

A Comparison of Floating Point and Logarithmic Number Systems for 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 Laboratories*
Albuquerque, NM
{kdunder, kshemme}@sandia.gov

Abstract

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 [1,4-9,13]. However, this advantage comes at the cost of complicated, inexact addition and subtraction, as well as the need to convert between the formats. In this work, we created a parameterized LNS library of computational units and compared them to an existing floating point library. Specifically, we considered multiplication, division, addition, subtraction, and format conversion to determine when one format should be used over the other and when it is advantageous to change formats during a calculation.

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. The requirements for performance and a large dynamic range lead to the use of floating point or logarithmic number systems in hardware. Reconfigurable hardware allows designers to select a range and precision most appropriate to their application which may lead to better performance.

*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.

FPGA designers began mapping floating point arithmetic units to FPGAs in the mid 90's [3,10,11,14]. 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 the 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. 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. Number Systems

2.1 Floating point

A floating point number F has the value [2]

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

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 [14]). 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:

S	Exponent E	Unsigned Significand f
1 bit	e bits	m bits

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 11 and e is 52, 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.

	f = 0	f ≠ 0
E = 0	0	Denormalized
E = max	±∞	NaN

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

2.2 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 [2]. 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:

	S_A	E_A	
flags	Sign	integer	fraction
2 bits	1 bit	K bits	l bits

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].

If $k=e$ and $l=m$ the 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.

3. 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 [14] to the same device for comparison.

Most of the commercial FPGAs contain dedicated RAM 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 look up tables 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 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.

3.1 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:

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

The sign is the XOR of the multiplier and multiplicand's sign bits and 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 $\pm\infty$.

Floating point 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:

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

Since both exponents each have a bias component one bias must be subtracted. Again, an overflow event must be detected. 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 possible overflow.

3.2 Division

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

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

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 zero.

Division in floating point is accomplished by dividing the mantissas and subtracting the divisor's exponent from the dividends exponent [2]. 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.

3.3 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 ($|A| \geq |B|$) 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

$$S_c = S_A$$

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

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

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 value of $f(x)$ shown in figure 1, 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[13]. Notice that if $x=0$ then $f(x)$ for subtraction is negative infinity so this needs to be detected.

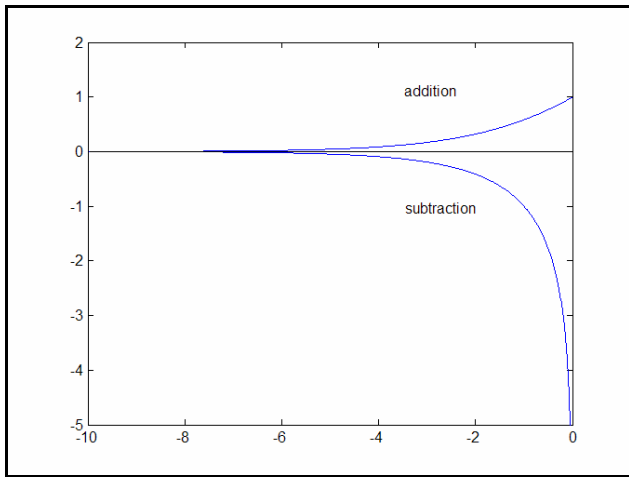


Figure 1. Plot of $f(x)$ for addition and subtraction [12].

For our LNS library, the algorithm from Lewis [5] 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. To increase the accuracy, the x axis is broken up into even size intervals where the intervals closer to zero have more points stored (ie. 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 requires 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. 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 [2]. 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.

3.5 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.

3.5.1 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:

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

$$= \log_2(1.xxx...) + \text{exp}$$

Notice that the $\log_2(1.xxx\dots)$ 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 $\log_2(1.x_1x_2\dots x_n)$. 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 floating point mantissa as

$$(1.x_1x_2\dots x_n) = (1.x_1x_2\dots x_m)(1.00\dots 0c_1c_2\dots c_m)$$

$$\log_2(1.x_1x_2\dots x_n) = \log_2[(1.x_1x_2\dots x_m)(1.00\dots 0c_1c_2\dots c_m)]$$

$$= \log_2(1.x_1x_2\dots x_m) + \log_2(1.00\dots 0c_1c_2\dots c_m)$$

where $m = n/2$ and

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

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

If $c = .c_1c_2\dots c_m$, $b = .x_{m+1}x_{m+2}\dots x_{2m}$ and $a = .x_1x_2\dots x_m$ then

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

In attempt to avoid a slow division, the computation can be rewritten as

$$c = b/(1+a) = (1+b)/(1+a) - 1/(1+a)$$

$$= 2^{\log(1+b) - \log(1+a)} - 2^{-\log(1+a)}$$

where $\log(1+b)$ and $\log(1+a)$ can be looked up in the same ROM for $\log_2(1.x_1x_2\dots x_m)$. 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 $4 \log_2(1.x_1x_2\dots x_m)$ and $1 \log_2(1.0\dots 0c_1c_2\dots c_m)$ 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 [13] 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

$$\log(1 + .x_1x_2\dots x_n) \cong \log(1 + q_1)$$

$$+ \log(1 + 0q_2) + \dots + \log(1 + 00\dots 0q_n)$$

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

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

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 the calculating the quotient in a binary division except the divisor is updated at each iteration. So modified restoring division is used to find all q'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 nxn 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.

3.52 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

$$.a_1a_2..a_n = x_1x_2..x_n \frac{y_1y_2..y_n}{k} \dots \frac{z_1z_2..z_n}{k} \dots$$

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 $(2^k \times k) \times n$ 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.

4 Results

The following tables show the results of mapping the floating point and logarithmic libraries to a 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 or the time from input to output as this metric is very circuit dependent.

4.1 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

Table 2. Area and latency of a multiplication.

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.

4.2 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

Table 3. Area and latency of a division.

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.

4.3 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

Table 4. Area and latency of an addition.

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.

4.4 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	0.78
norm. area	424	1491	836	3907
latency(ns)	45	95	67	110

Table 5. Area of a subtraction.

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.

4.5 Convert floating point to LNS

	single	double
slices	163	7631
multipliers	0	0
18K BRAM	24	0
units per FPGA	2.333	1.4
norm. area	832.6	7712
latency(ns)	76	380

Table 6. Area and latency of a conversion from floating point to LNS as discussed in section 3.5.1.

Converting from floating point to LNS has typically been done with ROMs [2] for small word sizes. For larger words, an approximation must be used. Two algorithms were used for this approximation, one for single precision [1] and another for double precision [13]. 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.

4.6 Convert LNS to floating point

	single	double
slices	236	3152
multipliers	20	226
18K BRAM	4	14
units per FPGA	2.8	0.25
norm. area	8.66	9713
latency(ns)	72	84

Table 7. Area and latency of a conversion from LNS to floating point.

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.

5 Analysis

5.1 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 2 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, 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.

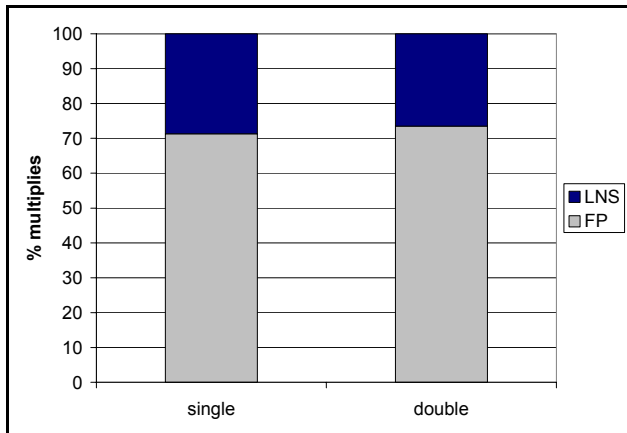


Figure 2. Plot of the percentage of multiplies versus additions in a circuit that make LNS beneficial as far as normalized area.

Figure 3 shows the same analysis as figure 2 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. 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

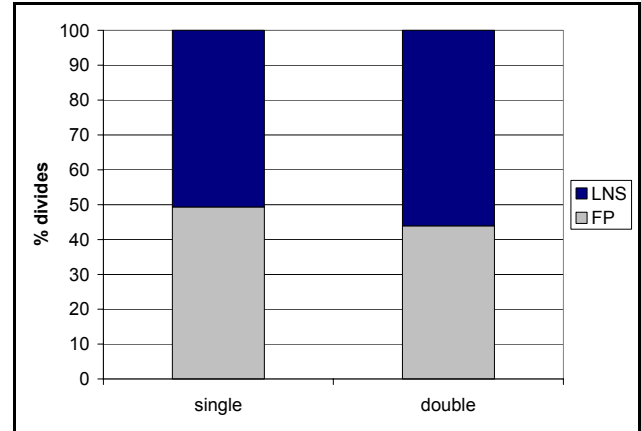


Figure 3. Plot of the percentage of divides versus additions in a circuit that make LNS beneficial as far as normalized area.

Notice that 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.

5.2 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 need per addition to make the conversion beneficial. The curves in figure 4 show the break even point of circuit size if it was done in floating point or 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.

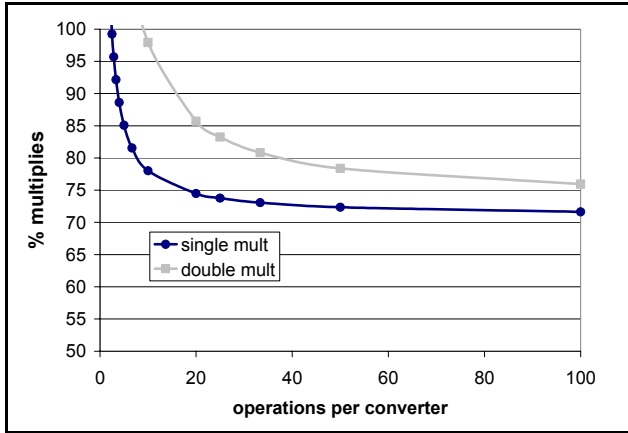


Figure 4. Plot of the operations per converter and percent multiplies in a circuit to make the area the same for LNS and floating point.

As can be seen in figure 4, 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 2 where there is assumed no conversion. This is because when many operations per conversion are being performed, the cost of conversion becomes irrelevant.

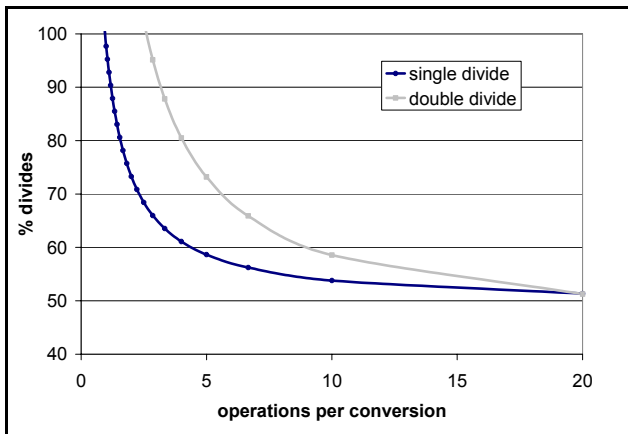


Figure 5. Plot of the percentage of divides versus additions required in a circuit to make LNS beneficial if a conversion is required.

Figure 5 is identical to figure 4 except that it is for divides versus additions instead of multiplies.

Notice that the 100% line in both figure 4 and 5 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.

5.3 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 6 show the percentage of multiplies versus additions on the critical path that is required to make LNS faster. For example, 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.

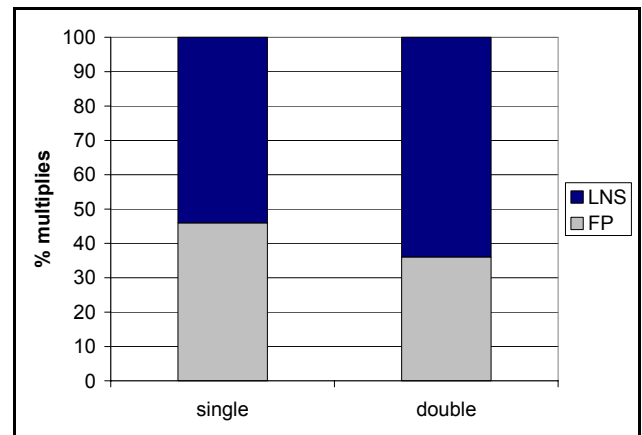


Figure 6. Plot of the percentage of multiplies versus additions in a circuit that make LNS beneficial in latency.

Figure 7 is identical to figure 6 but for division instead of multiplication.

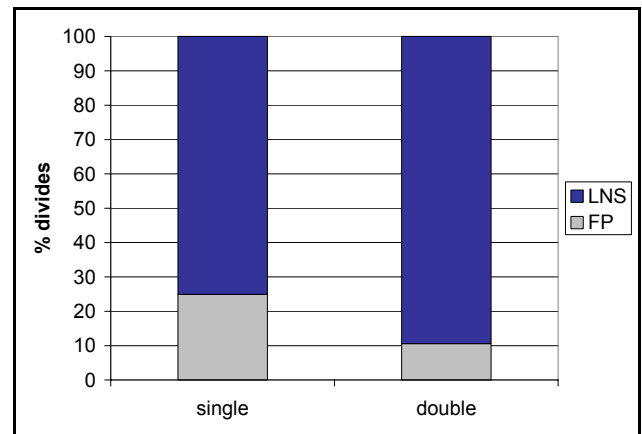


Figure 7. Plot of the percentage of multiplies versus additions in a circuit that make LNS beneficial in latency.

6 Conclusion

In this paper we created a guide for determining when an FPGA circuit should be performed in floating point or the LNS. We have also developed a parameterized LNS library of multiply, divide, addition, and subtraction along with converters for changing formats. We showed how the makeup of the computation and the word sizes determines what format would be faster and more area efficient to use. We also showed that if conversion from and to floating point at the I/O is required that the number of operations per conversion must be high enough to outweigh the overhead of conversion.

7 Acknowledgements

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. Initial work on floating point libraries was performed by Henry Lee.

8 References

- [1]Wan, Y. and Wey, C.L., "Efficient Algorithms for Binary Logarithmic Conversion and Addition," *IEE Proceedings, Computers and Digital Techniques*, Vol.146, No.3, pp.168-176, May 1999.
- [2]Koren, Israel, *Computer Arithmetic Algorithms*, 2nd Edition, A.K. Peters, Ltd., Natick, MA, 2002.
- [3]Pavle Belonovic and Miriam Leeser, "A Library of Parameterized Floating Point Modules and Their Use." *12th International Conference on Field Programmable Logic and Application*. September, 2002.
- [4]Detrey, Jérémie and Dinechin, Florent de, "A VHDL Library of LNS Operators", *Signals, Systems & Computers*, 2003 The Thirty-Seventh Asilomar Conference on , Volume: 2 , 9-12 Nov. 2003, Pages:2227 – 2231.
- [5]Lewis, D.M., "An Accurate LNS Arithmetic Unit Using Interleaved Memory Function Interpolator", *Computer Arithmetic*, 1993. Proceedings., 11th Symposium on , 29 June-2 July 1993, Pages:2-9.
- [6]Tsoi, K.H., Ho, C.H., Yeung, H.C., Leong, P.H.W., "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 Pages:68 – 78.
- [7]Lee, B.R.; Burgess, N., "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 Pages:76 – 83.
- [8]Coleman, J.N.; Chester, E.I.; Softley, C.I.; Kadlec, J., "Arithmetic on the European logarithmic microprocessor", *Computers, IEEE Transactions on* , Volume: 49 , Issue: 7 , July 2000 Pages:702 – 715.
- [9]Coleman, J.N., Chester, E.I., "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 Pages:142 – 151.
- [10]B. Fagin, C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic", *IEEE Transactions on VLSI Systems*, Vol. 2, No. 3, pp. 365-367, Sept. 1994.
- [11]Louca, Loucas, Cook, Todd A., Johnson, William H., "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs", *FPGA's for Custom Computing*, 1996.
- [12]Matousek, R., Tichy, M., Pohl, Z., Kadlec, J., Softley, C., Coleman, N., "Logarithmic Number System and Floating-Point Arithmetics on FPGA", *FPL*, 2002, LNCS 2438, pp. 627-636.
- [13]Wan, Y., Khalil, M.A., Wey, C.L., "Efficient conversion algorithms for long-word-length binary logarithmic numbers and logic implementation", *IEE Proc. Comput. Digit. Tech*", Vol. 146, No. 6. November 1999.
- [14]Underwood, Keith, "FPGA's vs. CPU's: Trends in Peak Floating Point Performance," *FPGA 04*.