A Tool for Unbiased Comparison between Logarithmic and Floating-point Arithmetic

JÉRÉMIE DETREY AND FLORENT DE DINECHIN
*Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon,
46, allée d'Italie, 69364, Lyon cedex 07, France*

Abstract. For applications requiring a large dynamic, real numbers may be represented either in floating-point, or in the logarithm number system (LNS). Which system is best for a given application is difficult to know in advance, because the cost and performance of LNS operators depend on the target accuracy in a highly non linear way. Therefore, a comparison of the pros and cons of both number systems in terms of cost, performance and overall accuracy is only relevant on a per-application basis. To make such a comparison possible, two concurrent libraries of parameterized arithmetic operators, targeting recent field-programmable gate arrays, are presented. They are unbiased in the sense that they strive to reflect the state-of-the-art for both number systems. These libraries are freely available at http://www.ens-lyon.fr/LIP/Arenaire/.

1. Introduction

1.1. Hardware Representations of Real Numbers

Digital signal processing (DSP) relies mostly on fixed-point arithmetic. However, some DSP applications such as adaptive filters compute on numbers with a large dynamic range. The mainstream solution in this case is the use of floating-point arithmetic, which is supported by recent high-end DSP processors. Another recent contender on the DSP market is the FPGA (for *Field Programmable Gate Array*), a programmable VLSI circuit which can be configured at the bit-level to emulate any digital circuit. Initially essentially used for the rapid prototyping of application-specific integrated circuits (ASIC), FPGAs are increasingly being used as hardware accelerators for specific computations. Here also, fixed-point is preferred when applicable, all the more as the fine-grained structure of FPGAs is optimized for fixed-point. However, as the capacity of FPGAs increases, so does the complexity of their applications: Many floating-point applications were published in the last years [3, 7, 10, 11, 12, 16, 19, 22, 24, 26]. Some of these works emulate the floating-point formats available in processors, but the flexibility of FPGAs also allows to adapt these formats to match the precision and dynamic requirements of a given application.

Floating-point is not the only way to represent real numbers in hardware circuits with a larger dynamic than fixed-point: One can also use a *logarithmic* coding (or LNS for *Logarithmic Number System*). A positive real number is then represented by its logarithm (usually in radix 2), and the hardware operators compute on these logarithms. The main advantage of this coding is that multiplications, divisions and square roots are trivial with logarithms. The main drawback is, of course, that additions and subtractions are much more complicated. Given this tradeoff, several publications have shown applications for which this system was more efficient in terms of speed and area than floating-point [5, 23].

1.2. Which Arithmetic for Which Application?

Qualitatively, it is clear that LNS arithmetic can be competitive only if the application matches two conditions: There has to be many easy operations ($\times$, $/$, x^2 et $\sqrt{x}$) and few additions, and the required precision has to be quite low, as the area of an LNS adder grows exponentially with precision (see Section 2.3). Quantitatively, it is much more difficult to have a precise answer. The best comparative study was that of Coleman *et al.* [5]: It considers several representative algorithms in two precisions, and studies both accuracy and performance. It is however clear that the authors took less care while designing floating-point operators than LNS ones, and the comparisons are biased. For example, their floating-point square root is a Newton iteration, which is quite inefficient in this context. Moreover, they only target ASIC applications. For FPGAs, there is a paper by Matoušek *et al.* [23], but the example algorithm they study is a mere caricature: Its iteration only has one addition for two divisions, three multiplications, two squares and one square root. Such an uncommon algorithm will not convince a designer to try LNS for more classical circuits.

Are there real applications for which LNS is better suited than FP, and more generally, how can we help designers evaluate the pros and cons of each arithmetics for their application? A problem is that the costs of some LNS operators with respect to precision are highly non-linear. These costs also depend on the target technology, and a variety of algorithms expose wide area/speed tradeoffs, as the sequel will show.

Another problem is the evaluation of the overall accuracy of the application (or its signal to noise ratio). On one side, both systems, for the same number of bits, represent numbers with comparable range and precision. On the other side, the rounding errors due to operations may be very different. In FP, all the operations may involve a rounding error. In LNS, multiplications and divisions are exact (as they are implemented as fixed-point addition and subtraction) but addition and subtraction involve rounding errors which may be larger than that of FP. The net effect of combining these errors in one's application is difficult to predict. If one of the number systems provides the same overall accuracy for a smaller precision of the operators, this in turn has an impact on their respective costs, as will be illustrated in Section 3.3.3.

The conclusion is that it is probably impossible in a publication to exhaustively cover the set of parameters controlling the speed/area/precision tradeoffs for both LNS and FP so that a designer can make an informed choice. As an example, an attempt by Haselman *et al.* [18] only covers the standard IEEE-754 single and double precision: it will not help if the application can accomodate lower or intermediate precisions, as is commonly the case for signal processing.

Therefore, our goal in this paper is not to publish comparisons, but a generic comparison tool.

1.3. A Tool for an Unbiased Comparison

This paper presents a library of operators, supporting both floating-point and LNS formats. This library is freely downloadable from http://www.ens-lyon.fr/LIP/Arenaire/. It allows to choose the precision and the dynamic of numbers, and the operators for the two number system share a common syntax and exceptional case handling, easing the switch from one to the other. It provides operators for addition, subtraction, multiplication, division and square root, along with some useful conversions, in combinatorial or pipelined flavor. It is written in portable VHDL, and all the operators have been designed with equivalent optimization effort.

Our objective is dual: First, to allow a more accurate study of the respective pros and cons of LNS and FP than what current literature offers. Second, to provide designers with all the elements to experiment and choose the number system that best suits a given application, with its operations, its cost/performance constraints, and its dynamic/precision constraints.

The first part of the article briefly describes the number formats and the architectures of the differents operators of the library, with the purpose of showing that this library reflects the state-of-the-art in both systems. The second part gives area and speed benchmarks of these operators according to the different parameters, and examples of the unbiased comparisons we hope this library allows.

2. A Library for Real Number Arithmetic

2.1. Number Representation

The representation of a real number in the library is parameterized by two integers, w_E which determines the dynamic of the represented numbers, and w_F their precision. The ranges of representable values for a

given (w_E, w_F) are not identical for both formats, but are as close as possible given the intrinsic differences between these formats. It is therefore possible to compare these two number systems for a same pair of parameters (w_E, w_F), as the dynamic and the precision of these arithmetics are equivalent. More accurately, relative coding errors between those two formats are within a $\log(2)$ ratio, as can be deduced from the following Eqs. (1) and (2).

2.1.1. Floating-point Representation.
For floating-point numbers, a format inspired by the IEEE-754 standard [1] is adopted: A number X is represented on $3 + w_E + w_F$ bits by two bits for coding exceptional cases, followed by a sign bit S_X, an exponent E_X biased by E_0 on w_E bits, and the fractional part F_X of the mantissa on w_F bits. The mantissa is normalized in $[1; 2[$, so its most significant bit is always 1 and is implicit in the coding:

$$X = (-1)^{S_X} \times 1.F_X \times 2^{E_X - E_0}. \qquad (1)$$

The two extra bits used by the internal format represent exceptional cases such as 0, $\pm\infty$ or *NaN* (*Not a Number*). As the IEEE-754 standard codes these numbers by specific exponent values, our library provides conversion operators from one format to the other. It is also possible to retrieve these bits for exception handling.

Our format diverges from the IEEE-754 standard because it does not support subnormal numbers [17].

2.1.2. LNS Representation.
The LNS format on $3 + w_E + w_F$ bits is composed of the same two bits
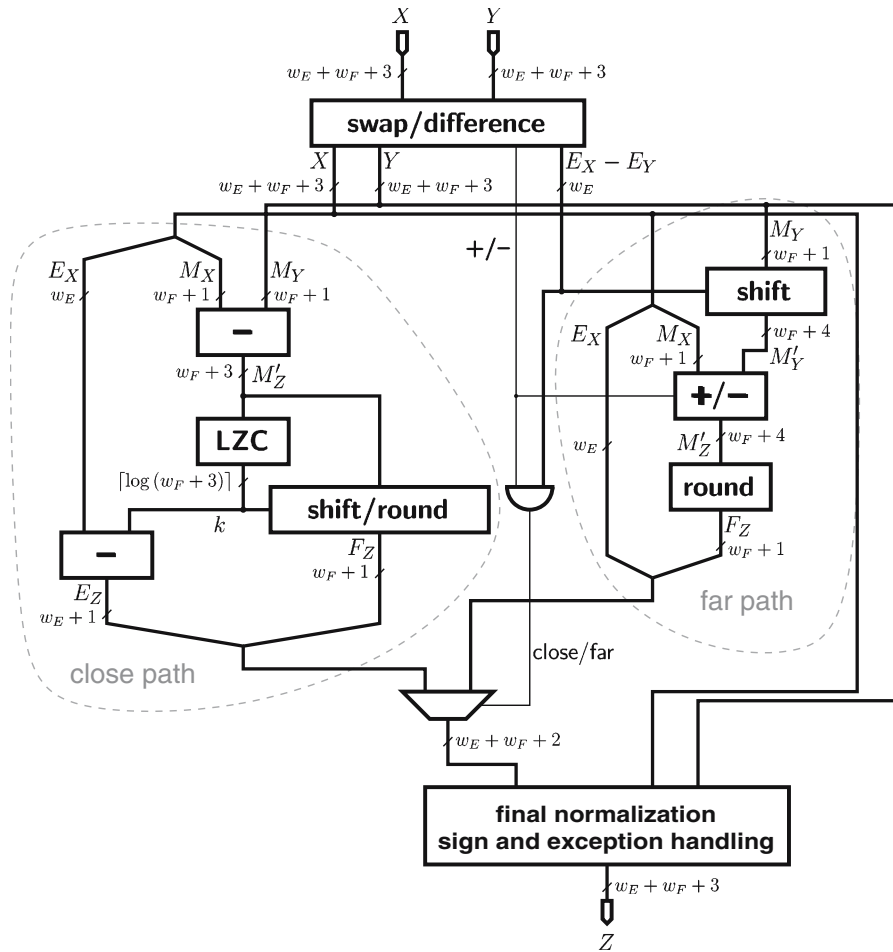

Figure 1. Architecture of the floating-point adder/subtracter.

for exceptional cases, a sign bit S_X, and a fixed-point 2's complement representation of the logarithm $L_X = \log_2(X)$, coded with w_E bits for its integer part E_{L_X} and w_F bits for its fractional part F_{L_X}:

$$X = (-1)^{S_X} \times 2^{E_{L_X}.F_{L_X}}. \tag{2}$$

Exceptional cases are coded exactly as for floating-point numbers. Exception handling is thus identical between FP and LNS, and we shall not detail it any further.

2.2. *Floating-point Operators*

This section does not intend to be a course about hardware floating-point operators: There are whole books discussing this subject [14, 15]. The goal here is to convince the reader that our library is sufficiently optimized to allow an unbiased comparison between LNS and floating-point.

2.2.1. *Addition/subtraction.* To compute a floating-point addition, one first has to align the mantissas,

then add/subtract them and last renormalize the result. But these steps are not always necessary: When the exponents are close (*close path*), the alignment of the mantissas becomes simple whereas conversely, when the exponents are far (*far path*), it is the final normalization which becomes trivial.

The architecture of this operator, shown Fig. 1, therefore uses two concurrent computational paths. Those two paths allow to reduce the critical path at the expense of increasing slightly the area of the operator.

2.2.2. *Multiplication.* The multiplier architecture is simpler than the adder/subtracter architecture: It only has to compute the product of the mantissas of the two operands, and the exponent is obtained by summing the two exponents minus the bias E_0.

The product of the mantissas is expressed as a * in VHDL. This ensures efficient portability: For example, on Xilinx Virtex-II, the synthesis tools can use the embedded small multipliers specific to this FPGA. It also ensures flexibility, as synthesis tools may provide several multiplier variants optimized for
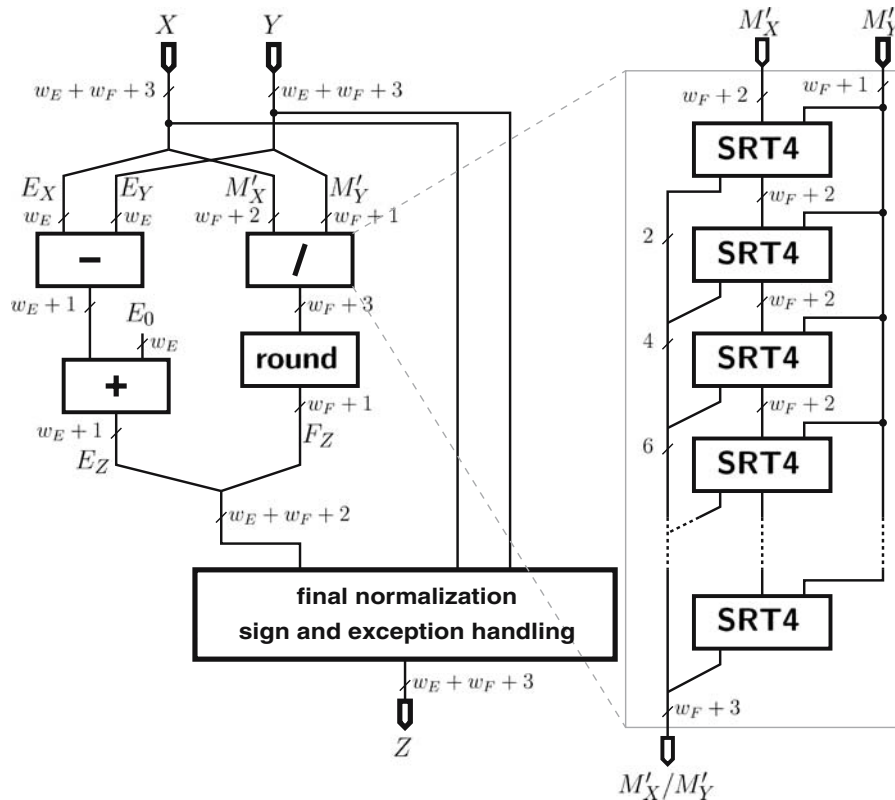

Figure 2.   Architecture of the floating-point divider.

various area/speed tradeoff. However, using a generic multiplier will generally be slightly sub-optimal. If this is a concern, a designer can always provide his own specific multiplier, such as those described in [4].

2.2.3. Division. The global architecture of the divider is shown in Fig. 2.

The result mantissa is the quotient of the mantissas of the two operands, and the exponent is the difference between the two exponents plus the bias E_0. The quotient is computed using a radix 4 SRT algorithm [13] with the digit set $\{-3, -2, -1, 0, +1, +2, +3\}$ which is maximally redundant. Radix 4 SRT was the best solution among the other radices we tried (radix 2, radix 8 or radix 4 with another digit set). These findings are consistent with those of Lee and Burgess [19]. As the quotient is computed in a redundant digit set, a final addition is needed to switch back to binary.

2.2.4. Square Root. The square root operator follows the same principle: The exponent is divided by two and corrected by adding half a bias $E_0/2$, while the square root of the mantissa is computed by a radix 2 SRT (appearing to be better than radix 4) [13].

2.3. LNS Operators

2.3.1. Multiplication, Division and Square Root. The main advantage of LNS is the simplicity of these operators:

$$L_{X \times Y} = L_X + L_Y,$$
$$L_{X/Y} = L_X - L_Y,$$
$$L_{\sqrt{X}} = \frac{1}{2} L_X.$$

Multiplication, division and square root are therefore implemented respectively by addition, subtraction and right shift of the logarithms of the
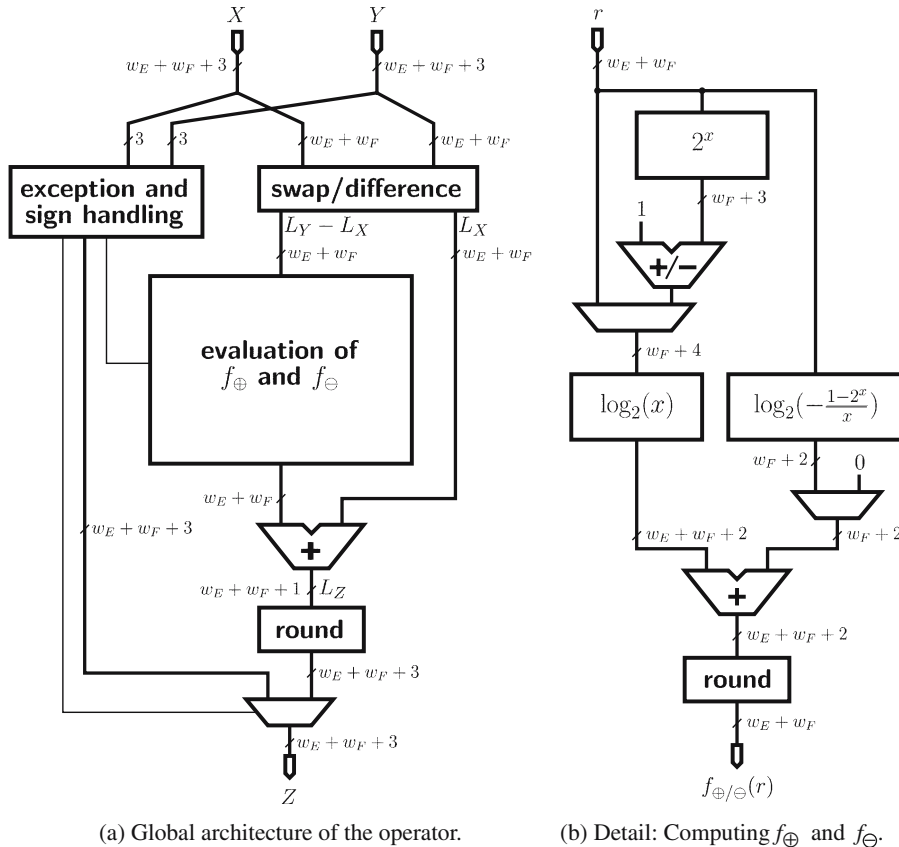

(a) Global architecture of the operator.

(b) Detail: Computing $f_\oplus$ and $f_\ominus$.

Figure 3.   Architecture of the LNS adder/subtracter.

operands. The following focuses only on the addition/subtraction.

2.3.2. General Architecture of the Addition/subtraction Operator.

Performing an addition or a subtraction in LNS is much more complicated than in floating-point, as it requires the evaluation of two non-linear functions $f_\oplus$ and $f_\ominus$ defined as follows (here, X and Y are both positive numbers such that $X > Y$):

$$L_{X+Y} = \log_2(2^{L_X} + 2^{L_Y}) = L_X + f_\oplus(L_Y - L_X),$$
$$\text{with} \quad f_\oplus(r) = \log_2(1 + 2^r),$$
$$L_{X-Y} = \log_2(2^{L_X} - 2^{L_Y}) = L_X + f_\ominus(L_Y - L_X),$$
$$\text{with} \quad f_\ominus(r) = \log_2(1 - 2^r).$$

The architecture of this operator is shown in Fig. 3a. The two main components evaluate approximations to $f_\oplus$ and $f_\ominus$ (also represented in the figure) on the interval $]-\infty; 0)$.

2.3.3. Implementation of $f_\oplus$ and $f_\ominus$.

Given the intrisically non-linear nature of $f_\oplus$ and $f_\ominus$, we have to use an approximation scheme for evaluating these functions. The literature proposes many solutions that cover a wide tradeoff between speed and operator precision. In [27], the functions are evaluated with only one table lookup on intervals finely tuned to minimize the table size (order 0). In [21], Lewis uses a first-order Taylor series that reduces the size of the tables while increasing the critical path. Finally, in [5], the authors introduce a method using a degree 2 polynomial which greatly reduces the size of the tables, but requires two table lookups, one multiplication and some additions in the critical path.

In the first release of our library, we limited ourselves to a multiplier-less first-order approximation scheme, using the multipartite table method [6]. This choice is widely discussed in [8], but limits the achievable precision for this operator to $w_F \leq 13$ bits.
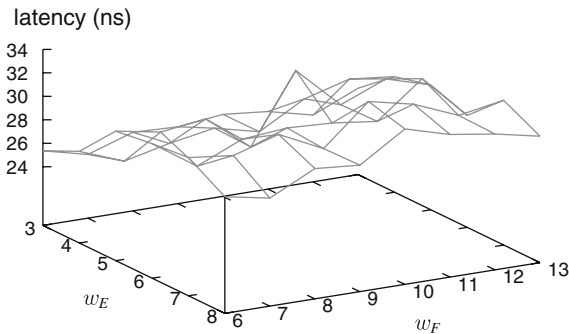
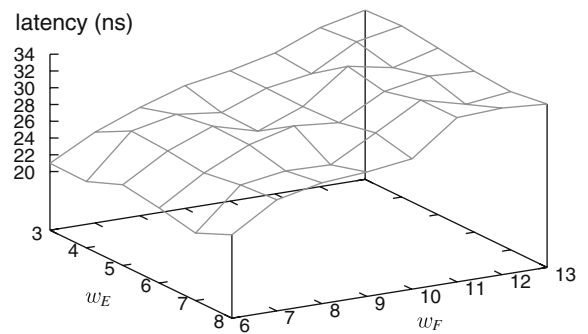We then studied other possible decompositions and simplifications of these functions, focusing particular-
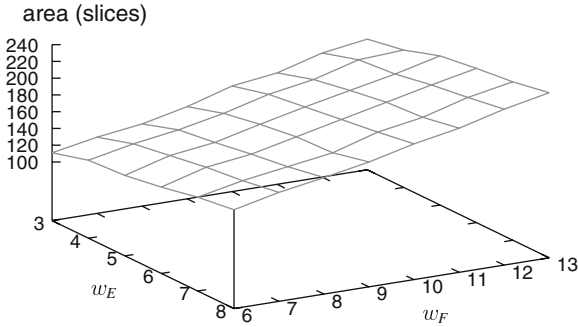
(a) FP adder area
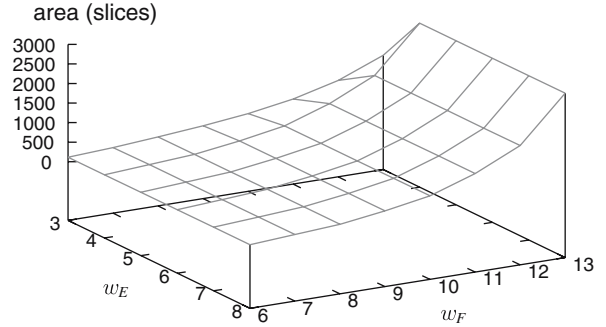
(b) LNS adder area

(c) FP adder latency

(d) LNS adder latency

Figure 4. Area/delay for floating-point (*on the left*) and LNS (*on the right*) adders. Note the different scales.

ly on the method presented in [25] and already used by [20]. This method, whose architecture is depicted Fig. 3b, relies on the evaluation of the functions 2^x, $\log_2(x)$ and $\log_2\left(-\frac{1-2^x}{x}\right)$ which can be faithfully approximated by second-order polynomials (here "faithfully" means that the error is smaller than the value of the least significant bit of the result), something not possible for $f_\oplus$ and $f_\ominus$. Nevertheless, the critical path is greatly increased, as the 2^x and $\log_2(x)$ functions have to be evaluated sequentially and also require some range reduction mechanism. The evaluation itself of those functions is achieved by using a second-order method presented in [9], which allows precisions up to $w_F = 23$ bits.

To summarize, our library currently proposes two flavors for the LNS adder, one based on the multipartite table method, noted O1 (order 1), which is fast but bulky and limited to $w_F \leq 13$ bits, and the other based on the decomposition from [25], noted O2 (order 2), which is much smaller but also slower. These two implementations reflect the state of the art.

3. Precision/Performance Tradeoff: Which Number System for Which Application?

As previously remarked, the characteristics (area and latency) of the operators depend on the chosen number format and range and precision parameters. If the application dictates the latter two, the designer still has to choose between floating-point and LNS. Therefore, this section first presents compared benchmarks of the various operators, allowing to compute a rough estimation the area and latency of a circuit for the two number systems. Then, with some examples,
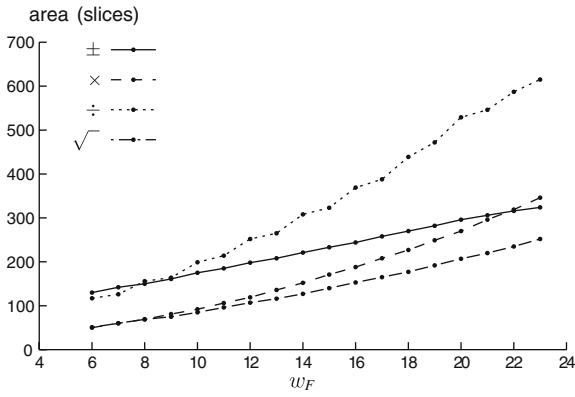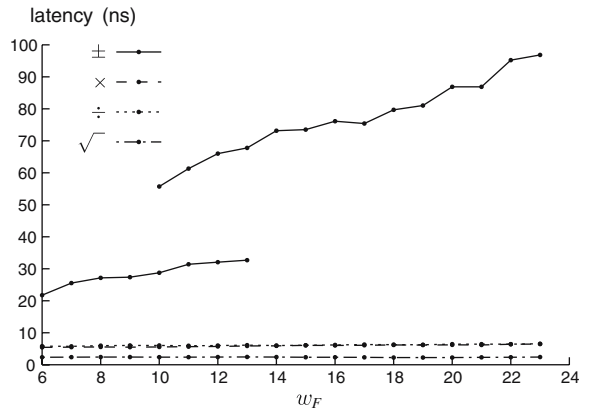

(a) FP operators area
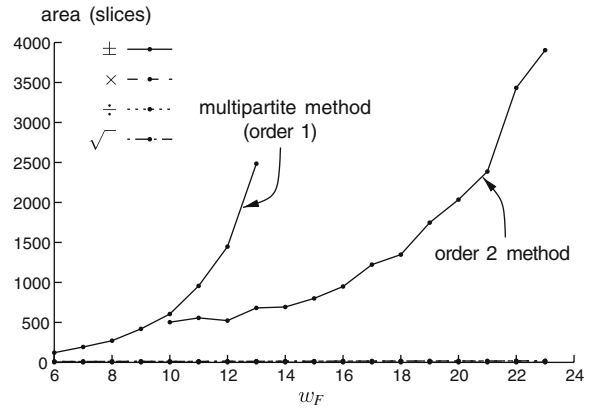
(b) LNS operators area
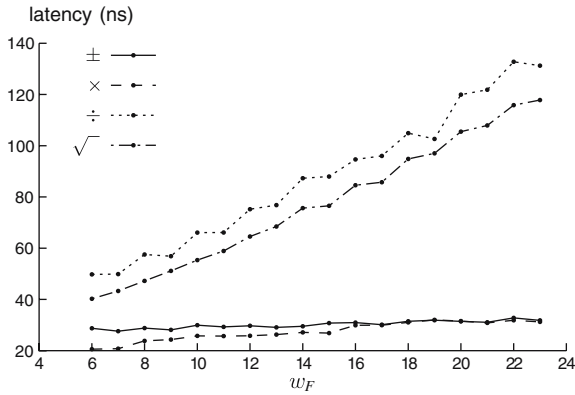
(c) FP operators latency

(d) LNS operators latency

Figure 5. Benchmarks for floating-point (*on the left*) and LNS (*on the right*) operators. Note the different scales.

Table 1. Area (slices) and latency (ns) comparison of some examples for various parameter combinations. The estimations for the compound operators are obtained by adding the values for the simple operators.

data width (w_E, w_F)	$A + B$	$A \times B$	$A \times B + C$	$\sqrt{A^2 + B^2}$
10 bits $(3, 6)$ FP	111 sl.—25 ns	46 sl.—19 ns	157 sl.—44 ns	250 sl.—84 ns
10 bits $(3, 6)$ LNS-O1	114 sl.—21 ns	9 sl.—5 ns	123 sl.—26 ns	133 sl.—28 ns
14 bits $(5, 8)$ FP	148 sl.—30 ns	68 sl.—25 ns	216 sl.—55 ns	354 sl.—103 ns
14 bits $(5, 8)$ LNS-O1	269 sl.—28 ns	11 sl.—5 ns	280 sl.—33 ns	292 sl.—36 ns
16 bits $(5, 10)$ FP	173 sl.—33 ns	92 sl.—26 ns	265 sl.—59 ns	442 sl.—114 ns
16 bits $(5, 10)$ LNS-O1	627 sl.—28 ns	12 sl.—5 ns	639 sl.—33 ns	652 sl.—35 ns
24 bits $(7, 16)$ FP	260 sl.—35 ns	191 sl.—30 ns	451 sl.—65 ns	866 sl.—153 ns
24 bits $(7, 16)$ LNS-O2	930 sl.—76 ns	16 sl.—6 ns	946 sl.—82 ns	1,877 sl.—84 ns
32 bits $(8, 23)$ FP	351 sl.—34 ns	351 sl.—31 ns	702 sl.—65 ns	1,310 sl.—183 ns
32 bits $(8, 23)$ LNS-O2	3,904 sl.—97 ns	20 sl.—7 ns	3,924 sl.—104 ns	7,829 sl.—106 ns

a second section shows that a finer estimation can be obtained by effective synthesis of the circuit. Three complete examples illustrate this methodology.

In this section, all the estimations are given by the Xilinx ISE 5.2 tool suite for a Virtex-II XC2V1000-4 FPGA.

3.1. Comparison of Isolated Operators

The plots of Fig. 4 give the area and latency for adders in the two number systems depending on the dynamic (w_E) and precision (w_F) parameters. Latencies are given mainly for comparative purpose, as the pipelined versions of these operators are more likely to be used (however, latency is still critical when dealing with loops in the circuit).

As expected, for floating-point addition/subtraction operators, area and latency grows linearly with

w_E and w_F. However, for LNS operators, the area grows linearly with w_E but exponentially with w_F, whereas the latency remains linear for w_E and w_F because the latency of table lookups is logarithmic in respect to the size of the tables.

For all operators, both in floating-point and LNS, area and latency estimations depend essentially on the required precision w_F and not on the dynamic w_E. For the sake of readability, only the O1 method was shown on Fig. 4 but the same conclusion applies to the O2 method. Therefore, all further comparisons will be presented here depending only on w_F.

The plots of Fig. 5 present area and latency estimates for all the operators provided by the libraries. It can be noted that the four floating-point operators have roughly the same area, but if addition/subtraction and multiplication have relatively low latencies, the digit recurrence algorithms of division
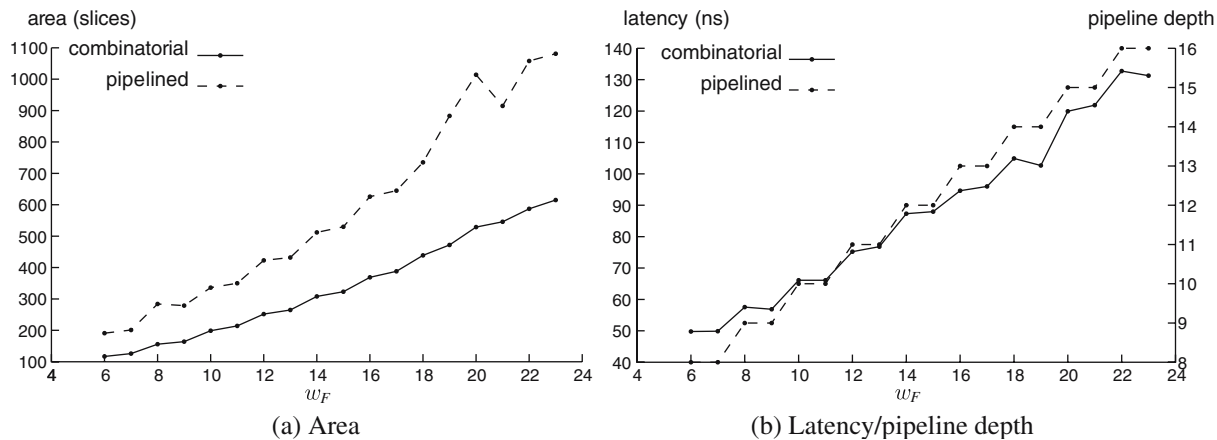

(a) Area (b) Latency/pipeline depth

Figure 6. Benchmarks for combinatorial and pipelined floating-point dividers.

```
1    library ieee;
2    use ieee.std_logic_1164.all;
3    library fplib;
4    use fplib.pkg_fplib.all;
5
6    entity Norm is
7      port ( A : in  std_logic_vector(6+13+2 downto 0);
8             B : in  std_logic_vector(6+13+2 downto 0);
9             R : out std_logic_vector(6+13+2 downto 0) );
10   end entity;
11
12   architecture arch of Norm is
13     constant fmt : format   := FP;
14     constant wE  : positive := 6;
15     constant wF  : positive := 13;
16
17     signal A2 : std_logic_vector(wE+wF+2 downto 0);
18     signal B2 : std_logic_vector(wE+wF+2 downto 0);
19     signal R2 : std_logic_vector(wE+wF+2 downto 0);
20   begin
21     mul_a_a : Mul
22        generic map ( fmt, wE, wF )
23        port map ( A, A, A2 );
24
25     mul_b_b : Mul
26        generic map ( fmt, wE, wF )
27        port map ( B, B, B2 );
28
29     add_a2_b2 : Add
30        generic map ( fmt, wE, wF )
31        port map ( A2, B2, R2 );
32
33     sqrt_r2 : Sqrt
34        generic map ( fmt, wE, wF )
35        port map ( R2, R );
36   end architecture;
```

(a) VHDL code
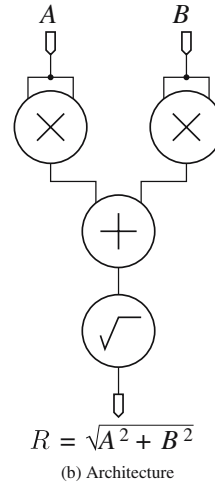
$$R = \sqrt{A^2 + B^2}$$

(b) Architecture

Figure 7. VHDL code and architecture of the norm operator $R = \sqrt{A^2 + B^2}$.

and square root are significantly slower. The area/latency tradeoff also clearly appears for the LNS adder: when applicable, the O1 multipartite method is faster but bulkier than the O2 method.

These graphs can be used to quickly compute a very rough estimation of the area and latency of a given circuit according to the dynamic, the precision and the number representation format. Examples are given in Table 1.

All the previous estimations are given only for combinatorial operators. These operators are also available in pipelined version, designed to run at 100 MHz. As shown by Fig. 6, the characteristics of the pipelined operators roughly follow those of their combinatorial counterpart: their area is slightly higher but remains proportional to the combinatorial area, and the pipeline depth is also proportional to the combinatorial delay.
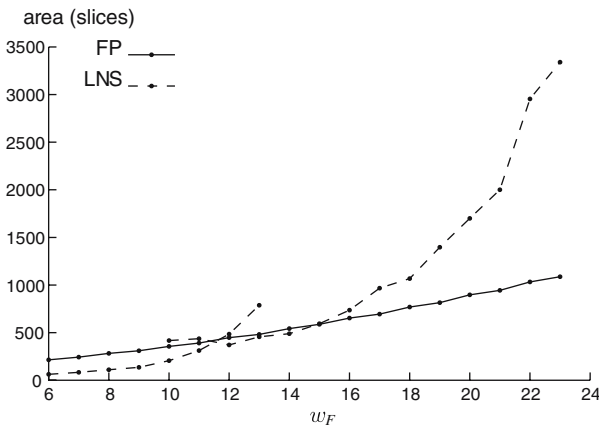
(a) Area
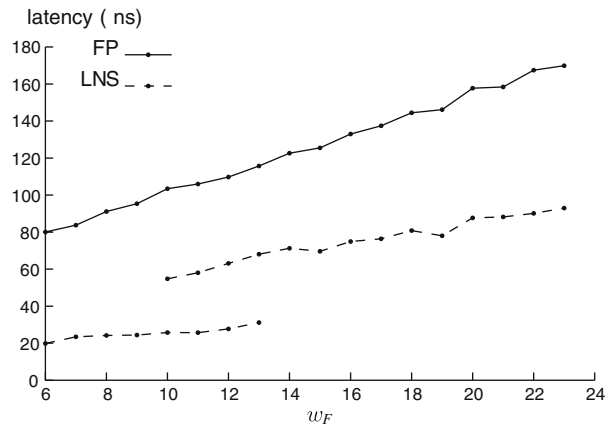
(b) Latency

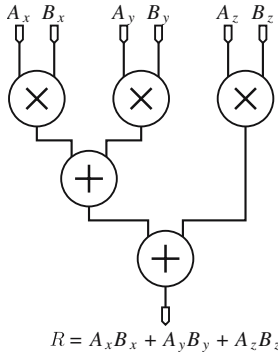Figure 8. Benchmarks for the norm operator $R = \sqrt{A^2 + B^2}$.

$$R = A_x B_x + A_y B_y + A_z B_z$$

Figure 9. Architecture of the dimension 3 dot product operator.

3.2. Comparing Operators in Context

The estimates presented in the previous section give a rough idea, but the goal of our library is to allow accurate estimations for each particular circuit. The simplicity of such a comparison is shown for a toy example, the computation of a norm $\sqrt{A^2 + B^2}$. The architecture of this operator is shown in Fig. 7b. The two squarings are performed in parallel by multipliers, as our libraries do not yet provide dedicated squaring operators, followed by addition and square root.

The corresponding VHDL code is given in Fig. 7a. As it is written here, it handles floating-point data, with a dynamic of $w_E = 6$ bits, and a precision of $w_F = 13$ bits. Those three parameters are represented in the code by the constants "fmt", "wE" and "wF" respectively (defined lines 13, 14 et 15), which are passed to each operator and define the bit width of the signals.
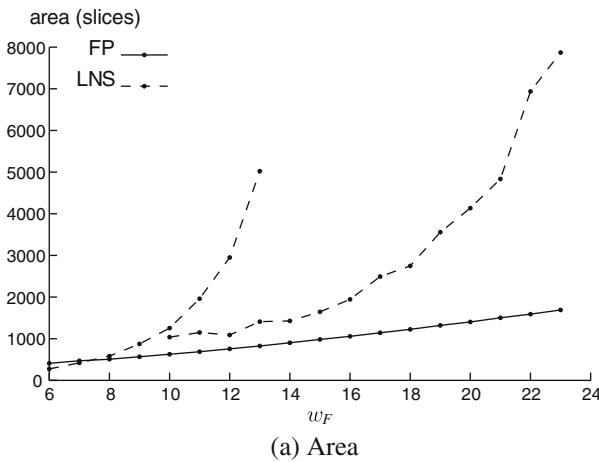
Therefore, to change the number representation format, the user just has to change the value of "fmt" from "FP" (for floating-point) to "LNS" (for logarithmic representation). The same principle applies for w_E and w_F, that can be modified by changing the value of "wE" or "wF", and of course adjusting the value of the width of the component ports (lines 7, 8 et 9).

For pipelined operators, the method is sensibly more complex, as the pipeline depth of the operators varies with the number representation and the precision. Scheduling the operations depends on these parameters. A reasonable approach is therefore to study the various parameter choices on a combinatorial circuit (while reserving area for the pipeline overhead), and then benchmark the pipelined version only for the most interesting parameter sets.

3.3. Comparison Examples

3.3.1. Norm $\sqrt{A^2 + B^2}$. Figure 8 compares the area and latency of the norm operator for floating-point and LNS. If the general aspect of the plots roughly matches the estimations from Table 1, area and latency values are lower than expected, especially for the LNS operator. This is because the VHDL synthesizer realizes that both A^2 and B^2 are nonnegative, and thus the subtraction part of the adder/subtracter operator is useless. This simplification is quite important in the case of LNS, as the subtraction tables contribute to a large part of the area of the operator. This effect, unsuspected when looking at the operators in isolation, illustrates the usefulness of a comparison in context.
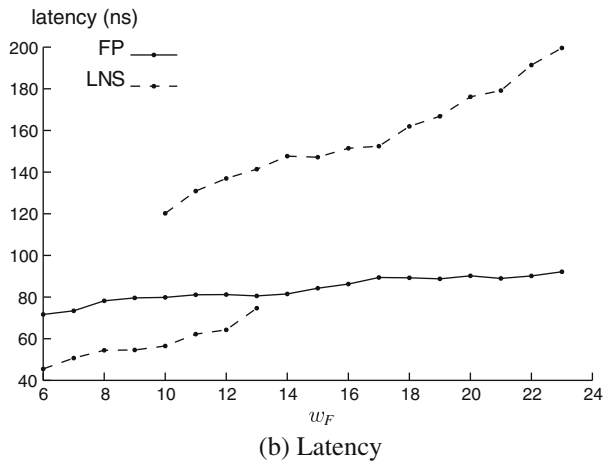

(a) Area

(b) Latency

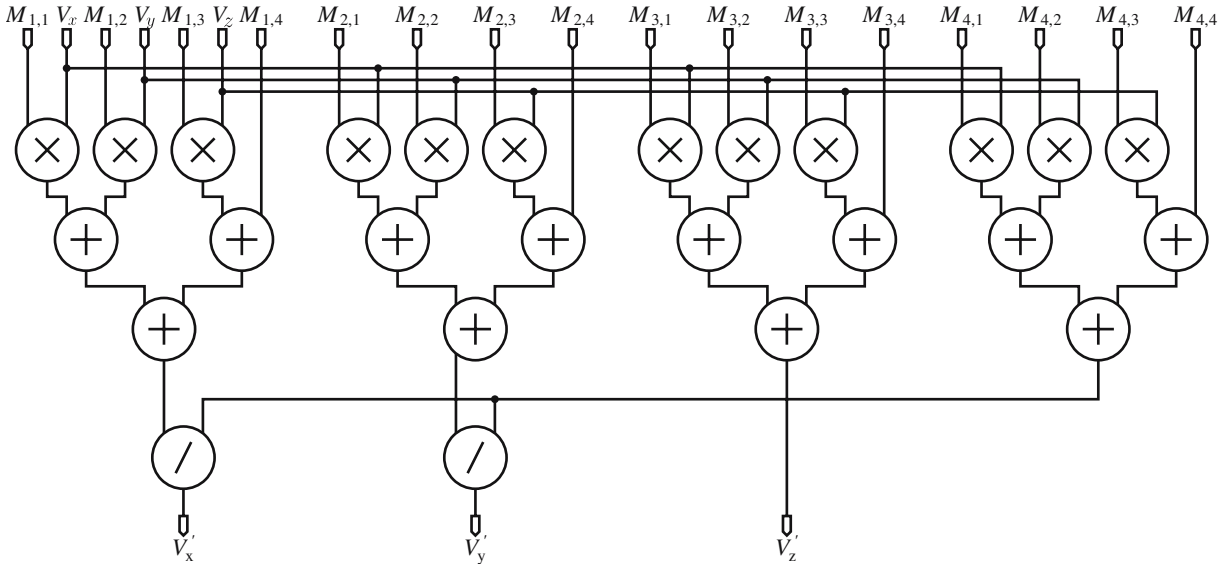*Figure 10.* Benchmarks for the dimension 3 dot product operator.

Figure 11. Architecture of the 3D transformation operator.

In this example, the designer will conclude that LNS is interesting for precisions up to $w_F = 16$ bits, but floating-point has to be used for higher precisions.

3.3.2. Dot Product.

As previously, a three-dimensional dot product operator is quite easy to implement using our operators. The naive architecture of this operator is shown in Fig. 9.

The area and delay estimations are shown in Fig. 10, where LNS operators appear to be too large very soon, for precision from $w_F = 9$ or 10 bits. Indeed, as the number of adders increases in this case, so does the area of LNS operators. However, these operators are still faster than the floating-point ones.

3.3.3. 3D Transformation Pipeline.

As last example, we choose to study a full scale application. Current 3D engines generate an image from a scene described as a list of vertices, a list of triangles and the position of the camera. The transformation stage transforms the vertices from the scene coordinates to the camera's viewing frustrum coordinates, including perspective computations. From an algorithmic point of view, this stage can be trivially parallelized, and only requires a dimension 4 matrix–vector product and two divisions as shown in Fig. 11. This stage is sensitive, as it determines the on-screen position of the triangles, and therefore requires some precision not to distort the objects.

The circuit has been fully implemented and tested on a Xilinx Virtex-E XCV2000E-6 based Celoxica RC1000-PP board. The complete application is also freely available, along with the library. Some screenshots are given in Figs. 12 and 13. Even at low precisions, as can be seen from Fig. 12, the general aspect of the objects remains correct, but as one zooms into the picture, more precision is required, as illustrated by Figs. 13 and 14.

Back to the comparison between FP and LNS, these screenshots give a new information: For the same precision, FP gives images which provide

Figure 12. 3D image, using method LNS-O1 and precision $(w_E, w_F) = (5, 8)$.
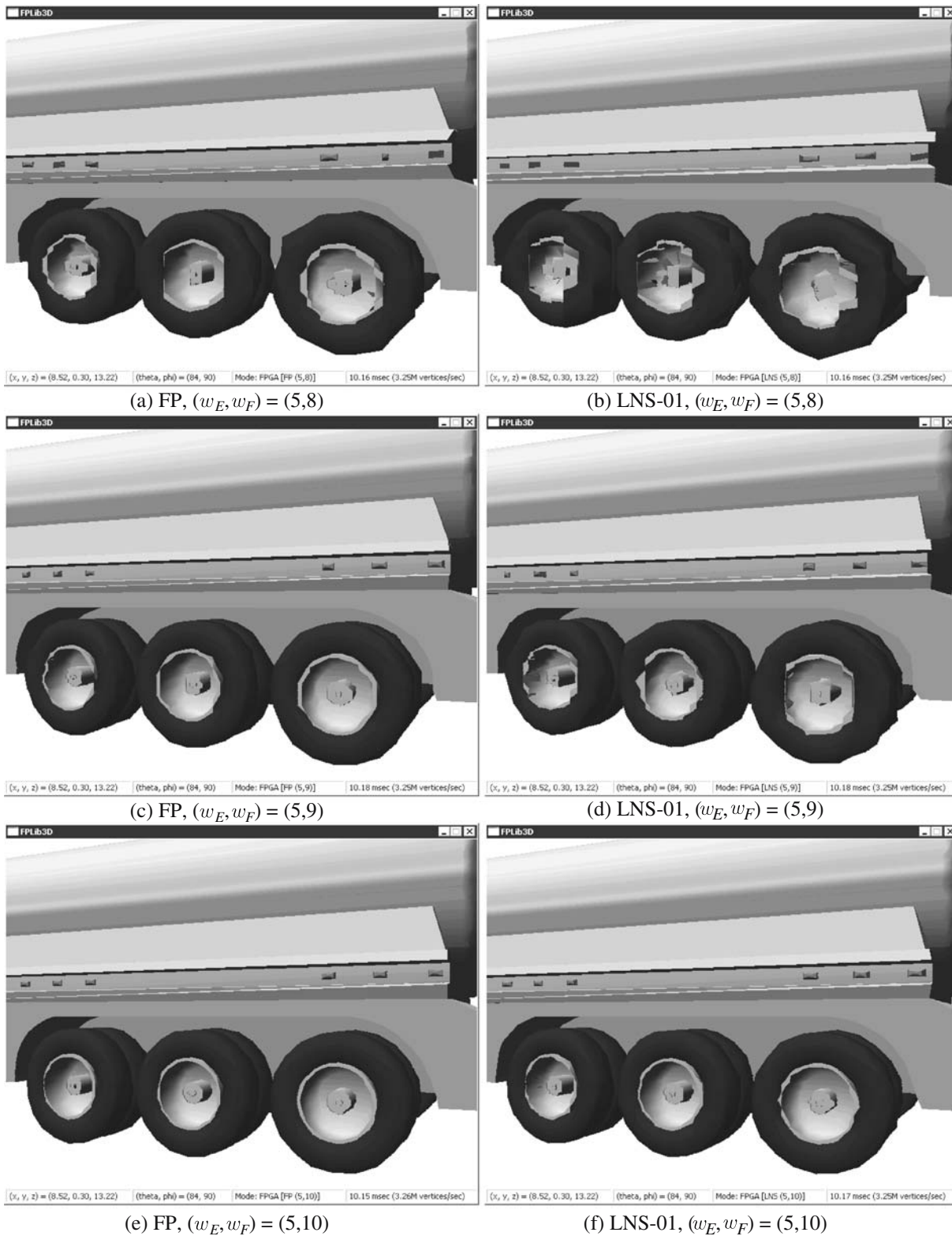
(a) FP, $(w_E, w_F) = (5,8)$

(b) LNS-01, $(w_E, w_F) = (5,8)$

(c) FP, $(w_E, w_F) = (5,9)$

(d) LNS-01, $(w_E, w_F) = (5,9)$

(e) FP, $(w_E, w_F) = (5,10)$

(f) LNS-01, $(w_E, w_F) = (5,10)$

Figure 13. 3D images for various precisions.

(a) FP, $(w_E, w_F) = (6,14)$
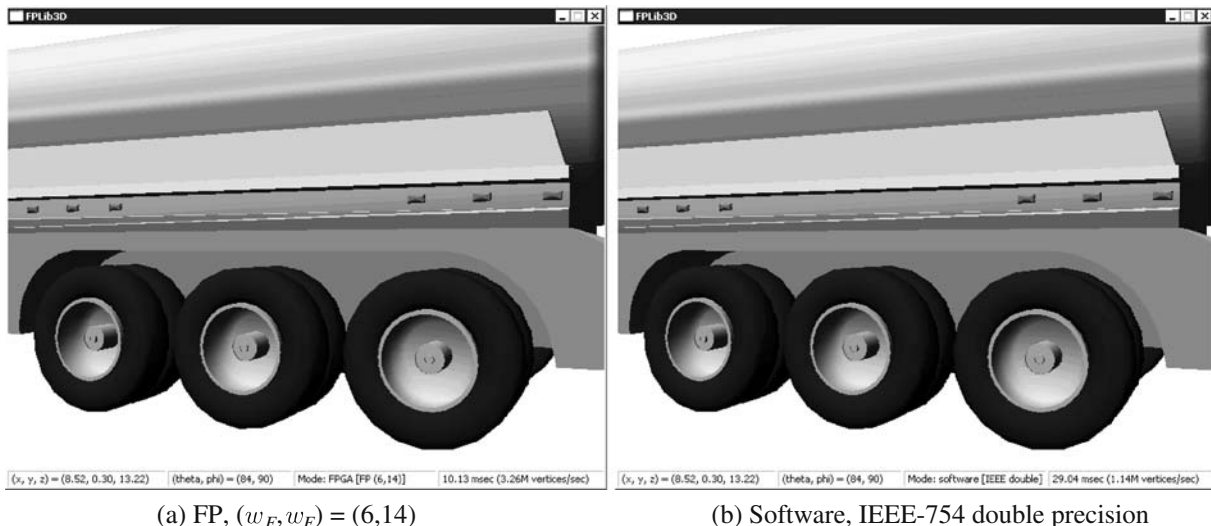

(b) Software, IEEE-754 double precision

Figure 14.   3D images for reference precisions.

slightly better visual quality than LNS. The rule of thumb here is that $FP(5, w_F)$ provides a visual quality better than $LNS(5, w_F)$ but worse than $LNS(5, w_F + 1)$. For this specific application, cost/performance should be compared accordingly. There was no easy way to get an intuition of this beforehand, and it shouldn't be generalised: For some applications, LNS will provide better overall accuracy or signal-to-noise ratio than FP. As LNS multiplications and divisions are without errors, this will indeed probably happen to applications for which LNS is also more efficient.

Table 2 gives a few results obtained for various precisions. Due to on-board memory bandwith limitations, along with the PCI bus limitations, the circuit can only process one vertex each 50 MHz cycle, and therefore parallelizing the design by replicating the operator is useless, although it could have been interesting as some designs occupy less than a quarter of the available FPGA area.

This example is still a toy example, as there is no hope that an FPGA will match the cost/performance ratio of current graphics cards. It illustrates how the library can be used to evaluate in situation the performance and accuracy of a whole application.

4.   Conclusion and Future Work

We hope to show with this work that, in order to discuss the compared pros and cons of floating-point

and logarithmic number systems, it is much more profitable to publicly release a library of finely crafted operators instead of publishing application-specific comparisons. Moreover, a non neglectable side-effect to this work is the existence of this library, which we will carry on extending and developing.

Improving the floating-point operators is probably difficult, considering the convergence between our

Table 2.   Area (slices and FPGA area percentage), latency (pipeline depth) and throughput benchmarks of the 3D transformation pipeline for various parameter combinations.

(w_E, w_F)	Area	Pipeline depth	Throughput (vertices/s)
FP, $(5, 8)$	4,446 slices (23%)	25	3.3 M
LNS-O1, $(5, 8)$	5,497 slices (28%)	14	3.3 M
FP, $(5, 9)$	4,802 slices (25%)	25	3.3 M
LNS-O1, $(5, 9)$	7,415 slices (38%)	14	3.3 M
FP, $(5, 10)$	5,246 slices (27%)	26	3.3 M
LNS-O1, $(5, 10)$	9,701 slices (50%)	14	3.3 M
FP, $(6, 14)$	7,408 slices (38%)	28	3.3 M
software, IEEE-754 single precision	–	–	1.2 M

library and that of Lee [19], developed independently. Our current focus is on developing parameterized elementary functions (exp, log, trigonometric) for this library.

The LNS addition, however, can be improved further by proposing various implementation methods in order to allow a designer to choose between several solutions, depending on his precision, area and latency requirements. For instance, future versions of FPLibrary will include LNS operators using the co-transformation approach by Arnold et al. [2], which should be smaller and more accurate but slower. Our current first-order and second-order approaches to LNS addition will not allow to go much further than single precision (we share this concern with Haselman et al, whose double-precision LNS adder [18] does not even fit in a Virtex-II 2000 FPGA, one of the largest available today). Reaching double-precision will require either a higher-order method, or an improvement in the range reduction.

Another direction for future work is to offer the same comparison tools when targeting application-specific integrated circuits (ASICs): If the FPGA is used for rapid prototyping, the cost/performance tradeoffs obtained on FPGA are probably of little significance to an ASIC implementation of the same application, because the metrics are very different. For example, the tables used in the LNS addition will be implemented as ROMs in an ASIC. On one hand, their area (relative to the area of the adders) will be much smaller than an LUT-based implementation on FGPAs. On the other hand, the synthesis tools are able to optimize the LUT-based implementation, but not the block ROM. The combined effect of this is difficult to predict, all the more as it depends on the table values. Another example is the synthesis of the adder trees: Where the fast carry propagation logic of modern FPGAs means that the simplest adder tree is optimal, we should use in an ASIC a carry-save adder tree followed by some sort of fast adder.

These differences have influenced the architectural choices made in our library: Although our portable VHDL can be compiled for ASIC, the result would be far from optimal in this case. However, our main claim is that only concurrent, high-quality operator libraries will allow an enlightened and unbiased choice between LNS and FP on a per-application basis. This claim also holds when targeting ASICs.

References

1. ANSI/IEEE, *Standard 754-1985 for Binary Floating-point Arithmetic (also IEC 60559)*, 1985.
2. M. Arnold, T. Bailey, J. Cowles, and M. Winkel, "Arithmetic Co-transformations in the Real and Complex Logarithmic Number Systems," *IEEE Trans. Comput.*, vol. 47, no. 7, (July) 1998, pp. 777–786.
3. P. Belanović and M. Leeser, "A Library of Parameterized Floating-point Modules and Their Use," in *Field Programmable Logic and Applications (LNCS, vol. 2438)*, Springer, (Sept.) 2002, pp. 657–666.
4. J.-L. Beuchat and A. Tisserand, "Small Multiplier-based Multiplication and Division Operators for Virtex-II Devices," in *Field-programmable Logic and Applications (LNCS, vol. 2438)*, Springer, (Sept.) 2002.
5. J. N. Coleman and E. I. Chester, "Arithmetic on the European Logarithmic Microprocessor," *IEEE Trans. Comput.*, vol. 49, no. 7, (July) 2000, pp. 702–715.
6. F. de Dinechin and A. Tisserand, "Some Improvements on Multipartite Table Methods," in *15th IEEE Symposium on Computer Arithmetic*, N. Burgess and L. Ciminiera (Eds.), June 2001, pp. 128–135. (Updated version of LIP research report 2000-38.)
7. M. deLorimier and A. DeHon, "Floating-point Sparse Matrix–vector Multiply for FPGAs," in *ACM/SIGDA Field-programmable Gate Arrays*, ACM Press, 2005, pp. 75–85.
8. J. Detrey and F. de Dinechin, "A VHDL Library of LNS Operators," in *37th Asilomar Conference on Signals, Systems and Computers*, Oct. 2003.
9. J. Detrey and F. de Dinechin, "Second Order Function Approximation Using a Single Multiplication on FPGAs," in *14th Intl Conference on Field-Programmable Logic and Applications (LNCS, vol. 3203)*, Springer, Aug. 2004, pp. 221–230.
10. J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, and D. Poirier, "A Flexible Floating-point Format for Optimizing Datapaths and Operators in FPGA Based DSPs," in *ACM/SIGDA Field-programmable Gate Arrays*, (Feb.) 2002, pp. 50–55.
11. C. Doss and R. Riley, "FPGA-based Implementation of a Robust IEEE-754 Exponential Unit," in *FPGAs for Custom Computing Machines*, IEEE, 2004.
12. Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit Floating-point FPGA Matrix Multiplication," in *ACM/SIGDA Field-programmable Gate Arrays*, ACM Press, 2005.

13. M. D. Ercegovac and T. Lang, *Division and Square Root: Digit-recurrence Algorithms and Implementations*, Kluwer, Boston, MA, 1994.

14. M. D. Ercegovac and T. Lang, *Digital Arithmetic*, Morgan Kaufmann, San Mateo, CA, 2003.

15. M. J. Flynn and S. F. Oberman, *Advanced Computer Arithmetic Design*, Wiley-Interscience, New York, 2001.

16. A. A. Gaffar, W. Luk, P. Y. K. Cheung, N. Shirazi, and J. Hwang, "Automating Customisation of Floating-point Designs," in *Field Programmable Logic and Applications(LNCS, vol. 2438)*, Springer, (Sept.) 2002, pp. 523–533.

17. D. Goldberg, "What Every Computer Scientist Should Know about Floating-point Arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, (Mar.) 1991, pp. 5–47.

18. M. Haselman, M. Beauchamp, K. Underwood, and K. S. Hemmert, "A Comparison of Floating-point and Logarithmic Number Systems for FPGAs," in *FPGAs for Custom Computing Machines*, 2005.

19. B. Lee and N. Burgess, "Parameterisable Floating-point Operators on FPGAs," in *36th Asilomar Conference on Signals, Systems, and Computers*, 2002, pp. 1064–1068.

20. B. Lee and N. Burgess, "A Dual-path Logarithmic Number System Addition/Subtraction Scheme for FPGA," in *Field-programmable Logic and Applications*, (Sept.) 2003, Lisbon.

21. D. M. Lewis, "An Architecture for Addition and Subtraction of Long Word Length Numbers in the Logarithmic Number System," *IEEE Trans. Comput.*, vol. 39, no. 11, (Nov.) 1990.

22. G. Lienhart, A. Kugel, and R. Männer, "Using Floating-point Arithmetic on FPGAs to Accelerate Scientific N-body Simulations," in *FPGAs for Custom Computing Machines*, IEEE, 2002.

23. R. Matoušek, M. Tichý, Z. Pohl, J. Kadlec, C. Softley, and N. Coleman, "Logarithmic Number System and Floating-point Arithmetics on FPGA," in *Field-programmable Logic and Applications*, Montpellier, (Sept.) 2002, pp. 627–636.

24. F. Ortiz, J. Humphrey, J. Durbano, and D. Prather, "A Study on the Design of Floating-point Functions in FPGAs," in *Field Programmable Logic and Applications (LNCS, vol. 2778)*, Springer, (Sept.) 2003, pp. 1131–1135.

25. V. Paliouras and T. Stouraitis, "A Novel Algorithm for Accurate Logarithmic Number System Subtraction," in *International Symposium on Circuits and Systems*, volume 4, IEEE, (May) 1996, pp. 268–271.

26. E. Roesler and B. Nelson, "Novel Optimizations for Hardware Floating-point Units in a Modern FPGA Architecture," in *Field Programmable Logic and Applications (LNCS, vol. 2438)* Springer, (Sept.) 2002, pp. 637–646.

27. F. J. Taylor, R. Gill, J. Joseph, and J. Radke, "A 20 Bit Logarithmic Number System Processor," *IEEE Trans. Comput.*, vol. 37, no. 2, (Feb.) 1988

Jérémie Detrey received his DEA from the *École Normale Supérieure de Lyon* (ENS-Lyon) in 2003, and is now completing a Ph.D. in the *Laboratoire de l'Informatique du Parallélisme* (LIP) at ENS-Lyon, under the direction of Florent de Dinechin and Jean-Michel Muller. His research is focused on designing operators for real arithmetic on FPGAs.

Florent de Dinechin received his DEA from the *École Normale Supérieure de Lyon* (ENS-Lyon) in 1993, and his Ph.D. from Université de Rennes 1 in 1997. After a postdoctoral position at Imperial College, London, he is now a permanent lecturer at ENS-Lyon in the *Laboratoire de l'Informatique du Parallélisme* (LIP). His research interests include computer arithmetic, software and hardware evaluation of functions, computer architecture and FPGAs.