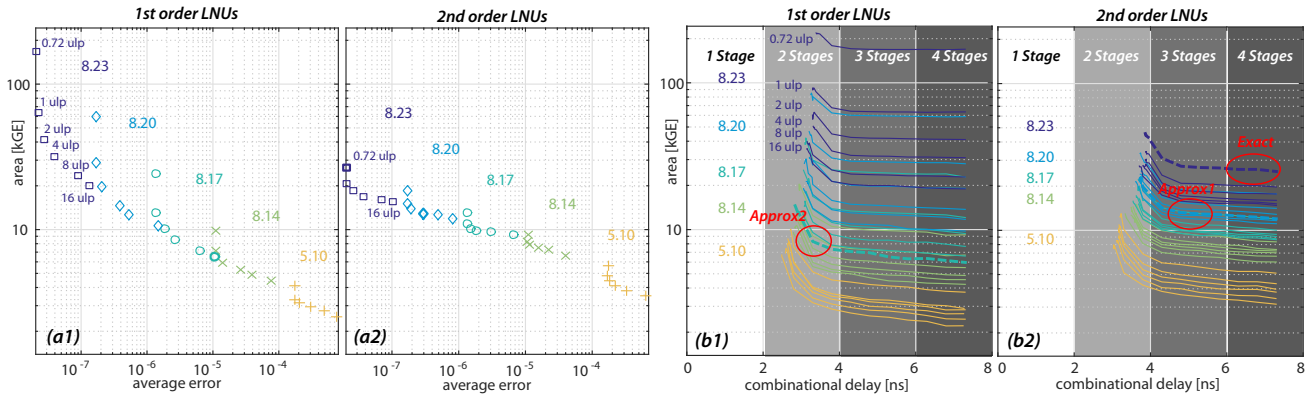

Figure 6. a) Area vs. average error for LNUs synthesized with 4.5 ns timing constraint. b) AT-plots, including 0.8 ns I/O delay for the sharing infrastructure.

the first write port is used by the ALU, the results of which are ready at the end of the execution-stage, the second write port is used to write back the values loaded from memory. Instead of adding a third write port for the LNU, this second write port has been multiplexed with the LNU interconnect. In a single issue in-order pipeline, it is mutually impossible that LNU and load operations are executed at the same time. Hence, this second write port can be shared without any contention.

During the exploration we have considered LNUs with different pipeline depths. An implementation with only 2 cycles latency can directly write back to the register file without causing the pipeline to stall. For implementations with longer pipeline depths, the core has to be stalled unless the LNU instruction is followed by another LNU instruction with same latency. LNU operations with 3 and 4 cycle latency generally result in 1 and 2 stall cycles, respectively. Hence, it is crucial to minimize the latency of the LNU in order to obtain good application level performance. In case of a 2 cycle latency LNU, stalls only occur when multiple cores try to access it in the same cycle, or when the result of the LNU is needed in the subsequent cycle which can typically be avoided by the compiler with instruction reordering.

VII. Results

In this section we first present a design space exploration of a family of exact, faithful and approximate LNUs generated with our framework and compare against related work. The resulting instances have been synthesized using the 8-metal UMC 65 nm LL CMOS technology with Synopsys Design Compiler version 2015.6 in order to get gate-level area and timing estimates at typical conditions, 25 °C and 1.2 V supply voltage. For cluster level evaluations, we have selected a set of representative LNU versions, pipelined them using the automatic retiming feature of Synopsys Design compiler, and for each cluster version we have performed a complete back-end design flow using Cadence EDI 14.24 in the same 8-metal UMC 65 nm LL CMOS technology. We have modified the back-end of the OpenRISC LLVM compiler to support the LNS format, and added new instructions to support the additional functionality provided by our LNU. A set of benchmarks written in C was compiled and executed on the FP and LNU cluster architectures, which have been simulated in Mentor QuestaSim 10.3a using back-annotated post-layout gate-level netlists. Finally, the obtained VCD files were used to analyze the power dissipation in Cadence EDI 14.24.

A. LNU Design Space Exploration

First we present a comparison of the fully exact 32 bit LNU generated with our framework to similar published works [8], [9], [11]. In Table I technology independent gate equivalent (GE) numbers show that our 2nd order LNU implementation is the smallest implementation reported in literature (by 33%), while delivering the lowest maximum error over the relevant range $[0, 24.588)$. More importantly, at 26.8 kGE complexity we show that exact LNUs can be designed with similar area overhead than single precision FPUs that include division support ($\geq$26 kGE with 12 cycle iterative division [32]).

Next we investigate the impact of approximation on the circuit area. There are three approaches to trade-off circuit area, delay and precision: changing the bit-width of the design, changing the interpolation order, or relaxing the precision requirements. In order to achieve smaller and faster designs, usually the bit-width and interpolation order are reduced and tables are calculated for exact or faithful representations. In this paper, for a given bit-width we also relax the precision requirements up to 16 ulp which allows simpler tables that reduce the overall circuit size. In Table II we show the normalized area of 1st and 2nd order LNUs under different bit-widths and approximation goals. As we can see, relaxing the

Table I
Comparison of exact 32 bit LNU with related work.

	[8]	[9]	[11], [27]	This Work
Functionality	ADD, SUB	ADD, SUB	ADD, SUB, I2F, F2I, LOG2, EXP2	
ADD Precision				
$\lvert\epsilon\rvert_{\text{max rel arith}}$	0.4544	0.4623	0.4618	0.3920
$\lvert\epsilon\rvert_{\text{avg rel arith}}$	0.1777	0.1745	0.1748	0.1744
SUB Precision				
$\lvert\epsilon\rvert_{\text{max rel arith}}$	0.4952	0.4987	0.4786	0.4504
$\lvert\epsilon\rvert_{\text{avg rel arith}}$	0.1776	0.1738	0.1748	0.1746
Implementation				
Technology	180 nm	180 nm	65 nm	65 nm
1 GE [μm^2]	9.374†	9.374†	1.44	1.44
Delay (min) [ns]	11.74	7.10	6.00	4.50
Delay (max) [ns]	13.50	14.79		
LUT size [kBit]	356.4	183.3	113.1	64.2
Area [mm^2]	0.906	0.589	0.057	0.039
Area [kGE]	96.6	62.9	40.0	26.8

† assumed NAND2 area for calculating gate equivalents (GE) for [8], [9].

precision of an exact 2nd order LNU from 0.72 ulp to 8 ulp leads to an area reduction of 40 %. The interesting result from Table II is that area-wise, similar results can be obtained by either reducing the bit-width or the precision. However, when considering the average error, the situation changes. In Figure 6 the *average* error for LNU designs are plotted against the circuit area. In this plot we can see that approximate configurations with larger bit-width are consistently more accurate (on average) than lower bit-width configurations. For example, a 2nd order 8.20 with 8 ulp precision LNU configuration is not only slightly smaller than a 2nd order 8.17 with 0.72 ulp precision, but the average error is lower by a factor of $\times$ 2.74. Another observation that can be made from Table II is the that for higher precision ranges (17-23 fractional bits), 2nd order LNUs are much more area efficient since fewer LUT entries are required. For designs with a $w_{\text{frac}} \leq 14$, 1st order interpolation is preferable.

Reducing the circuit complexity has an additional benefit as it also reduces the critical path through the LNU. Depending on the clock frequency of the system where the LNU will be integrated, this could change the number of required pipeline stages which in turn can have important consequences on the overall performance of the system. Figure 6 shows a design space exploration for 50 LNU configurations, mapped to hardware with different timing constraints. The target clock period (in our case 2 ns for 500 MHz operation) is overlayed in this graph, and it can be seen that LNUs with different area/precision trade-offs can be obtained with 2 to 4 pipeline stages. We have selected a representative set of three LNU variants *Approx2* (8.17 bit, 16 ulp, 1st order), *Approx1* (8.20 bit, 4 ulp, 2nd order), and *Exact* (8.23 bit, 0.72 ulp, 2nd order) that were implemented with different numbers of pipeline stages to be evaluated in the following section that compares overall system performance.

B. Performance of LNU in multi-core clusters

After evaluating the performance of a single LNU, we now present a more detailed performance analysis of a shared LNU in a real multi-core system running actual computation kernels. For this comparison we use a system comprised of four 32-bit OpenRISC processor cores running at 500 MHz in the UMC 65LL technology used throughout this work. Our reference (*FPU*) is a system that includes four IEEE-754 single precision compliant FPU units with support for ADD/SUB/MULT and casts. This is compared against three different LNU configurations (*Exact*, *Approx1*, and *Approx2*) selected from the design space exploration described previously.

Table II
RELATIVE AREA COMPARISON (IN PERCENT) OF EXACT AND APPROXIMATE LNUS SYNTHESIZED WITH 4.5NS TIMING CONSTRAINT.

Order	$w_{\text{int}}.w_{\text{frac}}$	Precision Constraint in the LNS Domain (ulp) 0.72	1	2	4	8	16
	8.23	618.6	232.7	153.6	116.0	86.6	73.0
	8.20	218.6	106.5	72.4	53.8	46.7	39.6
1	8.17	89.8	48.2	38.0	31.0	26.9	25.4
	8.14	36.3	26.8	22.3	19.8	18.3	17.2
	5.10	15.5	12.2	11.8	11.0	10.3	9.4
	8.23	**100.0**	76.5	68.4	62.9	59.9	57.8
	8.20	69.5	56.5	51.2	48.2	46.1	44.4
2	8.17	49.0	41.3	38.5	37.0	36.4	34.9
	8.14	34.9	31.1	29.4	28.3	27.0	24.9
	5.10	21.4	18.2	16.9	15.9	14.6	13.3

Table III
COMPARISON OF INSTRUCTION LATENCY AND ENERGY EFFICIENCY IN THE FPU AND LNU CLUSTER VARIANTS AT 1.2 V.

Format	IEEE754	LNS		
Name	FPU	Exact	Approx1	Approx2
Bitwidth	8.23	8.23	8.20	8.17
Precision	0.5 ulp	0.72 ulp†	4 ulp†	16 ulp†
Order N	-	2	2	1
FPU/LNU [kGE]	4×11	36	27	23
Total Area [kGE]	720	718	708	704
Operations	**Latency [cycles] / Energy [pJ/Op]**			
I2F/F2I	2 / n.a.	4 / n.a.	3 / n.a.	2 / n.a.
ADD	2 / 40.7	4 / 106.2	3 / 88.8	2 / 85.3
SUB	2 / 39.7	4 / 109.9	3 / 92.1	2 / 90.4
MUL	2 / 47.6	1 / 30.7	1 / 27.7	1 / 30.6
DIV	62 / 525.0*	1 / 31.5	1 / 28.7	1 / 31.6
SQRT	56 / 609.3*	1 / 16.0	1 / 14.5	1 / 15.2
EXP	51 / 566.6*	4 / 114.9	3 / 56.8	2 / 86.9
LOG	85 / 695.7*	4 / 104.9	3 / 54.5	2 / 73.7

* software emulation. † in the LNS domain.

Table III lists all four variants and their complexities. As can be seen, all LNU variants are smaller than the reference FPU implementation. The larger size of the LNUs is compensated by sharing them among the processor cores. Note that, even though these systems only have a single LNU, they can perform up to four MUL/DIV/SQRT single cycle LNS operations within the integer ALU of the cores that include the modifications described in VI-A.

The reference FPU at 11 kGE is very compact but does not include support for more complex operations which have to be emulated in software. For DIV operations, we perform a range reduction to [1,2) and generate a linear estimate for the inverse that is refined using three Newton-Raphson iterations. A similar technique is used for the SQRT, where the initial estimate is generated using the fast-inverse square-root approach. EXP/LOG operations combine range-reduction with a standard high-order interpolation techniques as described in [28]. Note that this is a common way to add FP support to small embedded processors [33]. Table III lists the number of cycles per instruction and the corresponding energy consumption in pJ/Op. While ADD/SUB operations (as expected) are costlier in the LNUs than FPU, all other operations can be performed more efficiently.

We have compiled a set of benchmarks written in C to reflect a variety of different signal processing applications. The benchmark set consists of linear algebra operations (AXPY, GEMM, GEMV), geometry calculations (2D homographies, reprojection error [35], 3D distances), matrix decompositions (QR, CHOL), regression (radial basis functions), and image (bilateral filter and FIR filters, gradient magnitude, DCT-II) and audio processing kernels (sine generation, Butterworth IIR lowpass). The floating point instruction ratio and the instruction mix of the benchmark applications is shown in Figure 8b. As can be seen, the ratio of ADD/SUB operations for most benchmarks is below 25% further reinforcing our sharing concept.

All performance evaluations were made with four configurations of the cluster, including 2 configurations (*Approx1* and *Approx2*) that have small precision relaxations. We have determined the impact of this range of approximations on image and audio processing kernels using PSNR and THD+N metrics which are shown in Figure 7. The *FPU* and *Exact* LNU configurations deliver identical

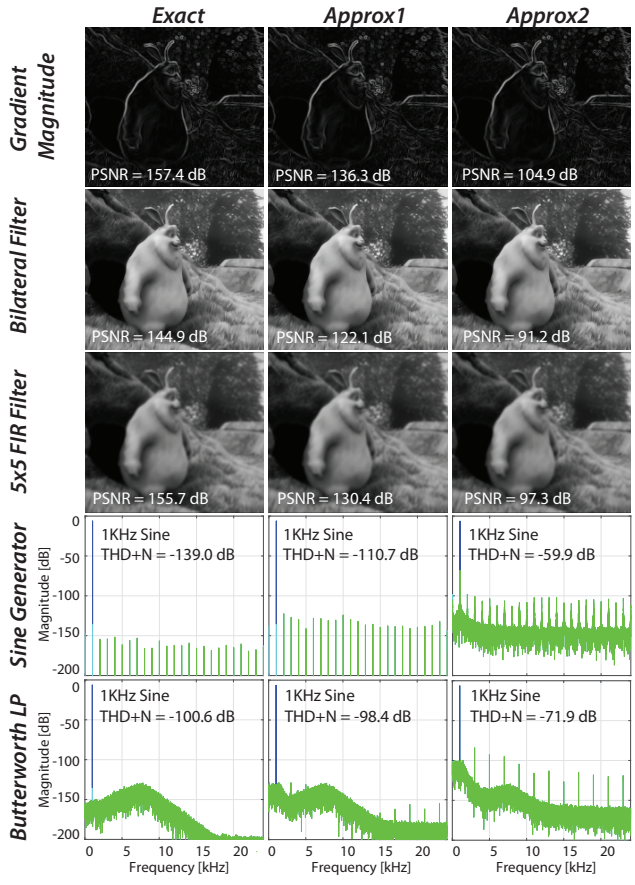

Figure 7. Images and audio streams processed with the cluster variants *Exact*, *Approx1* and *Approx2*. Image © copyright 2008, Blender Foundation.

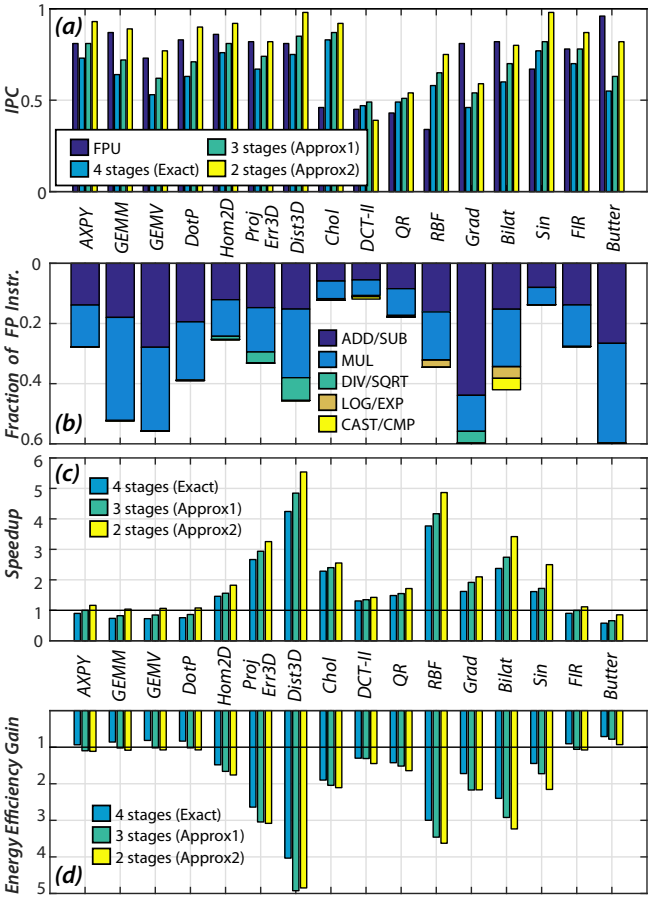

Figure 8. Application level performance for the four cluster variants.

results. We can observe that for all image processing kernels the PSNR values stay way above the 30 dB, below which artifacts start to be visible. Also for the audio processing kernels we see that THD+N values are below -59 dB for all LNUs. For all practical purposes, both approximate versions show no perceptible quality degradations. Normalized execution time and energy-efficiency improvements with respect to the reference *FPU* implementation mirror each other and are shown in Figure 8c and d. As can be seen for most tests at least one LNU configuration outperforms the reference. In the best case, a speedup of 5.54× can be achieved for the DIST3D case using *Approx2* configuration. Even when the *Exact* LNU configuration is used, on average the kernels can be calculated 1.71× faster and 1.65× more efficiently in terms of energy. As expected the shared LNU has the most difficulty with kernels containing many ADD/SUB operations (such as the linear algebra or FIR/IIR filters). Even so, it can be seen in Figure 8c and d that the *Approx2* LNU configuration can perform at least as well as the FPU design on all kernels except the Butterworth filter.

We also observe that the pipeline depth of the LNU implementation has significant impact on the overall system performance. In our example, the 4-stage in-order OpenRISC cores can be operated without stalling for LNUs with 2 stages. LNUs with more stages incur additional stalls, reducing the overall IPC as seen in Figure 8a.

VIII. Conclusions

We presented a framework to generate a family of accurate and approximate 1st and 2nd order LNUs capable of executing ADD, SUB, I2F, F2I, EXP2, and LOG2 instructions. The area overhead of an accurate LNU with equivalent accuracy offered by the IEEE 754 single precision format is reduced by 33% when compared to previous state-of-the-art implementations. Also, we show that by relaxing the precision requirements from exact to 16 ulp, significant area savings of 40 % can be achieved, bringing the complexity closer to standard FPU implementations. The area overhead of the LNU can further be reduced in a multi-core setting where one LNU is shared by multiple processor cores. Unlike standard FPUs, LNUs are much more amenable to a shared setting, as several LNS operations such as MUL/DIV/SQRT can be performed within the integer ALU of the cores. We have extracted real-life performance values of three different shared LNU versions in a four-core cluster, and compared it with a standard FPU implementation consisting of four FPUs and four cores. We show that in a shared setting, the shared LNU solutions are not only smaller but outperform the standard FP solution on *average* by a factor of 1.71× in execution time. We further show that by using approximate computing techniques these gains can be further increased, by up to 5.54× in the best case. Using examples from the image and audio processing domains, we analyze the incurred quality losses of the approximate designs in terms of PSNR and THD+N and demonstrate that the

errors are imperceptible in the evaluated precision range. We show that using LNUs in a shared setting is a very promising approach for multi-core processors especially in applications where energy-efficiency is paramount. For these applications we also show that relaxed-precision approximations can be more efficient than simple bit-width reductions.

ACKNOWLEDGMENTS

This research was supported by the IcySoC project, evaluated by the Swiss NSF and funded by Nano-Tera.ch with CH financing.

REFERENCES

[1] N. Kingsbury and P. Rayner, "Digital filtering using logarithmic arithmetic," *Electronics Letters*, vol. 7, no. 2, pp. 56–58, January 1971.

[2] E. Swartzlander and A. Alexopoulos, "The Sign/Logarithm Number System," *IEEE TOC*, vol. C-24, no. 12, pp. 1238–1242, Dec 1975.

[3] V. Paliouras and T. Stouraitis, "A Novel Algorithm for Accurate Logarithmic Number System Subtraction," in *IEEE ISCAS*, 1996.

[4] M.G. Arnold et al., "Arithmetic Co-Transformations in The Real and Complex Logarithmic Number Systems," *IEEE TOC*, Jul 1998.

[5] J. Coleman, "Simplification of Table Structure in Logarithmic Arithmetic," *Electronics Letters*, vol. 31, no. 22, pp. 1905–1906, Oct 1995.

[6] J.N. Coleman et al., "Arithmetic on the European Logarithmic Microprocessor," *IEEE TOC*, vol. 49, no. 7, pp. 702–715, Jul 2000.

[7] P. Vouzis et al., "LNS Subtraction Using Novel Cotransformation and/or Interpolation," in *IEEE ASAP*, July 2007, pp. 107–114.

[8] J.N. Coleman et al., "The European Logarithmic Microprocessor," *IEEE TOC*, vol. 57, no. 4, pp. 532–546, April 2008.

[9] R. Ismail and J. Coleman, "ROM-less LNS," in *ARITH*, July 2011.

[10] H. Fu et. al, "FPGA Designs With Optimized Logarithmic Arithmetic," *IEEE TOC*, no. 7, pp. 1000–1006, 2010.

[11] Y. Popoff, F. Scheidegger, M. Schaffner, M. Gautschi, F. K. Gürkaynak, and L. Benini, "High-Efficiency Logarithmic Number Unit Design Based on an Improved Co-Transformation Scheme," in *DATE*, 2016.

[12] H. Esmaeilzadeh et. al, "Architecture Support for Disciplined Approximate Programming," *ACM SIGPLAN*, vol. 47, no. 4, Mar. 2012.

[13] S. Venkataramani et. al, "Quality Programmable Vector Processors for Approximate Computing," in *IEEE/ACM MICRO*, 2013.

[14] K. Palem and A. Lingamneni, "Ten Years of Building Broken Chips: The Physics and Engineering of Inexact Computing," *ACM TECS*, 2013.

[15] F. de Dinechin and A. Tisserand, "Multipartite Table Methods," *IEEE TOC*, vol. 54, no. 3, pp. 319–330, March 2005.

[16] J. Detrey and F. de Dinechin, "Table-Based Polynomials for Fast Hardware Function Evaluation," in *IEEE ASAP*, July 2005.

[17] J. Detrey and F. De Dinechin, "A Tool for Unbiased Comparison Between Logarithmic and Floating-Point Arithmetic," *J VLSI SIG PROC SYST*, vol. 49, no. 1, pp. 161–175, 2007.

[18] J. Rust et al., "Low Complexity QR-Decomposition Architecture using the Logarithmic Number System," in *IEEE DATE*, March 2013.

[19] J. Garcia et al., "LNS Architectures for Embedded Model Predictive Control Processors," ser. ACM CASES, 2004, pp. 79–84.

[20] I. Kouretas et al., "Low-Power Logarithmic Number System Addition/Subtraction and Their Impact on Digital Filters," *TOC*, Nov 2013.

[21] M. Arnold and S. Collange, "A Real/Complex Logarithmic Number System ALU," *IEEE TOC*, vol. 60, no. 2, pp. 202–213, Feb 2011.

[22] M.G. Arnold et al., "Towards a Quaternion Complex Logarithmic Number System," in *IEEE ARITH*, 2011, pp. 33–42.

[23] R.C. Ismail, et al., "Hybrid Logarithmic Number System Arithmetic Unit: A Review," in *IEEE ICCAS*, Sept 2013, pp. 55–58.

[24] H. Fu et. al, "Optimizing Logarithmic Arithmetic on FPGAs," in *IEEE FCCM*. IEEE, 2007, pp. 163–172.

[25] "FloPoCo," http://flopoco.gforge.inria.fr/, accessed: 2015-11-05.

[26] J.-M. Muller, *Elementary Functions*. Springer, 2006.

[27] M. Gautschi, M. Schaffner, F. K. Gürkaynak, and L. Benini, "A 65nm CMOS 6.4-to-29.2pJ/FLOP@0.8V Shared Logarithmic Floating Point Unit for Acceleration of Nonlinear Function Kernels in a Tightly Coupled Processor Cluster," in *IEEE ISSCC*, 2016.

[28] J. F. Hart, *Computer Approximations*. Krieger Publishing Co., 1978.

[29] S. Chevillard et al., "Sollya: An Environment for the Development of Numerical Codes," in *ICMS*, September 2010, pp. 28–31.

[30] F. de Dinechin et al., "Automatic Generation of Polynomial-Based Hardware Architectures for Function Evaluation," in *IEEE ASAP*, 2010.

[31] M. Gautschi et al., "Tailoring Instruction-Set Extensions for an Ultra-Low Power Tightly-Coupled Cluster of OpenRISC Cores," in *VLSI-SoC*, 2015, pp. pp25–30.

[32] Taek-Jun Kwon et al., "Design Trade-Offs in Floating-Point Unit Implementation for Embedded and Processing-in-Memory Systems," in *IEEE ISCAS*, May 2005, pp. 3331–3334 Vol. 4.

[33] K. Karuri et al., "Design and Implementation of a Modular and Portable IEEE 754 Compliant Floating-Point Unit," in *DATE*, March 2006.

[34] S. Galal and M. Horowitz, "Energy-Efficient Floating-Point Unit Design," *IEEE TOC*, vol. 60, no. 7, pp. 913–922, July 2011.

[35] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge university press, 2003.