FPGA Designs with Optimized Logarithmic Arithmetic

Haohuan Fu, *Member*, *IEEE*,
Oskar Mencer, *Member*, *IEEE*, and
Wayne Luk, *Fellow*, *IEEE*

Abstract—Using a general polynomial approximation approach, we present an arithmetic library generator for the logarithmic number system (LNS). The generator produces optimized LNS arithmetic libraries that improve significantly over previous LNS designs on area and latency. We also provide area cost estimation and bit-accurate simulation tools that facilitate comparison between LNS and floating-point designs.

Index Terms—Reconfigurable hardware, special-purpose and application-based systems, computer systems organization, computer arithmetic, general, numerical analysis, mathematics of computing.

1 Introduction

LOGARITHMIC Number Systems (LNSs) were first introduced into computer systems for processing of low-precision FFT in the 1970s [1]. Unlike the floating-point (FLP) numbers defined by IEEE 754 standard, there is no commonly accepted standard for LNS numbers. In this paper, we use the signed logarithmic format similar to [2], which consists of a sign bit and a fixed-point number to record the logarithmic value, shown as follows:

Sign Bit	Fixed-point Logarithmic Value	
	Integer: m bits	Fractional: f bits
S	M	F

Its value is given by $(-1)^S \times 2^{M.F}$, which provides a similar representation range to FLP numbers with m-bit exponent, f-bit mantissa, and one sign bit. Similar to the infinity and "Not a Number" (NaN) cases in FLP, we use special encoding of the integer bits to indicate zero and exceptional cases.

Suppose a and b are logarithmic representations of positive numbers A and B (without loss of generality, we assume $A > B$), thus $A = 2^a$ and $B = 2^b$. The basic LNS arithmetic operations of these two numbers include:

$$\begin{aligned}
&\text{MUL:} && A \times B = 2^a \times 2^b = 2^{a+b},\\
&\text{DIV:} && A \div B = 2^a \div 2^b = 2^{a-b},\\
&\text{SQRT:} && \sqrt{A} = \sqrt{2^a} = 2^{a/2},\\
&\text{ADD:} && A + B = 2^a + 2^b = 2^{b+\log_2(2^{a-b}+1)},\\
&\text{SUB:} && A - B = 2^a - 2^b = 2^{b+\log_2(2^{a-b}-1)}.
\end{aligned}$$

As shown above, LNS MUL, DIV, and SQRT become simple operations, while LNS ADD and SUB become complicated as they require evaluation of two transcendental functions, $f_1(x) = \log_2(2^x + 1)$ and $f_2(x) = \log_2(2^x - 1)$.

- *H. Fu is with the Department of Geophysics, Stanford University, Stanford, CA 94305. E-mail: haohuan@stanford.edu.*
- *O. Mencer and W. Luk are with the Department of Computing, Imperial College London, UK. E-mail: {oskar, wl}@doc.ic.ac.uk.*

Manuscript received 19 Aug. 2008; revised 20 Feb. 2009; accepted 20 July 2009; published online 12 Feb. 2010.
Recommended for acceptance by P. Montuschi.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-08-0424.
Digital Object Identifier no. 10.1109/TC.2010.51.

For applications that compute over a wide dynamic range, LNS provides an alternative solution to FLP, and the possibility to implement the design with a smaller area or higher throughput. Based on our previous work on optimizing LNS arithmetic on FPGAs [3], we develop tools to enable convenient design and implementation of LNS application, and facilitate comparison between LNS and FLP implementations on area, precision, and throughput. Our contributions are:

1. A systematic optimization of LNS arithmetic on FPGAs, using a general polynomial approximation approach.
2. A library generator that produces LNS arithmetic libraries containing $+, -, *, /$ operators as well as convertors between FLP and LNS numbers.
3. Area cost estimation and bit-accurate simulation tools to facilitate comparison between LNS and FLP designs.

The basic arithmetic units are tested on an FPGA board as well as software simulation. When compared with existing LNS designs [4], our LNS units typically achieve 6-37 percent reduction in area (number of slices) and 20-50 percent reduction in latency, with a reduced or comparable usage of block RAM (BRAM) and 18×18 hardware multipliers (HMUL). We demonstrate our tools (area estimation tool and bit-accurate value simulator) on realistic applications, such as digital sine/cosine waveform generator, matrix multiplication, QR decomposition, radiative Monte Carlo simulation, and vector normalization. LNS shows better efficiency than FLP in QR decomposition and vector normalization with 73.5 percent fewer slices and 34.1-43.7 percent higher throughput. On the accuracy side, LNS achieves better results than FLP with two fewer bits in the radiative Monte Carlo simulation.

The tool infrastructure enables us to fast prototype LNS FPGA applications, and compare between LNS and FLP implementations on area, performance, and accuracy. Thus, users can efficiently study the potential benefits of using LNS number representation on various applications.

2 Related Work

2.1 LNS Arithmetic Design

In general, existing LNS arithmetic designs fall into three major categories: direct lookup table [1], piecewise polynomial approximation [5], [6], [7], and digit-serial methods [2], [8], [9].

Evaluation through direct lookup table [1] is straightforward and efficient for low-precision computations. With the precision requirement increasing above 20 bits, the size of direct lookup tables increases exponentially and becomes impractical. Other approaches, such as piecewise polynomial approximations, need to be used instead. Lewis [5] uses the Lagrange interpolation to compute functions f_1 and f_2 for 32-bit LNS numbers. This design calculates the coefficients on the fly based on an interleaved memory, thus reducing the storage cost. The European Logarithmic Microprocessor project [6] uses linear Taylor interpolation to implement 20-bit and 32-bit LNS arithmetic units, to achieve a short latency. Meanwhile, to achieve the same accuracy as FLP counterparts, the design uses a table-based error correction scheme for both f_1 and f_2. On the platform of FPGAs, Lee and Burgess [7] implement the 32-bit LNS arithmetic functions using Chebyshev polynomial approximation, and provide a Better-Than-Floating-Point (BTFP) accuracy.

Other than polynomial approximation methods, there are also digit-serial methods (also known as online or iterative methods), which calculate the result digit by digit. Arnold proposes a Dual Redundant Logarithmic Number System [2], which calculates the addition/subtraction with an online algorithm. Chen et al. [8] propose a pipelined addition/subtraction unit for LNS numbers of very large word length. The algorithm, which is similar to the

COordinate Rotation DIgital Computer (CORDIC) method [10], approximates the functions with digit-serial sequences of computations $\prod_{j=1}^{N}(1+s_j2^{-j})$ and $-\sum_{i=1}^{N}(1+q_i2^{-i})$ $(s_j, q_i = 1, 0, or -1)$. The work is improved in [9] with the base-e exponential function implementation, and half of the pipeline stages can be replaced by one stage of multiplication-and-accumulate (MAC) operation. In general, the digit-serial methods require much less hardware resource than polynomial approximation methods. However, the digit-by-digit computation pattern brings a high latency.

Due to the singularity near zero, the LNS SUB function f_2 is more difficult to evaluate than the LNS ADD function f_1. There are approaches specially proposed for function f_2. In [11], LNS SUB function is transformed into $f_2(x) = \log_2(x) + \log_2(\frac{2^x-1}{x})$. Both $\log_2(x)$ and $\log_2(\frac{2^x-1}{x})$ are easier to evaluate than f_2 itself. In [12], subtraction is performed through additions by either looking up the addition table reversely (as $\log_2(1+2^x)$ and $\log_2(1-2^x)$ are inverse functions to each other), or using an iterative method based on the following equation: $\log_2(1-2^x) = -\sum_{m=0}^{\infty}\log_2(1+2^{x\cdot 2^m})$. Cotransformation methods are also proposed in [13] to avoid the singularity at zero.

2.2 Comparisons between LNS and FLP Representations

Based on the work of the LNS microprocessor [6], Matousek [14] presents a comparison between LNS and FLP numbers on the QR-decomposition-based recursive least-squares (QRD-RLS) algorithm implementation on FPGA. Based on a VHDL library of parameterized LNS arithmetic operations for bit widths less than 32, Detrey and Dinechin provide a comparison tool between LNS and FLP arithmetic [15], and apply the tool to examples such as a 3D transformation pipeline. Another work is the comparison between FLP and LNS for FPGAs in [4], which compares all the basic operators in both 32-bit and 64-bit bit widths.

In general, most of the existing comparison work either only study bit width values below 32, or focus on the counterparts of IEEE 754 single and double-precision FLP formats; they lack a systematic analysis and reconfigurable support for a wide range of LNS bit widths.

3 Evaluation of LNS Arithmetic Functions

Since FLP numbers dominate most existing hardware and software applications, a complete LNS arithmetic library also needs conversion functions between LNS and FLP numbers. Thus, we include the logarithmic function $f_3(x) = \log_2(x)$ and the exponential function $f_4(x) = 2^x$ in our target LNS library. For these four functions (f_1-f_4), our library generator produces LNS arithmetic units for integer bit width m from 4 to 11, fractional bit width f from 13 to 52, and all the possible combinations between them, i.e., over 300 different bit width settings. The largest bit width setting $(m = 11, f = 52)$ corresponds to the double-precision FLP numbers.

We adopt the piecewise polynomial approximation method as a general approach to evaluate all the four functions, i.e., we divide the entire evaluation range into a number of segments, and evaluate each segment with a different minimax polynomial, which provides the minimum-maximum-approximation error over the range [16].

3.1 Accuracy Requirement

For MUL/DIV operations, the accuracy of LNS is better than FLP, as it introduces no rounding errors, compared to a 2^{-f-1} relative rounding error for FLP MUL/DIV. However, LNS has a worse error behavior than floating point for ADD/SUB. The transcendental functions (f_1 and f_2) can only be approximated, which already bring a half unit-in-the-last-place (ulp) error for the rounding in the last step.

Existing work [5], [7], [6] reports that two or three extra bits need to be calculated in order to achieve a Better-Than-Floating-Point (BTFP) error behavior for LNS ADD/SUB. On the other hand, Arnold and coworkers [17], [18] propose that faithful rounding (one ulp error requirement) is good enough for some LNS applications, and can greatly save hardware resource.

TABLE 1
Number of Segments Required for a 32-bit LNS ADD Function

	BTFP designs			faithful designs	
	[5]	[6]	33-bit[a]	[18]	32-bit
# Segments	768	1536	416	234/256[b]	264
Algorithm	Lagrange	Taylor	minimax	Lagrange	minimax
Degree	two	one[c]	two	two	two

[a]Our conservative error model shows two extra bits are required to achieve BTFP accuracy. However, the exhaustive precision test (detailed in section IV-C) shows that one extra bit already provides BTFP accuracy.

[b]234 for multiple-of-three segmentation, and 256 for multiple-of-four.

[c]The design applies an error correction step that needs another multiplier. The complexity is similar to a degree-two polynomial.

In our previous error model [3], we show that two extra bits are needed to achieve a BTFP precision for LNS ADD/SUB units. However, our precision experiments (detailed in Section 4.3) show that one extra bit is already enough to achieve BTFP accuracy for the 32-bit case.

3.2 Segmentation Method

An investigation into the properties of the LNS ADD/SUB functions show a huge difference for the approximation difficulty in different ranges [3]. To deal with the fast variation of approximation difficulty, we use a nonuniform adaptive divide-in-halves segmentation approach for f_1 and f_2. A detailed description of the algorithm can be found in [3]. The basic idea is to divide the range into two halves. For the right half, we try to find a proper K value (the right half is divided into 2^K uniform segments) that meets the error requirement. We then try to handle the left half with the same K. If it also meets the error requirement for the left half, the segmentation is finished; otherwise, we divide the left part into halves and continue the process recursively. By using this segmentation method, the encoding of the coefficient table address can be performed using a leading-one detector.

Table 1 shows the number of segments needed to evaluate a 32-bit $(m = 8,\ f = 23)$ LNS ADD function (f_1) in our design, compared to other existing LNS adder implementations. Using degree-two minimax polynomial approximation, our design with one extra bit achieves BTFP accuracy with only 416 segments, compared to 768 segments in [5] and 1,536 segments in [6]. Our faithful design uses 264 segments, slightly more than the 234 or 256 segments in [18] (234 for a multiple-of-three segmentation and 256 for a multiple-of-four segmentation).

Compared with f_1 and f_2, conversion functions f_3 and f_4 have a small evaluation range, and the approximation difficulty does not change much over the range. Thus, we use uniform power-of-two segmentation for them.

3.3 Selection of Polynomial Degree

In our previous work [3], we select the polynomial degree based on the storage architecture on FPGAs. The BRAM on Xilinx FPGAs stores 18K bits, which can be mapped into different organizations from 512 36-bit values to 16,384 1-bit values [19]. To improve the utilization rate of the BRAM storage, one strategy is to keep the number of segments close to the power-of-two values from 512 to 16,384.

The other issue is to keep a balance point between the BRAMs and HMULs consumed for one arithmetic unit. A higher polynomial degree costs more HMULs and logic slices while a lower degree consumes more BRAMs to store the coefficients. Minimizing one type of resource may lead to too much consumption of the other type, and increase the total cost consumed for one LNS arithmetic unit.

TABLE 2
Examples of Determining the Polynomial Degree with Smallest Resource Cost for LNS ADD Function $f_1(x)$

Bit Width	P	n	N_{HMUL}		N_{BRAM}		Total	
			esti.	exp.	esti.	exp.	esti.	exp.
1:7:13	**1**	**144**	**1**	**1**	**2**	**2**	**3**	**3**
	2	24	2	2	3	3	5	5
1:8:23	1	6272	4	4	24	23	28	27
	2	**264**	**8**	**8**	**3**	**3**	**11**	**11**
	3	60	12	10	4	4	16	14
1:11:52	4	1856	36	33	30	27	66	60
	5	**480**	**45**	**42**	**12**	**10**	**57**	**52**

"esti." and "exp." refer to estimated and experimental results. "Total" is the sum of HMUL and BRAM numbers.

To determine the proper balance point for a given bit width value, we run the segmentation method with different polynomial degrees, and find out the number of segments for each different degree. Using the number of segments and bit width values, we apply (1) and (2) to estimate the BRAM and HMUL costs. In (1) and (2), FBW denotes the fractional bit width of the LNS number. P denotes the polynomial degree. N_{HMUL}, N_{BRAM}, and n represent the numbers of HMULs, BRAMs, and segments

$$N_{HMUL} = \left\lceil \frac{FBW}{18} \right\rceil^2 \times P, \tag{1}$$

$$N_{BRAM} = \begin{cases} \left\lceil \frac{FBW}{36} \right\rceil \times (P+1), & : \quad n \in (0,512], \\ \left\lceil \frac{FBW}{18} \right\rceil \times (P+1), & : \quad n \in (512,1{,}024], \\ \cdots & : \quad \cdots \\ \left\lceil \frac{FBW}{1} \right\rceil \times (P+1), & : \quad n \in (8{,}192,16{,}384]. \end{cases} \tag{2}$$

To estimate the number of HMULs, we assume the bit widths of all the multiplication operands are FBW (in practical designs, the bit widths of the variables are optimized to different values in a range close to FBW). Thus, a multiplication consumes $\lceil \frac{FBW}{18} \rceil^2 18 \times 18$ HMULs on Xilinx FPGAs [19]), and we need P multiplications for a degree-P polynomial. For estimation of BRAMs' cost, based on the number of segments, we determine whether we can put a 36, 18, 9, 4, 2, or 1-bit value in one BRAM, and compute the number of BRAMs accordingly.

When we investigate LNS arithmetic at a unit level, we can simply define the balance point as the polynomial degree with the least total number of BRAMs and HMULs. Table 2 shows an example of LNS addition function f_1 with a number of typical bit width settings. The estimation numbers given by the equations match the experimental results quite well. We select the polynomial degree with the least sum of estimated HMUL and BRAM numbers, shown in bold in the table. In all the cases shown in Table 2, the estimations and the experimental results pick the same polynomial degree as the best choice.

3.4 Error Ratio Adjustment

The error of a hardware function evaluation unit consists of two parts [16]: 1) approximation error: the error due to the mathematical approximation method, provided that we compute with infinite precision; 2) quantization error: the rounding and truncation errors due to finite precision of hardware number representation. We define G as the ratio between the approximation error requirement and the total error requirement. Our LNS arithmetic library generator requires the maximum error of the units to be less than one ulp. As the rounding in the last step can already bring a maximum rounding error of 0.5 ulp, the approximation error requirement should be less than 0.5 ulp, i.e., G should be less than 0.5.

TABLE 3
Area and Latency of 64-bit ($m = 11$, $f = 52$) LNS ADD Units with Different G Values

G	segments	slices	BRAMs	HMUL	latency
0.05	832	1079	17	42	72.9 ns
0.3	544	1103	17	42	73.8 ns
0.45	480	1134	10	42	75.7 ns

G indicates the ratio between the approximation error requirement and the total error requirement.

Based on experimental results, setting G to 0.3 minimizes the total cost in general cases. However, for cases with an "edge" segment number (the segment number is slightly larger than the supported power-of-two dimension sizes from 512 to 16,384), we perform an extra adjustment of the error ratio G to identify the most appropriate setting.

Table 3 shows an example of 64-bit LNS ADD units with different G values. With $G = 0.3$, 544 segments are needed to calculate 64-bit f_1 with a degree-five polynomial. Since 544 is slightly larger than the bound of 512, the BRAM has to organize the data in an address space of 1,024 elements, which wastes almost 50 percent of the storage. To reduce this big waste, we explore different G values from 0.05 to 0.45.

When the value of G decreases from 0.3 to 0.05, the number of segments increases from 544 to 832. However, the number of BRAMs does not change as they are both using the $1{,}024 \times 16$-bit configuration of the BRAM, and the number of slices only decreases by 24. On the other hand, when the value of G increases from 0.3 to 0.45, the number of segments drops below 512, and the number of BRAMs falls from 17 to 10. Meanwhile, the number of slices only increases by 31.

3.5 Evaluation of f_1 and f_2

For large x values, $y = x$ gives an accurate approximation for f_1 and f_2. Thus, we can find a point $x = 2^r$, which assures that for all the x values on the right side of this point, the difference between $y = x$ and f_1 is within the error requirement. For the left side of the point, we perform the divide-in-halves segmentation approach, and evaluate each segment using the minimax polynomial approximation.

To circumvent the zero singularity of f_2's evaluation, for x values smaller than 4, we use the function decomposition approach [7], [11] to transform $f_2 = \log_2(2^x - 1)$ into $f_2 = g + f_3 = \log_2(\frac{2^x - 1}{x}) + \log_2(x)$.

4 LNS Arithmetic Library Generator

4.1 General Structure

Our LNS arithmetic library generator uses three major software tools: Maple, Matlab, and A Stream Compiler (ASC) [20]. The Maple program performs the segmentation algorithm and generates the minimax polynomial coefficients. The Matlab program optimizes the bit widths of intermediate variables using the affine arithmetic technique and the adaptive simulated annealing (ASA) method [21], and generates the arithmetic designs described in the C++ syntax format of ASC.

Using the operator overloading mechanism of C++, we hide the implementation details under the symbols of $+, -, *, /$. Therefore, with the LNS library file included, users can design their target applications using LNS in a similar way as a normal software data type. Meanwhile, as ASC already supports the hardware floating-point (HWfloat) data type, the users can simply change the declarations of variables between HWfloat and HWlns to perform a comparison between FLP and LNS implementations of the same design.

TABLE 4
Exhaustive Precision Test Results for 32-bit LNS Arithmetic Units

LNS ADD	# BTFP	# NBTFP	e_{max}
32-bit 'faithful'	268415966	19490	6.5e-8
one extra bit	268435456	0	4.9e-8
LNS SUB	# BTFP	# NBTFP	e_{max}
32-bit 'faithful'	268430674	4783	5.96e-8
one extra bit	268435456	0	5.03e-8

"# BTFP" and "# NBTFP" refer to the number of BTFP and non-BTFP cases, respectively. "e_{max}" is the recorded maximum relative error.

4.2 BRAM Cost Minimization

As a BRAM supports two concurrent reading ports, a pair of identical arithmetic units can read the coefficients from the same BRAMs. On the other hand, if the number of segments or the bit width of the coefficient is small, we can organize the coefficients of two different degrees into the same BRAM and set up their reading addresses with a predefined offset.

Generally, we share BRAMs between identical LNS ADD and SUB units, as applications normally consume a large number of ADD and SUB. For conversion units (f_3 and f_4), we compact coefficients of different degrees into the same BRAM, as conversion functions have a smaller number of segments and smaller bit widths than LNS ADD/SUB.

4.3 Precision Test

Our library generator produces LNS units with errors less than one ulp. To perform a verification of the precision, we include a precision test in the Matlab functions that generate the arithmetic units. The precision test generates 100,000 uniform random values within the input range, and performs a bit-accurate circuit simulation to check whether the errors are within one ulp. For all our experiments of different bit widths, the test has never failed.

For the most typical 32-bit LNS units, we run exhaustive precision tests. For x larger than 32, we approximate functions $f_1(x) = \log_2(2^x + 1)$ and $f_2(x) = \log_2(2^x - 1)$ using $y = x$, and hence the relative errors of $f_1(x)$ and $f_2(x)$ are assured to be less than $|2^{\log_2(2^{32}\pm 1)} - 2^{32}|/2^{\log_2(2^{32}\pm 1)} \approx 2.33e-10$. The maximum relative error of 32-bit (single precision) FLP ADD/SUB operations is $2^{-24} \approx 5.96e-8$. Thus, for x larger than 32, our units are assured to provide BTFP accuracy.

For values below 32, we test for every possible input. For 23 fractional bits, the total number of values to test is $2^{28} = 268{,}435{,}456$. As shown in Table 4, the precision of our 32-bit "faithful" LNS units is already very close to BTFP accuracy. For LNS ADD and SUB units, only 19,490 and 4,783 cases out of 268,435,456 produce relative errors larger than the maximum FLP relative error. With one extra bit, both LNS ADD and SUB designs achieve BTFP accuracy in all possible cases.

4.4 Experimental Results for LNS Arithmetic Units

Using ASC, we map all the arithmetic units onto Xilinx Virtex-II XC2V6000 FPGA to test their performance and hardware cost. Compared with the LNS arithmetic designs in [4], our designs consume 5.9 to 37.2 percent fewer slices, 41.7 to 75 percent fewer HMULs, but consume more BRAMs for 64-bit ADD. For LNS SUB units, we use 25 percent fewer BRAMs and 62.5 percent fewer HMULs in 32-bit case. In 64-bit case, our SUB units consume more resources than [4]. For converters, our designs consume much less resources, and support larger number of units on one FPGA board. Meanwhile, for most units, our designs also provide 20-50 percent reduction in latency.

In practical arithmetic computation, we normally do not know the signs of the two operands in advance. Depending on whether they have the same sign or not, the operation of $+/-$ can be mapped into either LNS ADD or SUB function. Thus, we also generate mixed units that can perform both addition and subtraction (not concurrently). Compared with other existing mixed LNS units [5], [14], [7], our automatically generated units cost more HMULs and a similar number of BRAMs. However, our unit consumes the least amount of slices, and provides the shortest latency.

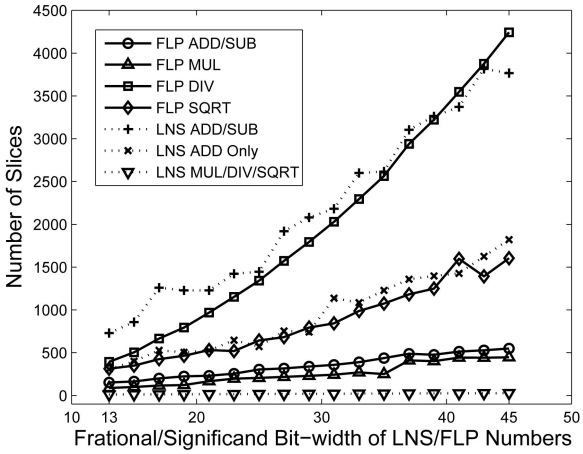

Fig. 1. Area comparison between LNS and FLP arithmetic units.

Note that a full set of comparison results can be found in our previous paper [3].

5 Comparing LNS and FLP Designs on FPGAs

Although our library generator supports LNS numbers up to 64 bits (with 52 fractional bits), we focus on the fractional bit widths from 13 to 45 for the comparison between LNS and FLP designs in this section. This is mainly because of the high resource consumption for LNS ADD and SUB units with a large fractional bit width around 50, which prevents high-precision designs from fitting into the FPGA.

5.1 Comparison of Arithmetic Units

Fig. 1 shows an area comparison between LNS and FLP arithmetic units. We apply the model based on relative silicon area proposed by Haselman et al. [4], to count a BRAM as 27.9 slices and an HMUL as 17.9 slices. With the fractional bit width changes from 13 to 45, the area of FLP ADD increases from 151 to 548 slices, while the area of LNS ADD/SUB unit increases from 729 to 3,766 slices, and the area cost of LNS ADD-only units increases from 305 to 1,820 slices. With the same change of bit width values, LNS MUL/DIV/SQRT only costs around 10-30 slices, while the area cost of FLP MUL increases from 87 to 445 slices, FLP DIV costs 451-4,365 slices, and FLP SQRT consumes 312-1,603 slices.

5.2 Area Cost Estimation Tool

A straightforward way to determine the area of an LNS design is to perform the mapping, placing, and routing using commercial CAD tools, and collect the results afterward. Depending on the circuit size, the process takes several to over 10 hours. To avoid the time spent on running the CAD tools, we develop an area cost estimation tool that acquires area cost results within a second, with an average error around 5 percent.

The area estimation tool serves as an extension of the ASC compiler. The tool derives a data flow graph of the design from the ASC description, and calculates the area cost of the arithmetic units based on the prerecorded exploration result of 231 different bit width settings (m from 4 to 10, f from 13 to 45). The calculation is performed through either a direct table lookup or an interpolation of the nearest values.

To illustrate the accuracy of our area estimation tool, we investigate a typical design, a degree-four polynomial (poly4). Horner's rule is used to evaluate the polynomial as follows: $y = ((c_d x + c_{d-1})x + \cdots)x + c_0$. We compare the estimated area costs against actual experiment results for 231 different bit width settings (m from 4 to 10, f from 13 to 45), and analyze the errors. For FLP designs, our area cost estimation gives a maximum relative error within 15.7 percent and an average error within 4.7 percent. However, the maximum and average errors for LNS cases increase to around 25 percent.

In order to improve the accuracy of LNS area estimation results, we apply a scaling procedure to the estimated area costs. Using a typical bit width setting (e.g., $m = 8$, $f = 23$), we map, place, and route a practical circuit, collect the area result, and calculate a scale factor between the experiment result and the estimation result. We can then use this factor to scale the estimation results for all the other bit width combinations. The scaled area modeling results for LNS designs fit the experimental results much better, providing a maximum error less than 14.2 percent and an average error less than 6.1 percent.

5.3 Bit-Accurate Simulator

We also develop a value simulator to investigate the precision behavior of different number representations. The simulator performs a bit-accurate value simulation of the basic hardware arithmetics, such as addition, subtraction, and rounding. Similar to the area modeling tool, the value simulator is also an extension of the original ASC compiler. In order to achieve a fast speed of the simulation, we use C++ 80-bit long double to perform the value calculations, and truncate or round the intermediate results at each step, according to the bit width settings of the variables.

The simulator currently supports all the basic arithmetic operations of fixed-point (HWfix) and floating-point (HWfloat) hardware variables with configurable bit widths up to 64-bit. Since the LNS arithmetic functions are evaluated using fixed-point variables, the simulator can also handle all the basic arithmetic operations of LNS hardware data types.

6 Case Studies

6.1 Brief Description

6.1.1 Digital Sine/Cosine Waveform Generator (DSCG)

A common tool in digital signal processing and communication applications, which generates a sequence of discrete sine or cosine values as follows:

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} \cos\theta & \cos\theta + 1 \\ \cos\theta - 1 & \cos\theta \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix}.$$

6.1.2 Matrix Multiplication (MM)

A fully pipelined 2×2 matrix multiplication unit, which consists of eight multiplications and four additions.

6.1.3 QR Decomposition (QRD)

A part of the QR-decomposition-based recursive least-squares (QRD-RLS) adaptive filtering algorithm. Similar to the work by Matousek et al. [14], we implement the diagonal arithmetic element of QRD algorithm, which includes one addition, three multiplications, two squares, two divisions, and one square root.

6.1.4 Radiative Monte Carlo Simulation (RMCS)

Gokhale et al. [22] present an acceleration of radiative Monte Carlo simulation on FPGA, using FLP numbers. The most inner loop, which is also the most computationally intensive part, is implemented on FPGA. RMCS performs 12 multiplications, one division, three additions, and seven subtractions.

6.1.5 Vector Normalization (VN)

A fully pipelined normalization for a vector of three elements, shown as follows:

$$w = \sqrt{a^2 + b^2 + c^2},$$
$$(a', b', c') = (a/w, b/w, c/w),$$

where (a, b, c) is normalized into (a', b', c').

Note that in this section, we only discuss the fully pipelined designs, which are mapped into Xilinx Virtex-4 FX100 FPGA.

6.2 Area Cost

Fig. 2 shows the comparison on area cost between LNS and FLP designs of the five case studies. Costs of different resources (logic slices, HMULs, and BRAMs) are converted into an equivalent total number of slices, using the same conversion rates as in Section 5.1. The scaled area estimation results, which are shown as the lines in the figure, provide a good match for the experimental results, which are shown as the markers. The maximum estimation error is around 15 percent and the average error is around 5 percent.

For DSCG, MM, and RMCS, which mainly consist of ADD/SUB, the LNS designs consume two to three times more resources than the FLP designs. For QRD and VN, which consume more multiplications, divisions, and square roots, the FLP designs consume two to four times more resources.

6.3 Accuracy

In our accuracy investigation, we use results from long-double software versions as the true values for error analysis. We then use our bit-accurate simulator to acquire the results for the designs with different number representations and different bit width settings.

Note that in our accuracy investigation, RMCS includes the conversion units that convert from/to FLP formats. DSCG, MM, QRD, and VN do not include the conversion units, and we use the arbitrary-precision computation utility in Maple to convert between the LNS and FLP values.

Fig. 3 shows the comparison on accuracy between LNS and FLP designs of the five case studies. For DSCG, MM, QRD, and VN, LNS provides a similar maximum and average error to the FLP designs with the same bit width. This shows that LNS arithmetic units with faithful roundings are able to provide similar accuracy as FLP units with rounding to nearest, which backs the arguments of [17].

RMCS is more complicated than the other three. Since it is performing a Monte Carlo simulation of photons' movements, the errors in each iteration get propagated to the next round and affect the accuracy of the final result. As shown in Fig. 3, for small bit width values, both FLP and LNS designs produce very large errors as the simulated photons take totally different movements from the accurate case. With the bit width increasing gradually, there emerges a turning point, where the simulation of photon movements turns from chaos to the accurate pattern and the error drops down dramatically. The LNS design gets to the dropping point with 16 fractional bits, while FLP design needs 17 mantissa bits. For the photon numbers calculated in the end of the simulation, the LNS design with 29 bits (1 sign bit, 8 integer bits, and 20 fractional bits) produces the same results as the long-double software version, while the FLP design needs 31 bits (1 sign bit, 8 exponent bits, and 22 significand bits) to produce the same results. Thus, even with two fewer bits, the LNS design manages to achieve the same or better accuracy than FLP.

6.4 Performance

Table 5 shows the performance results (given by Xilinx timing analysis tools) of the five designs for a typical 32-bit setting. Resource utilization details are also included in this table. For DSCG and MM, LNS designs consume over two times more resources than FLP designs, while providing 14.2-21.1 percent higher throughput. For QRD and VN, the FLP design consumes three times more resources than the LNS design, while the LNS design also improves the throughput by 34.1-43.7 percent.

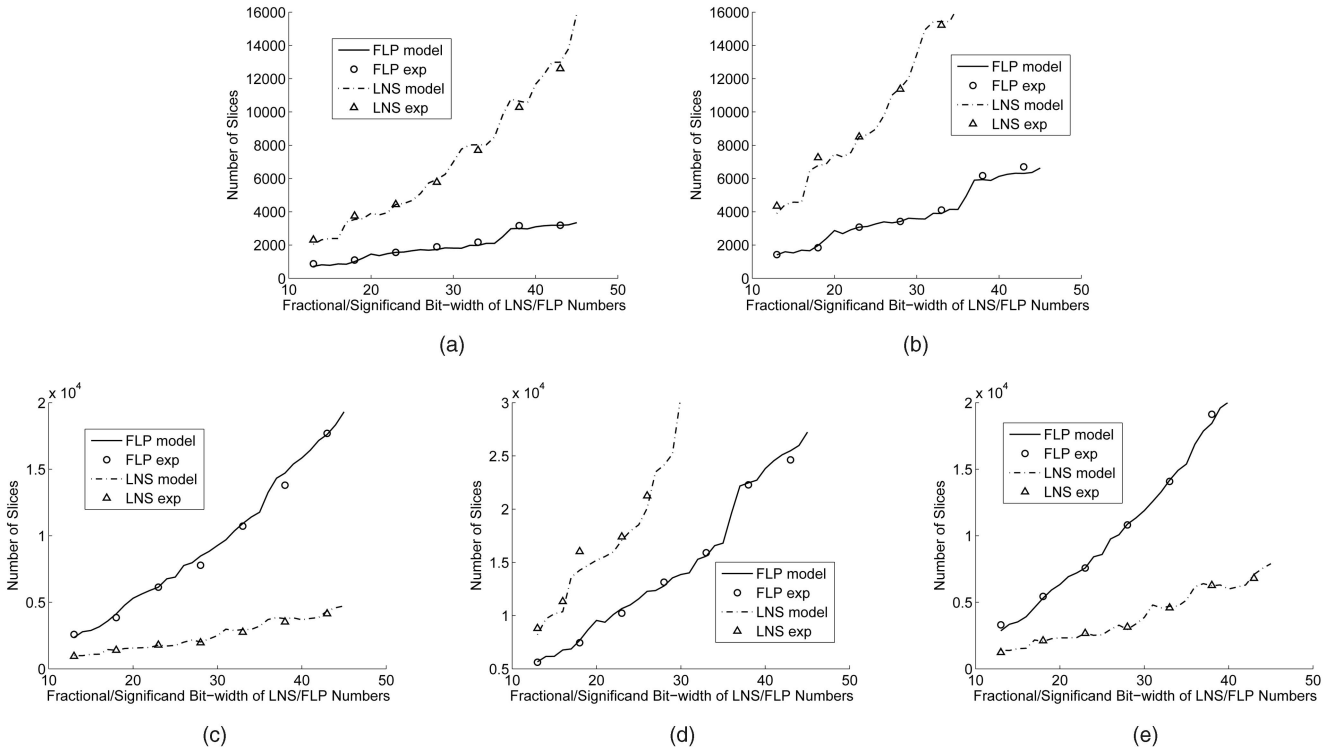

Fig. 2. Comparison of area cost between LNS and FLP designs. The line illustrates the scaled area modeling results (denoted as "model") while the markers show the experimental results (denoted as "exp") on a Virtex-4 FX100 FPGA. (a) Area costs of DSCG, LNS versus FLP. (b) Area costs of MM, LNS versus FLP. (c) Area costs of QRD, LNS versus FLP. (d) Area costs of RMCS, LNS versus FLP. (e) Area costs of VN, LNS versus FLP.

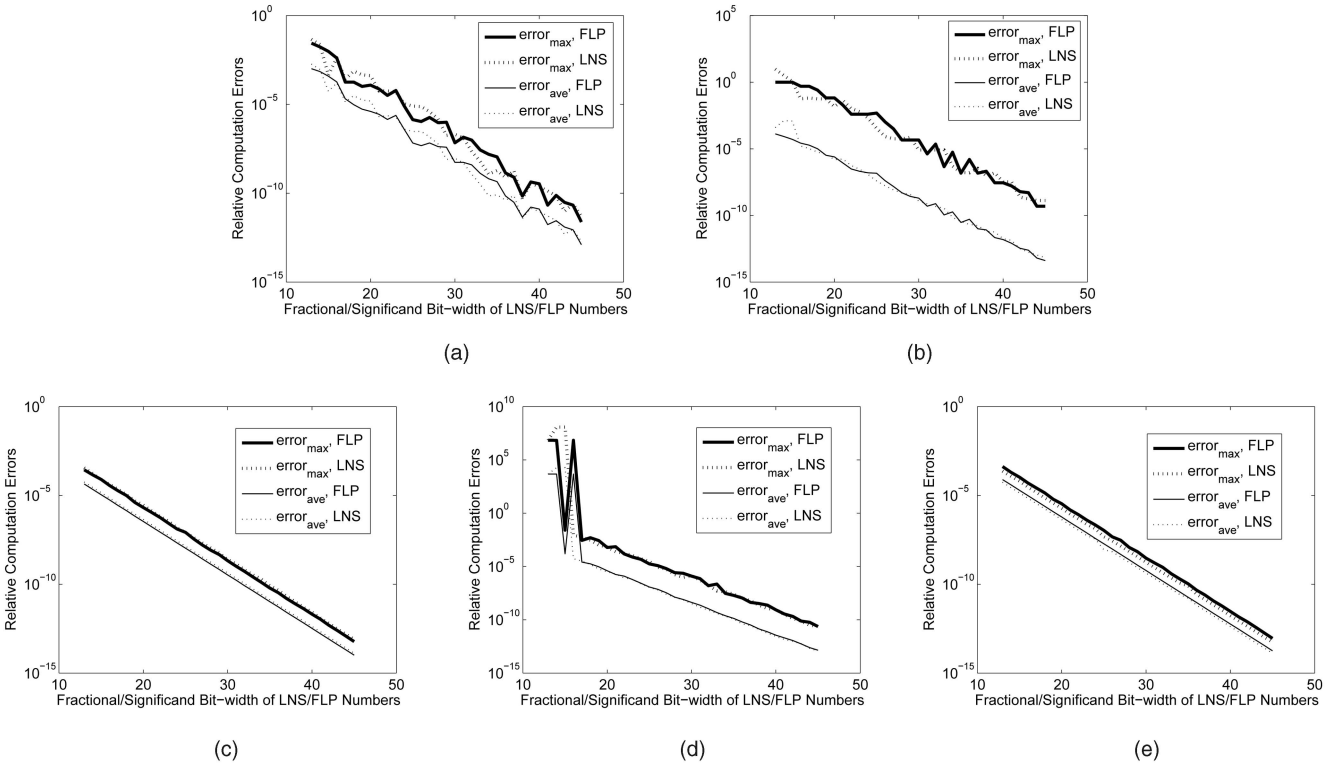

Fig. 3. Comparison of accuracy between LNS and FLP designs. We investigate both maximum and average relative errors. (a) Errors of DSCG, LNS versus FLP. (b) Errors of MM, LNS versus FLP. (c) Error of QRD, LNS versus FLP. (d) Error of RMCS, LNS versus FLP. (e) Error of VN, LNS versus FLP.

For RMCS, compared to FLP designs, LNS consumes 44.7 percent more slices, around 2.5 times HMULs with extra BRAMs. The throughput of the LNS design is 16.9 percent higher. Compared to the original FLP implementation of the RMCS computation core [22], the FLP circuits generated by our tool infrastructure consume 48 percent more slices, but use much fewer HMULs. Meanwhile, both our LNS and FLP circuits provide three to four times higher throughput than the original design.

TABLE 5
Comparison of Performance between LNS and FLP Designs

Number Format	BRAM	HMUL	slices	clk/ns	thr/MHz	Number Format	BRAM	HMUL	slices	clk/ns	thr/MHz
DSCG						**VN**					
LNS (1:8:23)	9	22	3798	6.2	161.4						
FLP (1:8:23)	0	16	1456	7.5	133.3	LNS (1:8:23)	3	8	2022	6.6	150.9
MM						FLP (1:8:23)	0	12	7642	9.5	105.0
LNS (1:8:23)	18	44	7218	6.2	161.3	**RMCS**					
FLP (1:8:23)	0	32	2501	7.1	141.2						
QRD						FLP (1:8:23) [22]	0	144	6758	29.9	33.4
LNS (1:8:23)	3	8	1562	6.4	155.6	LNS (1:8:23)	39	102	14485	7.3	137.5
FLP (1:8:23)	0	24	5886	8.4	116.0	FLP (1:8:23)	0	40	10012	8.5	117.6

The number format is described in the form of "sign:integer:fraction" for LNS, and "sign:exponent:significand" for FLP. "clk" and "thr" denote the clock cycle and throughput of the design.

7 Conclusion

This paper provides optimized LNS arithmetic targeting reconfigurable hardware designs. In particular, we introduce a general polynomial approximation approach for LNS arithmetic function evaluations, and develop a library generator that produces LNS arithmetic libraries containing $+, -, *, /$ operators as well as converters between FLP and LNS numbers. The generated arithmetic units are tested on FPGAs and in software simulation. Our evaluation shows that the generated LNS arithmetic units have significant improvements over existing LNS designs.

To facilitate comparison between LNS and FLP designs, we develop an area cost estimation tool that acquires results in less than a second, with a maximum error of 15.7 percent and an average error around 5 percent. We also provide a bit-accurate simulator to investigate the accuracy of LNS and FLP designs.

We demonstrate our tools on practical case studies, such as digital sine/cosine waveform generator, matrix multiplication, QR decomposition, radiative Monte Carlo simulation, and vector normalization. LNS shows better efficiency than FLP in QR decomposition and vector normalization with 73.5 percent less slices and 34.1-43.7 percent higher throughput. On the accuracy side, LNS achieves better results than FLP with two fewer bits in the radiative Monte Carlo simulation. The tool infrastructure enables us to fast prototype LNS FPGA applications and effectively study the logarithmic number representation and its tradeoffs in speed and size when compared with FLP designs.

Future work includes the following:

First, add the multipartite method [15] into our library generator. One problem with the current polynomial approach is that the number of segments is usually much smaller than 512, and the coefficients do not map to the BRAM structure quite efficiently. Using the multipartite method, which is proved to be suitable for low precision, we can make a more efficient utilization of the BRAMs and reduce the cost of HMULs, thus providing more efficient LNS units.

Second, extend the library generator to produce VHDL descriptions of the design as well as the ASC descriptions. One problem with ASC is that the scheduling and synthesis of the circuit are performed automatically by the compiler. By generating the VHDL descriptions, we can have more control over the scheduling and circuit architecture when needed.

Third, extend the library generator to produce multicycle LNS arithmetic units. By reusing the HMULs, the area cost of high-precision LNS units can be greatly reduced. The support for multicycle units also enables the users to study the tradeoff between performance and area cost.

Acknowledgments

The support of UK Engineering and Physical Sciences Research Council (grant number EP/C509625/1 and EP/C549481/1) and Xilinx, Inc., is gratefully acknowledged. Haohuan Fu was with the Department of Computing, Imperial College London, UK.

References

[1] E. Swartzlander, D. Chandra, H. Nagle, and S. Starks, "Sign/Logarithm Arithmetic for FFT Implementation," *IEEE Trans. Computers*, vol. 32, no. 6, pp. 526-534, June 1983.
[2] M. Arnold, T. Bailey, J. Cowles, and J. Cupal, "Redundant Logarithmic Arithmetic," *IEEE Trans. Computers*, vol. 39, no. 8, pp. 1077-1086, Aug. 1990.
[3] H. Fu, O. Mencer, and W. Luk, "Optimizing Logarithmic Arithmetic on FPGAs," *Proc. IEEE Int'l Symp. Field-Programmable Custom Computing Machines (FCCM)*, pp. 163-172, 2007.
[4] M. Haselman, M. Beauchamp, K. Underwood, and K. Hemmert, "A Comparison of Floating Point and Logarithmic Number Systems for FPGAs," *Proc. IEEE Int'l Symp. Field-Programmable Custom Computing Machines (FCCM)*, pp. 181-190, 2005.
[5] D. Lewis, "An Accurate LNS Arithmetic Unit Using Interleaved Memory Function Interpolator," *Proc. Symp. Computer Arithmetic (ARITH)*, pp. 2-9, 1993.
[6] J. Coleman, E. Chester, C. Softley, and J. Kadlec, "Arithmetic on the European Logarithmic Microprocessor," *IEEE Trans. Computers*, vol. 49, no. 7, pp. 702-715, July 2000.
[7] B. Lee and N. Burgess, "A Parallel Look-Up Logarithmic Number System Addition/Subtraction Scheme for FPGA," *Proc. Int'l Conf. Field-Programmable Technology (FPT)*, pp. 76-83, 2003.
[8] C. Chen, R. Chen, and C. Yang, "Pipelined Computation of Very Large Word-Length LNS Addition/Subtraction with Polynomial Hardware Cost," *IEEE Trans. Computers*, vol. 49, no. 7, pp. 716-726, July 2000.
[9] C. Chen and R. Chen, "Performance Improved Computation of Very Large Word-Length LNS Addition/Subtraction Using Signed-Digit Arithmetic," *Proc. IEEE Int'l Conf. Application-Specific Systems, Architectures and Processors (ASAP)*, pp. 337-347, 2003.
[10] J. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Trans. Electronic Computing*, vol. EC-8, pp. 330-334, 1959.
[11] V. Paliouras and T. Stouraitis, "A Novel Algorithm for Accurate Logarithmic Number System Subtraction," *Proc. IEEE Int'l Symp. Circuits and Systems (ISCAS)*, vol. 4, pp. 268-271, 1996.
[12] M. Arnold, "Iterative Methods for Logarithmic Subtraction," *Proc. IEEE Int'l Conf. Application-Specific Systems, Architectures and Processors (ASAP)*, pp. 315-325, 2003.
[13] M. Arnold, "Improved Cotransformation for LNS Subtraction," *Proc. IEEE Int'l Symp. Circuits and Systems (ISCAS)*, vol. 2, pp. 752-755, 2002.
[14] R. Matousek, M. Tichy, Z. Pohl, J. Kadlec, C. Softley, and N. Coleman, "Logarithmic Number System and Floating-Point Arithmetic on FPGA," *Proc. Int'l Conf. Field Programmable Logic and Applications (FPL)*, pp. 627-636, 2002.
[15] J. Detrey and F. Dinechin, "A Tool for Unbiased Comparison between Logarithmic and Floating-Point Arithmetic," *J. VLSI Signal Processing*, vol. 49, no. 1, pp. 161-175, Oct. 2007.
[16] J. Muller, *Elementary Functions: Algorithms and Implementation*. Springer, 2006.
[17] M. Arnold and C. Walter, "Unrestricted Faithful Rounding is Good Enough for Some LNS Application," *Proc. Symp. Computer Arithmetic (ARITH)*, pp. 237-246, 2001.
[18] M. Arnold, "Design of a Faithful LNS Interpolator," *Proc. Euromicro Symp. Digital Systems Design*, pp. 336-345, 2001.
[19] *Virtex-4 Family Overview*, Xilinx, Inc., http://www.xilinx.com, 2007.
[20] O. Mencer, "ASC, a Stream Compiler for Computing with FPGAs," *IEEE Trans. Computer-Aided Design*, vol. 25, no. 9, pp. 1603-1617, Sept. 2006.
[21] L. Ingber, *Adaptive Simulated Annealing (ASA) 25.15*, http://www.ingber.com, 2004.
[22] M. Gokhale, J. Frigo, C. Ahrens, J. Tripp, and R. Minnich, "Monte Carlo Radiative Heat Transfer Simulation on a Reconfigurable Computer," *Proc. Int'l Conf. Field Programmable Logic and Applications (FPL)*, pp. 95-104, 2004.