A Comparison of Floating Point and Logarithmic Number Systems on FPGAs

Michael Haselman, Michael Beauchamp,
Aaron Wood, Scott Hauck
Dept. of Electrical Engineering
University of Washington
Seattle, WA
{haselman, mjb7, arw82, hauck}@ee.washington.edu

Keith Underwood, K. Scott Hemmert
Sandia National Labratories*
Albuquerque, NM
{kdunder, kshemme}@sandia.gov

Abstract

There have been many papers proposing the use of the logarithmic number system (LNS) as an alternative to floating point (FP) because of simpler multiplication, division and exponentiation computations. However, this advantage comes at the cost of complicated, inexact addition and subtraction, as well as the possible need to convert between the formats. In this work, we created a parameterized LNS library of computational units and compared them to existing FP libraries. Specifically, we considered the area and latency of multiplication, division, addition and subtraction to determine when one format should be used over the other. We also characterized the tradeoffs when conversion is required for I/O compatibility.

1 Introduction

Digital signal processing (DSP) algorithms are typically some of the most challenging computations. They often need to be done in real-time, and require a large dynamic range of numbers. FP or LNS in hardware fulfill these requirements. FP and LNS provide numbers with large dynamic range while the hardware provides the performance required to compute in real time. Traditionally, these hardware platforms have been either ASICs or DSPs due to the difficulty of implementing FP computations on FPGAs. However, with the increasing density of modern FPGAs, FP computations have become feasible, but are still not trivial. Better support of FP or LNS computations will broaden the applications supported by FPGAs.

FPGA designers began mapping FP arithmetic units to FPGAs in the mid 90's [3, 10, 11, 14]. Some of these were full implementations that handled denormalized number and exceptions, but most make some optimizations to reduce the hardware. The dynamic range of FP comes at the cost of lower precision and increased complexity over fixed point. LNS provide a similar range and precision to FP but may have some advantages in complexity over FP for certain applications. This is because multiplication and division are simplified to fixed-point addition and subtraction, respectively, in LNS. However, FP number systems have become a standard while LNS has only seen use in small niches.

Most implementations of FP on DSPs or microprocessors use single (32-bit) or double (64-bit) precision. Using an FPGA gives us the liberty to use a precision and range that best suits an application, which may lead to better performance. In this paper we investigate the computational space for which LNS performs better than FP. Specifically, we attempt to define when it is advantageous to use LNS over FP number systems.

2 Backgrounds

2.1 Previous Work

Since the earliest work on implementing IEEE compliant FP units on FPGAs [10] there has been extensive study on FP arithmetic on FPGAs. Some of these implementations are IEEE compliant (see section 3.1 below for details) [10, 11, 14] while others leverage the flexibility of FPGAs to implement variable word

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

sizes that can be optimized per application [3, 24, 17]. There has also been a lot of work on optimizing the separate operations, such as addition/subtraction, multiplication, division and square root [11, 23, 22] to make incremental gains in area and performance. Finally, there have been many applications implemented with FP to show that not only can arithmetic units be mapped to FPGAs, but that useful work can be done with those units [20, 21].

In recent years, there has been work with the logarithmic number system as a possible alternative to FP. There has even been a proposal for a logarithmic microprocessor [8, 27, 30] and ALU [28]. Most of this work though has been algorithms for LNS addition/subtraction [5, 7, 8] or conversion from FP to LNS [1, 19] because these are complicated operations in LNS. There have also been some previous studies that compared LNS to FP [9, 12, 4]. Coleman et al. showed that LNS addition is just as accurate as FP addition, but the delay was compared on an ASIC layout. Matousek et al [12] report on a specific application where LNS is superior to FP. Detrey et al [4] also compared FP and LNS libraries, but only for 20 bits and less because of the limitation of their LNS library.

2.2 Field Programmable Gate Arrays

To gain speedups over software implementations of algorithms, designers often employ hardware for all or some critical portions of an algorithm. Due to the difficulty of FP computations, FP operations often are a part of these critical portions and therefore are good candidates for implementation in hardware. Current technology gives two main options for hardware implementations. These are the application specific integrated circuit (ASIC) and the field programmable gate array (FPGA). While an ASIC implementation will produce larger speedups while using less power, it will be very expensive to design and build and will have no flexibility after fabrication. FPGAs on the other hand will still provide good speedup results while retaining much of the flexibility of a software solution at a fraction of the startup cost of an ASIC. This makes FPGAs a good hardware solution at low volumes or if flexibility is a requirement.

3 Number Systems

3.1 FP

A FP number F has the value [2]

$$F = -1^{S} \times 1.f \times 2^{E} \tag{1}$$

where *S* is the sign, *f* is the unsigned fraction, and *E* is the exponent, of the number. The mantissa (also called the significand) is made up of the leading "1" and the fraction, where the leading "1" is implied in hardware. This means that for computations that produce a leading "0", the fraction must be shifted. The only exception for a leading one is for gradual underflow (denormalized number support in the FP library we use is disabled for these tests [14]). The exponent is usually kept in a biased format, where the value of *E* is

$$E = E^{true} + bias. \tag{2}$$

The most common value of the bias is

$$2^{e-1} - 1, \tag{3}$$

where *e* is the number of bits in the exponent. This is done to make comparisons of FP numbers easier.

FP numbers are kept in the following format:

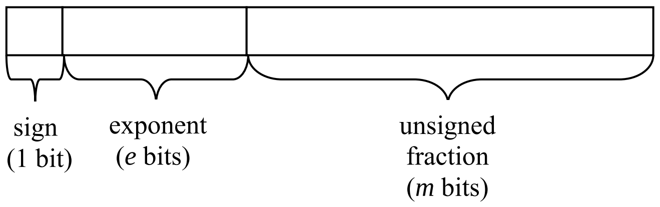


Figure 1: Binary storage format of the FP number.

The IEEE 754 standard sets two formats for FP numbers: single and double precision. For single precision, *e* is 8 bits, *m* is 23 bits and *S* is one bit, for a total of 32 bits. The extreme values of the exponent (0 and 255) are for special cases (see below) so single precision has a range of $\pm(1.0 \times 2^{-126})$ to $(1.11\ldots \times 2^{127})$, $\approx$ $\pm 1.2 \times 10^{-38}$ to 3.4×10^{38} and resolution of 10^{-7}. For double precision, where *m* is 11 and *e* is 52, the range is $\pm(1.0 \times 2^{-1022})$ to $(1.11\ldots \times 2^{1023})$, $\approx$ $\pm 2.2 \times 10^{-308}$ to 1.8×10^{308} and a resolution of 10^{-15}.

Finally, there are a few special values that are reserved for exceptions. These are shown in **Error! Reference source not found.**.

Table I.
FP exceptions (f is the fraction of M).

	$f = 0$	$f \neq 0$
$E = 0$	0	Denormalized
$E = \max$	$\pm\infty$	NaN

3.2 LNS

Logarithmic numbers can be viewed as a specific case of FP numbers where the mantissa is always 1, and the exponent has a fractional part [2]. The number *A* has a value of

$$A = -1^{S_A} \times 2^{E_A} \tag{4}$$

where S_A is the sign bit and E_A is a fixed point number. The sign bit signifies the sign of the whole number. E_A is a 2's complement fixed point number where the negative numbers represent values less than 1.0. In this way LNS numbers can represent both very large and very small numbers. The logarithmic numbers are kept in the following format:

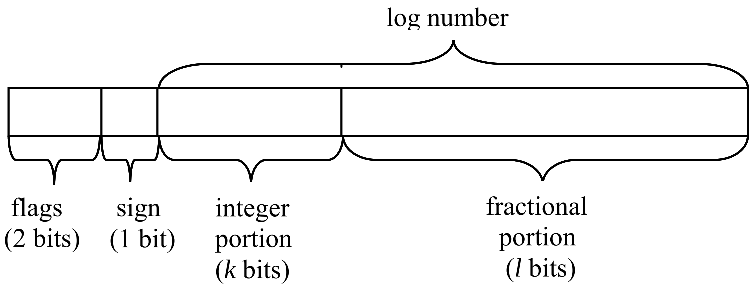


Figure 2: Binary storage format of the LNS number.

Since there are no obvious choices for special values to signify exceptions, and zero cannot be represented in LNS, we decided to use flag bits to code for zero, +/- infinity, and NaN in a similar manner as Detrey et al [4]. Using two flag bits would increase the storage requirements for logarithmic numbers but they give the greatest range available for a given bit width. If *k*=*e* and *l*=*m*, LNS has a very similar range and precision as FP numbers. For *k*=8 and *l*=23 (single precision equivalent) the range is $\pm 2^{-129}$ to $2^{128-ulp}$, $\approx\ \pm 1.5 \times 10^{-39}$ to 3.4×10^{38} (ulp is unit of least precision). The double precision equivalent has a range of $\pm 2^{-1025}$ to $2^{1024-ulp}$, $\approx\ \pm 2.8 \times 10^{-309}$ to 1.8×10^{308}. Thus, we have an LNS representation that covers the entire range of the corresponding floating-point version.

4 Implementation

A parameterized library was created in Verilog of LNS multiplication, division, addition, subtraction and converters using the number representation discussed previously in section 3.2. Each component is parameterized by the integer and fraction widths of the logarithmic number. For multiplication and division, the formats are changed by specifying the parameters in the Verilog code. The adder, subtractor and converters involve look up tables that are dependent upon the widths of the number. For these units, a C++ program is used to generate the Verilog code. The Verilog was mapped to a VirtexII 2000 (speed grade -4) using Xilinx ISE software. We then mapped two FP libraries [3, 14] to the same device for comparison. The results and analysis use the best result from the two FP libraries.

All of the units are precise to one half of an ulp. It has been shown that for some applications, the precision can be relaxed to save area while only slightly increasing overall error [29], but we felt that to do a fair comparison, the LNS units need to have a worst case error equal to or less than that of FP. The LNS units were verified by Verilog simulations. The adders and converters that used an algorithm instead of a direct look-up table have errors that depend on the inputs. In theses cases, a C++ simulation was written to find the inputs that resulted in the worst case error and then the Verilog was tested at those points.

Most commercial FPGAs contain dedicated memories in the form of random access memories (RAMs) and multipliers that will be utilized by the units of these libraries. This makes area comparisons more difficult because a computational unit in one number system may use one or more of these coarse-grain units while the equivalent unit in the other number system does not. For example, the adder in LNS uses memories and multipliers while the FP equivalent uses neither of these. To make a fair comparison we used two area metrics. The first is simply how many units can be mapped to the VirtexII 2000; this gives a practical notion to our areas. If a unit only uses 5% of the slices but 50% of the available block RAMs (BRAMs), we say that only 2 of these units can fit into the device because it is BRAM-limited. However, the latest generation of Xilinx FPGAs has a number of different RAM/multiplier to logic ratios that may be better suited for a design, so we also computed the equivalent slices of each unit. This was accomplished by determining the relative silicon area of a multiplier, slice and block RAM in a VirtexII from a die photo [16] and normalizing the area to that of a slice. In this model a BRAM counts as 27.9 slices and a multiplier as 17.9 slices.

4.1 Multiplication

Multiplication becomes a simple computation in LNS [2]. The product is computed by adding the two fixed point logarithmic numbers. This is from the following logarithmic property:

$$\log_2(x \cdot y) = \log_2(x) + \log_2(y) \qquad (5)$$

The sign is the XOR of the multiplier's and multiplicand's sign bits. The flags for infinities, zero, and NANs are encoded for exceptions in the same ways as the IEEE 754 standard. Since the logarithmic numbers are 2's complement fixed point numbers, addition is an exact operation if there is no overflow or underflow events. Overflows (correctly) result in ±∞ and underflow result in zero. Overflow events occur when the two numbers being added sum up to a number too large to be represented in the word width while underflow results when the added sum is too small to be represented.

FP multiplication is more complicated [2]. The two exponents are added and the mantissas are multiplied together. The addition of the exponents comes from the property:

$$2^x \times 2^y = 2^{x+y}. \tag{6}$$

Since both exponents each have a bias component one bias must be subtracted. The exponents are integers so there is a possibility that the addition of the exponents will produce a number that is too large to be stored in the exponent field, creating an overflow event that must be detected to set the exceptions to infinity. Since the two mantissas are in the range of [1,2), the product will be in the range of [1,4) and there will be a possible right shift of one to renormalize the mantissas. A right shift of the mantissa requires an increment of the exponent and detection of another possible overflow.

4.2 Division

Division in LNS becomes subtraction due to the following logarithmic property [2]:

$$\log_2\left(\frac{x}{y}\right) = \log_2(x) - \log_2(y). \tag{7}$$

Just as in multiplication, the sign bit is computed with the XOR of the two operands' signs and the operation is exact. Overflow and underflow are possible exceptions for LNS division as they were for addition.

Division in FP is accomplished by dividing the mantissa and subtracting the divisor's exponent from the dividend's exponent [2]. Because the range of the mantissa is [1,2) the quotients range will be in the range (.5,2), and a left shift by one may be required to renormalize the mantissa. A left shift of the mantissa requires a decrement of the exponent and detection of possible underflow.

4.3 Addition/Subtraction

The ease of the above LNS calculations is contrasted by addition and subtraction. The derivation of LNS addition and subtraction algorithms is as follows [2]. Assume we have two numbers A and B (|A|≥|B|) represented in LNS: $A = -1^{S_A} \cdot 2^{E_A}, B = -1^{S_B} \cdot 2^{E_B}$. If we want to compute $C = A \pm B$ then

$$S_C = S_A \tag{8}$$

$$E_c = \log_2 |(A \pm B)| = \log_2 \left| A\left(1 \pm \frac{B}{A}\right)\right| \tag{9}$$

$$= \log_2|A| + \log_2\left|1 \pm \frac{B}{A}\right| = E_A + f(E_B - E_A) \tag{10}$$

where $f(E_B - E_A)$ is defined as

$$f(E_B - E_A) = \log_2\left|1 \pm \frac{B}{A}\right| = \log_2\left|1 \pm 2^{(E_B - E_A)}\right|. \tag{11}$$

The ± indicates addition or subtraction (+ for addition, - for subtraction). The value of *f*(x) shown in Figure 3, where $x = (E_B - E_A) \leq 0$, can be calculated and stored in a ROM, but this is not feasible for larger word sizes. Other implementations interpolate the nonlinear function [1]. Notice that if $E_B - E_A = 0$ then $f(E_B - E_A) = -\infty$ for subtraction so this needs to be detected.

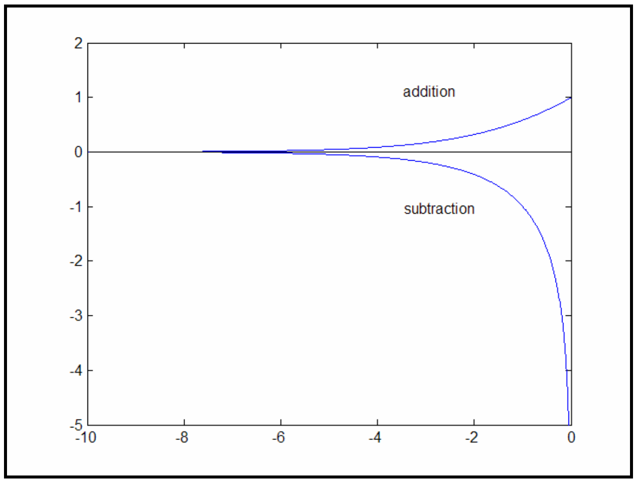


Figure3: Plot of *f*(x) for addition and subtraction [12].

For our LNS library, the algorithm from Lewis [5, 18] was implemented to calculate *f*(x) because of its ease of parameterization. This algorithm uses a polynomial interpolation to determine *f*(x). Instead of storing all values, only a subset of values is stored. When a particular value of *f*(x) is needed, the three closest stored points are used to build a 2nd order polynomial that is then used to approximate the answer. Lewis chose to store the actual values of *f*(x) instead of the polynomial coefficients to save memory space. To store the coefficients would require three coefficients to be stored for each point. To increase the accuracy, the x axis is broken up into even size intervals where the intervals closer to zero have more points stored (i.e., the steeper the curve, the closer the points need to be). For subtraction, the first interval is even broken up further to generate more points; hence it will require more memory. It is the extra memory requirement for subtraction that made us decide to implement the adder and subtractor separately because if only one computation is needed, a significant amount of overhead would be incurred to include the other operation.

FP addition on the other hand is a much more straightforward computation. It does however require that the exponents of the two operands be equal before addition [2]. To achieve this, the mantissa of the operand with the smaller exponent is shifted to the right by the difference between the two exponents. This requires a variable length shifter that can shift the smaller mantissa up to the length of the mantissa. This is significant because variable length shifters are costly to implement in FPGAs. Once the two operands are aligned, the mantissas can be added and the exponent becomes equal to the greater of the two exponents. If the addition of the mantissa overflows, it is right shifted by one and the exponent is incremented. Incrementing the exponent may result in an overflow event which requires the sum be set to $\pm\infty$.

FP subtraction is very similar to FP addition. One exception is that the mantissas are subtracted [2]. A big exception though is the possibility of “catastrophic” cancellation. This occurs when the two mantissas are nearly the same value resulting in many leading zeros in the difference mantissa. This will require another variable length shifter.

4.4 Conversion

Since FP has become a standard, modules that covert FP numbers to logarithmic numbers and vice versa may be required to interface with other systems. Conversion during a calculation may also be beneficial for certain algorithms. For example, if an algorithm has a long series of LNS suited computations (multiply, divide, square root, powers) followed by a series of FP suited computations (add, subtract) it may be beneficial to perform a conversion in between. However, these conversions are not exact and error can accumulate for multiple conversions.

4.5 FP to LNS

The conversion from FP to LNS involves three steps that can be done in parallel. The first part is checking whether the FP number is one of the special values in table 1 and thus encoding the two flag bits.

The remaining steps involve computing the following conversion:

$$\begin{aligned}\log_2(1.xxx \times 2^{\exp}) &= \log_2(1.xxx...) + \log_2(2^{\exp}) \\ &= \log_2(1.xxx...) + \exp\end{aligned} \quad (12)$$

Notice that $\log_2(1.xxx...)$ is in the range of (0,1) (notice that $\log_2(1) = 0$, but this would result in the zero flag being set) and exp is an integer. This means that the integer portion of the logarithmic number is simply the exponent of the FP minus a bias. Computing the fraction involves evaluating the nonlinear function $\log_2(1.x_1x_2...x_n)$. Using a look up for this computation is viable for smaller mantissas but becomes unreasonable for larger word sizes.

For larger word sizes up to single precision an approximation developed by Wan et al [1] was used. This algorithm reduces the amount of memory required by factoring the FP mantissa as (for $m = n/2$)

$$(1.x_1x_2...x_n) = (1.x_1x_2...x_m)(1.00...0c_1c_2c_m) \quad (13)$$

$$\begin{aligned}\log_2(1.x_1x_2...x_n) &= \log_2[(1.x_1x_2...x_m)(1.00...0c_1c_2c_m)] \\ &= \log_2(1.x_1x_2...x_m) + \log_2(1.00...0c_1c_2c_m),\end{aligned} \quad (14)$$

where

$$.c_1c_2..c_m = (.x_{m+1}x_{m+2}..x_{2m}) / (1 + .x_1x_2..x_m). \quad (15)$$

Now $\log_2(1.x_1x_2..x_m)$ and $\log_2(1.0...0c_1c_2...c_m)$ can be stored in a ROM that is $2^m \times n$.

In an attempt to avoid a slow division we can do the following:
If $c = .c_1c_2..c_m$, $b = .x_{m+1}x_{m+2}..x_{2m}$ and $a = .x_1x_2..x_m$ then

$$c = b/(1+a). \quad (16)$$

The computation can rewritten as

$$\begin{aligned}c = b/(1+a) &= (1+b)/(1+a) - 1/(1+a) \\ &= 2^{\log(1+b)-\log(1+a)} - 2^{-\log(1+a)},\end{aligned} \quad (17)$$

where log (1+b) and log (1+a) can be looked up in the same ROM for $\log_2(1.x_1x_2..x_m)$. All that remains in calculating 2^z which can be approximated by

$$2^z \approx 2 - \log_2[1 + (1-z)], \text{ for } z \in [0.1), \quad (18)$$

where $\log_2[1+(1-z)]$ can be evaluated in the same ROM as above. To reduce the error in the above approximation, the difference

$$\Delta z = 2^z - (2 - \log_2[1 + (1-z)]) \quad (19)$$

can be stored in another ROM. This ROM only has to be 2^m deep and less than m wide because Δz is small. This algorithm reduces the lookup table sizes from $2^n \times n$ to $2^m \times (m + 5n)$ (m from Δz ROM and 5n from 4 $\log_2(1.x_1x_2..x_m)$ and 1 $\log_2(1.0...0c_1c_2...c_m)$ ROMs). For single precision (23 bit fraction), this reduces the memory from 192MB to 32KB. The memory requirement can be even further reduced if the memories are time multiplexed or dual ported.

4.6 LNS to FP

In order to convert from LNS to FP we created an efficient converter via simple transformations. The conversion involves three steps. First, the exception flags must be checked and possibly translated into the FP exceptions shown in table 1. The integer portion of the logarithmic number becomes the exponent of the FP number. A bias must be added for conversion, because the logarithmic integer is stored in 2's complement format. Notice that the minimum logarithmic integer value converts to zero in the FP exponent since it is below FP's precision. If this occurs, the mantissa needs to be set to zero for FP libraries that do not support denormalized (see table 1).

The conversion of the logarithmic fraction to the FP mantissa involves evaluating the non-linear equation 2^n. Since the logarithmic fraction is in the range from [0,1) the conversion will be in the range from [1,2), which maps directly to the mantissa without any need to change the exponent. Typically, this is done in a look-up-table, but the amount of memory required becomes prohibitive for reasonable word sizes. To reduce the memory requirement we used the property:

$$2^{x+y+z} = 2^x \times 2^y \times 2^z \tag{20}$$

The fraction of the logarithmic number is broken up in the following manner

$$.a_1a_2..a_n = x_1x_2..x_{\frac{n}{k}}y_1y_2..y_{\frac{n}{k}}...z_1z_2..z_{\frac{n}{k}}... \tag{21}$$

where *k* is the number of times the integer is broken up. The values of $2^{.x_1x_2..x_{n/k}}$, $2^{.00..0y_1y_2..y_{n/k}}$ etc. are stored in *k* ROMs of size $2^k \times n$. Now the memory requirement is $\left(2^k \times k\right) \times n$ instead of $2^n \times n$. The memory saving comes at the cost of *k-1* multiplications. *k* was varied to find the minimum area usage for each word size.

5 Results

The following tables show the results of mapping the FP and logarithmic libraries to a Xilinx VirtexII 2000 – 4bf957. Area is reported in two formats, equivalent slices and number of units that will fit on the FPGA. The number of units that fit on a device gives a practical measure of how reasonable these units will be on current devices. Normalizing the area to a slice gives a value that is less device specific. For the performance metric, we chose to use latency of the circuit (time from input to output) as this metric is very circuit dependent. The latency was determined by placing registers on the inputs and outputs and reporting the summation of all register-to-register times as reported by the tools. For example, if a computation requires BRAMs, the time from input to the BRAM and from BRAM to output were summed up because BRAMs are registered.

We looked at four different bit width formats: 4_8, 4_12, 6_18 and 8_23. In some cases the mantissa of the single precision bit width is 24 because the algorithm required an even number of bits. In the format we use the first number is the bit width of the LNS integer or FP exponent and the second number is the LNS or FP fractions. 8_23 is the normal IEEE single precision FP, and a corresponding LNS format. We didn't go over single precision because the memory for the LNS adder and subtractor becomes too large to fit on a chip.

5.1 Multiplication

Table II.
Area and latency of a multiplication.

	4_8		4_12		6_18		8_23	
	FP	LNS	FP	LNS	FP	LNS	FP	LNS
slices	127	9	177	11	240	15	312	18
multipliers	1	0	1	0	3	0	4	0
18K BRAM	0	0	0	0	0	0	0	0
units per FPGA	56.0	1194.7	56.0	977.5	18.7	716.8	14.0	597.3
area(eq. slices)	144.9	9.0	194.9	11.0	293.7	15.0	383.6	18.0
latency(ns)	10.5	6.6	13.9	6.9	17.1	8.3	23.9	11.2

FP multiplication is complex because of the need to multiply the mantissas and add the exponents. In contrast, LNS multiplication is a simple addition of the two formats, with a little control logic. Thus, in terms of normalized area Table II shows that the LNS unit is 21x smaller for single precision (8_23). The benefit grows for larger precisions because the LNS structure has a near linear growth, while the mantissa multiplication in the floating-point multiplier grows quadratically.

5.2 Division

Table III.
Area and latency of a division.

	4_8		4_12		6_18		8_23	
	FP	LNS	FP	LNS	FP	LNS	FP	LNS
slices	148	9	153	13	301	15	412	18
multipliers	2	0	2	0	7	0	8	0
18K BRAM	1	0	1	0	1	0	7	0
units per FPGA	28.0	1194.7	28.0	827.1	8.0	716.8	7.0	597.3
area(eq. slices)	211.7	9.0	216.7	13.0	454.2	15.0	750.6	18.0
latency(ns)	21.1	6.1	32.5	6.6	48.7	7.9	54.4	11.1

FP division is larger because of the need to divide the mantissas and subtract the exponent. Logarithmic division is simplified to a fixed point subtraction. Thus in terms of normalized area Table III shows that the LNS unit is 42x smaller for single precision.

5.3 Addition

Table IV.
Area and latency of an addition.

	4_8		4_12		6_18		8_23	
	FP	LNS	FP	LNS	FP	LNS	FP	LNS
slices	114	33	148	440	247	827	306	629
multipliers	0	0	0	9	0	12	0	12
18K BRAM	0	3	0	0	0	0	0	4
units per FPGA	94.3	18.7	43.4	6.2	43.5	4.7	35.1	4.7
area(eq.slices)	114.0	116.8	248.0	601.1	247.0	1041.8	306.0	955.5
latency(ns)	21.6	16.3	25.3	60.3	30.2	68.5	33.2	72.3

As can be seen in Table IV, the LNS adder is more complicated than the FP version. This is due to the requirement of memory and multipliers to compute the non-linear function. This is somewhat offset by the need for lots of control and large shift registers required by the FP adder. With respect to the normalized area the LNS adder is 3.1x for single. One thing to note is that even though the LNS adder uses 4 18Kbit RAMs for 8_23, it does not mean that 72 Kbits were required. In fact only 20 Kbits were used for the single precision adder, but because our implementation used four memories to get the three interpolation points and simplify addressing, we have to consume four full memories. This is a drawback to FPGAs because in ASICs, the memories can be built to suit required capacity.

5.4 Subtraction

Table V.
Area and latency of a subtraction.

	4_8		4_12		6_18		8_23	
	FP	LNS	FP	LNS	FP	LNS	FP	LNS
slices	114	33	148	448	246	736	306	10981
multipliers	0	0	0	3	0	12	0	12
18K BRAM	0	3	0	8	0	12	0	4
units per FPGA	94.3	28.0	72.6	7.0	43.7	4.7	35.1	1.0
pct of FPGA	1.1	3.6	1.4	14.3	2.3	21.4	2.8	102.1
norm. area	114.0	88.8	148.0	725.1	246.0	1285.8	306.0	11307.5
latency(ns)	21	17	23.2	63.6	31.2	66.7	33.2	95.4

The logarithmic subtraction shown in Table V is even larger than the adder because of the additional memory needed when the two numbers are very close. FP subtraction is very similar to addition. In terms of normalized area, the LNS subtraction is 37x larger for single precision. The large increase in slice usage from 6_18 to 8_23 for LNS subtraction is because the large memories are mapped to distributed memory instead of BRAMs. Also note that the growth of the LNS subtraction and addition is exponential with the increase of the LNS fraction bit width. This is because as the precision increases, the spacing between the points used for interpolation must decrease in order to keep the error to one ulp. This means that the LNS subrtactor has very large memory requirements above single precision.

5.6 Convert FP to LNS

Table VI.
Area of a conversion from FP to LNS.

	4_8	4_12	6_18	8_24
slices	7	19	107	160
multipliers	0	0	0	0
18K BRAM	1	3	9	24
units per FPGA	56.0	18.7	6.2	2.3
area(eq. slices)	34.9	102.8	358.3	830.1

Table VI shows the area requirements for converting from FP to LNS. Converting from FP to LNS has typically been done with ROMs [2] for small word sizes. For larger words, an approximation as discussed in section 4.5 must be used.

5.7 Convert LNS to FP

Table VII.
Area and latency of a conversion from LNS to FP.

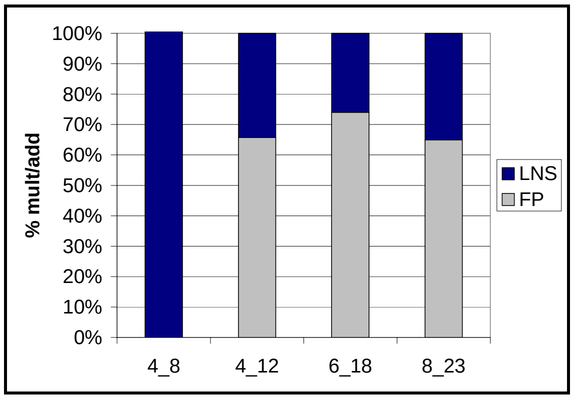
	4_8	4_12	6_18	8_24
slices	8	10	87	574
multipliers	0	0	10	48
18K BRAM	1	3	3	6
units per FPGA	56.0	18.7	5.6	1.2
area(eq. slices)	35.9	93.8	349.8	1600.7

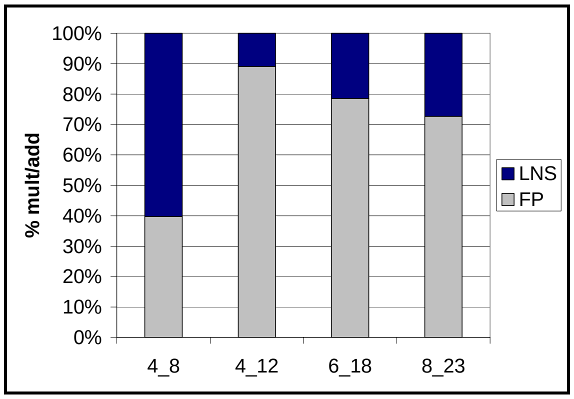
Table VII shows the area requirements for converting from LNS to FP. The area of the LNS to FP converter is dominated by the multipliers as the word sizes increases. This is because in order to offset the exponential growth in memory, the LNS integer is divided more times for smaller memories.

6 Analysis

6.1 Area benefit without conversion

The benefit or cost of using a certain format depends on the makeup of the computation that is being performed. The ratio of LNS-suited operations (multiply, divide) to FP-suited operations (add, subtract) in a computation will have an effect on the size of a circuit. Figure 4 shows the plot of the percentage of multiplies to addition that is required to make LNS have a smaller normalized area than FP. For example, for a single precision circuit (8_23), the computation makeup would have to be about 2/3 multiplies to 1/3 additions for the two circuits to be of even size if implemented in either format. More than 2/3 multiplies would mean that an LNS circuit would be smaller, while anything less means a FP circuit would be smaller. For 4_8 (4 bits exponent or integer, 8 bits fraction) the LNS multiplier and adder are smaller in equivalent slices so it doesn’t make sense to use FP at all in that instance. This is because it is possible to compute the non-linear function discussed in 4.3 with a straight look-up table without any interpolations. The implementation for the 4_8 format uses block RAM when equivalent slices are compared (a) and LUTs when percentage of FPGA used (b) is compared. At 4_12 precision, the memory requirements become too great and interpolation is required to do LNS addition. When the percentage of the FPGA used is evaluated in Figure 4b, the 4_8 format has a cutoff point of about 60% (i.e. 60% multiply 40% add). This is because the LNS adder uses memory which is compact but limited on a current generation FPGA.

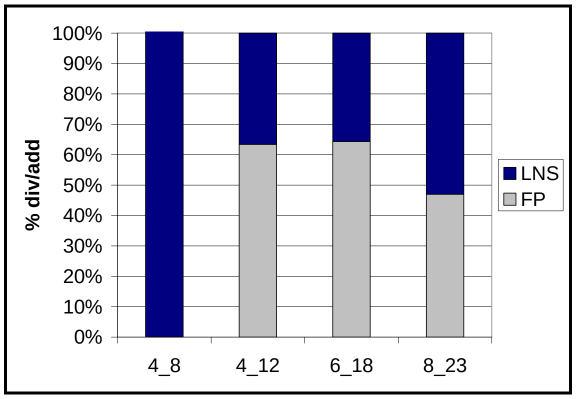


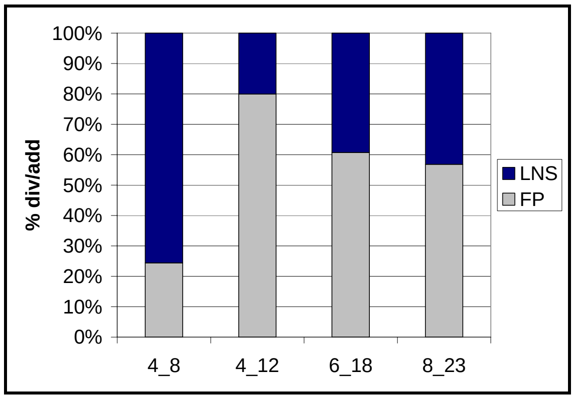


(a) (b)

Figure 4: Plot of the percentage of multiplies versus additions in a circuit that make LNS beneficial as far as equivalent slices (a) and the percentage of FPGA used (b). The bar labels are formatted as exponent/integer_fraction widths. For example, 8_23 is single precision.

Figure 5 shows the same analysis as Figure 4 but for division versus addition instead of multiplication versus addition. For division, the break even point for single precision is about 1/2 divides and 1/2 additions for equivalent slices (Figure 5a). Again, the 4_8 format has smaller LNS adders and dividers when analyzed with equivalent slices. The large decline in percentages from multiplication to division is a result of the large increase in size of the FP division over FP multiplication.

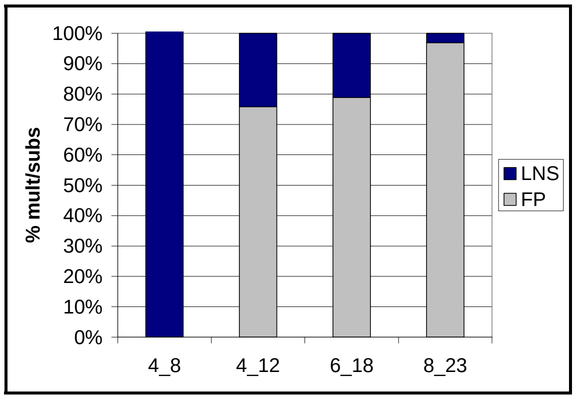




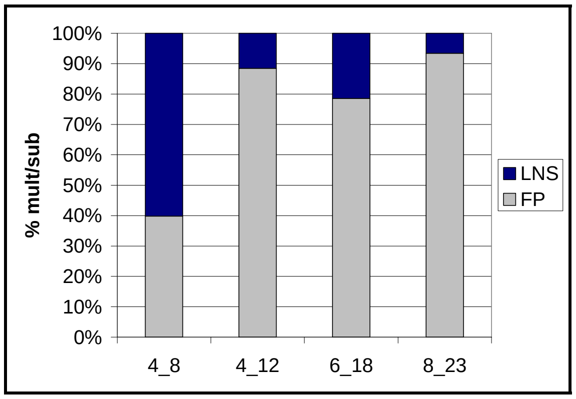
(a) (b)

Figure 5. Plot of the percentage of divisions versus additions in a circuit that make LNS beneficial as far as equivalent (a) and the percentage of FPGA used (b). The bar labels are formatted as exponent/integer_fraction widths. For example, 8_23 is single precision.

Figure 6 shows the equivalent slices and percentage of FPGA used for multiplication versus subtractions and Figure 7 analyzes divisions versus subtraction. These results show that when subtraction is required the graphs generally favor FP. This is due to the increase in memory required for LNS addition. The only exception is at 4_8 precision where again, the interpolation can be implemented in a look-up-table stored in RAM. If the LNS addition and subtraction units were combined, the datapath can be shared but the memory for both would still be required so the results would be even more favored towards FP.

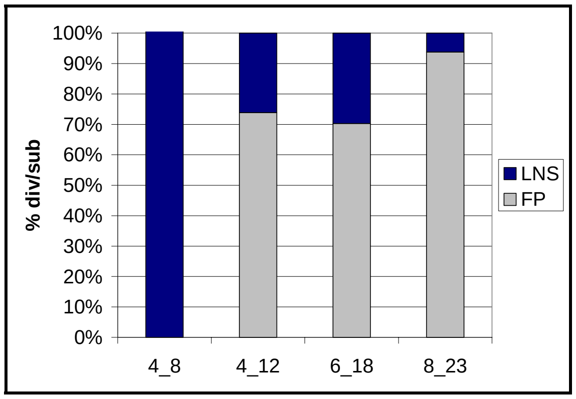


(a)

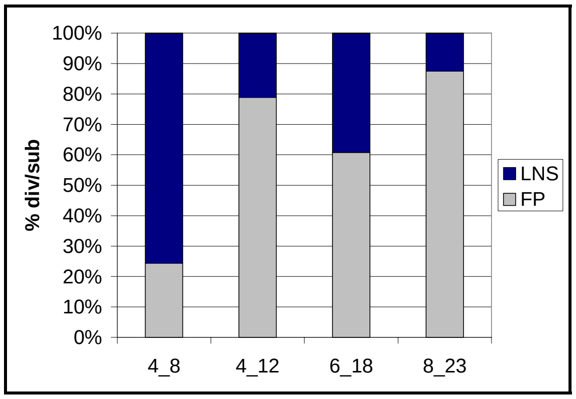


(b)

Figure 6. Plot of the percentage of multiplies versus subtractions in a circuit that make LNS beneficial as far as equivalent (a) and the percentage of FPGA used (b). The bar labels are formatted as exponent/integer_fraction widths. For example, 8_23 is single precision.



(a)



(b)

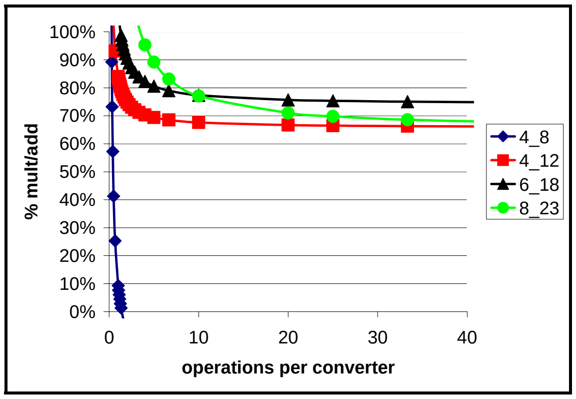
Figure 7. Plot of the percentage of divisions versus subtractions in a circuit that make LNS beneficial as far as equivalent (a) and the percentage of FPGA used (b). The bar labels are formatted as exponent/integer_fraction widths. For example, 8_23 is single precision.

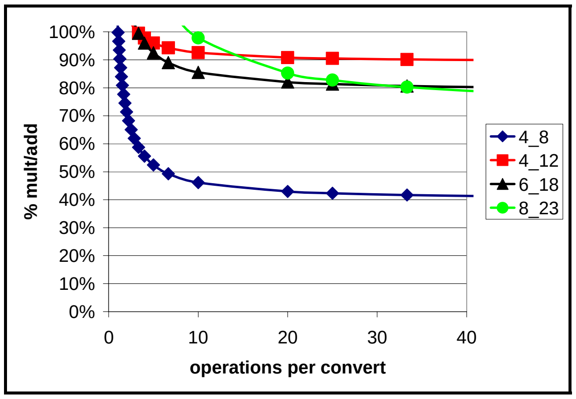
There area several things to note about the area analysis without conversion. Even though the single precision LNS multiplier is 21x smaller than the FP multiplier while the LNS adder is only 3x larger than the FP adder, a large percentage of multiplies is required to make LNS a smaller circuit. This is because the actual number of equivalent slices for the LNS adder is so large. For example, the LNS single precision adder is 649 slices larger than the FP adder while the LNS multiplier is only 365 slices smaller than the FP multiplier. Also when the percentage of FPGA used is compared the results generally favor FP even more. This is because LNS addition and subtractions make more use of block memories and multipliers which are scarcer than slices on the FPGA.

6.2 Area benefit with conversion

What if a computation requires FP numbers at the I/O or a specific section of a computation is LNS suited (i.e., high percentage of multiplies and divides), making conversion required? This is a more complicated space to cover because the number of converters needed in relation to the number and makeup of the operations in a circuit affects the tradeoff. The more operations done per conversion the less overhead each conversion contributes to the circuit. With lower conversion overhead, fewer multiplies and divides versus additions or subtraction is required to make LNS smaller. Another way to look at this is for each added converter, more multiplies and divides are needed per addition to offset the conversion overhead. By looking at operations per conversion, converters on the input and possible output are taken into account.

The curves in Figure 8 show the breakeven point of circuit size if it was done strictly in FP, or in LNS with converters on the inputs and outputs. Anything above the curve would mean a circuit done in LNS with converters would be smaller. Any point below means it would be better to just stay in FP. As can be seen in Figure 6a, a single precision circuit needs a minimum of 2.5 operations per converter to make conversion beneficial even if a circuit has 100% multiplies. Notice that the curves are asymptotic to the break even values in Figure 4 where there is assumed no conversion. This is because when many operations per conversion are being performed, the cost of conversion becomes irrelevant. Notice that the 4_12 curve approaches the 2/3 multiplies line very quickly. This is because at 12 bits of precision, the converters can be done with look-up tables in memory and therefore it only takes a few operations to offset the added area of the 12 bits of precision converters.

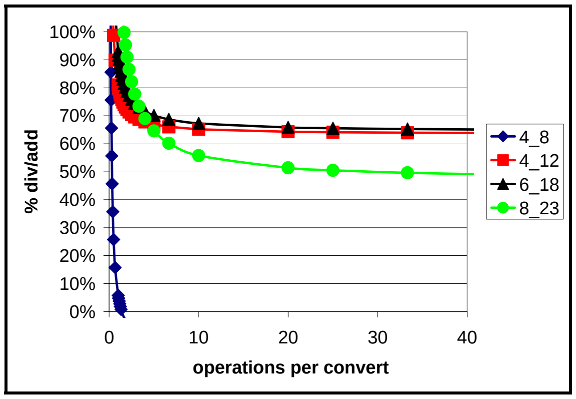


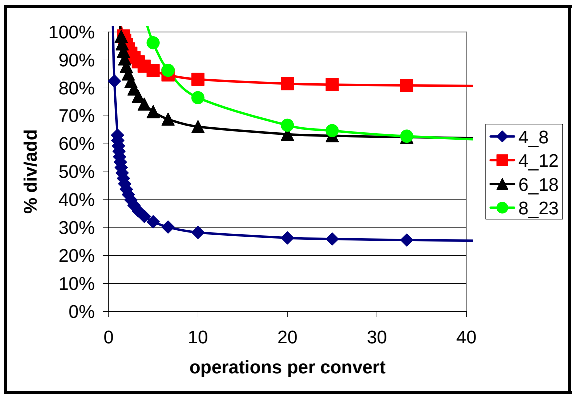


(a) (b)

Figure 8. Plot of the operations per converter and percent multiplies versus additions in a circuit to make the normalized area (a) and percentage of FPGA (b) the same for LNS with converters and FP. The bar labels are formatted as exponent/integer_fraction widths. For example, 8_23 is single precision.

Figure 9 is identical to Figure 8 except it compares divisions and additions instead of multiplications and divisions. Notice that while 4_12 shows up in Figure 9b it only approaches 90% divides.

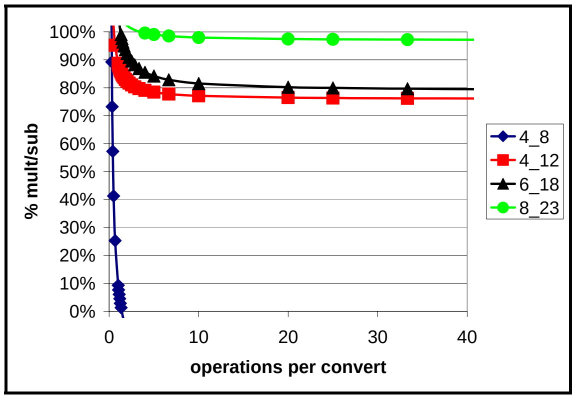


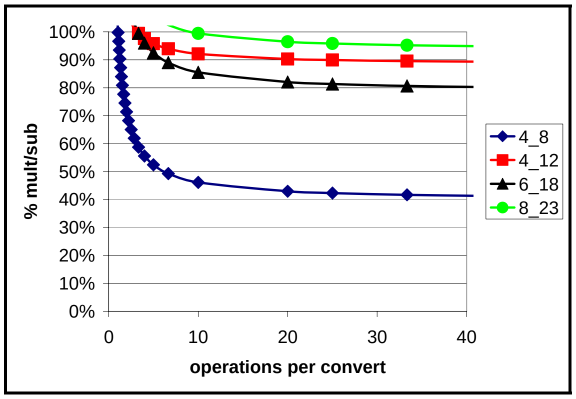


(a) (b)

Figure 9. Plot of the operations per converter and percent divides versus additions in a circuit to make the normalized area (a) and percentage of FPGA (b) the same for LNS with converters and FP. The bar labels are formatted as exponent/integer_fraction widths. For example, 8_23 is single precision.

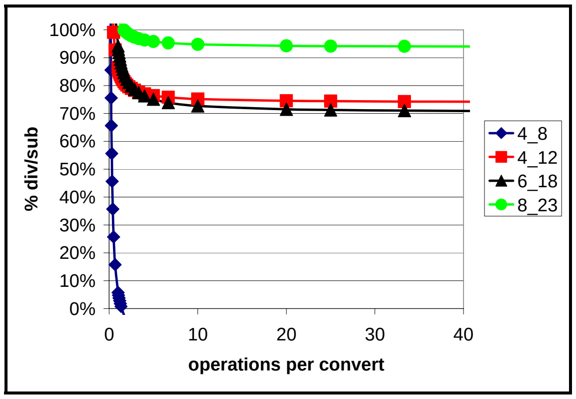
Figure 10 analyzes the area of multiplies versus subtractions and Figure 11 analyzes the area of divisions versus subtractions.

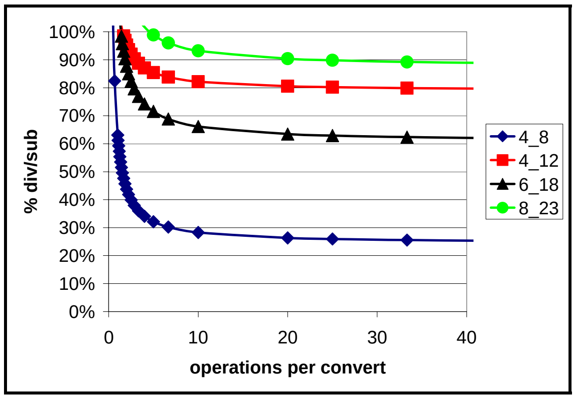




(a) (b)

Figure 10. Plot of the operations per converter and percent multiplies versus subtractions in a circuit to make the normalized area (a) and percentage of FPGA (b) the same for LNS with converters and FP. The bar labels are formatted as exponent/integer_fraction widths. For example, 8_23 is single precision.





(a) (b)

Figure 11. Plot of the operations per converter and percent divides versus subtractions in a circuit to make the normalized area (a) and percentage of FPGA (b) the same for LNS with converters and FP.

Notice that the 100% line of Figure 8 through Figure 11 show how many multiplies and divides in series are required to make converting to LNS for those operations beneficial. For example, in Figure 11b for single precision, as long as the ratio of divisions to converters is at least 4:1, converting to LNS to do the divisions in series would result in smaller area than staying in FP.

6.3 Performance benefit without conversion

A similar tradeoff analysis can be done with performance as was performed for area. However, now we are only concerned with the makeup of the critical path and not the circuit as a whole. Figure 12 shows the percentage of multiplies versus additions on the critical path that is required to make LNS faster. For example, for the single precision format (8_23), it shows that at least 75% of the operations on the critical path must be multiplies in order for an LNS circuit to be faster. Anything less than 75% would make a circuit done in FP faster.

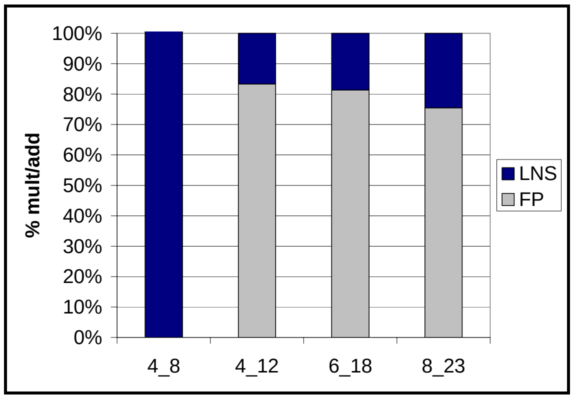


Figure 12. Plot of the percentage of multiplies versus additions in a circuit that make LNS beneficial in latency. The bar labels are formatted as exponent/integer_fraction widths. For example, 8_23 is single precision.

Figure 13 is identical to Figure 12 but for division versus addition instead of multiplication. The difficulty of FP division caused the breakeven points to lower when division versus addition is analyzed.

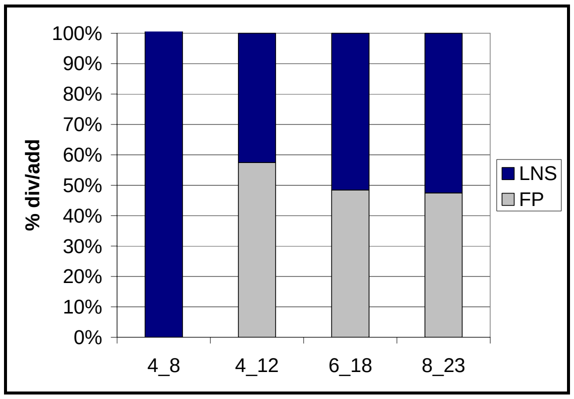


Figure 13. Plot of the percentage of divisions versus additions in a circuit that make LNS beneficial in latency. The bar labels are formatted as exponent/integer_fraction widths. For example, 8_23 is single precision.

Figure 14 and Figure 15 show the timing analysis for multiplication versus subtraction and division versus subtraction respectively. Notice that the difficulty of LNS subtraction causes the break even points to generally be higher than those involving addition.

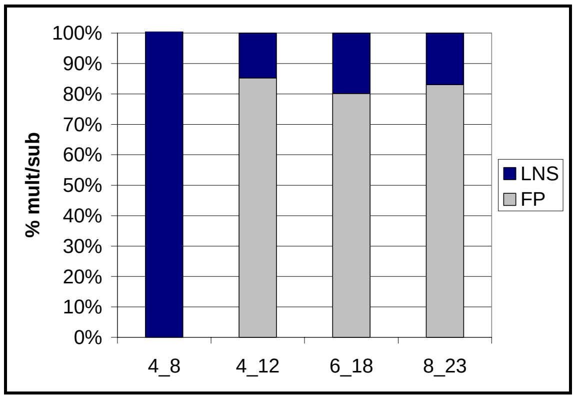


Figure 14. Plot of the percentage of multiplications versus subtractions in a circuit that make LNS beneficial in latency. The bar labels are formatted as exponent/integer_fraction widths. For example, 8_23 is single precision.

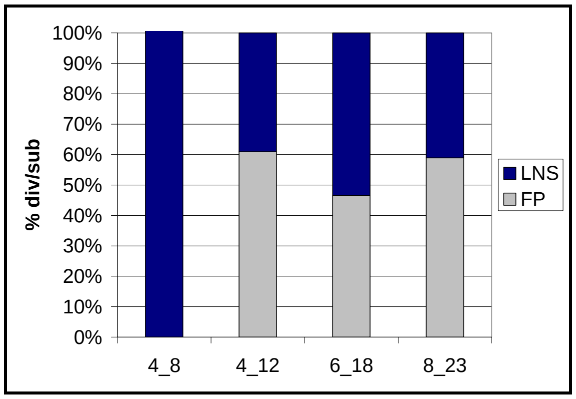


Figure 15. Plot of the percentage of divisions versus subtractions in a circuit that make LNS beneficial in latency. The bar labels are formatted as exponent/integer_fraction widths. For example, 8_23 is single precision.

The timing analysis follows the same trends as the area analysis without conversion in section 6.1 especially for addition comparisons. For comparisons involving division, the timing break even points are slightly lower that the area points. This is because the LNS subtractions greatly increase in area over LNS addition while there is only a slight increase in latency. The additional area for LNS subtraction is mostly due to extra memory requirements but the additional memory doesn't greatly increase the read time.

7 Conclusion

In this paper we set out to create a guide for determining when an FPGA circuit should be performed in FP or LNS in order to better support number systems with large dynamic range on FPGAs. While the results we obtained are dependent on the implementation of the individual LNS and FP units as well as the particular FPGA we implemented them on, we feel that meaningful conclusions can be obtained. The results show that while LNS is very efficient at multiplies and divides, the difficulty of addition and conversion at bit width greater than four bits exponent and eight bits fraction are too great to warrant the use of LNS except for a small niche of algorithms. For example, if a designer using single precision is interested in saving area, then the algorithm will need to be at least 65% multiplies or 48% divides in order for LNS to realize a smaller circuit if no conversion is required. If conversion is required or desired for a multiply/divide intensive portion of an algorithm, all precisions above 4_8 require a high percentage of multiplies or divides and enough operations per conversion to offset the added conversion area. If latency of the circuit is the top priority then an LNS circuit will be faster if 60-70% of the operations are multiply or divide. These results show that for LNS to become a suitable alternative to FP at greater precisions for FPGAs, better addition and conversion algorithms need to be employed to efficiently compute the non-linear functions.

BIBLIOGRAPHY

[1] Y. Wan, C.L. Wey, "Efficient Algorithms for Binary Logarithmic Conversion and Addition," *IEEE Proc. Computers and Digital Techniques,* vol.146, no.3, 1999, pp. 168-172.

[2] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed., A.K. Peters, Ltd., 2002.

[3] P. Belanovic, M. Leeser, "A Library of Parameterized Floating Point Modules and Their Use,*" 12th Int'l Conf. Field Programmable Logic and Applications (FPL 02)*, 2002, pp. 657-666.

[4] J. Detrey, F. Dinechin, "A VHDL Library of LNS Operators," *The 37th Asilomar Conf. Signals, Systems & Computers,* vol. 2, 2003, pp. 2227–2231.

[5] D.M. Lewis, "An Accurate LNS Arithmetic Unit Using Interleaved Memory Function Interpolator," *Proc. 11th IEEE Symp. Computer Arithmetic*, 1993, pp. 2-9.

[6] K.H. Tsoi et al., "An Arithmetic Library and its Application to the N-body Problem," *12th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM 04)*, 2004, pp. 68–78.

[7] B.R. Lee, N. Burgess, "A Parallel Look-Up Logarithmic Number System Addition/Subtraction Scheme for FPGAs," *Proc. 2003 IEEE Int'l Conf. Field-Programmable Technology (FPT 03)*, 2003 pp. 76–83.

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

[9] J.N. Coleman, E.I. Chester, "A 32 bit Logarithmic Arithmetic Unit and its Performance Compared to Floating-Point," *Proc. 14th IEEE Symp. Computer Arithmetic*, 1999, pp.142–151.

[10] B. Fagin, C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic," *IEEE Trans. VLSI Systems*, vol. 2, no. 3, 1994, pp.365-367.

[11] L. Louca, T.A. Cook, W.H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs," *Proceedings. IEEE Symp. FPGA's for Custom Computing,* 1996, pp.107-116.

[12] R. Matousek et al., "Logarithmic Number System and Floating-Point Arithmetics on FPGA," *12th Int'l Conf. Field-Programmable Logic and Applications* (*FPL 02)*, 2002, vol. 2438, pp. 627-636.

[13] Y. Wan, M.A. Khalil, C.L Wey, "Efficient Conversion Algorithms for Long-Word-Length Binary Logarithmic Numbers and Logic Implementation," *IEEE Proc. Computer Digital Technology,* vol. 146, no. 6, 1999, pp. 295-331.

[14] K. Underwood, "FPGA's vs. CPU's: Trends in Peak Floating Point Performance," *Proc. the 2004 ACM/SIGDA 12th Int'l Symp. Field Programmable Gate Arrays (FPGA 04),* 2004, pp. 171-180.

[15] Chipworks. Xilinx_XC2V1000_die_photo.jpg www.chipworks.com.

[16] J. Liang, R. Tessier, O. Mencer, "Floating Point Unit Generation and Evaluation for FPGAs," *11th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM 03),* 2003, pp. 185–194.

[17] D. M. Lewis, "Interleaved Memory Function Interpolators with Applications to an Accurate LNS Arithmetic Unit," *IEEE Trans. Computers*, vol. 43, no. 8, 1994, pp. 974-982.

[18] K.H. Abed, R.E. Siferd, “CMOS VLSI Implementation of a Low-Power Logarithmic Converter,” *IEEE Trans. Computers*, vol. 52, no. 11, 2003, pp. 1421–1433.

[19] G. Lienhart, A. Kugel, R. Manner, “Using Floating-Point Arithmetic on FPGAs to Accelerate Scientific N-Body Simulations,” *Proc. 10th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM 02)*, 2002, pp. 182–191.

[20] A. Walters, P. Athanas, “A Scaleable FIR Filter Using 32-Bit Floating-Point Complex Arithmetic on a Configurable Computing Machine,” *Proc. IEEE Symp. FPGAs for Custom Computing Machines, 1998*, pp. 333–334.

[21] W. Xiaojun, B.E. Nelson, “Tradeoffs of Designing Floating-Point Division and Square Foot on Virtex FPGAs,” *Proc. 11th Ann. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM 2003)*, 2003, pp. 195–203.

[22] Y. Li, W. Chu, “Implementation of Single Precision Floating Point Square Root on FPGAs,*” Proc., The 5th Ann. IEEE Symp. FPGAs for Custom Computing Machines, 1997*, pp. 226–232.

[23] C.H. Ho et al., “Rapid Prototyping of FPGA Based Floating Point DSP Systems,” *Proc. 13th IEEE Int’l Workshop on Rapid System Prototyping, 2002*, pp.19–24.

[24] A.A. Gaffar, “Automating Customization of Floating-Point Designs,” *12th Int’l Conf. Field Programmable Logic and Application* (*FPL 02)*, 2002, pp. 523-533.

[25] Xilinx, “Virtex-II Platform FPGAs: Complete Data Sheet,” v3.4, March 1, 2005.

[26] F. J. Taylor et al., “A 20 Bit Logarithmic Number System Processor," *IEEE Trans. on Computers*, vol. 37, no. 2, 1988, pp. 190-200.

[27] M. Arnold, "A Pipelined LNS ALU," *IEEE Workshop on VLSI*, 2001, pp. 155-161.

[28] M. Arnold and C. Walter, "Unrestricted Faithful Rounding is Good Enough for Some LNS Applications," *15th Int’l Symp. Computer Arithmetic*, 2001, pp. 237-245.

[29] J. Garcia, et al., "LNS Architectures for Embedded Model Predictive Control Processors," *Int’l Conf. Compilers, Architecture, and Synthesis for Embedded Systems (CASES 04)*, 2004, pp. 79-84.