

A Comparison of Floating Point and Logarithmic Number Systems on FPGAs

Michael Haselman

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in Electrical Engineering

University of Washington
2005

Program Authorized to Offer Degree:
Electrical Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a master's thesis by

Michael Haselman

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

Scott Hauck

Jeff Bilmes

Date: _____

In presenting this thesis in partial fulfillment of the requirements for a master's degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Any other reproduction for any purposes or by any means shall not be allowed without my written permission.

Signature _____

Date _____

University of Washington

Abstract

A Comparison of Floating Point and Logarithmic Number Systems on FPGAs

Michael Haselman

Chair of the Supervisory Committee:

Associate Professor Scott Hauck

Electrical Engineering

There have been many papers proposing the use of logarithmic numbers (LNS) as an alternative to floating point because of simpler multiplication, division and exponentiation computations [Lewis93, Wan99, Coleman99, Coleman00, Detrey03, Lee03, Tsoi04]. 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 floating point 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.

Table of Contents

	Page
List of Figures	ii
List of Tables.....	iii
1 Introduction	1
2 Background	2
2.1 Previous Work.....	2
2.2 Field Programmable Gate Arrays	2
3 Number Systems	5
3.1 Floating Point	5
3.2 Logarithmic Number System (LNS)	6
4 Implementation.....	8
4.1 Multiplication	9
4.2 Division	10
4.3 Addition/Subtraction	10
4.4 Conversion	12
4.5 Floating point to LNS.....	13
4.6 LNS to floating point.....	16
5 Results	18
5.1 Multiplication	18
5.2 Division	19
5.3 Addition.....	19
5.4 Subtraction	20
5.5 Another floating point library.....	20
5.6 Multiplication	21
5.7 Division	21
5.8 Addition.....	21
5.9 Subtraction	22
5.10 Converting floating point to LNS.....	22
5.11 Converting LNS to floating point.....	23
6 Analysis.....	24
6.1 Area benefit without conversion	24
6.2 Area benefit with conversion	25
6.3 Performance benefit without conversion.....	29
7 Conclusion	31
Bibliography.....	32

List of Figures

Figure Number	Page
1. FPGA architecture	4
2. LNS addition/subtraction curves.....	11
3. Area breakeven for multiplication vs. addition, no conversion	24
4. Area breakeven for division vs. addition, no conversion.....	25
5. FP to LNS conversion overhead for multiplication	26
6. FP to LNS conversion overhead for division.....	27
7. LNS to FP conversion overhead for multiplication	28
8. LNS to FP conversion overhead for division.....	28
9. Latency breakeven for multiplication vs. addition, no conversion	29
10. Latency breakeven for division vs. addition, no conversion.....	30

List of Tables

Table Number	Page
1. Floating point exceptions.....	6
2. Area and latency of multiplication.....	18
3. Area and latency of division.....	19
4. Area and latency of addition.....	19
5. Area and latency of subtraction.....	20
6. Area and latency of Leaser multiplication.....	21
7. Area and latency of Leaser division.....	21
8. Area and latency of Leaser addition.....	21
9. Area and latency of Leaser subtraction.....	22
10. Area and latency of conversion FP to LNS.....	22
11. Area and latency of conversion LNS to FP.....	23

Acknowledgements

The author would like to thank his advisor, Scott Hauck for providing his conception of this project and for his support and guidance throughout the term of the project. The author would also like to thank, Michael Beauchamp, Aaron Wood, and Henry Lee for their initial work on the arithmetic libraries. Finally, the author would like to thank Keith Underwood and Scott Hemmert for their support with their floating point library.

This work was supported in part by grants from the National Science Foundation, and from Sandia National Labs. Scott Hauck was supported in part by a Sloan Research Fellowship.

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. The requirements for performance and a large dynamic range lead to the use of floating point or logarithmic number systems in hardware.

FPGA designers began mapping floating point arithmetic units to FPGAs in the mid 90's [Fagin94, Louca96, Belonovic02, Underwood04]. Some of these were full implementations, but most make some optimizations to reduce the hardware. The dynamic range of floating point comes at the cost of lower precision and increased complexity over fixed point. Logarithmic number systems (LNS) provide a similar range and precision to floating point but may have some advantages in complexity over floating point for certain applications. This is because multiplication and division are simplified to fixed-point addition and subtraction, respectively, in LNS. However, floating point number systems have become a standard while LNS has only seen use in small niches.

Most implementations of floating point on DSPs or microprocessors use single (32-bit) or double (64-bit) precision. Using FPGAs give 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 floating point. Specifically, we attempt to define when it is advantageous to use LNS over floating point number systems.

2 Background

2.1 Previous Work

Since the earliest work on implementing IEEE compliant floating point units on FPGAs [Fagin94] there has been extensive study on floating point arithmetic on FPGAs. Some of these implementations are IEEE compliant (see the Number Systems section below for details) [Fagin94, Louca96, Underwood04] while others leverage the flexibility of FPGAs to implement variable word sized that can be optimized per application [Belanovic02, Ho02, Liang03, Liang03]. There has also been a lot of work on optimizing the separate operations, such as addition/subtraction, multiplication, division and square root [Loucas96, Li97, Wang03] to make incremental gains in area and performance. Finally, there have been many applications implemented with floating point to show that not only can arithmetic units be mapped to FPGAs, but that useful work can be done with those units [Walter98, Leinhart02].

In recent years, there has been a lot of work with the logarithmic number system as a possible alternative to floating point. There has even been a proposal for a logarithmic microprocessor [Colmen00]. Most of this work though has been algorithms for LNS addition/subtraction [Lewis93, Colmen00, Lee03] or conversion from floating point to LNS [Wan99, Wan99, Abed03] because these are complicated operations in LNS. There have also been some previous studies that compared LNS to floating point [Colmen99, Matousek02, Detrey03]. Colmen et al. showed that LNS addition is just as accurate as floating point addition, but the delay was compared on a ASIC layout. Matousek et al report size and speed numbers for LNS and floating point arithmetic units on a FPGA for a 20-bit and 32-bit word size but offer no insight to when one format should be used over the other. Detrey et al also compared floating point 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

floating point computations, floating point operations are often part of these critical portions and therefore are good candidates for implementation in hardware to obtain speedup of an algorithm. 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 implementations 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.

FPGAs have successfully filled a niche between of the flexibility of software and the performance of an ASIC by using a large array of configurable logic blocks, and dedicated block modules all connected by a reconfigurable communication structure (see figure 1). The reconfigurable logic blocks and communications resources are controlled by SRAM bits. Therefore, by downloading a configuration into the control memory, a complex circuit can be created.

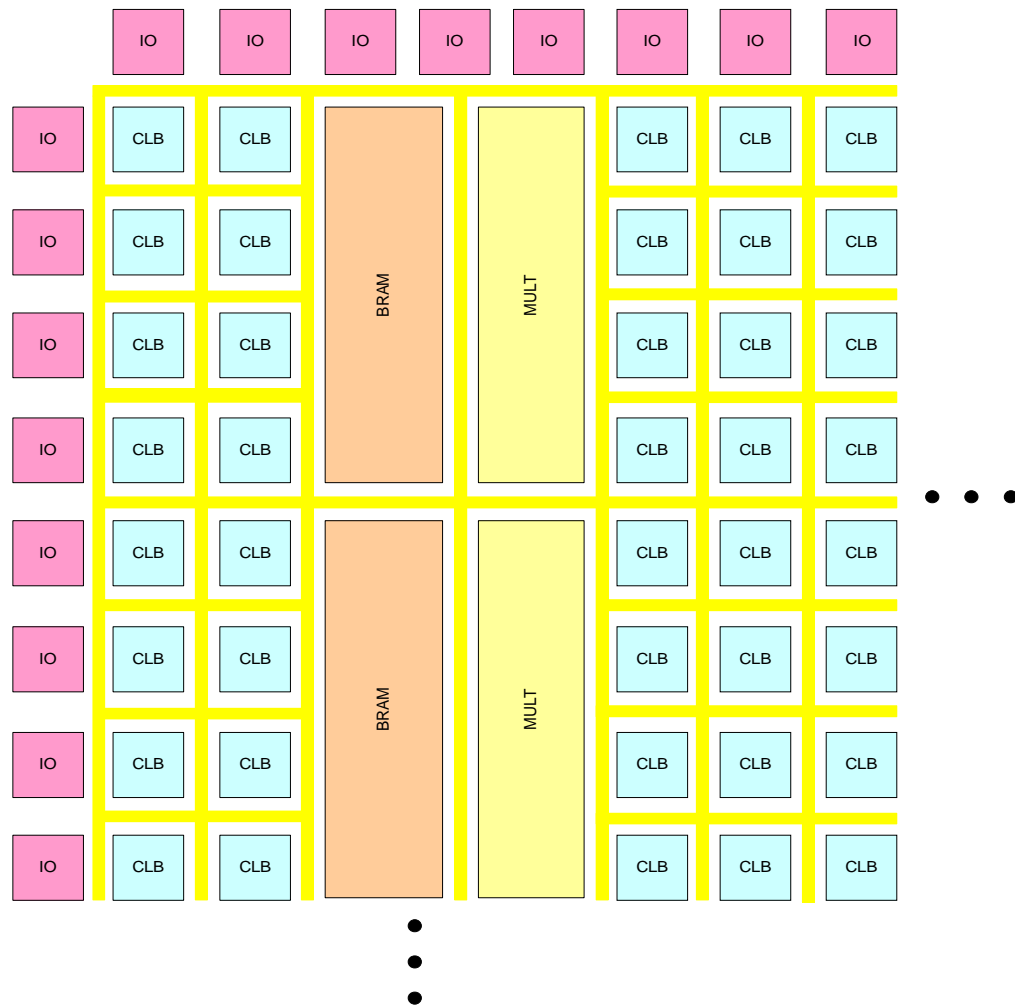


Figure 1. Architecture overview of a typical FPGA.

The configurable logic blocks (CLBs) typically contain multiple Look-up Tables (LUT) each capable of implementing any 4 to 1 function, along with some additional resources for registering data and fast addition [Xilinx05]. The typical dedicated block modules are multipliers and block memories (BRAMs). While the LUTs can be used for memory, the BRAM allows for more compact and faster large memories. The multipliers allow multiplications to be done faster while taking up less space and using less power than would be required if done with CLBs. The Xilinx VirtexII XC2V2000 (the device used in our experiments) has 2,688 CLBs, 56 18 Kbit BRAMs and 56 18bit x 18bit multipliers [Xilinx05]. The CLBs each contain four slices and the slices each contain two LUTs. The slices will be used in our area metric as discussed in implementation section.

3 Number Systems

3.1 Floating point

A floating point number F has the value [Koren02]

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

where S is the sign, f is the fraction, and E is the exponent, of the number. The mantissa 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 floating point library we use is disabled for these tests [Underwood04]). The exponent is usually kept in a biased format, where the value of E is

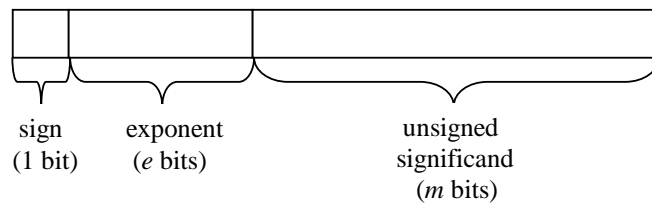
$$E = E^{true} + bias.$$

The most common value of the bias is

$$2^{e-1} - 1$$

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

Floating point numbers are kept in the following format:



The IEEE 754 standard sets two formats for floating point 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 $(1.11... \times 2^{127})$, $\approx \pm 1.2 \times 10^{-38}$ to 3.4×10^{38} and resolution of 10^{-7} . For double precision, where m is 52 and e is 11, the range is $\pm(1.0 \times 2^{-1022})$ to $(1.11... \times 2^{1023})$, $\approx \pm 2.2 \times 10^{-308}$ 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 table 1.

Table 1. Floating point exceptions (f is the fraction of M).

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

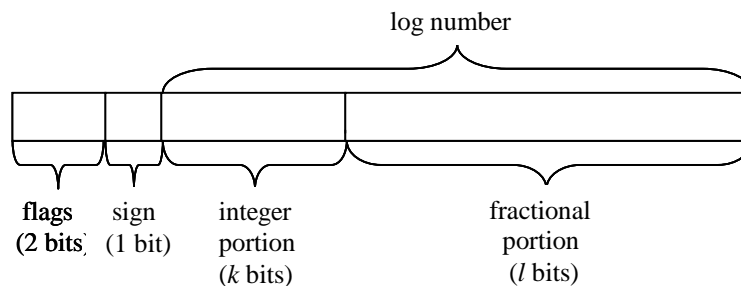
3.2 LNS

Logarithmic numbers can be viewed as a specific case of floating point numbers where the mantissa is always 1, and the exponent has a fractional part [Koren02]. The number A has a value of

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

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:



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 [Detrey03].

If $k=e$ and $l=m$ LNS has a very similar range and precision as floating point numbers. For $k=8$ and $l=23$ (single precision equivalent) the range is $\pm 2^{-129+ulp}$ 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 $\pm 2^{-1025+ulp}$ 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 2.2. 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, subtracter and converters involve look up tables that are dependent on 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 using Xilinx ISE software. We then mapped a floating point library from Underwood [Underwood04] to the same device for comparison.

As discussed previously most of the commercial FPGAs contain dedicated RAM and multipliers that will be utilized by the units of these libraries (see figure 1). 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 floating point 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 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 have 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 [Chipworks] 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 [Koren02]. The product is computed by adding the two fixed point logarithmic numbers. This is from the following logarithmic property:

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

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 event, while overflows (correctly) result in $\pm\infty$. Overflow events occur when the two numbers being added sum up to a number too large to be represented in the word width.

Floating point multiplication is more complicated [Koren02]. The two exponents are added and the mantissas are multiplied together. The addition of the exponents comes from the property:

$$2^x \times 2^y = 2^{x+y}$$

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 [Koren02]:

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

Just as in multiplication, the sign bit is computed with the XOR of the two operands' signs and the operation is exact. The only difference is the possibility of underflow due to the subtraction which results in a difference equal to 0. Underflow occurs when the result of the subtraction is a larger negative number than can be represented in word width.

Division in floating point is accomplished by dividing the mantissas and subtracting the divisor's exponent from the dividend's exponent [Koren02]. Because the range of the mantissas 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 [Koren02]. Assume we have two numbers A and B ($|A| \geq |B|$) 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

$$\begin{aligned} S_c &= S_A \\ E_c &= \log_2|(A \pm B)| = \log_2\left|A\left(1 \pm \frac{B}{A}\right)\right| \\ &= \log_2|A| + \log_2\left|1 \pm \frac{B}{A}\right| = E_A + f(E_B - E_A) \end{aligned}$$

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

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

The \pm indicates addition or subtraction (+ for addition, - for subtraction). The value of $f(x)$ shown in figure 2, 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 [Wan99]. Notice that if $E_A - E_B = 0$ then $f(E_A - E_B) = -\infty$ for subtraction so this needs to be detected.

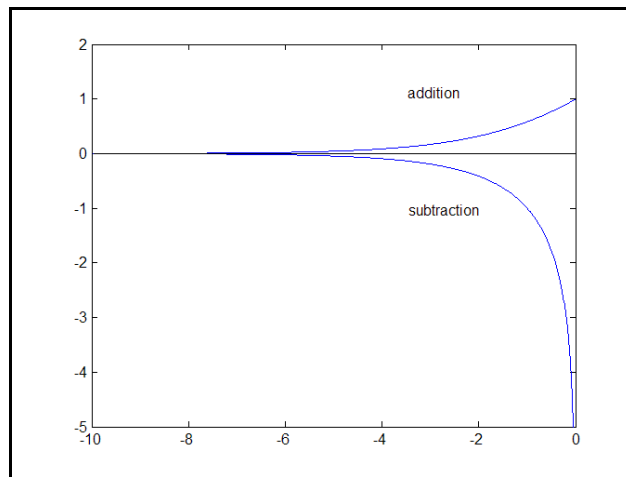


Figure 2. Plot of $f(x)$ for addition and subtraction [Matousek02].

For our LNS library, the algorithm from Lewis [Lewis93] was implemented to calculate $f(x)$. 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 points are closer together the steeper the curve). For subtraction, the first interval is even broken up further to generate more points, hence it will require more memory.

Floating point addition on the other hand is a much more straight forward computation. It does however require that the exponents of the two operands be equal before addition [Koren02]. 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 possible the length of the mantissa. This is significant because variable length shifters are a difficult process 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 $\pm\infty$.

Floating subtraction is identical to floating point addition except that the mantissas are subtracted [Koren02]. If the result is less than one, a left shift of the mantissa and a decrement of the exponent are required. The decrement of the exponent can result in underflow, which means the result is set to zero.

4.4 Conversion

Since floating point has become a standard, modules that covert floating point numbers to logarithmic numbers and vice versus 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 floating point 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 Floating point to LNS

The conversion from floating point to LNS involves three steps that can be done in parallel. The first part is checking if the floating point 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:

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

Notice that the $\log_2(1.xxx...)$ is in the range of (0,1] and exp is an integer. This means that the integer portion of the logarithmic number is simply the exponent of the floating point 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 [Wan99] was used. This algorithm reduces the amount of memory required by factoring the floating point mantissa as (for $m = n/2$)

$$\begin{aligned}(1.x_1x_2...x_n) &= (1.x_1x_2...x_m)(1.00...0c_1c_2...c_m) \\ \log_2(1.x_1x_2...x_n) &= \log_2[(1.x_1x_2...x_m)(1.00...0c_1c_2...c_m)] \\ &= \log_2(1.x_1x_2...x_m) + \log_2(1.00...0c_1c_2...c_m)\end{aligned}$$

where

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

Now $\log_2(1.x_1x_2...x_m)$ and $\log_2(1.00...0c_1c_2...c_m)$ can be stored in a ROM that is $2^m \times n$.

In 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

$$c = \frac{b}{(1+a)}$$

The computation can rewritten as

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

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

$$2^z \approx 2 - \log_2[1 + (1 - z)], \text{ for } z \in [0,1)$$

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

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

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.

For double precision the above algorithm requires 2.5Gbits of memory. This is obviously an unreasonable amount of memory for a FPGA, so an algorithm [Wan99] that does not use memory was used for double precision. This algorithm uses factorization to determine the logarithm of the floating point mantissa. The mantissa can be factored as

$$1 + .x_1x_2\dots x_n \cong (1 + .q_1)(1 + .0q_2)\dots(1 + .00\dots 0q_n)$$

then

$$\begin{aligned} \log(1 + .x_1x_2\dots x_n) &\cong \log(1 + q_1) \\ &\quad + \log(1 + 0q_2) + \dots + \log(1 + 00\dots 0q_n) \end{aligned}$$

where q_i is 0 or 1. Now the task is to efficiently find the q_i 's. This is done in the following manner:

$$1 + .x_1x_2\dots x_n \cong (1 + x_1)(1 + .0a_{12}a_{13}\dots a_{1n})$$

where $q_1 = x_1$ and

$$.0a_{12}a_{13}\dots a_{1n} = (.x_2x_3\dots x_n)/(1 + .q_1)$$

Similarly, $(1 + .0a_{12}a_{13}\dots a_{1n})$ can be factored as

$$1 + .0a_{12}a_{13}\dots a_{1n} \cong (1 + .0a_{12})(1 + .00a_{23}a_{24}\dots a_{2n})$$

where $q_2 = a_{12}$ and

$$.a_{23}a_{24}\dots a_{2n} = (.a_{13}a_{14}\dots a_{1n})/(1 + .0q_2)$$

This process is carried out for n steps, where n is the width of the floating point fraction. Notice that determining q is very similar to calculating the quotient in a binary division except the divisor is updated at each iteration. So, a modified restoring division is used to

find all q_i 's. At each step, if $q_i = 1$ then the value $\log_2(1 + .00..0q_i)$ is added to a running sum. The value of $\log_2(1 + .00..0q_i)$ is kept in memory. This memory requirement is $n \times n$ which is not stored in BRAM because there is a possibility of accessing all "addresses" on one cycle and the memory space is very small.

4.6 LNS to floating point

In order to convert from LNS to floating point 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 floating point exceptions shown in table 1. The integer portion of the logarithmic number becomes the exponent of the floating point 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 floating point exponent since it is below floating point's precision. If this occurs, the mantissa needs to be set to zero for floating point libraries that do not support denormalized (see table 1).

The conversion of the logarithmic integer to the floating point mantissa involves evaluating the non-linear equation 2^n . Since the logarithmic integer 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:

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

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

$$.a_1 a_2 \dots a_n = \underbrace{x_1 x_2 \dots x_n}_k \underbrace{y_1 y_2 \dots y_n}_k \dots \underbrace{z_1 z_2 \dots z_n}_k \dots$$

where k is the number of times the integer is broken up. The values of $2^{x_1 x_2 \dots x_{n/k}}$, $2^{.00..0y_1 y_2 \dots y_{n/k}}$ etc. are stored in k ROMs of size $2^k \times n$. Now the memory requirement is $(2^k \times k) \times n$ 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 floating point and logarithmic libraries to a Xilinx VirtexII 2000. 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.

5.1 Multiplication

Table 2. Area and latency of a multiplication.

	single		double	
	FP	LNS	FP	LNS
slices	297	20	820	36
multipliers	4	0	9	0
18K BRAM	0	0	0	0
units per FPGA	14	537	6.2	298
norm. area	368	20	981	36
latency(ns)	65	10	83	12

Floating point 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 the LNS unit is 18.4x smaller for single, and 27.3x smaller for double. 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 3. Area and latency of a division.

	single		double	
	FP	LNS	FP	LNS
slices	910	20	3376	36
multipliers	0	0	0	0
18K BRAM	0	0	0	0
units per FPGA	12	537	3.2	298
norm. area	910	20	3376	36
latency(ns)	150	10	350	12.7

Floating point 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 the LNS unit is 45.5x smaller for single precision and 93.8x smaller for double precision.

5.3 Addition

Table 4. Area and latency of an addition.

	single		double	
	FP	LNS	FP	LNS
slices	424	750	836	1944
multipliers	0	24	0	72
18K BRAM	0	4	0	8
units per FPGA	25	2.33	12.9	0.78
norm. area	424	1291	836	3456
latency(ns)	45	91	67	107

As can be seen, the LNS adder is more complicated than the floating point 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 floating point adder. With respect to the normalized area the LNS adder is 3.0x for single, and 4.1x larger for double precisions.

5.4 Subtraction

Table 5. Area and latency of a subtraction.

	single		double	
	FP	LNS	FP	LNS
slices	424	838	836	2172
multipliers	0	24	0	72
18K BRAM	0	8	0	16
units per FPGA	25	2.33	12.9	0.78
norm. area	424	1491	836	3907
latency(ns)	45	95	67	110

The logarithmic subtraction is even larger than the adder because of the additional memory needed when the two numbers are very close. Floating point subtraction is very similar to addition. In terms of normalized area, the LNS subtraction is 3.3x larger for single and 4.4x larger for double precision. Note that the growth of the LNS subtraction and addition are fairly linear with the number of bits. This is because the memory only increases in width and not depth as the word size increases.

5.5 Another floating point library

After discussing our results with the community, a few engineers versed in the area of computer arithmetic on FPGAs suggested that the implementation of the divider used for floating point division has a large impact on the area and latency of the floating point divider. They also discussed that current uses of LNS are for applications that require word sizes smaller than single precision. The smaller word sizes make it possible to do the converters with look-up table in memory which results in smaller and faster circuits. This prompted us to make a comparison to another floating point library that had a different floating point divider implementation and was capable of doing word sizes other than single and double precision. The second library from Leeser et al [Belonovic 02] is fully parameterizable and computes the mantissa division with reciprocal and multiply method to contrast the restoring division used in the Underwood library. The reciprocals are stored in a look-up table. This means that double precision division is unrealizable due to memory requirements.

The following results include a “half” precision word size which is four bits of exponent and 12 bits of significand. The single and double precision LNS data is identical to the results above but is include for easy comparison

5.6 Multiplication

Table 6. Area and latency of Leaser floating point and LNS multiplication.

	half		single		double	
	FP	LNS	FP	LNS	FP	LNS
slices	178	13	377	20	1035	36
multipliers	1	0	4	0	9	0
18K BRAM	0	0	0	0	0	0
units per FPGA	113	1442	52	537	24	298
eq slices	196	13	449	20	1196	36
latency(ns)	19.8	6.9	30.6	10	44.8	12

5.7 Division

Table 7. Area and latency of Leaser floating point and LNS division.

	half		single	
	FP	LNS	FP	LNS
slices	183	13	412	20
multipliers	2	0	8	0
18K BRAM	1	0	7	0
units per FPGA	28	1442	7	537
eq slices	246	13	751	20
latency(ns)	35.9	6.6	44.9	10

5.8 Addition

Table 8. Area and latency of Leaser floating point and LNS addition.

	half		single		double	
	FP	LNS	FP	LNS	FP	LNS
slices	166	341	360	750	757	1944
multipliers	0	9	0	24	0	72
18K BRAM	0	4	0	4	0	8
units per FPGA	113	6.2	52	2.33	24	0.78
eq slices	166	613	360	1291	757	3456
latency(ns)	25.3	63.2	33.2	91	44.9	107

5.9 Subtraction

Table 9. Area and latency of Leeser floating point and LNS subtraction.

	half		single		double	
	FP	LNS	FP	LNS	FP	LNS
slices	167	418	360	838	756	2172
multipliers	0	9	0	24	0	72
18K BRAM	0	8	0	8	0	16
units per FPGA	113	6	52	2.33	24	0.78
eq slices	167	802	360	1491	756	3907
latency(ns)	23.2	65	33.2	95	44.9	110

There are a few things of interest about the above data. Notice that for the single precision floating point divider, the Leeser library divider is 18% smaller and 4x faster than the Underwood library. The large difference in latency is due to the use of look-up tables versus computing the mantissa division. Also notice that the LNS adder is still quite large and slow at half precision. This is because even at half precision, the memory requirement for a look-up table approach is prohibitive.

5.10 Convert floating point to LNS

Table 10. Area and latency of a conversion from floating point to LNS.

	half	single	double
slices	19	163	7631
multipliers	0	0	0
18K BRAM	3	24	0
units per FPGA	18.7	2.3	1.4
norm. area	103	832.6	7712
latency(ns)	7.6	76	380

Converting from floating point to LNS has typically been done with ROMs [Koren 02] for small word sizes. For larger words, an approximation must be used. Two algorithms were used for this approximation, one for single precision [Wan 99] and another for double precision [Wan99]. Even though the algorithm for double precision conversion does not use any memory, it performs 52 iterations of a modified binary division which explains the large normalized area.

5.11 Convert LNS to floating point

Table 11. Area and latency of a conversion from LNS to floating point.

	half	single	double
slices	10	236	3152
multipliers	0	20	226
18K BRAM	3	4	14
units per FPGA	14	2.8	0.25
norm. area	93.8	866	9713

The area of the LNS to floating point 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 floating point suited operations (add, subtract) in a computation will have an effect on the size of a circuit. Figure 3 shows the plot of the percentage of multiplies to addition that is required to make LNS have a smaller normalized area than floating point. For example, for a single precision circuit in the Underwood library (figure 3a), the computation makeup would have to be 71% multiplies to 29% additions for the two circuits to be of even size. More than 71% multiplies would mean that an LNS circuit would be smaller and anything less means a floating point circuit would be smaller. For double precision, the break even point is 73% multiplications versus additions. Notice that for half precision, the number of multiplies required remains about the same. This is because half precision still requires too much memory to make a look-up implementation of LNS addition possible.

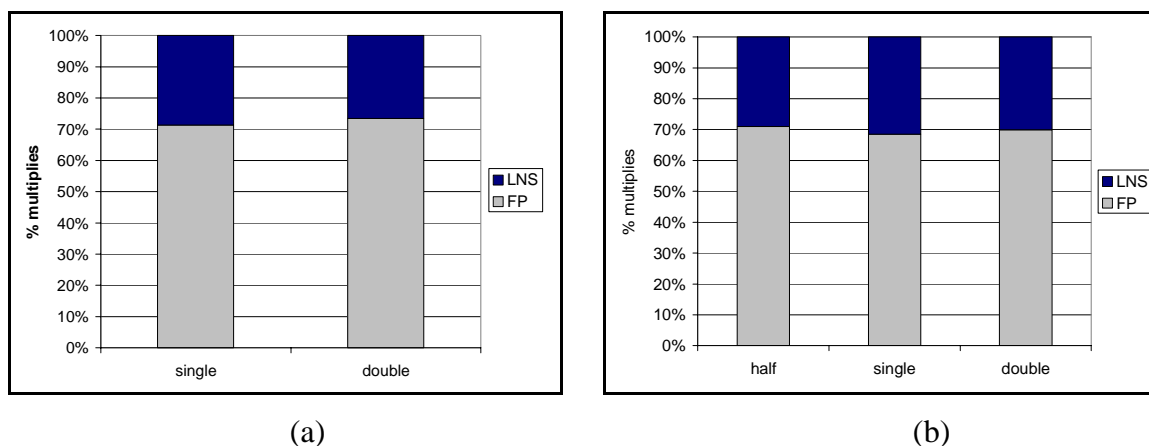


Figure 3. Plot of the percentage of multiplies versus additions in a circuit that make LNS beneficial as far as equivalent slices for the Underwood library (a) and the Leeser (b) library.

Figure 4 shows the same analysis as figure 3 but for divisions versus addition instead of multiplication versus addition. For division, the break even point for single precision is 49% divides and 44% divides for double precision for the Underwood library. The large

decline in percentages from multiplication to division is a result of the large increase in size of the floating point division over floating point multiplication. Notice that for single precisions, in the Leeser library, the percentage of divides is greater reflecting the 18% reduction in the Leeser floating point multiplier over the Underwood divider.

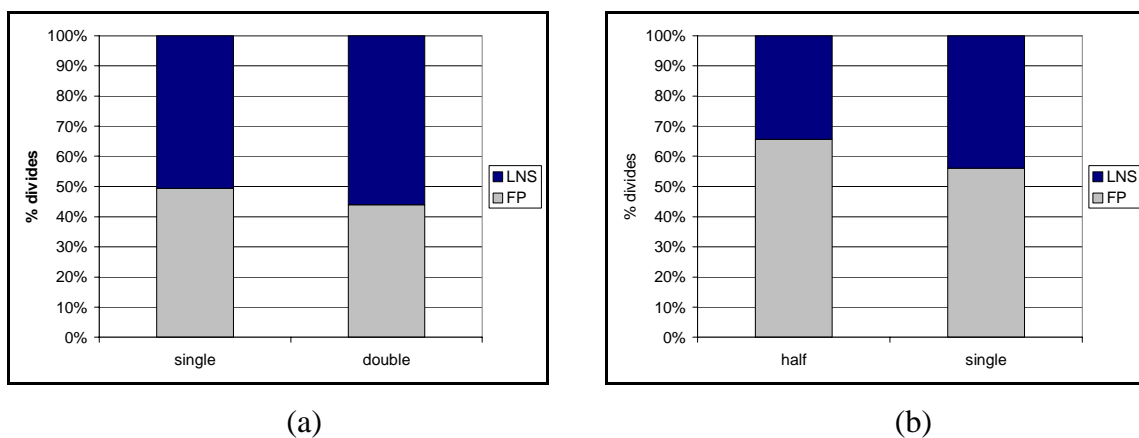


Figure 4. Plot of the percentage of divides versus additions in a circuit that make LNS beneficial as far as equivalent slices for Underwood library (a) and the Leeser (b) library.

Even though the LNS multiplier is 45x and 94x smaller than the floating point multiplier and the LNS adder is only 3x and 4x larger than the floating point 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 867 slices larger than the floating point adder while the LNS multiplier is only 348 slices smaller than the floating point multiplier.

6.2 Area benefit with conversion

What if this computation requires floating point numbers at the I/O, 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 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 make the conversion beneficial.

The curves in figure 5 show the break even point of circuit size if it was done strictly in floating point, 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 floating point. As can be seen in figure 5a, a single precision circuit needs a minimum of 2.5 operations per converter to make conversion beneficial even if a circuit has 100% multiplies. This increases to 8 operations per conversion for double precision. Notice that the two curves are asymptotic to the break even values in figure 3 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 half precision curve approaches the 70% multiplies line very quickly. This is because at half 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 half precision converters. Single and double precision have to use an approximation algorithm which are large as can be seen in table 6 and 7.

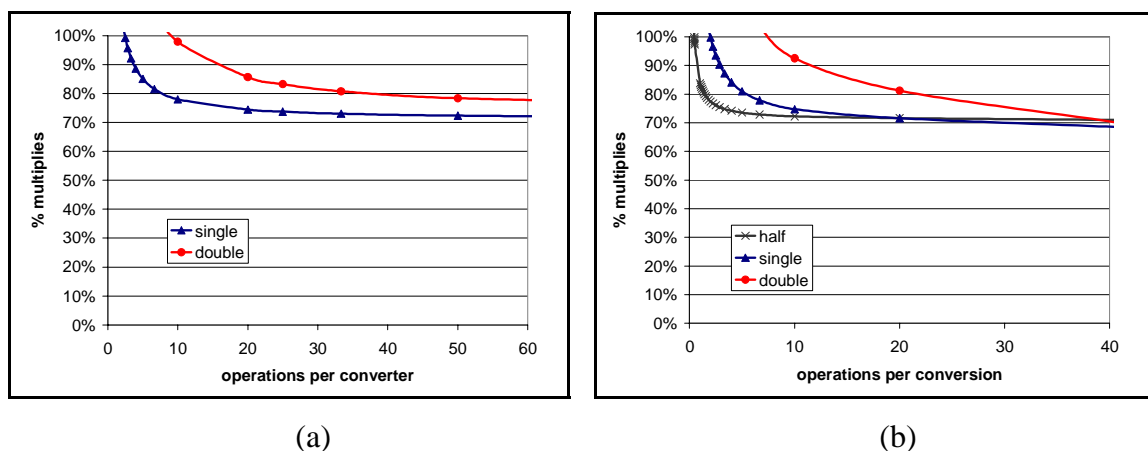


Figure 5. Plot of the operations per converter and percent multiplies versus additions in a circuit to make the area the same for LNS with converters and floating point for the Underwood library (a) and the Leeser (b) library.

Figure 6 is identical to figure 5 except that it is for divides versus additions instead of multiplies. Again, the break even point for the Leeser (figure 6b) single precision divider is slightly higher than the Underwood divider (figure 6a) due to it's smaller equivalent slice count.

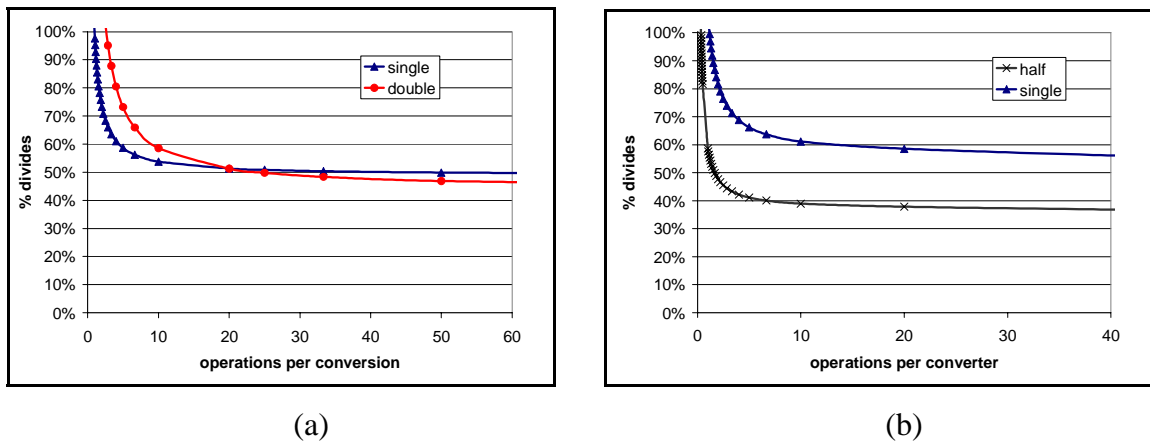


Figure 6. Plot of the operations per converter and percent divides versus additions in a circuit to make the area the same for LNS with converters and floating point for the Underwood library (a) and the Leeser (b) library.

Notice that the 100% line in both figure 5 and 6 show how many multiplies and divides in series are required to make converting to LNS for those operations beneficial. For example, for double precision division, as long as the ratio of divisions to converters is at least 3:1, converting to LNS to do the divisions in series would result in smaller area.

The conversion can be looked at in the other direction as well. While it is less likely that a floating point computation would need LNS at the I/O, it could be beneficial to change a LNS computation to a floating point computation for a section of a computation that has a high ratio of additions. Figure 7 plots the break even point for converting from LNS to floating point and back to LNS for the number of additions versus multiplications. Notice that in general, the break even curves are lower than when conversion from floating point to LNS was considered. This is because of the cost of

LNS addition over floating point addition is larger than the cost of floating point multiplication or division.

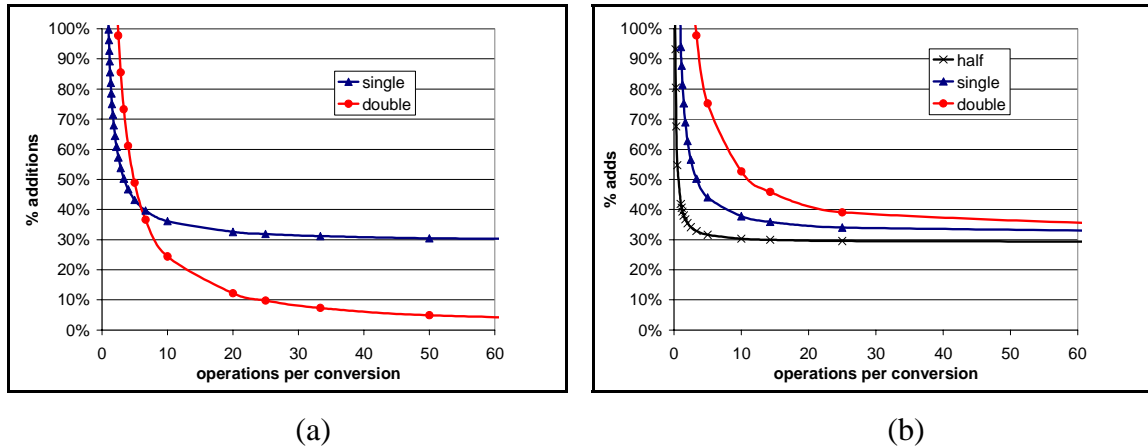


Figure 7. Plot of the operations per converter and percent addition versus multiplication in a circuit to make the area the same for LNS and floating point with converters for the Underwood library (a) and the Leeser (b) library.

Figure 8 is identical to figure 7 but is for additions versus divisions instead of additions versus multiplications.

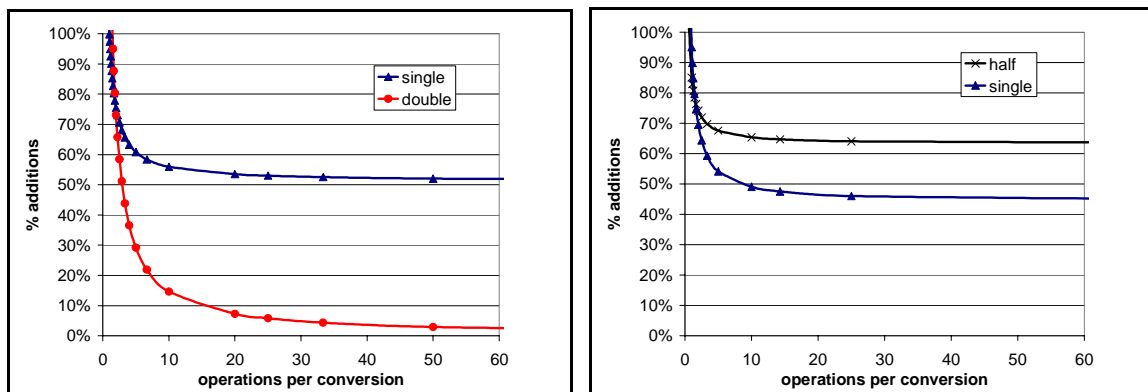


Figure 8. Plot of the operations per converter and percent addition versus division in a circuit to make the area the same for LNS and floating point with converters for the Underwood library (a) and the Leeser (b) library.

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 9 show the percentage of multiplies versus additions on the critical path that is required to make LNS faster. For example, for the Underwood library (figure 6a) it shows that for single precision, at least 45% of the operations on the critical path must be multiplies in order for an LNS circuit to be faster. Anything less than 45% would make a circuit done in floating point faster. For the Leeser library, the break even points are much higher due to the reduced latency of the circuits (see results).

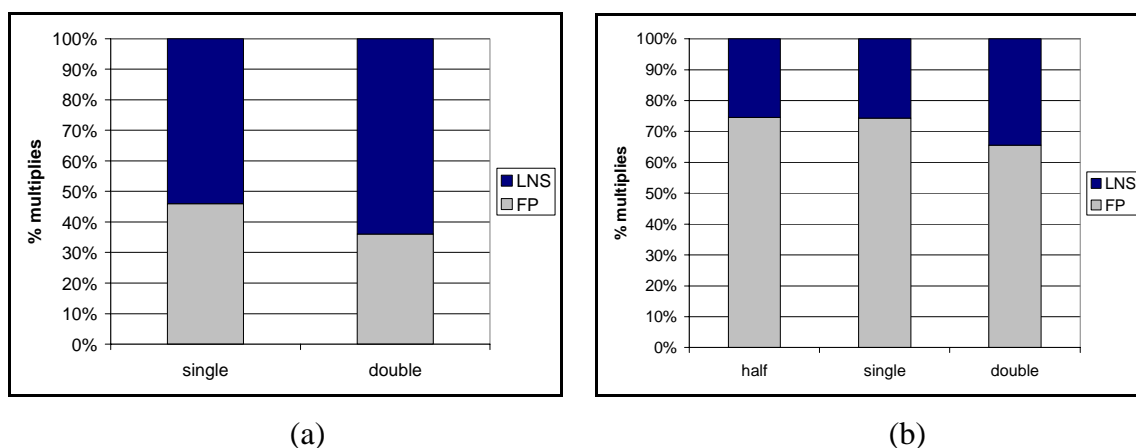


Figure 9. Plot of the percentage of multiplies versus additions in a circuit that make LNS beneficial in latency for the Underwood library (a) and the Leeser (b) library.

Figure 10 is identical to figure 9 but for addition versus division instead of multiplication. The large difference in the break even points of the two for single precision are due to the differences in the mantissa divider as discussed previously.

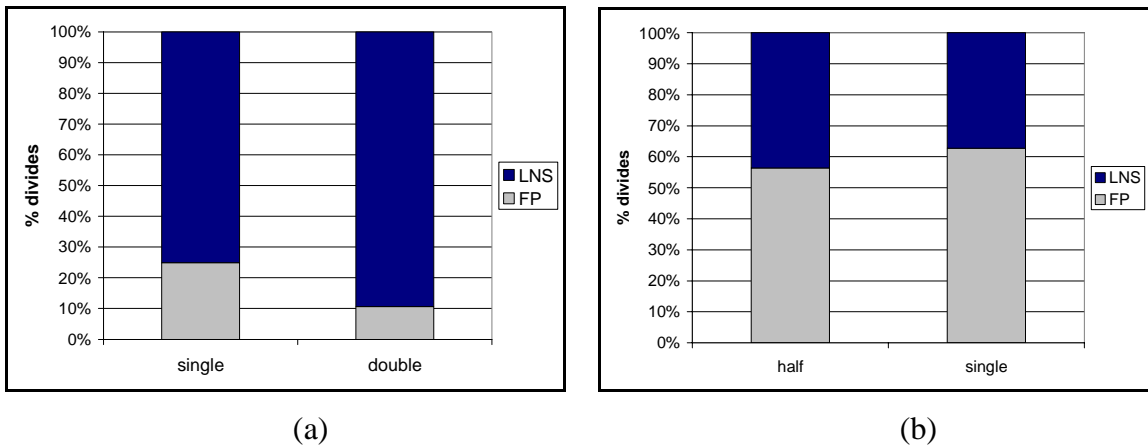


Figure 10. Plot of the percentage of divisions versus additions in a circuit that make LNS beneficial in latency for the Underwood library (a) and the Leeser (b) library.

7 Conclusion

In this paper we created a guide for determining when an FPGA circuit should be performed in floating point or LNS. We showed that while LNS is very efficient at multiplies and divides, the difficulty of addition and conversion are great enough to warrant the use of LNS only for a small niche of algorithms. For example, if a designer is interested in saving area, then the algorithm will need to be at least 70% multiplies or 50-60% 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 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. We also showed that the implementation of the mantissa divider has a larger effect on the latency of the circuit than the area. These results show that for LNS to become a suitable alternative to floating point for FPGAs, better addition and conversion algorithms need to be developed to more efficiently compute the non-linear functions.

BIBLIOGRAPHY

Y. Wan, C.L. Wey, "Efficient Algorithms for Binary Logarithmic Conversion and Addition," *IEEE Proceedings, Computers and Digital Techniques*, vol.146, no.3, May 1999.

I. Koren, *Computer Arithmetic Algorithms*, 2nd Edition, A.K. Peters, Ltd., Natick, MA, 2002.

P. Belonovic, M. Leeser, "A Library of Parameterized Floating Point Modules and Their Use" *12th International Conference on Field Programmable Logic and Applications*. September, 2002.

J. Detrey, F. Dinechin, "A VHDL Library of LNS Operators", *Signals, Systems & Computers, 2003 The Thirty-Seventh Asilomar Conference on* , vol. 2 , 9-12 Nov. 2003, pp. 2227 – 2231.

D.M. Lewis, "An Accurate LNS Arithmetic Unit Using Interleaved Memory Function Interpolator", *Computer Arithmetic, 1993. Proceedings., 11th Symposium on* , 29 June-2 July 1993, pp. 2-9.

K.H. Tsoi et al., "An arithmetic library and its application to the N-body problem", *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on* , 20-23 April 2004 pp. 68 – 78.

B.R. Lee, N. Burgess, "A parallel look-up logarithmic number system addition/subtraction scheme for FPGA", *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on* , 15-17 Dec. 2003 pp. 76 – 83.

J.N. Coleman et al., "Arithmetic on the European logarithmic microprocessor", *Computers, IEEE Transactions on* , vol. 49 , issue 7 , July 2000 pp. 702 – 715.

J.N. Coleman, E.I. Chester, "A 32 bit logarithmic arithmetic unit and its performance compared to floating-point", *Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on*, 14-16 April 1999 pp. 142 – 151.

B. Fagin, C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic", *IEEE Transactions on VLSI Systems*, Vol. 2, No. 3, Sept. 1994, pp. 365-367.

L. Louca, T.A. Cook, W.H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs", *FPGA's for Custom Computing*, 1996.

R. Matousek et al., "Logarithmic Number System and Floating-Point Arithmetics on FPGA", *FPL*, 2002, LNCS 2438, pp. 627-636.

Y. Wan, M.A. Khalil, C.L. Wey., "Efficient conversion algorithms for long-word-length binary logarithmic numbers and logic implementation", *IEE Proc. Comput. Digit. Tech*", vol. 146, no. 6. November 1999.

K. Underwood, "FPGA's vs. CPU's: Trends in Peak Floating Point Performance," *FPGA 04*.

J.N. Coleman et al., "Arithmetic on the European logarithmic Microprocessor" *IEEE Transactions on Computers*, vol. 49, no. 7, 2000, pp. 702-715.

Chipworks. Xilinx_XC2V1000_die_photo.jpg www.chipworks.com

J. Liang, R. Tessier, O. Mencer, "Floating Point Unit Generation and Evaluation for FPGAs" *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, 9-11 April 2003 pp. 185 – 194.

R. Matousek et al., "Logarithmic Number Systems and Floating Point Arithmetics on FPGA" *FPL 2002*, pp. 627-636.

K.H. Abed, R.E. Siferd, "CMOS VLSI implementation of a low-power logarithmic converter" *Computers, IEEE Transactions on*, vol. 52, issue 11, Nov. 2003, pp. 1421 – 1433.

G. Lienhart, A. Kugel, R. Manner, "Using floating-point arithmetic on FPGAs to accelerate scientific N-Body simulations", *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, 22-24 April 2002, pp. 182 – 191.

A. Walters, P. Athanas, "A scaleable FIR filter using 32-bit floating-point complex arithmetic on a configurable computing machine", *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, 15-17 April 1998, pp. 333 – 334.

W. Xiaojun, B.E. Nelson, "Tradeoffs of designing floating-point division and square root on Virtex FPGAs", *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, 9-11 April 2003, pp. 195 – 203.

Y. Li, W. Chu, "Implementation of single precision floating point square root on FPGAs", *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, 16-18 April 1997, pp. 226 – 232.

C.H. Ho et al., "Rapid prototyping of FPGA based floating point DSP systems", *Rapid System Prototyping, 2002. Proceedings. 13th IEEE International Workshop on*, 1-3 July 2002, pp.19 – 24.

A.A. Gaffar, "Automating Customisation of Floating-Point Designs", *FPL 2002*, pp. 523-533.

Xilinx, "Virtex-II Platform FPGAs: Complete Data Sheet", v3.4, March 1, 2005.