Floating point or LNS: Choosing the right arithmetic on an application basis

Sylvain Collange Jérémie Detrey Florent de Dinechin
Laboratoire de l'Informatique du Parallélisme
École Normale Supérieure de Lyon
46, allée d'Italie
F-69364 Lyon cedex 07

{Sylvain.Collange, Jeremie.Detrey, Florent.de.Dinechin }@ens-lyon.fr

Abstract

For applications requiring a large dynamic range, real numbers may be represented either in floating-point (FP), 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. In doubt, designers will choose floating-point. This article demonstrates a methodology for a better informed choice thanks to FPLibrary, a freely available, dual FP/LNS arithmetic operator library. FPLibrary may be used in the prototype phase of an application to obtain, with low design effort, accurate measures of performance, cost and accuracy of both LNS and FP approaches. Two case studies demonstrate the benefits of this methodology.

1 Introduction

When an application requires a large dynamic range in the numbers processed, floating-point (FP) arithmetic is a well-known solution, but it is not the only one. An alternative is the *Logarithmic Number System* (LNS), where a positive real number is represented by its logarithm (usually in radix 2), and the hardware operators compute on these logarithms. The main interest of this coding is that multiplications, divisions and square roots are trivial with logarithms: They are implemented respectively by fixed-point addition and subtraction (as on a slide rule), and bit shift. Besides, contrary to their FP counterpart, LNS multiplication and division entail no rounding error. Conversely, LNS addition and subtraction are much more complicated than their FP equivalent. Overall, several publications have shown applications (ASIC or FPGA) for which LNS is more efficient in terms of speed and area than floating-point [3, 15, 17].

However, many designers are unaware of LNS, or unwilling to try it. The main reason is probably that floating-point is well-known and well standardised, being implemented in virtually all contemporary processors. This also allows an efficient and reliable numerical simulation of an application before it is cast to hardware. Another reason is simply the lack of exposure of LNS in general, be it literature, textbooks, supporting design tools and libraries. Finally, and perhaps more importantly, for those designers aware of the existence of LNS, there is the knowledge that LNS only rarely outperforms floating-point, and that in these rare cases where LNS stands a chance of having some benefit, the cost of the design efforts required to confirm this benefit outweigh the possible benefit itself. For reasons exposed in the next section, it is difficult for a designer to get a fiable intuition of the relevance of LNS for a given application.

The purpose of this article is to help designers make a more informed choice between floating-point and LNS, at an acceptable cost. We describe a generic comparison tool which a designer may target to his application, with its budget and constraints. This tool is a dual library – implementing both LNS and floating-point – of arithmetic operators. The purpose is to help a designer chose the best arithmetic for a given application by simply synthesising both alternatives. As each operator is parametrised in terms of range (exponent size) and precision (mantissa size), the accuracy/performance/cost trade-offs can be explored relatively easily, and independently for each arithmetic. The library tries to avoid biased comparison by reflecting the state of the art of operator design in both arithmetic systems.

The next section presents in more details the issues involved in the accuracy/cost/performance trade-offs in LNS and in FP. Section 3 then briefly presents the comparison tool, and section 4 presents case studies of comparisons.

2 Accuracy, cost and performance trade-offs in FP and in LNS

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 and $\sqrt{x}$) with respect to difficult ones ($+$ and $-$), and the required precision has to be quite low (less than 16 bits), as the cost of these difficult operations in LNS grows exponentially with precision, as the sequel will show. Quantitatively, it is much more difficult to have a precise answer. The best application-based comparative study was that of Coleman *et al.* [3]: 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.* [15], but this case study concerns an iteration with 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.

It is difficult for a designer to get an accurate intuition of the problem beforehand. A first problem is that the costs of the difficult LNS operators with respect to precision (or number size) are highly non-linear. These costs also depend on the target technology, and a variety of algorithms expose wide area/speed trade-offs – probably wider than those in floating-point operators. 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. In practise, one of the number systems may provide a better overall accuracy for a given number size, as illustrated in Section 4.2.

The conclusion is that it is probably impossible in a publication to exhaustively cover the set of parameters controlling the speed/area/precision trade-offs for both LNS and FP so that a designer can make an informed choice. As an example, a comparison by Haselman et al [11] only covers the standard IEEE-754 single and double precision, although the authors claim they implemented a parametrised library. Due to the high non-linearity of the cost, it will not help if the application can accommodate lower or intermediate precisions, as is commonly the case for signal processing.

Therefore, our approach is not to publish comparisons, but a generic comparison tool which is presented in the remainder of this paper. The library itself is briefly presented in next section, and cases studies of application-specific comparisons enabled by FPLibrary are presented in section 4.

3 FPLibrary

FPLibrary is a library of arithmetic 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 range 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 flavour. It is written in portable VHDL, and all the operators have been designed with equivalent optimisation effort.

3.1 Number representation

The representation of a real number in the library is parametrised by two integers, w_E which determines the dynamic range of the represented numbers, and w_F their precision. The sets 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 the formats. More accurately, relative coding errors between those two formats are within a $\log(2)$ ratio, as can be deduced from the following Equations (1) and (2).

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 normalised in $[1; 2[$, so its most significant bit is always 1 and is implicit in the coding:

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

Our format diverges from the IEEE-754 standard in that it does not support subnormal numbers [10], and uses two additional bits to code other exceptional cases ($\pm\infty$, Not a Number, Zero), where IEEE-754 reserves special exponents. As exception handling is identical between FP and LNS, we shall not detail it any further.

The LNS format on $3+w_E+w_F$ bits is composed of the two exception bits, 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}:

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

3.2 Operator architectures

The library operators are described in detail in the FPLibrary documentation (`http://www.ens-lyon.fr/LIP/Arenaire/`) as well as in a companion research report [5]. This section briefly exposes the design philosophy for these operators.

The library has two main goals which often conflict: be *portable*, and reflect the *state of the art* for both floating-point and LNS operators, to allow an unbiased comparison.

Concerning portability, the library is written in portable VHDL without use of any vendor-specific library. For instance, fixed-point multipliers are expressed as `*` in the code, and we rely on the synthesis tools to use whatever best implementation of such multipliers is available. This also ensures flexibility, as synthesis tools may provide several multiplier variants optimised for various area/speed trade-off. For example, on Xilinx Virtex-II, the synthesis tools can use the embedded small multipliers specific to this FPGA. Of course, using such a generic multiplier may be slightly sub-optimal [2].

Another example is tables of values used in the LNS operators, whose optimal implementation is very technology-dependent. Again, such tables are expressed in the most portable way.

There is another, deeper sub-optimality issue: The architectures themselves may involve many parameters. This is the case for the floating-point divide and square root which use high-radix SRT algorithms [9, 8], and for LNS operators [4] which use sophisticated table-based methods for function evaluation [6]. In such cases, these parameters have been optimised for Virtex FPGAs, and the architecture may be sub-optimal for other FPGAs, or when targeting ASICs. The solution here is to write more flexible operator generators, which is one of our future research directions.

Concerning more specifically the architectures of our floating-point operators, the subject is very mature with good recent books [9, 8]. The reader is referred to [5] for detailed architecture discussion and description.

Concerning LNS, multiplication, division and square root are implemented respectively by addition, subtraction and right shift of the logarithms of the operands. Addition and subtraction are performed as follows (X and Y are both positive numbers such that $X > Y$) :

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

The LNS addition and subtraction therefore resumes to the evaluation of the two non-linear functions $f_\oplus$ and $f_\ominus$. The literature discusses many algorithms for that [18, 14, 16, 3, 13, 4, 11] with a range of area/speed trade-offs. FPLibrary currently offers two flavours for the LNS adder. The first (noted *method 1* on Figure 1) uses a simple argument reduction, and is fast but bulky and limited in practise to $w_F \leq 13$ bits. The second (noted *method 2* on Figure 1) uses a range reduction from [16], which is much smaller but also slower. Both implementations use advanced table-based techniques [6] for the evaluation of $f_\oplus$ and $f_\ominus$. These two implementations reflect the state of the art, at least as far as precisions lower than single-precision are concerned.

4 Case studies

In this section, all the estimations are given by the Xilinx ISE 5.2 tool suite for a Virtex-II XC2V2000-4 FPGA.

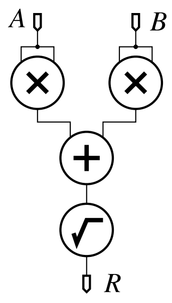
4.1 Norm $\sqrt{A^2 + B^2}$

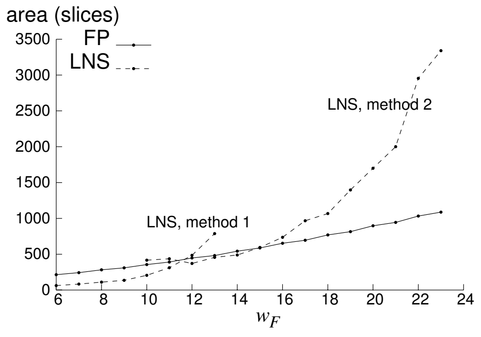
Figure 2 shows the VHDL implementation of a 2D norm using our library. As written here, it handles floating-point data, with a dynamic range of $w_E = 6$ bits, and a precision of $w_F = 13$ bits. Those three parameters are represented in the code by the constants “fmt”, “wE” and “wF” respectively (defined lines 13, 14 and 15).

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.

Figure 1 compares the area and latency of the norm operator for floating-point and LNS. Only the precision w_F





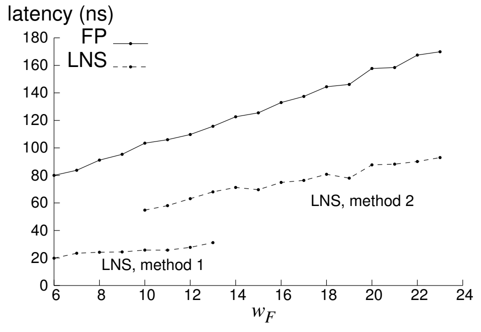


Figure 1. Area and speed for the norm operator $R = \sqrt{A^2 + B^2}$ implemented on a VirtexII.

```
library ieee;
use ieee.std_logic_1164.all;
library fplib;
use fplib.pkg_fplib.all;

entity Norm is
  port ( A : in  std_logic_vector(6+13+2 downto 0);
         B : in  std_logic_vector(6+13+2 downto 0);
         R : out std_logic_vector(6+13+2 downto 0) );
end entity;

architecture arch of Norm is
  constant fmt : format   := FP;
  constant wE  : positive := 6;
  constant wF  : positive := 13;

  signal A2 : std_logic_vector(wE+wF+2 downto 0);
  signal B2 : std_logic_vector(wE+wF+2 downto 0);
  signal R2 : std_logic_vector(wE+wF+2 downto 0);
begin
  mul_a_a : Mul
    generic map ( fmt, wE, wF )
    port map ( A, A, A2 );

  mul_b_b : Mul
    generic map ( fmt, wE, wF )
    port map ( B, B, B2 );

  add_a2_b2 : Add
    generic map ( fmt, wE, wF )
    port map ( A2, B2, R2 );

  sqrt_r2 : Sqrt
    generic map ( fmt, wE, wF )
    port map ( R2, R );
end architecture;
```

Figure 2. VHDL code for a 2D norm ($R = \sqrt{A^2 + B^2}$).

varies, as the dependence on w_E (the dynamic range) is very similar in both arithmetic systems, and is more or less linear both for area and delay.

This example is interesting because area and latency values are lower than expected, especially for the LNS operator. This is because the VHDL synthesiser realises that both A^2 and B^2 are non-negative, 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.

In this example, the designer will conclude that LNS is very interesting for precisions up to $w_F = 15$ bits (smaller and faster than floating point). LNS remains faster up to single precision (23 bits), but with increasing overhead in term of area.

4.2 3D transformation pipeline

This other example demonstrates the use of FPLibrary to obtain information on the overall accuracy of an 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 frustum coordinates, including perspective computations. From an algorithmic point of view, this stage can be trivially parallelised, and only requires a dimension 4 matrix-vector product and two divisions as shown in Figure 3. 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 based Celoxica RC1000-PP board. The complete application is also freely available, along with the library. Some screenshots are given in Figures 4 and Figure 5.

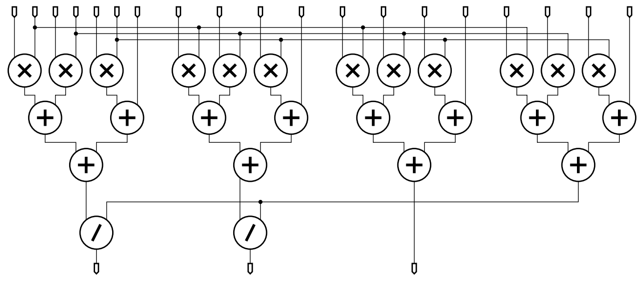
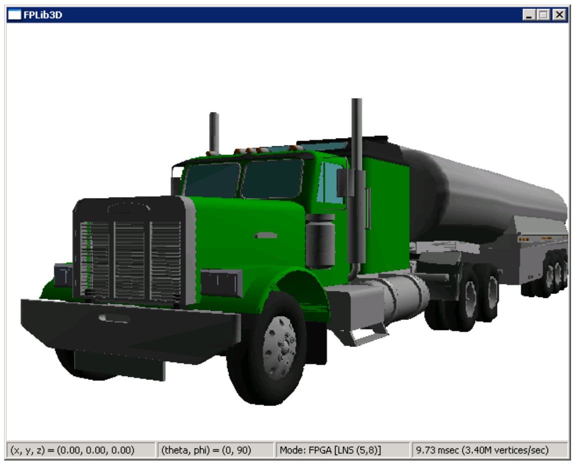


Figure 3. Architecture of the 3D transformation operator.

Figure 4 shows that low precision ($w_F = 8$) is enough for a good visual feedback. The distortions due to low-precision arithmetic become visible as one zooms into the picture – how much one is allowed to zoom is a parameter of the application. Figure 5 shows a given zoom with a range of arithmetic parameters, each with its area and delay. The delays are expressed in cycles as the pipelined operators are used (their target frequency is 100MHz on Virtex-II). Conversions from and to LNS are performed on the host PC.

As far as the FP vs LNS comparison is concerned, these screenshots give a new information: For the same precision, FP gives images which provide slightly better visual quality than LNS. The rule of thumb observed 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 competitive.

We also observe that LNS is always almost twice as fast, but becomes almost twice as large if one wants acceptable visual quality at the zoom levels of Figure 5. Again, 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 serves is purpose to illustrate how the library can be used to evaluate in situation the performance and accuracy of a whole application.



(x, y, z) = (0.00, 0.00, 0.00) | (theta, phi) = (0, 90) | Mode: FPGA [LNS (5,8)] | 9.73 msec (3.40M vertices/sec)

Figure 4. Low-precision LNS image $(w_E = 5, w_F = 8)$

5 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 release a library of finely crafted operators than to publish application-specific comparisons. Of course, a non neglectable side-effect to this work is the existence of this library, which we will carry on extending and developing.

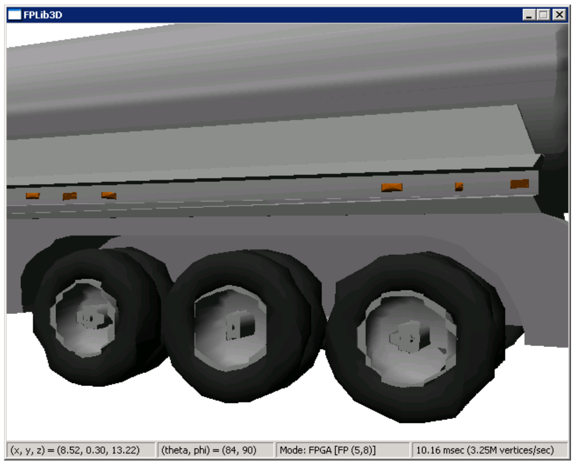
Improving the floating-point operators is probably difficult, considering for instance the convergence between our library and that of Lee [12], developed independently. Our current focus is on developing parametrised elementary functions (exp, log, trigonometric) for this library [7].

The LNS addition, however, can be improved further, by proposing more implementation methods. This will give more freedom to the designer in the performance/cost aspects. We also need to automate the (currently mostly manual) optimisation process which defines the parameters of an architecture. If we want this process to target efficiently ASICs and non-Xilinx FPGAs, we have to define cost functions for the various building blocks (adders, multipliers, ROM tables, ...)

Our current 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 [11] does not even fit in a Virtex-II 2000 FPGA. Reaching double-precision will require either a higher-order method, or an improvement in the range reduction – all this probably at the expense of the delay. Therefore, before researching this direction, we have to convince ourselves that there exists applications for which LNS will be competitive with floating-point at such large precisions.

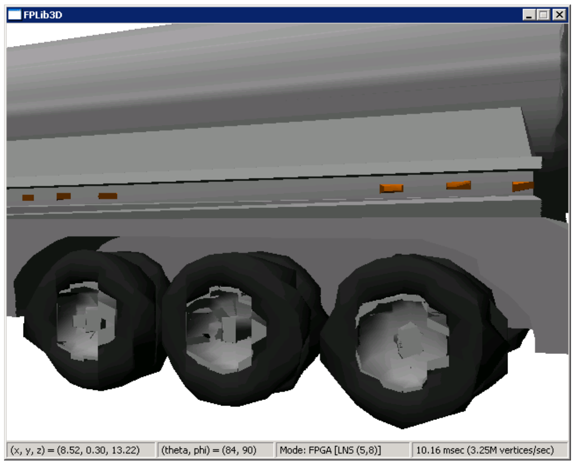
Acknowledgements

The authors would like to thank Arnaud Tisserand for many interesting discussions on this topic, and also for administrating the CAD tool server on which all the syntheses presented in this paper were performed.



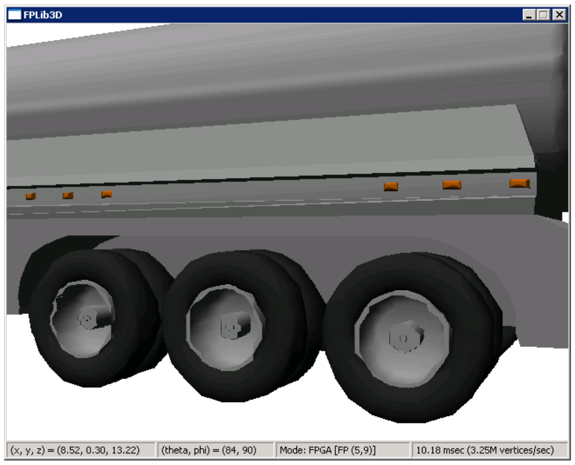
(x, y, z) = (8.52, 0.30, 13.22) | (theta, phi) = (84, 90) | Mode: FPGA [FP (5,8)] | 10.16 msec (3.25M vertices/sec)

FP(5,8), 4446 slices (23%), 25 cycles



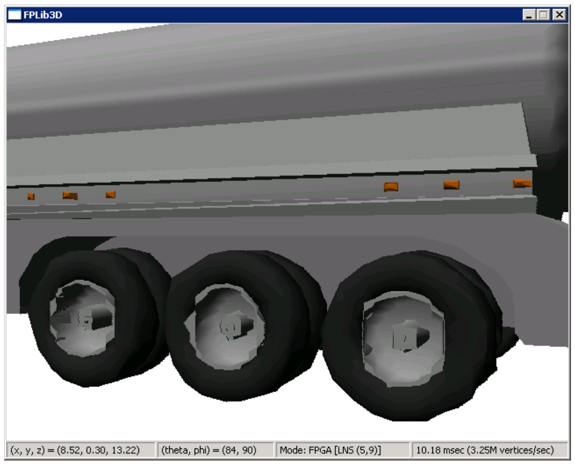
(x, y, z) = (8.52, 0.30, 13.22) | (theta, phi) = (84, 90) | Mode: FPGA [LNS (5,8)] | 10.16 msec (3.25M vertices/sec)

LNS(5,8), 5497 slices (28%), 14 cycles



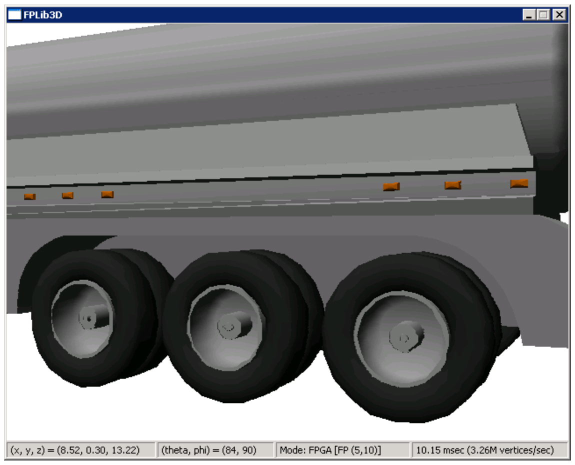
(x, y, z) = (8.52, 0.30, 13.22) | (theta, phi) = (84, 90) | Mode: FPGA [FP (5,9)] | 10.18 msec (3.25M vertices/sec)

FP(5,9), 4802 slices (25%), 25 cycles



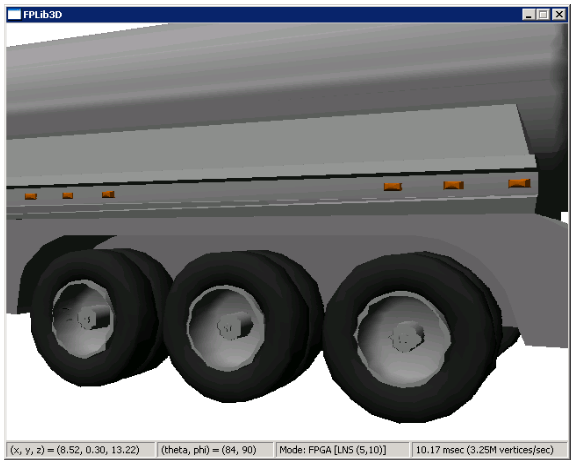
(x, y, z) = (8.52, 0.30, 13.22) | (theta, phi) = (84, 90) | Mode: FPGA [LNS (5,9)] | 10.18 msec (3.25M vertices/sec)

LNS(5,9), 7415 slices (38%), 14 cycles



(x, y, z) = (8.52, 0.30, 13.22) | (theta, phi) = (84, 90) | Mode: FPGA [FP (5,10)] | 10.15 msec (3.26M vertices/sec)

FP(5,10), 5246 slices (27%), 26 cycles



(x, y, z) = (8.52, 0.30, 13.22) | (theta, phi) = (84, 90) | Mode: FPGA [LNS (5,10)] | 10.17 msec (3.25M vertices/sec)

LNS(5,10), 9701 slices (50%), 14cycles

Figure 5. Accuracy/cost/performance trade-off in a graphic pipeline

References

[1] ANSI/IEEE. *Standard 754-1985 for Binary Floating-Point Arithmetic (also IEC 60559)*. 1985.

[2] J.-L. Beuchat and A. Tisserand. Small multiplier-based multiplication and division operators for Virtex-II devices. In *Field-Programmable Logic and Applications*, volume 2438 of *LNCS*. Springer, Sept. 2002.

[3] J. N. Coleman and E. I. Chester. Arithmetic on the European logarithmic microprocessor. *IEEE Transactions on Computers*, 49(7):702–715, July 2000.

[4] J. Detrey and F. de Dinechin. A VHDL library of LNS operators. In *37th Asilomar Conference on Signals, Systems and Computers*, Oct. 2003.

[5] J. Detrey and F. de Dinechin. A tool for unbiased comparison between logarithmic and floating-point arithmetic. Technical Report RR2004-31, LIP, École Normale Supérieure de Lyon, Mar. 2004.

[6] J. Detrey and F. de Dinechin. Table-based polynomials for fast hardware function evaluation. In *16th Intl Conference on Application-specific Systems, Architectures and Processors*. IEEE Computer Society Press, July 2005.

[7] J. Detrey and F. de Dinechin. Parameterized floating-point logarithm and exponential functions for FPGAs. *Journal of Microprocessors and Microsystems*, 2006. To appear.

[8] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.

[9] M. J. Flynn and S. F. Oberman. *Advanced Computer Arithmetic Design*. Wiley-Interscience, 2001.

[10] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, Mar. 1991.

[11] 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.

[12] B. Lee and N. Burgess. Parameterisable floating-point operators on FPGAs. In *36th Asilomar Conference on Signals, Systems, and Computers*, pages 1064–1068, 2002.

[13] B. Lee and N. Burgess. A dual-path logarithmic number system addition/subtraction scheme for FPGA. In *Field-Programmable Logic and Applications*, Lisbon, Sept. 2003.

[14] D. M. Lewis. An architecture for addition and subtraction of long word length numbers in the logarithmic number system. *IEEE Transactions on Computers*, 39(11), Nov. 1990.

[15] 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*, pages 627–636, Montpellier, Sept. 2002.

[16] V. Paliouras and T. Stouraitis. A novel algorithm for accurate logarithmic number system subtraction. In *International Symposium on Circuits and Systems*, volume 4, pages 268–271. IEEE, May 1996.

[17] J. Ruan and M. Arnold. Combined LNS adder/subtractors for DCT hardware. In *1st Workshop on Embedded Systems for Real-Time Multimedia*, Oct. 2003.

[18] F. J. Taylor, R. Gill, J. Joseph, and J. Radke. A 20 bit logarithmic number system processor. *IEEE Transactions on Computers*, 37(2), Feb. 1988.