Optimizing Logarithmic Arithmetic on FPGAs

Haohuan Fu, Oskar Mencer, Wayne Luk

Department of Computing, Imperial College
London, United Kingdom
{hfu,oskar,wl}@doc.ic.ac.uk

Abstract

This paper proposes optimizations of the methods and parameters used in both mathematical approximation and hardware design for logarithmic number system (LNS) arithmetic. First, we introduce a general polynomial approximation approach with an adaptive divide-in-halves segmentation method for evaluation of LNS arithmetic functions. Second, we develop a library generator that automatically generates optimized LNS arithmetic units with a wide bit-width range from 21 to 64 bits, to support LNS application development and design exploration. The basic arithmetic units are tested on practical FPGA boards as well as software simulation. When compared with existing LNS designs, our generated units provide in most cases 6% to 37% reduction in area and 20% to 50% reduction in latency. The key challenge for LNS remains on the application level. We show the performance of LNS versus floating-point for realistic applications: digital sine/cosine waveform generator, matrix multiplication and radiative Monte Carlo simulation. Our infrastructure for fast prototyping LNS FPGA applications allows us to efficiently study LNS number representation and its tradeoffs in speed and size when compared with floating-point designs.

1. Introduction

Logarithmic number system (LNS) was first introduced into computer systems for processing of low-precision FFT in 1970s [1]. Due to its similar representation range and better relative error behavior, LNS has long been considered as an alternative to the floating-point (FLP) representation. However, since its addition and subtraction are difficult, LNS is not widely used in practical hardware designs.

To provide a reconfigurable LNS arithmetic library with convenient support for all various bit-width cases, we look into the characteristics of LNS arithmetic functions, design and adjust the approximation schemes accordingly, and develop a configurable LNS library generator. Our contributions are:

1. a systematic improvement of LNS arithmetic on FPGA, and a general polynomial approximation approach based on adaptive segmentation;
2. an easy-to-use LNS library generator with reconfigurable bit-width settings to support LNS application development and design exploration; the library generator allows various parameters, such as integer and fractional bits of the LNS number, degree of the polynomial to be used for the design, the error ratio between approximation error and quantization error, etc.

The basic arithmetic units are tested on practical FPGA board as well as software simulation. When compared with existing LNS designs [2], most of our LNS units achieve a 6% to 37% reduction in area (number of slices) and 20% to 50% reduction in latency, with a reduced or comparable usage of block RAM (BRAM) and 18 by 18 hardware multipliers (HMUL). We also demonstrate the performance of LNS versus floating-point on realistic applications, such as digital sine/cosine waveform generator, matrix multiplication, and radiative Monte Carlo simulation. Our infrastructure for fast prototyping LNS FPGA applications allows us to efficiently study the LNS number representation and their impact on other applications that LNS may achieve better performance.

A number of methods have been proposed to tackle the problem of LNS ADD/SUB design, and to provide implementations with acceptable hardware cost. For bit-widths around 10 bits, we can implement LNS ADD/SUB directly through look-up tables with Read-Only Memories (ROMs) [1], with $10 \times 2^{10} = 10$K bits of storage. When the precision requirement goes from 10 bits to 20, the size of the table increases exponentially to $20 \times 2^{20} = 20$M bits. Direct look-up table becomes impractical, thus other techniques are needed to make practical LNS designs. Piecewise polynomial approximations, such as Lagrange interpolation [3] and Chebyshev approximation [4], solve the problem by dividing the input value range into small segments and approximating the value with a different polynomial in each segment. On the other hand, on-line methods (also known as digit-serial or iterative methods) [5], [6],

calculate the result digit by digit, with a smaller memory cost but increased delay. Meanwhile, a number of different techniques are also proposed to enhance the design by either improving the accuracy or reducing the resource costs, such as error correction [7], coefficient generation on the fly [3], and function transformations [8], [9].

Based on the above work, some LNS arithmetic libraries are provided for FPGA platforms. Using the arithmetic designs of the European LNS microprocessor [7], Matousek et al. [10] present an arithmetic library on FPGA for 20-bit and 32-bit LNS numbers. J. Detrey et al. [11] provide a VHDL library of LNS operators with a configurable precision up to 32 bits. More recent work includes comparison study between FLP and LNS [2], which demonstrates a parameterized Verilog arithmetic library for both 32-bit and 64-bit LNS numbers. Most of the existing libraries either work on bit-width values below 32 bits, or focus on the counter-parts 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.

2. Background

2.1. LNS Representation and Arithmetic

Unlike the FLP numbers defined by IEEE 754 standard, there is no commonly accepted standard for LNS numbers. In this paper, we use the signed-logarithmic representation format similar to [5], 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 significand and one sign bit. Note that the first bit of the fixed-point logarithmic value is also a sign bit, while the other bits indicate the absolute magnitude.

Suppose a and b are logarithmic representations of positive numbers A and B ($A > B$), thus $A = 2^a$ and $B = 2^b$. The basic LNS arithmetic operations of these two numbers include:

MUL: $A \times B = 2^a \times 2^b = 2^{a+b}$.
DIV: $A \div B = 2^a \div 2^b = 2^{a-b}$.
ADD: $A + B = 2^a + 2^b = 2^{a+\log_2(1+2^{b-a})}$.
SUB: $A - B = 2^a - 2^b = 2^{b+\log_2(2^{a-b}-1)}$.

Thus, we can implement LNS MUL and DIV with fixed-point addition or subtraction, but for LNS ADD and SUB we need to evaluate two transcendental functions, $f_1(x) = \log_2(1+2^x)$ and $f_2(x) = \log_2(2^x - 1)$, which are difficult to approximate. The function $f_2(x) = \log_2(2^x - 1)$ and its derivatives have singularities at zero, which make the evaluation even more difficult. By determining which operand is larger before the addition/subtraction, we can assure $x \geq 0$ in the above two functions, thus need to evaluate f_1 and f_2 in the range of $[0, 2^m]$.

On the other hand, since FLP numbers are used for most of the existing hardware/software applications, a complete LNS arithmetic library should also contain conversion functions between LNS and FLP numbers. Thus, we also need to design the logarithmic function $f_3(x) = \log_2(x)$ and the exponential function $f_4(x) = 2^x$. As the exponential part of a IEEE 754 conformal FLP number directly maps to the integer part of its logarithmic representation, we only need to convert between the significand part of a FLP number (in the range of $[1,2)$) and its base-two logarithmic value (in the range of $[0,1)$). Hence, the evaluation range for f_3 and f_4 are $[1,2)$ and $[0,1)$ respectively.

2.2. Accuracy Requirement

The relative representation error of a floating-point number has a variation range of $(2^{-f-2}, 2^{-f-1}]$, while the LNS number has a constant relative error of $2^{2^{-f-1}} - 1 \approx ln2 \times 2^{-f-1} \approx 0.693 \times 2^{-f-1}$. Thus LNS has a inherent better worst-case relative error compared to FLP. For MUL/DIV operations, LNS also wins by introducing no rounding errors, compared to a 2^{-f-1} relative rounding error for FLP MUL/DIV. However, LNS becomes the worse one for the four LNS arithmetic functions (f_1 to f_4). These four transcendental functions can only be approximated, which already bring a half unit-in-the-last-place (ulp) rounding error at the last step of approximation.

Existing work [3], [4], [7] 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 et al. [12] propose that faithful rounding (the evaluation result is the nearest or the next nearest machine number representation) is good enough for some LNS applications, and can greatly save hardware resource compared with BTFP designs.

In our library generator, we require only faithful roundings for the evaluation of the four functions, which means the maximum error of the hardware designs for the four functions should be less than 1 ulp, i.e. 2^{-f}. As the rounding of the evaluation result already introduces an error of 2^{-f-1}, we require the error introduced by other parts to be below 2^{-f-1}.

If an application requires BTFP units, as we provide LNS arithmetic units for various bit-widths, we can simply modify the last rounding step of a faithful rounding design for a larger bit-width, so as to round to the re-

Table 1: Number of bits for the integer part (m) and the fractional part (f) of the LNS numbers used in this paper.

integer part	7	8	9	10	11
fractional part	13-23	23-32	32-42	42-52	52
sign bit	1	1	1	1	1
total bit-width	21-31	32-41	42-52	53-63	64

quired bit-width with BTFP accuracy. Suppose we modify a faithful rounding LNS arithmetic unit for $x+\Delta$ bits to get a BTFP LNS arithmetic unit for x bits. In logarithmic value domain, the error bound for the parts excluding the last rounding is $2^{-x-1-\Delta}$, and the error bound for the last rounding is 2^{-x-1}. Thus, the bound for the total error in logarithmic value domain is $2^{-x-1}+2^{-x-1-\Delta}$. In order to achieve BTFP maximum relative error for x bits, we have to assure $2^{2^{-x-1}+2^{-x-1-\Delta}}-1<2^{-x-1}$, which approximates as $ln2\times(2^{-x-1}+2^{-x-1-\Delta})<2^{-x-1}$, and gives $\Delta>1.18$. Thus, to achieve a BTFP LNS arithmetic unit for x bits, we utilize the faithful rounding unit for $x+2$ bits and modify the last step to round to x bits.

Another problem concerning the accuracy requirements is the ratio of different error parts. The error of a hardware function evaluation unit consists of two parts [13]: (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. Therefore, when we design the mathematical approximation approach for a function, we need to reserve error space for the hardware errors. For convenience, we define G as the ratio between the approximation error requirement and the total error requirement. Based on experiment results, we set G to be 0.3 in general cases. Thus, with 1 ulp accuracy requirement for the units, the bound for approximation error is 0.3 ulp, the rounding error in the last step takes up 0.5 ulp, and the bound for quantization error in other parts is 0.2 ulp.

3. Evaluation of LNS Arithmetic Functions

For LNS numbers with a small bit-width, existing work [1] already provides efficient hardware designs based on direct look-up table approaches. In this paper, we focus on arithmetic units for medium and large bit-widths from 21 to 64 bits. The values of integer bit-width m and fractional bit-width f used in our experiments are shown in Table 1.

In our library generator, we adopt the piecewise polynomial approximation method as a general approach to evaluate all the four functions. To develop an approximation design, we need to firstly divide the evaluation range into a number of small segments, and then calculate the polynomial coefficients ($c_0 \cdots c_n$) of each segments. We can then use the coefficients to perform the evaluation with the polynomial $y=c_0\cdot x^n+\cdots+c_{n-1}\cdot x+c_n$. We use the minimax algorithm as the polynomial approximation method, because it provides the best maximum approximation error over all the methods. The following sections give a more detailed discussion about the techniques concerned.

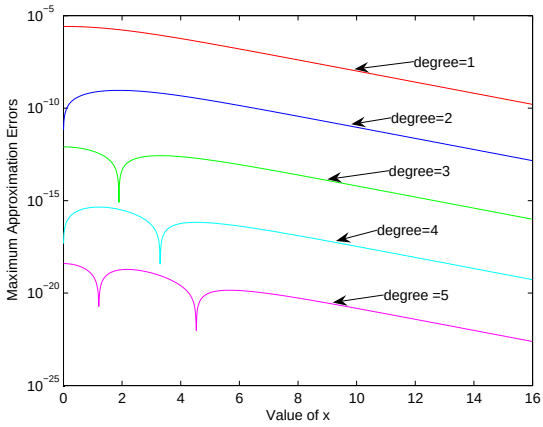


Figure 1: Approximation difficulty of function f_1, indicated by maximum approximation errors in each segment.

3.1. Segmentation Method

To find a proper segmentation approach to divide the evaluation range, we first check the approximation difficulty of the four functions by investigating their approximation errors over the range with a fixed approximation setting.

Figure 1 shows the results for function f_1 of 32-bit LNS numbers. We divide the range [0, 16] into uniform segments with length of 2^{-4}, and record the maximum approximation error of each segment using degree-one to degree-five polynomials. The error values provide a rough indication of the approximation difficulty within this segment. For all the five different degrees, when x goes from 16 to 0 (or from 16 to 2 for degree-two), the error values increase by more than 10^5 times. Function f_2 shows an even faster increase in the error values. Dividing the range [0, 16] into uniform segments with length of 2^{-6}, when x goes from 16 to 0, the error values increase by more than 10^{15}.

To deal with the fast variation of approximation difficulty in different ranges, we use a non-uniform adaptive divide-in-halves segmentation approach for f_1 and f_2. Figure 2 gives a detailed description of the approach in a right-to-left manner, which suits functions that become more difficult to approximate from right to left. Generally, we divide the range into two halves, and try to approximate the

Segmentation of function $f(x)$ in [a, b] with an error requirement of E.

```
begin = a; end = b; K = 1;
while ( end != begin )
{    //handle the right half.
   mid = (begin+end)/2;
   seg_len = (b−a)/2^K;
   compute the max_error for approximation of segment [mid, mid+seg_len];
   if ( max_error ≥ E ) {
      //test fails,need to compute with more segments;
      K = K+1;
      continue; //jump to the next iteration of while
   }
   else {
      //test passes, divide with the current K;
      divide [mid, end] into 2^K uniform segments;
      end = mid;
   }

   //handle the left half.
   compute the max_error for approximation of segment [begin, begin+seg_len];
   if ( max_error ≥ E ) {
      //test fails, jump to the next iteration;
      continue;
   }
   else {
      //test passes, segmentation is finished;
      divide [begin, mid] into 2^K uniform segments;
      end = begin;
   }
}
```

Figure 2: Right-to-left adaptive divide-in-halves segmentation approach.

right half with a uniform segmentation using the current K (the segment length equals to the length of the whole evaluation range divided by 2^K). If it fails to meet the error requirement, we increase the K and try again. If it succeeds, we try to handle the left half with the same K. If it also succeeds for the left half, the segmentation is finished; otherwise, we divide the left part into halves and continue the process recursively. For the ranges where the approximation difficulty increases monotonically, we only need to compute the maximum approximation error of the leftmost segment to check whether the segmentation with the current K satisfies the error requirement.

Compared with f_1 and f_2, conversion functions f_3 and f_4 have a small evaluation range. Moreover, although their approximation errors increase or decrease monotonously, the variation is only seven times for f_3 and two times for f_4, compared to 10^5 for f_1 and 10^{15} for f_2. Thus, we use uniform power-of-two segmentation for them.

3.2. Selection of Polynomial Degree

In piecewise polynomial approximation, there is a clear tradeoff between the polynomial degree and the density of segmentation. Given a fixed error requirement, if we use a higher polynomial degree, the function can be approximated with a smaller number of segments. On the other hand, if the approximation is performed with a larger number of segments, a smaller polynomial degree can be used.

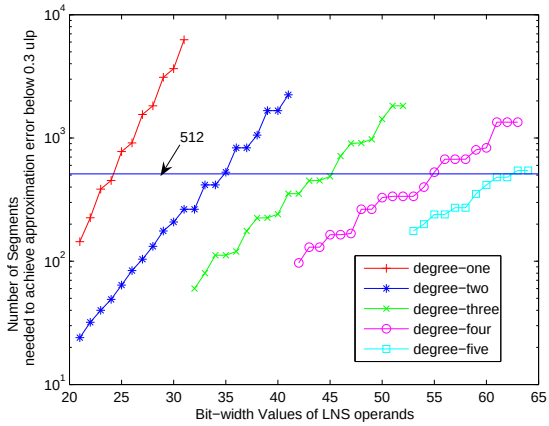


Figure 3: Number of segments needed to evaluate f_1 for different bit-widths using different polynomial degrees.

Figure 3 shows the number of segments needed to evaluate f_1 for different precision requirements with different polynomial degrees. For the memory units on FPGA, one Block RAM (BRAM) can hold up 18K bits, which is 512 36-bit values. Based on this consideration, we hold a general constraint that the number of segments shall be around 512, so that the coefficient data of one degree can be filled into one or two BRAMs. According to this constraint, as shown in Figure 3, degree-one polynomial is only used for bit-widths below 24, degree-two polynomial is suggested for bit-widths from 25 to 36, degree-three for bit-widths from 36 to 45, degree-four for 46 to 55, and degree-five for 56 to 64. Function f_2 shows a similar case as f_1, while f_3 and f_4 are easier to approximate and can be generally handled with smaller degrees than f_1 and f_2.

3.3. Function Evaluation of f_1

As shown in Figure 1, the error decreases very quickly for all the degrees when x grows into a large value. This is because linear function $y = x$ gives a quite accurate approximation for f_1 for large x values. Therefore, we 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. The approximation of f_1 is then divided into three ranges:

- If $x \geq 2^r$, f_1 is approximated with $y = x$.
- If $2 \leq x < 2^r$, we use the adaptive divide-in-halves segmentation approach for f_1. From the far end of the evaluation range to the point $x = 2$, we gradually reduce the segment length to meet the precision re-

quirement. Meanwhile, the adaptive adjustments only take place at half points, which provide a relatively easy address encoding mechanism for the coefficient tables when we map the approximation method into hardware design.

- If $0 \leq x < 2$, for degree-one, three, four and five cases, we continue the adaptive divide-in-halves approach until the end. Although there are several turning points in degree-three, four and five cases, the errors decrease and increase again in a very fast manner, which make the change of adaptive pattern (right-to-left or left-to-right) not useful. In degree-two case, as the error keeps on decreasing after the point of $x = 2$, we originally try to perform adaptive segmentation in a left-to-right manner from 0 to 2. However, since the error value at the mid-point ($x = 1$) is still close to the error value at $x = 2$, the result of the divide-in-halves approach turns out to approximate the whole range with the same segment length. Hence, we move the dividing point from the mid-point to $x = 0.5$. Both [0, 0.5] and [0.5, 2] are approximated with uniform segmentation, but [0, 0.5] uses a smaller segment length than [0.5, 2].

3.4. Function Evaluation of f_2

Evaluation of f_2 is the most difficult part of the design of LNS arithmetic. Especially for the range near zero, the existence of singularity makes the effect of polynomial approximation very poor. In order to circumvent this difficulty, we use the function decomposition approach [4], [8] to transform $f_2 = log_2(2^x - 1)$ into $f_2 = g + f_3 = log_2(\frac{2^x-1}{x}) + log_2(x)$, thus f_2 can be evaluated through two other functions which are relatively easier to approximate.

To find out under what kind of condition we shall replace f_2 with g and f_3, we perform a similar approximation difficulty investigation as section 3.1 to f_2 and g. Function f_3 is not considered, as it can be shifted into a small range of [1, 2) and is much easier to approximate. As shown in Figure 4, we divide the evaluation range into uniform segments with length of 2^{-6}, and record the maximum approximation error of f_2 and g in each segment. When x gets close to zero, f_2 becomes much more difficult to approximate very fast. From the point around $x = 6.703125$, g becomes easier to approximate than f_2. However, as the error of g and f_3 are combined to make the total error, we require the error of g to be below half of the error of f_2. Based on these considerations, we use the decomposition for the range of [0, 4] (at point $x = 4$, error of g is about 16.5% of f_2), rather than the range of [0, 2] used in [4].

Similar to f_1, when x becomes very large, $y = x$ provides a good approximation of f_2, and we can also find out a point

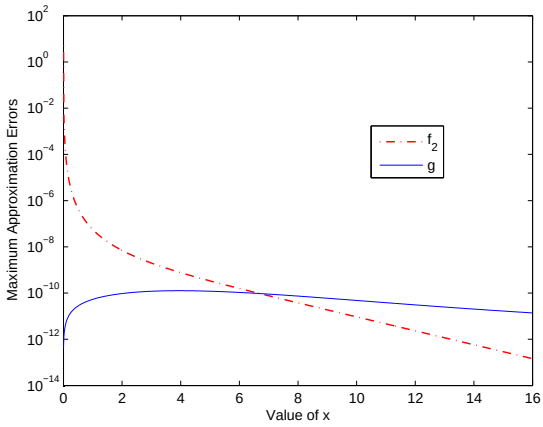


Figure 4: Approximation difficulty of f_2 and g, indicated by the maximum approximation error in each segment.

$x = 2^r$ as the starting point to use $y = x$ for evaluation. The approximation of f_2 is then divided into three ranges:

- If $x \geq 2^r$, f_2 is approximated with $y = x$.
- If $4 \leq x < 2^r$, we approximate f_2 with a polynomial. The right-to-left adaptive divide-in-halves approach introduced in section 3.1 produces the segmentation in this range.
- If $0 \leq x < 4$, approximation of f_2 is decomposed into g and f_3. As the approximation difficulty of both g and f_3 does not vary too much, we use uniform segmentation for function g and f_3.

4. LNS Library Generator

4.1. General Structure

The general structure of our library generator is shown in Figure 5. We use Maple as the mathematical approximation design tool, as it provides convenient support for polynomial approximation methods, such as Chebyshev and minimax algorithms. Our Maple programs divide the function evaluation range into proper segments with the segmentation approach discussed in section 3.1 and compute the coefficients of each segment. The result files of the Maple programs, which contain the information of all the segments' ranges, errors and coefficients, are stored into the coefficient file repository.

We use Matlab as the interface tool between the mathematical approximation design level and the hardware mapping design level. The Matlab functions read in the segmentation file from the coefficient file repository, produce

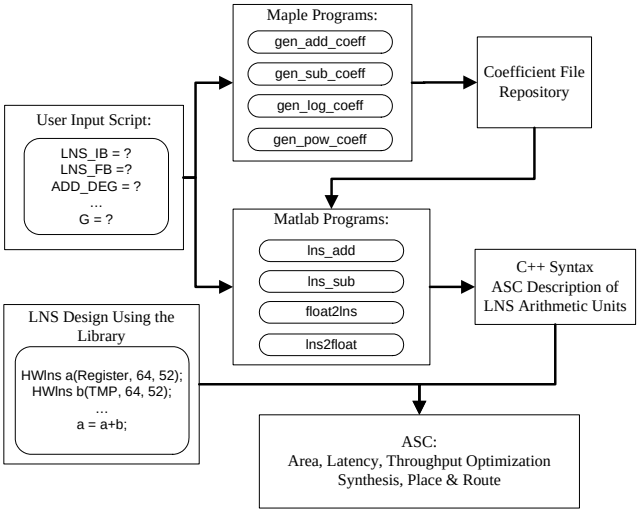


Figure 5: General structure of the library generator.

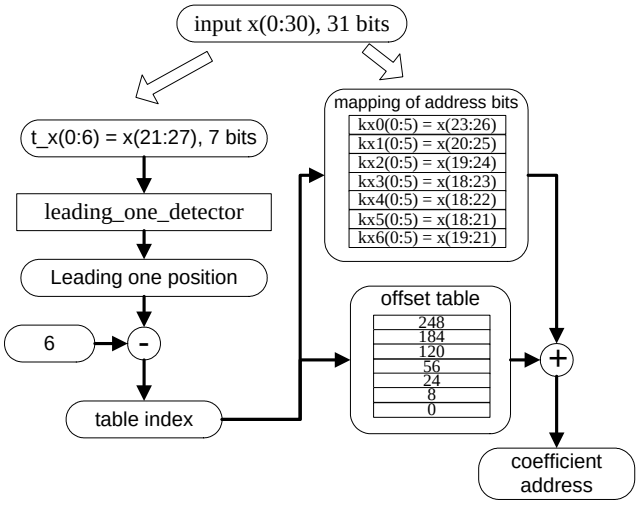


Figure 6: Address encoder circuit for 32-bit LNS ADD.

the design for the arithmetic units, and generate the description file of the design in the C++ syntax format of A Stream Compiler [14], ASC.

As a high-level FPGA programming tool, ASC provides hardware data-types, such as integer, fixed-point and floating-point number, with configurable bit-widths. It also provides configurable optimization mode, such as area, latency and throughput. By specifying the throughput mode, ASC automatically generates a fully-pipelined circuit design for the application. These features make ASC an ideal hardware compilation tool for our library generator.

To generate an LNS arithmetic library with specific settings, the user only needs to write a simple script file which specifies the bit-widths of the LNS number, the approximation error ratio G and the polynomial degrees to be used for the four functions. With these parameters, the script file calls the corresponding Maple and Matlab programs, which automatically generate the LNS library file that contains a full set of LNS arithmetic units described in ASC syntax.

With the LNS library file, users can then design their target applications using LNS as a normal data-type that supports general operators such as $+, -, *, /$.

4.2. Address Encoder of Coefficient Table

The first part of a polynomial approximation design is the address encoder of the coefficient tables. If the function uses a uniform power-of-two segmentation such as f_3 and f_4, after transforming the original range into [0,1], we can simply use the first K bits of the number as the address of the coefficient table. The address is more complicated to calculate for functions that use divide-in-halves adaptive segmentation. Suppose we segment the range [0, 2^m] with right-to-left adaptive divide-in-halves segmentation, the range is divided into a number of sections s_0, s_1, s_2, $\cdots$, s_n, and each section s_i is divided into 2^{K_i} uniform segments. Because the sections are actually formed in a divide-in-halves manner, the distribution shall be $\{s_0 = [0, 2^p], s_1 = [2^p, 2^{p+1}], \cdots, s_n = [2^{p+n-1}, 2^{p+n}]\}$. Thus, given an input value x, based on its leading one position, we can determine the section the input value falls in, and then use the next K_i bits after the leading one to find out its segment number in the section.

Figure 6 shows the hardware architecture of the address encoder for a 32-bit LNS adder. Two tables are used to calculate the coefficient table address based on the input value's leading one position, which identifies its corresponding section. The offset table records the address offset of each section, i.e. the address of the first element in that section. The table of address bits maps the corresponding K_i bits of the input value into a number. The sum of the section offset and the K_i address bits give the table address of the coefficients.

4.3. Circuit Structure of Arithmetic Units

The circuit structure of f_1, f_3 and f_4 units consists of three parts: an address encoder, coefficient tables, and the polynomial calculation module. Using the result of the address encoder, we can then fetch the coefficients from the BRAM and calculate the value of the polynomial. We evaluate an n-degree polynomial using Horner's method, $y = c_n + (c_{n-1} + \cdots + (c_1 + c_0 \cdot x) \cdot x) \cdots) \cdot x$. The hardware unit contains $n+1$ coefficients ($c_0 \cdots c_n$), and $2n$ intermediate results ($b_1 = c_0 \cdot x$, $d_1 = b_1 + c_1$, $b_2 = d_1 \cdot x$, $\cdots$, $b_n = d_{n-1} \cdot x$, $d_n = b_n + c_n$). To perform bit-width op-

timization on all the variables, we adopt the error model of [13] to give a strict bound to the error of the hardware design. Meanwhile, we build up a cost function that estimates the hardware overhead, which includes the number of slices occupied, number of BRAMs and number of hardware multipliers. With the error function and the cost function, we use the ASA method [15] to find the heuristic best solution in the multi-dimensional parameter space.

The circuit structure of f_2 is more complicated, as it is decomposed into g and f_3 for the range [0, 4]. It contains two sets of address encoders, coefficient tables and polynomial calculation modules, one used for f_2 ($x \geq 4$) and g ($0 < x < 4$), the other for f_3.

In the arithmetic operations of most applications, we 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 addition or subtraction. Thus, we also generate mixed units that can perform both addition and subtraction (not concurrently). The mixed units have three sets of address encoders and coefficient tables, but just two sets of polynomial calculation modules, as f_1 shares the polynomial calculation module with f_2 and g. The comparison result of the two operands' signs is used to determine which coefficients are used to perform the calculation.

The coefficients are stored in BRAMs on FPGA, which provide two ports that can be read concurrently. Based on this consideration, we provide an automatic BRAM sharing mechanism in our LNS library. When the program maps the operator into adders/subtractors, it keeps a record of the BRAMs' usage and try to share them among consecutive calls to the same type of unit. In applications with even number of $+/-$ operations, this mechanism can reduce the number of BRAMs by half.

4.4. Error Ratio Adjustment

As mentioned in section 2.2, in general we set G to be 0.3, which leaves the other 0.2 ulp for quantization errors. However, for cases with an 'edge' segment number, we perform adjustment of the error ratio G to identify the optimal setting. For instance, with G=0.3, 544 segments are needed to calculate 64-bit f_1 with a degree-five polynomial. Since 544 is above the bound of 512, a BRAM-based coefficient table has to organize the data in an address space of 1024 elements, which produces almost 50% waste of the BRAM resource. To reduce this big waste, we try with different G values from 0.05 to 0.45 with a step size of 0.05.

Table 2 shows the area and latency of LNS ADD designs with several different G values. 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 1024 by 16bit 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 also falls from 17 to 10. Although the number of slices increases by 31, it is small when compared with the BRAM reduction.

Table 2: 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	12	72.9 ns
0.3	544	1103	17	12	73.8 ns
0.45	480	1134	10	12	75.7 ns

4.5. Experiment Results for LNS Arithmetic Units

Using ASC, we map all the arithmetic units onto the Sepia card [16] and Xilinx Virtex-II XC2V6000 FPGA to test their performance and hardware cost. Table 3 shows the area and latency of the LNS arithmetic units automatically produced by our generator, compared with the designs in [2]. There is no clear description about whether a faithful rounding or a BTFP accuracy is achieved in [2], thus we compare the design with both our faithful rounding and BTFP units. For LNS ADD units, our designs consume 5.9% to 37.2% fewer slices, 41.7% to 75% fewer HMULs, but consume more BRAMs for 64-bit ADD. For LNS SUB units, we use 25% fewer BRAM and 62.5% fewer HMULs in 32-bit case. In 64-bit case, our SUB units consume more resources than [2]. For convertors, our designs generally consume much less resources, and support larger number of units on one FPGA board. Meanwhile, for most units, our designs also provide 20% to 50% reduction in latency.

Most of the existing work on LNS arithmetic designs do not provide results of FPGA platforms, and usually only have designs below 32 bits. Based on the results summarized in [4], Table 4 gives a comparison of some existing designs with our BTFP mixed units that can perform both ADD and SUB operations. Note that, Lewis' LNS design [3] is not implemented on FPGA, but Arnold [12] gives an estimation for the hardware cost of its data-path in slices. For the estimation of BRAM cost, Lewis' design requires 91K bits of ROM for the 'weak' error mode (relax the error requirement for f_2 when x gets near zero), which is equivalent to about five 18Kbit BRAM. Compared with these designs, our automatically generated units consume the most HMULs and a medium number of BRAMs. However, our unit consumes the least amount of slices, and provides the shortest latency.

The other advantage of our library generator is the automatic generation of different bit-width arithmetic units

Table 3: Area and latency of our auto-generated LNS arithmetic units, compared with the designs in [2]. F stands for our faithful rounding units (1 ulp error bound in logarithmic domain) while B stands for our BTFP units (less than 0.5 ulp error bound in floating-point domain). Number of units per FPGA is calculated based on Xilinx Virtex-II XC2V6000.

		LNS ADD			LNS SUB			FLP to LNS			LNS to FLP		
		[2]	F	B	[2]	F	B	[2]	F	B	[2]	F	B
32-bit LNS $m=8, f=23$	slices #	**750**	**471**	**490**	838	989	1044	163	286	327	236	376	408
	BRAM #	**4**	**3**	**3**	**8**	**6**	**6**	**24**	**2**	**2**	**4**	**2**	**2**
	HMUL #	**24**	**8**	**6**	**24**	**9**	**9**	0	3	3	**20**	**3**	**3**
	latency (ns)	**91**	**48**	**58**	95	90	91	**76**	**51**	**52**	**72**	**47**	**46**
	units per FPGA	**6**	**18**	**24**	**6**	**16**	**16**	**6**	**48**	**48**	**7**	**48**	**48**
64-bit LNS $m=11, f=52$	slices #	**1944**	**1727**	**1830**	2172	3140	3410	**7631**	**1029**	**1063**	**3152**	**996**	**1026**
	BRAM #	8	10	18	16	20	30	0	5	5	**14**	**5**	**5**
	HMUL #	**72**	**42**	**42**	72	84	101	0	36	33	**226**	**30**	**30**
	latency (ns)	**107**	**81**	**84**	110	98	122	**380**	**77**	**73**	84	111	111
	units per FPGA	2	3	3	1	1	1	4	4	4	**0**	**4**	**4**

Table 4: Area and latency of our 32-bit mixed BTFP LNS arithmetic units, compared with other existing designs.

design	Accuracy	BRAM	HMUL	slices	latency
MIX	BTFP	9	11	1066	83
[3]	BTFP	5	0	2300	x
[10]	BTFP	96	0	1300	160
[4]	BTFP	6	4	1210	125
[17]	faithful	0	0	3904	97

according to the user requirement. Figure 7 shows the latency and hardware cost of our ADD/SUB units at typical bit-widths. The basic arithmetic unit designs are tested on Xilinx Virtex-II XC2V6000 FPGA as well as software simulation. When the bit-width increases from 21 to 64, the consumed slices increase from 0.8% to 5.1% of Virtex-II XC2V6000 for ADD, and from 1.6% to 9.3% for SUB. BRAMs increase from 1.4% to 9.0% for ADD (64-bit ADD use fewer BRAMs than 53-bit, because we perform the G ratio adjustment for 64-bit case), and from 2.1% to 14.9% for SUB. These costs are acceptable for applications with various accuracy requirements. However, the usage of HMULs increases from 0.7% to 29.2% for ADD and from 2.8% to 58.3% for SUB. This makes it difficult to map high precision LNS arithmetic units onto an FPGA board.

5. Benchmark Performance Evaluation

5.1. Evaluation Approach

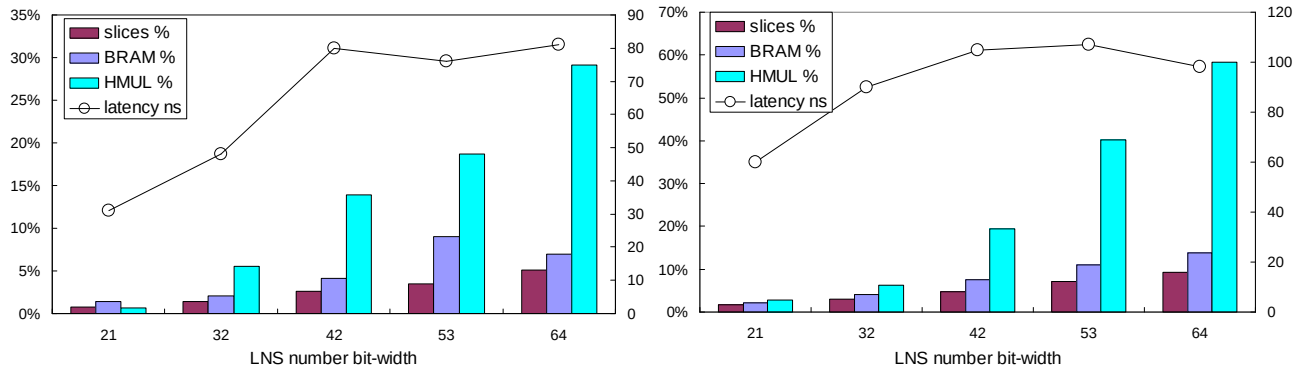
With the automatic generation of LNS library and the easy-to-use programming interface of ASC, we can efficiently perform fast prototyping of LNS applications on FPGA. We implement a number of practical benchmarks, such as digital sine/cosine waveform generator (DSCG), matrix multiplier, and radiative Monte Carlo simulation, using different bit-width LNS units, to compare their accuracy, throughput and hardware cost with single and double precision FLP designs. All the designs are fully-pipelined in the throughput optimization mode of ASC.

To perform accuracy investigation, we develop a stream value simulator, which performs a strict value simulation of all the hardware arithmetics, such as addition, subtraction and roundings. The value simulator conforms to the C++ syntax of ASC, and provides value results of the circuit which is accurate to the bit-level. It enables us to analyze the maximum and average errors of LNS designs. For FLP designs, we assume that the accuracy behavior shall conform to IEEE 754 standard, and use C++ float and double data-types to provide an estimation of the errors. Values calculated with C++ 128-bit long double data-type are used as the true values for error analysis.

In DSCG and matrix multiplication, we compare our LNS design with the FLP designs in [18]. These circuits does not include convertors between LNS and FLP numbers. To compare the accuracy, we use 100-digit high-precision calculation of Maple to convert the LNS results into FLP values and evaluate the maximum and average errors. For the radiative Monte Carlo simulation, we compare with the FLP designs in [19]. As the design needs to communicate with software part that uses FLP, convertors from and to FLP numbers are included.

5.2. Comparison Results

DSCG: a DSCG generates a sequence of discrete values representative of a sine or cosine wave. The generator is a common tool in digital signal processing and communication applications. In our benchmark evaluation, we implement a simple generator as follows:



(a) LNS ADD units for different bit-width.

(b) LNS SUB units for different bit-width.

Figure 7: Hardware resource cost and latency of faithful rounding LNS arithmetic units for different bit-widths. The integer bits and fractional bits of each bit-width value are the same as the values shown in Table 1. The percentage of slices, BRAM and HMUL are calculated based on Virtex-II XC2V6000, which has 33792 slices, 144 BRAMs and HMULs.

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

LNS shows a much better accuracy in this application compared to FLP. As shown in Table 5, for single-precision cases, compared with 32-bit FLP design, 29-bit LNS design already provides 11.1% smaller average error and 14.7% higher throughput, with 9% more slices and 11 additional BRAM. 32-bit LNS design provides even better accuracy, but also consumes more hardware resources.

In 64-bit case, as mentioned in section 4.5, the high-precision arithmetic units have a large overhead of multipliers, because all the multiplications in our polynomial approximation use HMUL. A 64-bit mixed $+/-$ unit already contains 84 HMULs, thus only one unit can be supported on common FPGA boards. Since this application requires two mixed $+/-$ units, instead of implementing all of them with HMUL, we map 40% of the multipliers into 4-input look-up tables. This provides a 64-bit unit with 60 HMULs but more slices. Due to a big usage of large-bitwidth multipliers, our LNS designs consume much more resource than FLP designs in [18], but they provide a 15.9% higher throughput and also smaller errors. The LNS-61 design provides a better accuracy than LNS-64 in this benchmark, possibly because some roundings at 61-bit randomly produce better results.

Matrix Multiplication: as the whole processing of 3 by 3 matrix multiplication is generally too large to fit in one FPGA, we compute one vector multiplication at one time, with a state machine to calculate the matrix address and schedule the process of the whole multiplication. As shown in Table 5, our 32-bit LNS design provides a similar accuracy to 32-bit FLP, with 11.4% higher throughput, but consumes over 50% more slices. For 64-bit units, the LNS design again consumes more resource than FLP design due to the large number of multipliers needed for LNS ADD/SUB, but it provides a 14.6% higher throughput.

Radiative Monte Carlo Simulation: M. Gokhale et al. [19] presents an acceleration of Monte Carlo radiative heat transfer simulation on FPGA, with FLP numbers. The simulation traces photons emitted from the surfaces of a 2-D enclosure. The most inner loop, which is also the most computationally intensive part, is implemented on FPGA. It performs 12 multiplications, 1 division, 3 additions and 7 subtractions. Different from other benchmarks, we calculate the errors based on the count of photons collected at the end of the simulation rather than the values computed during the process.

As a 64-bit LNS design is too large to map into a FPGA board, only a 32-bit LNS design is implemented. Compared with the single-precision FLP design [19], our design doubles the throughput, uses 42% fewer HMULs, and has 16% smaller error in the photon counts, at the expense of consuming over three times more slices and an additional 49 BRAMs.

6. Conclusion

This paper provides optimized LNS arithmetic targeting reconfigurable hardware designs. In particular, we introduce a polynomial approximation approach based on adaptive segmentation, and develop a library generator for LNS application development and design exploration. The basic arithmetic units are tested on practical FPGAs as well as software simulation. We also evaluate our method to

Table 5: Hardware cost, performance and accuracy of LNS benchmark designs, compared with FLP designs in [18] and [19]. BR stands for number of BRAMs used, and HM stands for number of HMULs used.

Number Format	BR	HM	slices	clock cycle	Error max	Error average	Number Format	BR	HM	slices	clock cycle	Error max	Error average
DSCG 32-bit							**Matrix Multiplication**						
FLP-32[18]	0	16	4000	11.6	6.03E-5	2.52E-6	FLP-32[18]	0	12	3375	12.3	3.86E-4	1.43E-7
LNS-29	11	16	4360	9.9	5.71E-5	2.24E-6	LNS-32	13	18	5191	10.9	2.81E-4	1.51E-7
LNS-32	11	18	5149	9.9	5.09E-5	2.06E-6	FLP-64[18]	0	27	8618	19.32	1.98E-13	2.05E-16
DSCG 64-bit							LNS-64	34	120	27530	16.5	4.50E-14	2.30E-16
FLP-64[18]	0	36	9503	22.7	9.70E-14	3.98E-15	**Monte Carlo Simulation**						
LNS-61	32	108	25364	19.0	3.44E-14	2.46E-15	FLP-32[19]	0	144	6758	29.9	1.93E-3	6.56E-4
LNS-64	32	120	27996	19.1	7.31E-14	3.07E-15	LNS-32	49	84	23239	14.2	1.68E-3	5.52E-4

show that the generated LNS arithmetic units have significant improvements over existing LNS designs. Our infrastructure for fast prototyping LNS FPGA applications allows us to effectively study the logarithmic number representation and its tradeoffs in speed and size when compared with floating-point designs.

Current and future work includes further reducing the hardware overhead for large bit-width LNS arithmetic units, and investigating LNS on other applications that require more multiplication and division operations.

Acknowledgements. The support of UK Engineering and Physical Sciences Research Council (grant number EP/C509625/1) and Xilinx Inc. is gratefully acknowledged. We also thank Mark Shand of HP Labs for providing the Sepia card and his technical support.

References

[1] E. Swartzlander, D. Chandra, H. Nagle, and S. Starks, "Sign/Logarithm Arithmetic for FFT Implementation," *IEEE Trans. Comput.*, vol. 32, no. 6, pp. 526–534, 1983.

[2] M. Haselman, M. Beauchamp, K. Underwood, and K. Hemmert, "A Comparison of Floating Point and Logarithmic Number Systems for FPGAs," in *Proc. FCCM*, 2005, pp. 181–190.

[3] D. Lewis, "An Accurate LNS Arithmetic Unit Using Interleaved Memory Function Interpolator," in *Proc. ARITH*, 1993.

[4] B. Lee and N. Burgess, "A Parallel Look-up Logarithmic Number System Addition/Subtraction Scheme for FPGA," in *Proc. FPT*, 2003, pp. 76–83.

[5] M. Arnold, T. Bailey, J. Cowles, and J. Cupal, "Redundant Logarithmic Arithmetic," *IEEE Trans. Comput.*, vol. 39, no. 8, pp. 1077–1086, Aug. 1990.

[6] C. Chen, R. Chen, and C. Yang, "Pipelined Computation of Very Large Word-Length LNS Addition/Subtraction with Polynomial Hardware Cost," *IEEE Trans. Comput.*, vol. 49, no. 7, pp. 716–726, July 2000.

[7] J. Coleman, E. Chester, C. Softley, and J. Kadlec, "Arithmetic on the European Logarithmic Microprocessor," *IEEE Trans. Comput.*, vol. 49, no. 7, pp. 702–715, July 2000.

[8] V. Paliouras and T. Stouraitis, "A Novel Algorithm for Accurate Logarithmic Number System Subtraction," in *Proc. ISCAS*, vol. 4, 1996, pp. 268–271.

[9] M. Arnold, "Improved Cotransformation for LNS Subtraction," in *Proc. ISCAS*, vol. 2, 2002, pp. 752–755.

[10] R. Matousek, M. Tichy, Z. Pohl, J. Kadlec, C. Softley, and N. Coleman, "Logarithmic Number System and Floating-point Arithmetic on FPGA," in *Proc. FPL*, 2002, pp. 627–636.

[11] J. Detrey and F. de Dinechin, "A VHDL Library of LNS Operators," in *Proc. 37th Asilomar Conference on Signals, Systems and Computers*, vol. 2, 2003, pp. 2227–2231.

[12] M. Arnold and C. Walter, "Unrestricted Faithful Rounding is Good Enough for Some LNS Application," in *Proc. ARITH*, 2001, pp. 237–246.

[13] D. Lee, A. A. Gaffar, R. Cheung, O. Mencer, W. Luk, and G. Constantinides, "Accuracy guaranteed bit-width optimization," *IEEE Trans. Computer-Aided Design*, Nov. 2005.

[14] O. Mencer, "ASC, A Stream Compiler for Computing with FPGAs," *IEEE Trans. Computer-Aided Design*, 2006.

[15] D. Lee, A. A. Gaffar, O. Mencer, and W. Luk, "Optimizing hardware function evaluation," *IEEE Trans. Comput.*, vol. 54, no. 12, pp. 1520–1531, Dec. 2005.

[16] L. Moll, M. Shand, and A. Heirich, "Sepia: Scalable 3D Compositing Using PCI Pamette," in *Proc. FCCM*, 1999, pp. 146–155.

[17] J. Detrey and F. Dinechin, "A Tool for Unbiased Comparison between Logarithmic and Floating-point Airthmetic." http://www.ens-lyon.fr/LIP/Pub-/Rapports/RR/RR2004/RR2004-31.ps.gz.

[18] C. H. Ho, P. H. W. Leong, W. Luk, S. J. E. Wilton, and S. Lopez-Buedo, "Virtual Embedded Blocks: A Methodology for Evaluating Embedded Elements in FPGAs," in *Proc. FCCM*, 2006, pp. 35–44.

[19] M. Gokhale, J. Frigo, C. Ahrens, J. Tripp, and R. Minnich, "Monte Carlo Radiative Heat Transfer Simulation on a Reconfigurable Computer," in *Proc. FPL, LNCS 3203*, 2004, pp. 95–104.