Customizing Floating-Point Operators for Linear Algebra Acceleration on FPGAs

Bogdan Pasca

ENS Lyon
Université de Lyon
LIP (UMR 5668 CNRS - ENS Lyon - INRIA - UCBL)
École Normale Superiéure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France

M2 internship report in the Arénaire Project
Supervisor – Florent de Dinechin

June 16, 2008

Abstract

Accelerating the execution of algorithms involving floating-point computations is currently a necessity in the scientific community. A solution – FPGAs – are believed to provide the best balance between costs, performance and flexibility. The FPGA's flexibility can be best exploited when used to accelerate "exotic operators"(log, exp, dot product) and operators tailored for the numerics of each application. These requirements gave birth to FloPoCo, a floating-point core generator written in C++ available under GPL at [1].

The purpose of this work was to bring FloPoCo to maturity, by framework stabilization and operator implementation. In term of framework stabilization we have implemented a novel *automatic pipeline generation* feature. In term of implemented operators, our work includes the basic blocks of FloPoCo: IntAdder, IntMultiplier, FPMultiplier, DotProduct, Karatsuba multiplication, FPAdder, LongAccumulator.

The obtained results are promising, ranking higher than the results obtained by FPLibrary [2] and not far from the operators generated with Xilinx CoreGen [3]. However, we generate portable VHDL code which is target independent, while CoreGen work only with Xilinx FPGAs. Nevertheless, work is still needed to bring some operators to CoreGen level.

We also studied the possibilities to implement *interval arithmetic* operators on FPGAs. We have proposed and discussed architectures for the four basic operations, for both infimum-supremum and midpoint-radius representations. We have also proposed an application for testing different trade-offs in terms of precision and size of these operators.

[1] http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo
[2] http://www.ens-lyon.fr/LIP/Arenaire/Ware/FPLibrary/
[3] http://www.xilinx.com/ipcenter/coregen

1 Introduction

1.1 Floating-Point Arithmetic

Several different representations of real point numbers exist, for instance the *signed logarithm* [1], *floating slash* and the *floating-point* [2] representations. Among these, by far the most popular is the floating-point one. It's definition can be stated as:

Definition. *In a floating point system of base b, mantissa length n and exponent range $E_{min}...E_{max}$, a number t is represented by a mantissa (or significand) $M_t = t_0.t_1...t_{n-1}$ which is a n-digit number in base b, satisfying $0 \leq M_t < b$, a sign $S_t = \pm 1$, and an exponent E_t, $E_{min} \leq E_t \leq E_{max}$, such that:*

$$t = S_t \times M_t \times b^{E_t}$$

To ensure the uniticity of representation it is usually required that $1 \leq M_t < b$ (a non-zero first digit).

The IEEE-754 standard defines the representation of floating-point single and double precision numbers [2]. These are the most common floating-point formats provided by computer hardware. The parameters of these formats are presented in table 1.

Name	b	n	E_{min}	E_{max}	Max Value
single precision	2	23+1	-126	127	$3.4... \times 10^{38}$
double precision	2	52+1	-1022	1023	$1.8... \times 10^{308}$

Table 1: IEEE 754 single and double precision floating-point formats

Rounding errors are inherent in floating-point computations. The simplest operations, like the sum or product of floating-point numbers, do not always generate a representable floating-point number for the result. In order to represent the result in the floating-point system under consideration, it might need to be *rounded*. We denote by *machine number* the floating-point number which can be exactly represented in a floating-point system.

The IEEE-754 standard defines four accepted rounding modes [2]:

- rounding towards $-\infty$: $\nabla(x)$ is the largest machine number less than or equal to x;
- rounding towards $+\infty$: $\triangle\ (x)$ is the smallest machine number greater than or equal to x;
- rounding towards 0 : $Z(x)$ is $\nabla(x)$ when $x > 0$ and $\triangle\ (x)$ when $x \leq 0$
- round to nearest : $\circ(x)$ is the closest machine number when x is between two machine numbers. When x is exactly in the middle of two machine numbers then the one which is *even* will be returned.

1.2 Field Programmable Gate Arrays

Field Programmable Gate Arrays or for short FPGA(s) are part of a larger family of reconfigurable hardware. They were first introduced in 1985, by Xilinx®. In present time, the FPGA market has grown considerably, large companies like Altera® or smaller ones like Actel® becoming more and more influent.

At a structural level, an FPGA consists of three parts [3]:

- a set of *configurable logic blocks (CLB)*,
- a *programmable interconnection network* also called the switch matrix,
- a set if *programmable input/output cells* around the device.

The top-level view of an FPGA architecture is presented in figure 1. The logic in an FPGA chip is organized hierarchical. The names and notations differ from manufacturer to manufacturer, but the organization is mostly the same. For example, in the case of Xilinx®FPGAs: at the top level of the hierarchy, defining the most generic organization of logic is the CLB, next are the slices 2, which are the constituents of the the CLBs. Depending on the FPGA, the number of slices in a CLB can be different.

At low level, the implementation of a function in an FPGA translates to the following steps:

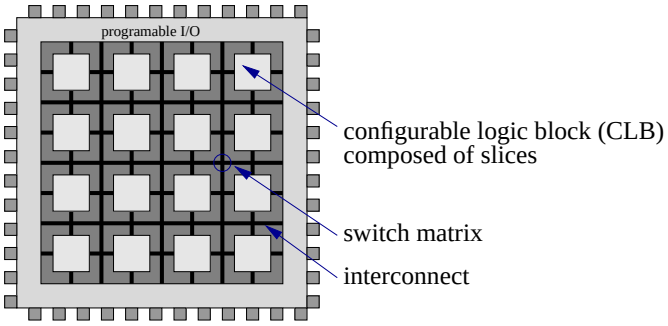


Figure 1: The basic architecture of an FPGA chip

- the function to be implemented is partitioned into modules which can be implemented into CLB,
- using the programmable interconnection network, the CLB are connected together,
- these are further connected to programmable I/Os for completing the implementation.

1.2.1 Slice architecture

Let us now consider the features of Xilinx®Virtex IV FPGA which are relevant to this work. The CLBs present in this FPGA contains four slices. A simplified diagram of the slice structure is presented in figure 2. The slices are equivalent and contain: two function generators, two storage elements, a fast carry chain, logic gates and large multiplexers [4]. It must be noted that Altera® chips share the same functionality.

The function generators are represented by configurable 4-input look-up tables (LUTs). The LUTs can be either used as 16-bit shift registers or as 16-bit distributed memories. In addition, the two storage elements are either edge triggered D-type flip-flops or level sensitive latches.

Each CLB has local fast interconnect resources for connecting internal slices, and connects to a switch matrix to access global routing resources. The fast carry chain uses dedicated routing resources of the FPGA. It is used to implement fast additions, multiplications and comparisons.

1.2.2 DSP blocks

In addition to the previous, most FPGAs today have all the discrete elements essential for digital signal processing (DSP). The modern DSP blocks are dedicated silicon blocks with dedicated, high-speed routing. The internal architecture is different from manufacturer to manufacturer but they generally share the same functionality.

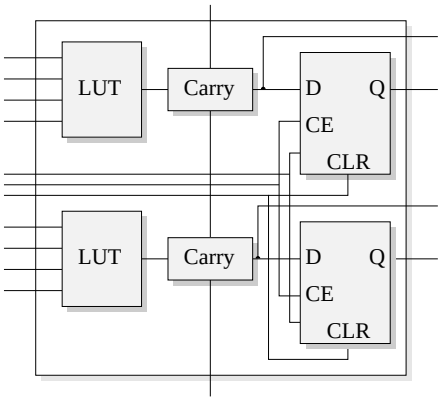


Figure 2: The basic architecture of an FPGA slice

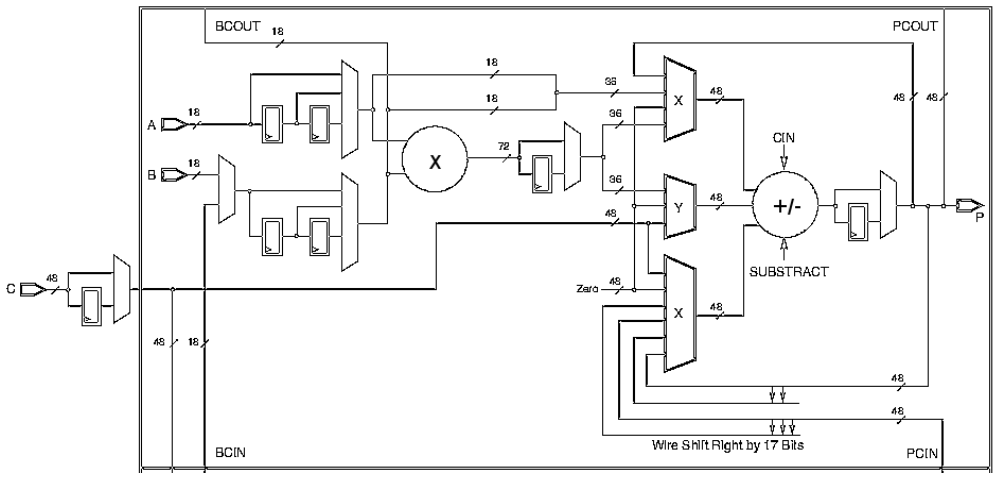


Figure 3: The Xilinx Virtex IV DSP48 block

For instance, the DSP block of a Xilinx Virtex IV FPGA 3 can either implement: an 18 bit x 18 bit signed integer multiplication, a multiplication followed by a 48-bit accumulation (MACC), or a multiplier followed by an adder/subtracter. The DSP blocks have built-in pipeline stages which provide enhanced performance for a throughput of up to 500 MHz. Moreover, due to their fabrication process the DSP blocks are also very power efficient.

The DSP blocks of the Altera Stratix device family have four 18x18-bit multipliers that can also be configured to support eight 9x9-bit multiplication or one 36x36-bit multiplication for different applications. The DSP block also contains an adder/subtractor/accumulator unit can be configured as an adder, a subtracter, or as an accumulator on 9-bit, 18-bit, or 36-bits, as necessary. In the accumulator mode, the unit may act as a 52-bit accumulator useful for building double-precision floating-point arithmetic. However, the DSP blocks in Stratix devices run at a maximum of 333 MHz.

2 State of the art

2.1 Floating-point computation using FPGAs

Most scientific algorithms require some form of fractional representation for their internal variables. In most popular programming languages the solution is to declare variables as floating-point numbers. The representation of these numbers is often based on one of the two IEEE-754 floating-point formats (see table 1).

It is generally believed that reconfigurable logic (such as the FPGA), has the potential to speed up many of these scientific algorithms. Due to modest initial densities of FPGAs, porting scientific algorithms on them usually meant transforming all the internal variable representation into fixed point. However, in some algorithms, the dynamic range of variables is impractical for a fixed point implementation. Together with the commodity of floating-point formats, the motivation for implementing floating-point operators on FPGA conducted numerous studies ([5], [6], [7], [8]). However, floating-point computations started to become feasible on FPGAs only during the mid 90'. This was mainly due to the increases in speed and density of FPGAs.

The studies conducted on this subject took two different approaches.

For the first, in order to take full advantage of the FPGA resources, the operators are tailored in order to conform to specific input/output constraints constraints. Shirazi et al. [6] provided two custom floating-point formats (16 bits and 18 bits total) and designed specific architectures for them of the operations of addition and multiplication. In [9], Shirazi et al. suggest customizing the representation of floating-point numbers in order to exploit the flexibility of reconfigurable hardware.

Consequently, many parameterized floating-point operators have been proposed. Some of the parameterizations take advantage of specific characteristics for the deployment target [10]. Other are simply parameterized in precision so that a good compromise between precision, speed and accuracy is obtained [7]. Some parameterizable operators are distributed under the form of generic libraries ([11], [1], [12]) so that the operators can be used with off-the-shelf

FPGAs.

The second research approach consisted of designing single and double-precision operators for FPGAs and examining the speedup of these operators when compared to those present in high-end general purpose processors. The first feasible single-precision implementation results were presented in [13] but no significant speedup was obtained on a Pentium processor. In [8] Underwood presented designs for single and double precision IEEE compliant floating-point addition, multiplication, and division. His analysis on the trends of floating-point computing estimates that the performance peak of FPGAs will surpass by one order of magnitude that of general-purpose processors in 2009.

Another idea was to use not only custom precision, but also custom operators and architectures. Following this approach this order of magnitude was already reached in 2005 ([14] and [15]) for elementary functions. This further conducted to the idea of building a tool for generating custom operators for FPGAs, exotic operators, ones which would not be economical for implementing in general purpose processors.

3 The FloPoCo project

3.1 Context

With the current approach of parameterizable libraries for VHDL operators reaching it's limits, the time came for exploring a different perspective in generating VHDL operators. This motivation gave birth to FloPoCo, a floating-point core generator for FPGAs.

FloPoCo is the successor of FPLibrary, a library of parameterizable floating-point operators for FPGAs. Developing FloPoCo became a necessity due to problems regarding the user experience in using the library and the difficulties encountered during late project development. From the user's point of view, the unpleasant experiences were regarding:

- importing the library to the project; tedious work in adding files manually to project.
- the difficulty of re-pipelining operators.
- lack of design space exploration, i.e. due to their generic nature, libraries cannot choose the best among possible implementation solutions.

The advantage with having a generator of operators is that the VHDL code remains clean of generic and recursive code. The complexity of the code is transferred from the VHDL sources to the generation software. Consequently, the VHDL code is simplified and synthesis takes less. Moreover, the generation of operators may take into account different architectural parameters (number of LUT inputs, carry propagation delay, etc.) and also constraints imposed by the user regarding area, speed or accuracy. Therefore, generators offer the possibility to adapt the generated operator pipeline to multiple constraints. Additionally, generators allow a high-level input specification for operators together with a complex design space exploration in order to find the design which best fits the input constraints.

The purpose of FloPoCo is to generate state of the art operators which satisfy multiple constraints. We now present detailed examples for operators we implemented in FloPoCo.

3.2 Integer addition

We consider here the addition of two positive integers represented in radix 2. The global architecture of this adder is shown in the left of figure 4. The inputs of this operator are X and Y satisfying $0 \leq X, Y \leq 2^n - 1$ and the carry in bit, denoted by $c_{in} \in \{0, 1\}$. The output are the sum $0 \leq S \leq 2^n - 1$ and the carry out bit $c_{out} \in \{0, 1\}$ such that:

$$X + Y + c_{in} = 2^n c_{out} + S$$

In the case when $n = 1$ this the above formula reduces to a primitive called the *full-adder* represented in the right part of figure 4.

FloPoCo is able to generate a range of operators performing integer addition. Available parameters for this operator comprise:

- `pipeline=yes/no`. The architecture generated according to this input parameter is either sequential if the parameter value is "yes", or combinatorial otherwise.

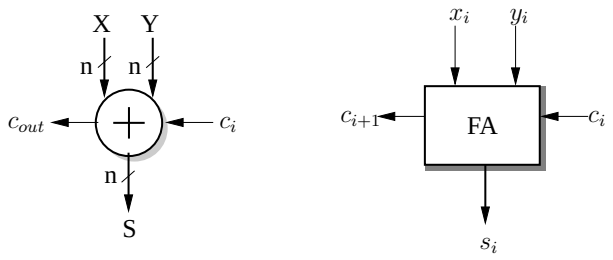


Figure 4: Left: generic n-bit adder architecture. Right: 1-bit full-adder module

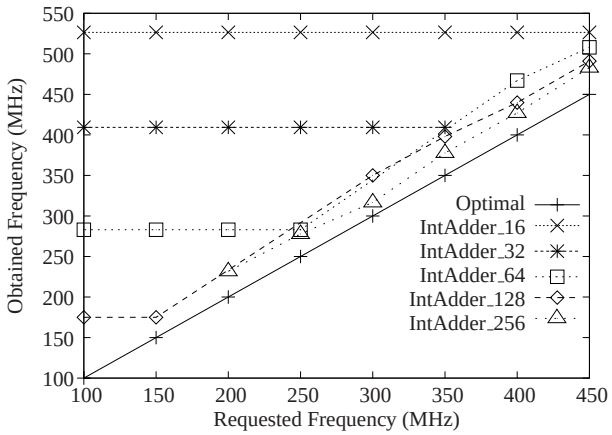


Figure 5: Obtained frequencies against requested frequencies

- `frequency=f`. This option, used together with `pipeline=yes` determines the pipeline structure of the output operator. The pipeline is modified so that the frequency of the output operator be as close as possible to the requested frequency . Figure 5 plots the obtained frequencies against requested frequencies for different sizes of integer addition. The solid line plots the optimal output frequencies for each of the input case. One can observe that the obtained frequencies for this operator are always greater than the optimal frequencies. The difference between these frequencies is larger for small addition size but this comes at no expense from the resource point of view. For instance, the addition of two 16-bit numbers realized by the IntAdder_16 component has an output frequency of more than 500MHz without using any registers to pipeline this addition. Moreover, for IntAdder_256 the obtained frequencies are closely controlled through addition pipelining.

In order to reach the requested input frequencies, FloPoCo needs to pipeline operators. FloPoCo implements *automatic pipelining*, i.e. the register levels are inserted into the operator architecture automatically. In order for an operator to reach a certain frequency, the critical path delay of that operator needs not be greater than the period at the given frequency. Due to complex combinatorial operations, the input signals accumulate delays from the circuit's input towards the output. The registers represent memory elements (storage) which are able to store the information delivered at their input, when requested. By inserting memory elements for storing the intermediary results in an operation, we stop the accumulation of signal delays, and thus we are able to constrain our circuit delays under a given bound.

As addition uses the fast-carry logic, we are able to derive a rather precise mathematical formulation for the delay of an n-bit adder. This delay is greatly influenced by the specific characteristics of the deployment target:

$$delay_{nBitAdd} = delay_{LUT} + n \cdot delay_{CarryPropagation}$$

where $delay_{LUT}$ represents the LUT delay, and $delay_{CarryPropagation}$ represents the delay for propagating the carry bit between adjacent logic elements by using the dedicated fast carry logic routing. For a given input frequency f to be reached, the accumulated delays need to be smaller than $1/f$. If we impose this on the above equation and extract n we have:

$$delay_{LUT} = 1.5 \cdot 10^{-9}$$
$$delay_{CarryPropagation} = 3.4 \cdot 10^{-11}$$
$$f = 400 \cdot 10^{6} \text{ (400 MHz)}$$

Computing n:

$$n = \left\lfloor \frac{1/f - delay_{LUT}}{delay_{CarryPropagation}} \right\rfloor = 29$$

```
X -> X[79..58] X[57..29]
X[28..0]; Y -> Y[79..58]
Y[57..29] Y[28..0];
```

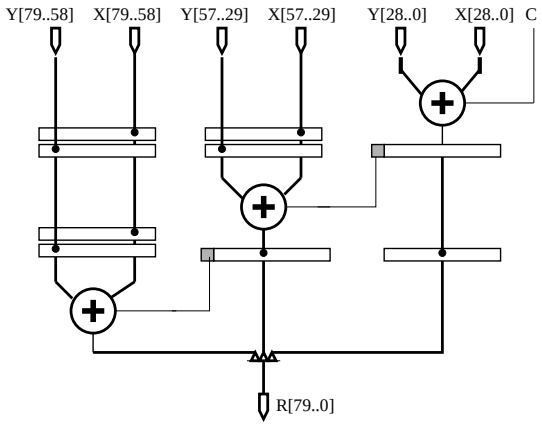


Figure 6: Pipelining the 80-bit addition of X and Y at 400 MHz on Virtex 4 device. Pipeline levels for splitting an 80-bit addition

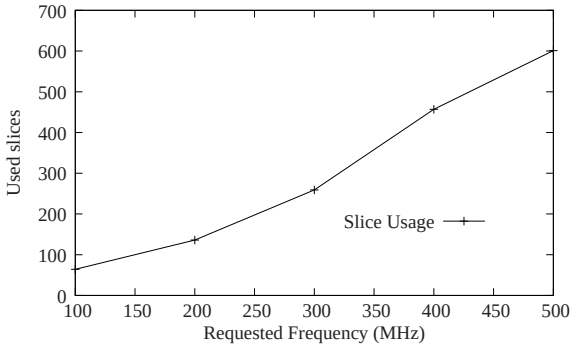


Figure 7: Dependency between requested frequency and used slices for IntAdder_128

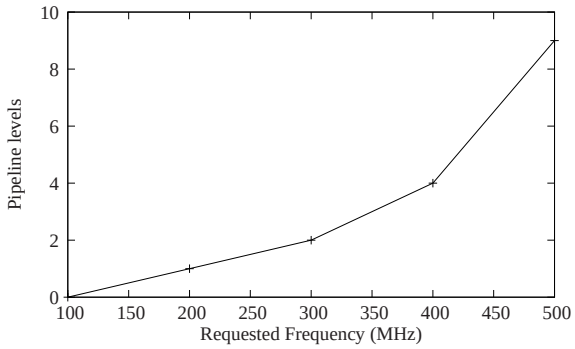


Figure 8: Dependency between requested frequency and architecture pipeline levels for IntAdder_128

$$n < \frac{1/f - delay_{LUT}}{delay_{CarryPropagation}}$$

with the constraint that n is always an integer value. If the result of this equation is less or equal to 0, it follows that the requested frequency cannot be reached for the n-bit addition and consequently, the addition needs to be broken into smaller additions. An example of this is shown in figure 6 where the input n-bit addition needs to be broken into tree parts so that the requested frequency is reached.

Figure 7 shows the dependency between the requested frequency and the number of slices used by the architecture IntAdder_128. For the simple pipeline model of the integer adder, one can observe a linear increase in the number of slices used by the architecture. On the other hand, we see in figure 8 that the number of pipeline levels of the architecture has a more than linear increase.

Now that we studied and implemented the operator performing the integer addition, let us move-on to a more complex operator – the integer multiplier.

3.3 Integer multiplication

We consider here the multiplication of two positive integers represented in radix 2. The global top level schematic for an n-bit multiplier is shown in figure 9. The inputs of this operator are X and Y satisfying $0 \leq X, Y \leq 2^n - 1$ and the output is the product of X and X having the property $0 \leq P \leq 2^{2n} - 1$.

The FloPoCo implementation of integer multiplication takes full advantage of the special FPGA target features such as DSP blocks. As previously stated, these blocks are able to perform multiplications of unsigned numbers of up

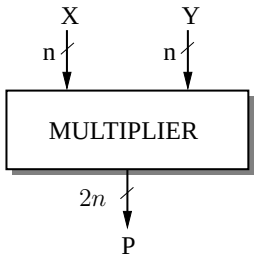


Figure 9: Top level schematic of an n-bit integer multiplication

to 17-bits wide in the case of Virtex IV device.

Let us denote by X and Y the n-bit input numbers which we want to multiply and by r the width in bits of the embedded multipliers. We now look at the multiplication $X \times Y$ as a multiplication in radix 2^r. If necessary, we pad the inputs with zeros to the left such that the width of the inputs transforms from n to n_r where n_r is the smallest number satisfying the property $n_r \geq n$ and $\frac{n_r}{r} \in \mathbb{N}$.

The two input numbers are now divided into p parts, each part containing $\frac{n_r}{r}$ digits. We denote the parts for X by X_1,...,X_p and similarly for Y. The denote product of two 1-digit radix 2^r numbers by:

$$X_j \times Y_i = {P_{i,j}}^H \cdot 2^r + {P_{i,j}}^L.$$

where ${P_{i,j}}^H$ and ${P_{i,j}}^L$ are the are the two digits of the result.

Now writing the multiplication in radix 2^r becomes:

$$X_p X_{p-1} ... X_1 \times Y_p Y_{p-1} ... Y_1 = \sum_{j=1}^{p} \sum_{i=1}^{p} {P_{i,j}}^L \cdot r^{i+j} + \sum_{j} \sum_{i} {P_{i,j}}^H \cdot r^{i+j+1}$$

This equation shows how the n-bit base 2 multiplication can be transformed into p^2 1-digit base 2^r multiplications and summations. The basis 2^r multiplication can be directly feed to the DSP block on the FPGA. The result of this multiplications will now need to be shifted and added. Fortunately, we can concatenate into bit-vectors with the names Low_i the products of the form ${P_{i,j}}^L$ and with the name $High_i$ the products the form ${P_{i,j}}^H$ for fixed values of i. Now, the only shifts which appear are those between bit-vectors Low_i and Low_{i+1} and correspondingly between $High_i$ and $High_{i+1}$ and one global shift on all bit-vectors of the form ${P_{i,j}}^H$.

An example of how this is transformed into an architecture is given in figure 10 for the case when $p = 3$. Additionally, a simplified version of the adder structure which sums all partial products is shown. The low part of the final product is obtained gradually by summing available vectors. For instance, the least significant r-bits of the final product are available immediately after the multiplications. Following the addition of the first two vectors from the low part of the result, the next r bits of final product are computed by summing-up the least significant r bits of the first two low bit-vector sum (Low_1+Low_2), and the least significant r bits of the first bit-vector of the high part of the result ($High_1$). Then the process repeats.

For simplicity of representations technical details were removed from the schematic. The technical details include pipelining of the long additions between concatenation vectors, together with pipelining of the last addition which computes the high part of the result. These pipeline stages permits arbitrary frequencies to be attained by this operator.

This architecture is optimized for the use of DSP blocks as it is performing the computation of the partial products in parallel. The results showing the dependency between the requested frequency and the obtained frequency for the architectures IntMultiplier_24_24 and IntMultiplier_53_53 are shown in figure 11. This integer multiplications have the same size as the mantissa multiplications for the IEEE-754 single and double precision representations. We remark that in general, we provide higher frequencies than requested. The exceptions can be explained with the help of figure 13 which presents the dependency between the requested frequency and the number of pipeline levels of the architectures. We can observe the rapid increase in the number of pipeline levels as the requested frequencies grow above 200MHz. Consequently, the additional number of registers inserted for the pipeline levels causes the area of the architecture to increase. Therefore, the routing delays become more significant and reduce the frequency of the architecture. In this cases the solution is to request larger frequencies than desired, and repeat the process if unsuccessful.

Let us pass to an even more complex operator, the floating-point multiplier.

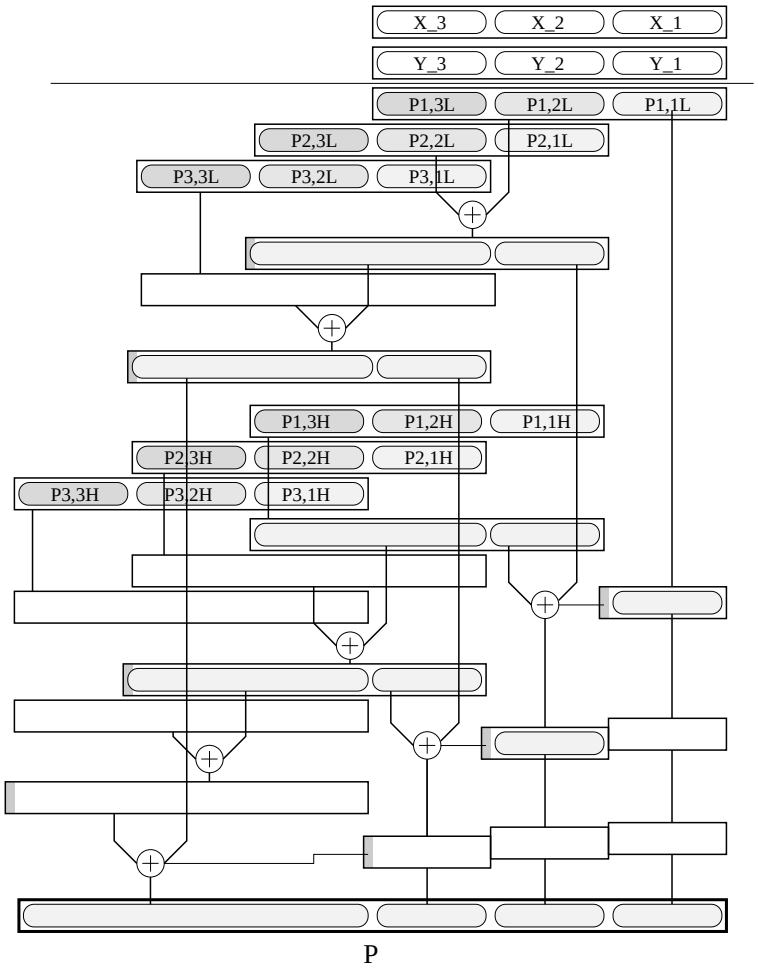


Figure 10: A example architecture of pipelined addition tree for integer multiplication. Each input is split into three chunks

3.4 Floating-point multiplication

We now consider the multiplication of two floating-point numbers, X and Y. These operands have the representation (M_x^*, E_x) and (M_y^*, E_y) respectively. The significands of the operands are signed and normalized. The product of X and Y is written as:

$$Z = X \times Y$$

where Z is denoted by the pair (M_Z^*, E_Z) which is also signed and normalized. From the algorithmic point of view, the floating-point multiplication has the following steps:

1. multiply the significands and add the exponents:

$$M_Z^* = M_X^* \times M_Y^*$$
$$E_Z = E_X + E_Y$$

2. normalize M_Z^* and update exponent accordingly,

3. perform rounding,

4. set the exception bits.

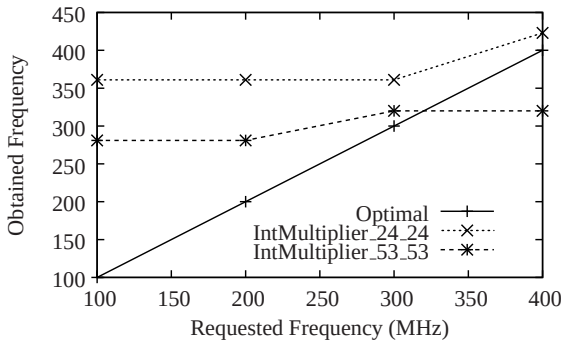


Figure 11: Dependency between requested frequency and obtained frequency for IntMultiplier_24_24 and IntMultiplier_53_53

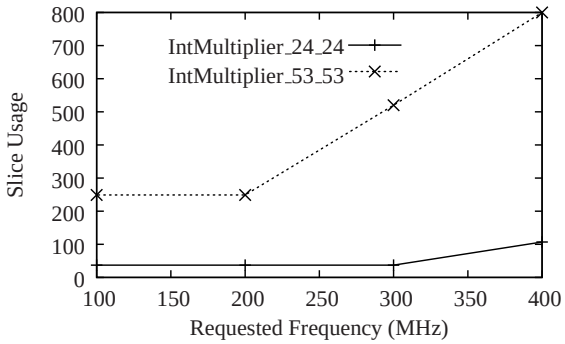


Figure 12: Dependency between requested frequency and used slices for IntMultiplier_24_24 and IntMultiplier_53_53

The floating-point (FP) number format used in FloPoCo is inspired from the IEEE-754 standard [2]. The key idea of the format is to represent numbers with a fixed-point normalized mantissa multiplied by an order of magnitude. This representation is parameterized by two bit-widths: w_E and w_F. The floating-point number is now represented as a vector of $w_E + w_F + 3$ bits which is partitioned in 4 fields as shown Figure 14. The description of the 4 fields is given below:

- exn (2 bits): the exception tag, used for infinities and NotANumber(NaN), zero.
- S_X (1 bit): the sign bit;
- E_X (w_E bits): the exponent (biased by $E_0 = 2^{w_E-1} - 1$);
- F_X (w_F bits): the fraction (mantissa).

The top level diagram for floating-point multiplication is presented in figure 15. The architecture is based on that of the integer multiplier, which is the main component of the floating-point multiplier. The FloPoCo implementation of the FPMultiplier performs rounding to nearest, as described by the IEEE-754 standard. The details regarding pipelining and it's synchronization have been left out for the sake of simplicity. It must be noted however, that the current architecture is able to multiply numbers having different precisions, output the result on the a desired precision and adapt the pipelining levels to the desired input frequency automatically.

The dependency of the output frequency with respect to the input frequency is presented in figure 16. It can be observed that the frequencies are usually better than expected, except when the frequencies become high. The bottleneck in this case is the integer multiplier. The number of slices used by the FPMultiplier is shown in figure 17 together with the number of slices used internally by the integer multiplier. As we can observe, the overhead induced

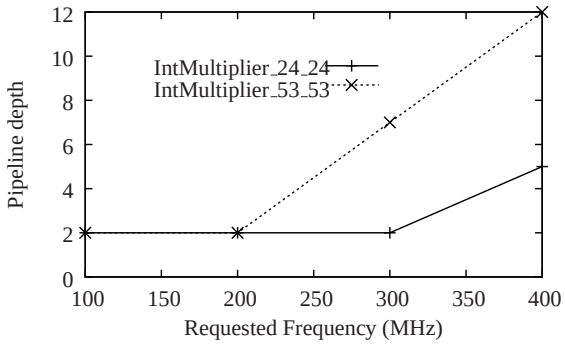


Figure 13: The dependency between requested frequency and the number of pipe levels for IntMultiplier_24_24 and IntMultiplier_53_53

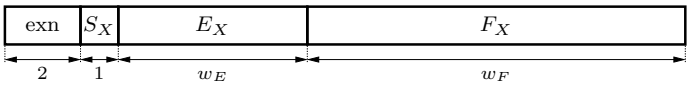


Figure 14: Floating-point number format.

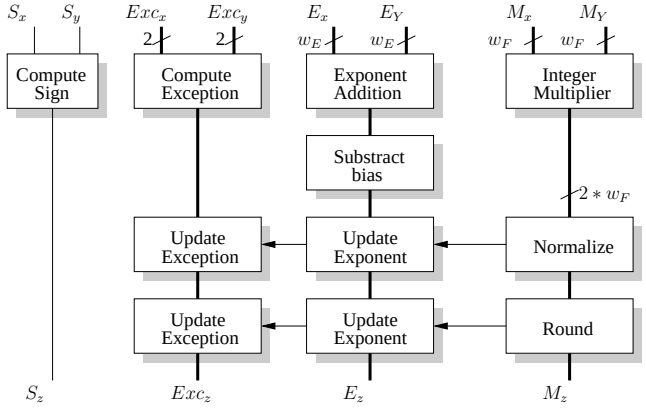


Figure 15: FPMultiplier top level diagram

by the rest of the circuitry of the FPMultiplier is less than the size of the integer multiplier. Figure 18 shows the growth in number of pipeline levels of the generated architecture as the requested frequency increases. We can observe that the growth rate is similar to the growth rate of the integer multiplier, suggesting again that this is the largest resource consuming component of the FPMultiplier.

Let us now see where this results stand. For comparisons we used FPLibrary, a VHDL library of parameterizable

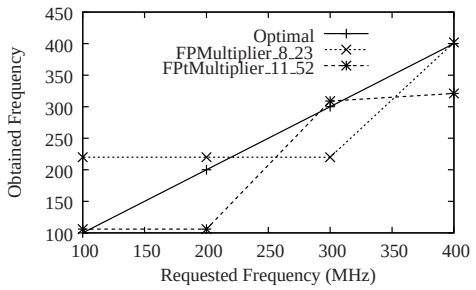


Figure 16: Dependency between requested frequency and obtained frequency for FPMultiplier_8_23 and IntMultiplier_11_52

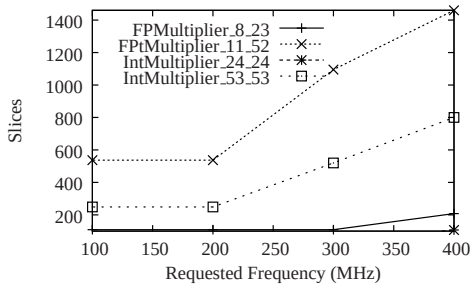


Figure 17: Dependency between requested frequency and used slices for FPMultiplier_8_23 and FPMultiplier_11_52. Area results for integer multiplier of mantissas is also presented

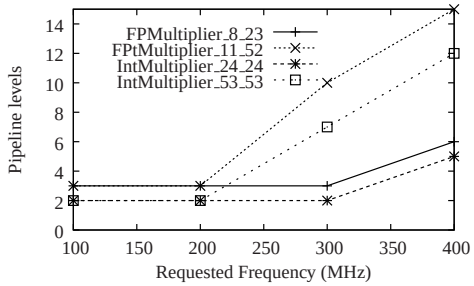


Figure 18: The dependency between requested frequency and the number of pipelevels for FPMultiplier_8_23 and IntMultiplier_8_53. Pipelining levels of integer multiplier for mantissas is also presented

floating point operators available at: http://www.ens-lyon.fr/LIP/Arenaire/Ware/FPLibrary/ for the case of the IEEE single and double precision representations. The result are presented in table 2. In the case of single precision, the operator generated by FPLibrary takes more than twice the area compared to the operator generated by FloPoCo (450 slices vs. 208 slices). The frequency of the FPLibrary operator is also much lower (163 MHz vs. 402 Mhz) than the frequency of FPMultipler operator. Moreover, FloPoCo supports different precision for the inputs, and also different precision for the output. This is not the case when working with FPLibrary.

We also made a rather unfair comparison with the Xilinx CoreGen tool, a core generator which is the state of the art for Xilinx platforms. Table 2 shows the performances for the operators generated by CoreGen. In the case of single precision, the FloPoCo operator is not far behind in terms of frequency (471Mhz vs 402Mhz) and in terms of slices used (171 slices vs. 208 slice). However, the latency of the our operator is only 6 when compared to 10 for the CoreGen operator. Moreover, CoreGen only supports high speed modes, so in the case when a particular frequency is enough for the operator, CoreGen generates the best operator it can, using the same amount of resources. For instance, for a single precision operator at 100Mhz, FloPoCo uses 109 slices whereas CoreGen still uses 171 slices. Moreover, CoreGen does not support different input precision. In the case of double precision operator we observe that when only 9 DSP blocks are used by the CoreGen operator, this operator is slightly inferior to the one generated by FloPoCo. However, when giving CoreGen the possibility to use the full number of DSP blocks, it's performance becomes superior to that of FloPoCo. Nevertheless, CoreGen does not generate portable VHDL files and works only for Xilinx targets.

Unlike CoreGen, FloPoCo is a generic floating-point core generator. The internal code of FloPoCo is not architecture specific, but parameterized function of the deployment target. The aim of FloPoCo is to generate architectures for any targets. For retargeting a specific operator, the code which generates the architectures does not need to be modified. What is needed is the addition of a Target class bearing the internal characteristics of the deployment target. This way, the portion of code which needs modifications is minor. However, the price that FloPoCo pays for being generic is that it's generated operators are still a little bit slower and larger than the state of the art operators generated by CoreGen. This price becomes lower if we also consider that the time needed for deploying a new operator is greatly reduced.

Software	Representation	Frequency	Slices	DSP
FPLibrary	sp	216 MHz	450	1
FPLibrary	dp	163 MHz	1339	30
CoreGen	sp	471 MHz	171	4
CoreGen	dp	265 MHz	1016	9
CoreGen	dp	361 MHz	469	16
FloPoCo	sp	402 MHz	208	4
FloPoCo	dp	309 MHz	1095	16

Table 2: Synthesis result for floating point multiplication with FPLibrary, CoreGen and Flopoco (sp means single precision and dp means double precision)

3.5 Other operators

We have also implemented in FloPoCo the dual-path floating-point addition. However, this architecture needs more work at the level of pipelining the leading zero counter component. This component uses long comparisons which could be performed using the DSP blocks of modern FPGAs.

In order to reduce the number of DSP blocks used by the integer multiplication we have designed a combinatorial version of the Karatsuba multiplication. Early results are promising, since we have managed, for a 53 bit multiplication, to reduce the number of used DSP blocks from 16 to just 9. We have currently in plan to pipeline this operator.

We have also worked on using these multipliers in specific architectures for dot product and sum of squares. Due to lack of space we omit this work here.

3.6 Conclusion

We are now happy that automatic pipelining works well even for composed operators. So, we can now attack more complex problems, for example, interval arithmetic operators which, at their basis, are composed of optimized basic operators. We will try to develop operators which outperform their equivalents in general purpose processors, at least in some cases.

4 Opportunities in deploying interval arithmetic on FPGA

4.1 Notions of interval arithmetic

In real life, due to the inevitable measurement inaccuracy, the exact values of the measured quantities are unknown. What are know are, at best, the intervals of possible values for those quantities.

When we use a computer to make computations involving real numbers, we have to use the finite set of floating-point numbers that the hardware makes available. It is known that most real values do not have a finite floating-point representation. The possibilities in representing real numbers in computers consist of either using floating-point numbers obtained by means of rounding, or by using an interval consisting of two floating-point numbers with the property:

$$f_1 \leq r \leq f_2 \text{ where } r \in \mathbb{R} \text{ and } f_1, f_2 \in \mathbb{F}$$

Here, $\mathbb{F}$ represents the set of representable floating-point numbers and $\mathbb{R}$ the set of real numbers.

Interval arithmetic is based on the idea of bounding real values with representable intervals. On one hand, it provides means for reasoning about all the possible results of a computation involving an input interval. On the other hand, it is used to provide guarantees on the result of a floating-point computation, i.e. for a given computation, no matter what rounding errors occur, the real result is always included in the output interval. This is a fundamental property of interval arithmetic and is called the *inclusion property*.

There are two common representations of intervals over $\mathbb{R}$. First, we have the *infimum-supremum* representation which denotes an interval by specifying it's two ends:

$$[a_1, a_2] := \{x \in \mathbb{R} : a_1 \leq x \leq a_2\} \text{ for some } a_1, a_2 \in \mathbb{R},\ a_1 \leq a_2$$

Second, we have the *midpoint-radius* representation which denotes an interval by specifying it's center (also called midpoint), and it's radius.

$$< a, \alpha >:= \{x \in \mathbb{R} : |x - a| \leq \alpha\} \text{ for some } a \in \mathbb{R},\ 0 \leq \alpha \in \mathbb{R}$$

In this study we are concerned only with interval arithmetic operations for which intervals are represented over $\mathbb{IF}$, which is the set of intervals represented using floating-point numbers. The set $\mathbb{F}_{Exp,Frac}$ contains the floating-point numbers having an exponent Exp and a fractional part $Frac$. For instance, the IEEE-754 single precision floating-point representation is denoted by $\mathbb{F}_{8,23}$. Consequently, the following relation holds over the set of floating-point numbers:

$$\mathbb{F} = \bigcup_{i,j \in \mathbb{N}} \mathbb{F}_{i,j}$$

The two different interval representations yield different $\mathbb{IF}$ sets. For instance, not all intervals which are representable in midpoint-radius format have the equivalent in infimum-supremum representation. Accordingly, we customize the notation for the set of intervals derived from the notation in the two cases. As a result, the notation for the set of intervals using the infimum-supremum representation, is $\mathbb{IF}^{IS}_{Exp,Frac}$. Redefining the set of intervals with this notation gives:

$$\mathbb{IF}^{IS}_{Exp,Frac} := \{[a_1, a_2] : a_1, a_2 \in \mathbb{F}_{Exp,Frac},\ a_1 \leq a_2\}$$

In the case of the midpoint-radius representation, an optional parameter appears in the notation denoting the floating-point characteristic of the radius. Consequently, we denote the set of floating point intervals by $\mathbb{IF}^{MR}_{(Exp,Frac),(Exp_r,Frac_r)}$. Using this new notation, the set of intervals using the midpint-radius representation is:

$$\mathbb{IF}^{MR}_{(Exp,Frac),(Exp_r,Frac_r)} := \{< a, \alpha >: a \in \mathbb{F}_{Exp,Frac}\ , \alpha \in \mathbb{F}_{Exp_r,Frac_r},\ \alpha \geq 0\}$$

Throughout this section, whenever the center and the radius belong to the same $\mathbb{F}_{Exp,Frac}$, we forget the second parameter of the representation for the set of intervals determined by using the midpoint-radius representation.

Our efforts to study the opportunities of interval arithmetic on FPGAs are motivated by the desire to provide more performance, at least in some situations, than general purpose processor can offer for interval arithmetic operators. Particularly, for the two representations of intervals, we design optimized architectures and compare these architectures

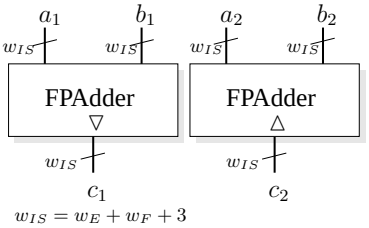


Figure 19: Infimum-Supremum addition architecture

in terms of resource usage. We then present an application for which the operator architectures designed for FPGAs have the opportunity to accelerate computations.

4.2 An analysis of algebraic operations

In this section the four basic operations: addition, substraction, multiplication and division will be analysed for each of the two interval representations. Issues regarding resource requirements and performance expectations will be also covered.

Let us now define the intervals which will be used to perform the basic operations:

Let $A = [a_1, a_2]$ and $B = [b_1, b_2]$ denote interval operands having the infimum-supremum representation, where $a_1, a_2, b_1, b_2 \in \mathbb{F}$ and $a_1 \leq a_2$, $b_1 \leq b_2$, $A, B \in \mathbb{IF}^{IS}_{Exp,Frac}$. Let $C = [c_1, c_2]$ with $c_1, c_2 \in \mathbb{F}$ and $c_1 \leq c_2$ denote the result of an interval operation in infimum-supremum representation $C \in \mathbb{IF}^{IS}_{Exp,Frac}$.

Let $A =< a, \alpha >$ and $B =< b, \beta >$ be intervals in midpoint-radius representation for which $a, \alpha, b, \beta \in \mathbb{F}$ and $\alpha \geq 0$, $\beta \geq 0$. Let $C =< c, \gamma >$ with $c, \gamma \in \mathbb{F}$ and $\gamma \geq 0$ denote the result of the interval operation in midpoint-radius representation. When we talk about operators in midpoint-radius representation, we say that $A, B \in \mathbb{IF}^{MR}_{(Exp,Frac),(Exp_r,Frac_r)}$.

4.2.1 Addition

In the case of infimum-supremum interval representation, the result of $A + B$ is [16]:

$$C = [c_1, c_2] \text{ where } c_1 = \nabla(a_1 + b_1) \text{ and } c_2 = \triangle\ (a_1 + b_1)$$

The architecture of this operator is presented in figure 19. For this interval addition to be implemented in an FPGA, the cost is of two floating-point adders as the one shown in figure 20. The left adder one computes the sum $a_1 + b_1$ rounding towards $-\infty$ while the right one computes the sum $a_2 + b_2$ rounding towards $+\infty$.

For general-purpose processors, the possibilities in accelerating floating-point computations are by use of pipelining the operations or by use of a single instruction multiple data (SIMD) instruction set with support for floating-point operations. The first who gave support to the instruction set was AMD© with the 3dNow!© technology, and then, 1 year later came Intel© with the SSE© technology. This technologies permit multiple floating-point computations be performed in parallel. Both instruction sets support the IEEE-754 rounding modes, but cannot use different rounding mode in parallel for the same instruction.

Consequently, if a large number of interval additions need to be performed, then performing the all additions which round towards $-\infty$ and then all additions which round towards $+\infty$ would give best performance for the processor. This would however require extra programming overhead. Otherwise, the feature behind the SIMD technology cannot accelerate the computations.

In the case of intervals represented in midpoint-radius form, adding two intervals reduces to adding their midpoints and then adding their radii. As the result of the adding of two floating-point numbers is not necessarily a floating-point number, when adding the centers of the intervals, the error which occurs due to rounding might cause a violation of the fundamental inclusion property. In order to prevent this, we must add to the sum of the radii a quantity which corrects this rounding error.

So, the result of the addition $A + B$ becomes [17]:

$$C =< c, \gamma > \text{ where } c = \circ(a + b) \text{ and } \gamma = \triangle\ (\varepsilon' \cdot |c| + \alpha + \beta)$$

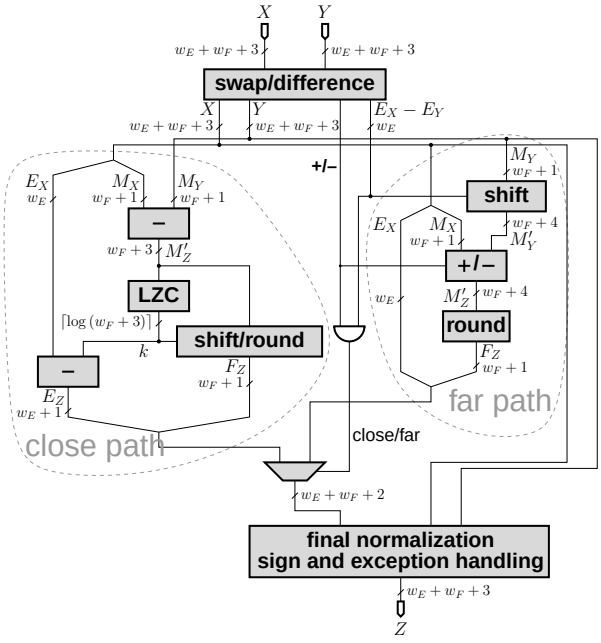


Figure 20: The architecture of a dual-path floating-point adder

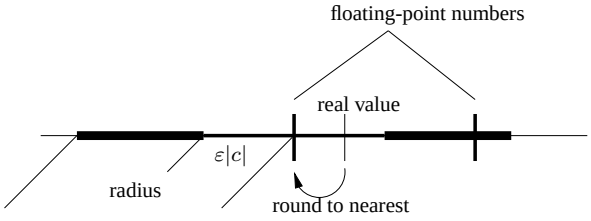


Figure 21: The overestimation caused by the correctly rounding the sum of midpoint values

The quantity ε denotes the relative rounding error unit. For example, in the case of the IEEE-754 double-precision format, $\varepsilon = 2^{-52}$. Furthermore, ε' is set to $\frac{1}{2}\varepsilon$. Figure 21 shows the overestimation introduced by the midpoint rounding in computing the result radius. This overestimation is caused by the term $\varepsilon'|c|$. which is necessary for preserving the inclusion property of the operation.

The architecture of this operation is presented in figure 22. We make some considerations form architectural point of view:

- a data dependency appears between the computation of result midpoint and the computation of the result radius. From technical point of view, this data dependency consists in adding register levels for delaying the computation of the radius. Fortunately, this is not necessary, due to the fact that the addition of midpoints and radii can be performed concurrently as can be observed from figure 22. Moreover, no register levels need to be adders for synchronization due to the fact that the floating-point adder has the same number of pipeline levels independent of the rounding mode.

- the product $\varepsilon' \cdot |c|$, where $\log \varepsilon' < 0$ requires only a short 2's complement integer addition $Exp_c - \log \varepsilon'$ for the exponent computation which can be fastly computed in an FPGA. The fractional part of $\varepsilon' \cdot |c|$ remains unchanged.

- the dual-path floating-point adder is a fast architecture for performing floating-point additions. The two paths have each a bottleneck. For the close path, the bottleneck is the leading zero counter, while for the far path, the bottleneck is the large shifter. When one path adders are used, the two bottlenecks appear on the critical path and approximately double the latency of the operator. Dual-path adders were introduced to reduce the operation

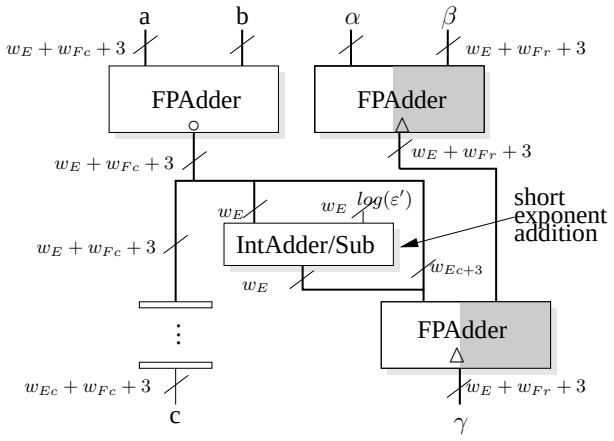


Figure 22: Midpoint-radius addition architecture

latency. Thus, the close path is only used for performing additions of X and Y, where $sign(X) * sign(Y) = -1$ and $|Exp_X - Exp_X| \leq 1$.

Now considering that $\triangle\ (\alpha + \beta) \geq 0$ and that $\varepsilon' \cdot |c| \geq 0$, i.e. the addition operands are greater or equal to 0, the floating point addition $\triangle\ (\varepsilon' \cdot |c| + \alpha + \beta)$ is implemented using only the far path of the FPAdder, thus saving appropriately half of the area of a FPAdder.

4.2.2 Subtraction

For infimum-supremum representation of intervals, the result of the substracting B from A is:

$$C = [c_1, c_2] \text{ where } c_1 = \triangledown(a_1 - b_1) \text{ and } c_2 = \triangle\ (a_1 + b_1)$$

In architectural terms, substraction in infimum-supremum representation has the same architecture as addition (see figure 19).

For intervals represented in midpoint-radius form, the result of substracting the interval B from A is:

$$C = < c, \gamma > \text{ where } c = \circ(a - b) \text{ and } \gamma = \triangle\ (\varepsilon' \cdot |c| + \alpha + \beta) \tag{1}$$

The architectural considerations are similar as for addition (figure 22).

4.2.3 Multiplication

For infimum-supremum representation of intervals, the result of the multiplying A and B depends on the position of the interval ends with respect to the origin (signs of the interval ends). Therefore, $A \cdot B$ gives [16]:

	$b_1 \geq 0$	$b_1 < 0 \leq b_2$	$b_2 < 0$
$a_1 \geq 0$	$[\triangledown(a_1 \cdot b_1), \triangle\ (a_2 \cdot b_2)]$	$[\triangledown(a_2 \cdot b_1), \triangle\ (a_2 \cdot b_2)]$	$[\triangledown(a_2 \cdot b_1), \triangle\ (a_1 \cdot b_2)]$
$a_1 < 0 \leq a_2$	$[\triangledown(a_1 \cdot b_2), \triangle\ (a_2 \cdot b_2)]$	$[\tilde{z}, \tilde{\zeta}]$	$[\triangledown(a_2 \cdot b_1), \triangle\ (a_1 \cdot b_1)]$
$a_2 < 0$	$[\triangledown(a_1 \cdot b_2), \triangle\ (a_1 \cdot b_1)]$	$[\triangledown(a_1 \cdot b_2), \triangle\ (a_1 \cdot b_1)]$	$[\triangledown(a_2 \cdot b_2), \triangle\ (a_1 \cdot b_1)]$

where $\tilde{z} = min(\triangledown(a_1 \cdot b_2), \triangledown(a_2 \cdot b_1))$ and $\tilde{\zeta} = max(\triangle\ (a_1 \cdot b_1), \triangle\ (a_2 \cdot b_2))$.

The architecture of this operator is presented in figure 23. There are a few considerations that need to be made:

- The four floating-point multipliers of this architecture are special in the sense that they output two results. The difference between the two outputs is that the first outputs the result with rounding towards $-\infty$ and the second towards $+\infty$. At internal level, this translates to replacing the rounding logic for correct rounding with the logic for rounding towards $-\infty$ and $+\infty$. In terms of area, the new rounding logic has as size two times larger than the one which performes correct rounding.

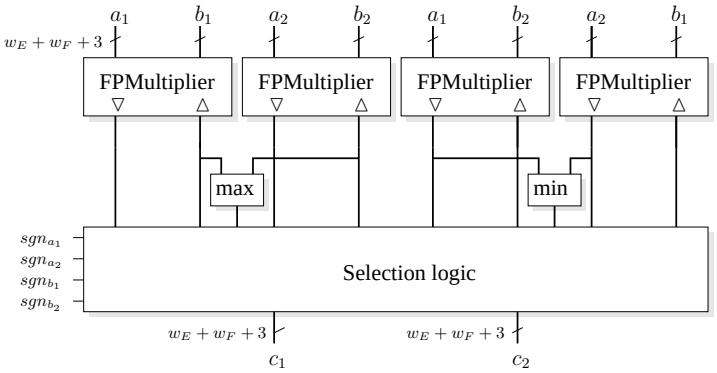


Figure 23: Infimum-Supremum multiplication architecture

- The circuitry which computes the max and the min should be implemented using the fast-carry logic present on current FPGAs. At the architectural level, this functions are translated as comparisons on integer numbers (the floating-point number is seen as an integer on $w_E + w_F + 1$ bits). This comparison is most of the time implemented combinationally while running at high frequencies. However, when this circuit becomes the bottleneck of the architecture in terms of latency, pipelining of this circuit is needed. This will consequently lead to delaying all the multiplier outputs so they become synchronized with the outputs from the min/max circuitry. The cost of this synchronization is high in the number of registers required.
- The selection logic is implemented with multiplexers and depends only on the signs of the interval ends of the inputs.

For intervals represented in midpoint-radius form, the result of multiplying the interval A with B is:

$$C =< c, \gamma > \text{ where } c = \circ(a \cdot b) \text{ and } \gamma =\triangle\ (\eta + \varepsilon' \cdot |c| + (|a| + \alpha)\beta + \alpha|\beta|)$$

In the radius computation of the result, η denotes the smallest representable positive floating point.

The are several considerations regarding the midpoint-radius multiplier architecture (figure 24):

- due to input constraints, all the floating-point adders of the architecture contain only the far path. Moreover, these adders have a rounding logic for rounding only towards $+\infty$.
- there is a number of three floating-point multipliers among which two use round towards $+\infty$ and one performes correct rounding.
- the exponent addition is a short operation which does not generally require pipelining. Therefore, the output of this operation has to be delayed with the equivalent of two floating point additions. This increases the number of registers used by the operator.
- the operator latency is high, with a latency given by the formula:

$$\begin{aligned} latency =& max(FPMultiplier_{\circ}, FPAdder_{\triangle}, FPMultiplier_{\triangle}) + \\ & max(FPMultiplier_{\triangle}, FPAdder_{\triangle}) + \\ & 2FPAdder_{\triangle} \end{aligned}$$

4.2.4 Division

For infimum-supremum representation of intervals, the result of the dividing A by B depends on the position of the interval ends with respect to the origin and whether or not the interval B contains the origin. For the case when $0 \notin B$:

	$b_1 \geq 0$	$b_2 < 0$
$a_1 \geq 0$	$[\nabla(a_1/b_2), \triangle\ (a_2/b_1)]$	$[\nabla(a_2/b_2), \triangle\ (a_1/b_1)]$
$a_1 < 0 \leq a_2$	$[\nabla(a_1/b_1), \triangle\ (a_2/b_1)]$	$[\nabla(a_2/b_2), \triangle\ (a_1/b_2)]$
$a_2 < 0$	$[\nabla(a_1/b_1), \triangle\ (a_2/b_2)]$	$[\nabla(a_2/b_1), \triangle\ (a_1/b_2)]$

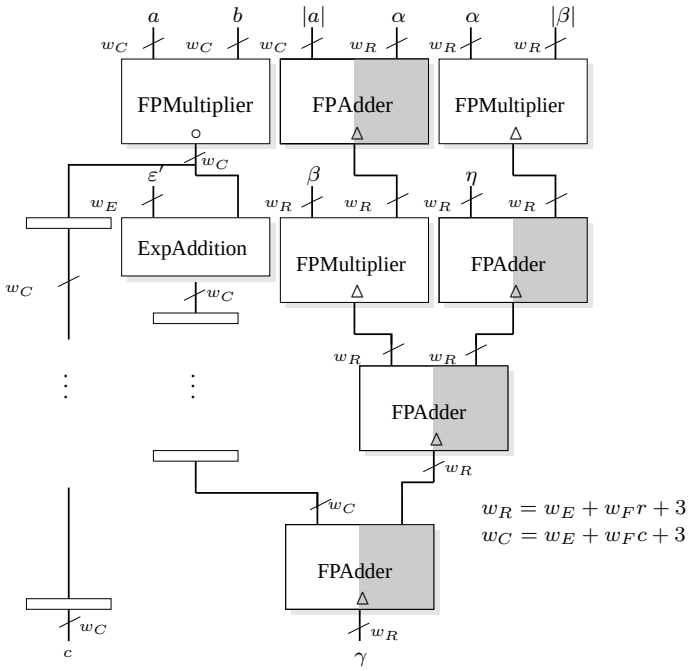


Figure 24: Midpoint-radius multiplication architecture

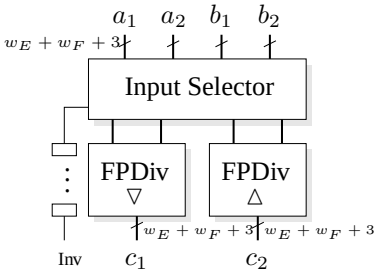


Figure 25: Infimum-supremum division architecture

In the case when $0 \in B$ we have:

	$b_1 = b_2 = 0$	$b_1 < b_2 = 0$	$b_1 < 0 < b_2$	$0 = b_1 < b_2$
$a_1 \geq 0$	$[+NaN, -NaN]$	$[\nabla(a_2/b_1), +\infty]$	$[\nabla(a_2/b_1), \triangle\,(a_2/b_2)]^*$	$[-\infty, \triangle\,(a_1/b_2)]$
$a_1 < 0 \leq a_2$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$
$a_2 < 0$	$[+NaN, -NaN]$	$[-\infty, \triangle\,(a_1/b_1)]$	$[\nabla(a_1/b_2), \triangle\,(a_1/b_1)]^*$	$[\nabla(a_2/b_1), +\infty]$

The architecture of the operator is presented in figure 25. It consists of two floating-point divisors preceded by combinational logic. The combinational logic selects the four inputs of the two divisors function of the exception bits and the signs of the inputs. In addition, an additional bit is provided to the output which selects between the interval type. A value of 1 for this bit suggests that the interval is of the form $[-\infty, c_1] \cup [c_2, +\infty]$ and a value of 0 suggests that the interval type is $[c_1, c_2]$. In terms of cost, the division operator for infimum-supremum interval representation requires two floating-point divisors.

In the case of the midpoint-radius interval representation, the division requires many more elementary operations. Firstly, in order to compute A/B one needs to compute the inverse of B and then multiply this by A. Formally, the division of A by B is written as:

$$C = A \times (1/B)$$

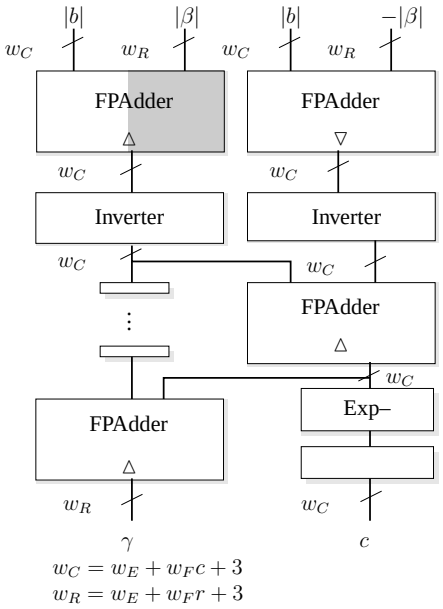


Figure 26: The inversion in midpoint-radius representation

The inverse of B is computed according to the formula given by Rump [17]. That is, $C = 1/B$ with:

$$
\begin{aligned}
c_1 &= \nabla(1/(|b| + \beta)) \\
c_2 &= \Delta\ (1/(|b| - \beta)) \\
c &= \nabla(c_1 + 0.5 \times (c_2 - c_1)) \\
\gamma &= \Delta\ (c - c_1) \\
c &= sign(B) \times c
\end{aligned}
$$

The architecture of the inversion operator is given in figure 26. It basically consists of four floating point adders, out of which one contains only the far path of the addition, and two inverters. In order to compare the architectures for the two representations, we can consider that the FPDiv components of figure 25 are in fact floating point inverters followed by floating point multipliers. In this case, the difference between the two architectures is determined only by the difference between the areas of the adders and multipliers. The importance of this difference decreases when we consider the fact that in the case of double-precision, the area of a FPMultiplier is 8 times smaller than the area of a FPDiv, and the area of FPAddition is 4 times smaller than the area of FPDiv (results obtained with Xilinx CoreGen).

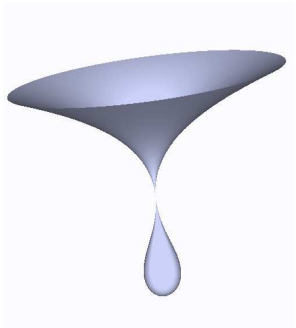
Consequently, the final architecture of the divisor requires one more multiplication of the inversion result (1/B) by A. This is done using a multiplier as presented in figure **??**.

The division algorithm proves more costly in the case of the midpoint-radius representation. However, the division generally proves costly for the FPGA, a double precision divisor, generated with CoreGen for a Virtex4vlx15 board occupies more than half of the slices of the target but is able to output a result at each 260MHz, with a latency of 57 clock periods.

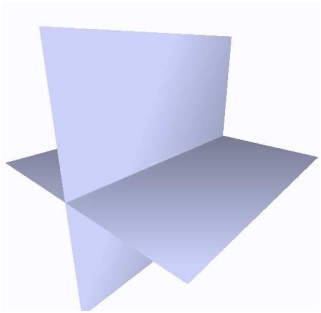
Now that we have designed all the interval arithmetic operator architectures, let us take an application which could benefit from the FPGA acceleration in computing interval arithmetic.

4.3 Ray tracing of implicit functions

In graphics, geometry is often modeled explicitly as a piecewise-linear mesh. However, one alternative is a higher-order analytical representation in implicit or parametric form. While implicit have not experienced as widespread adoption as parametric surfaces in 3D modeling, they are common in other fields, such as mathematics, physics and biology. Moreover, they serve as geometric primitives for isosurface visualization of point sets and volume data. An implicit



$0.5x^5 + 0.5x^4 - y^2 - z^2$



xy

Table 3: Possible implicit functions for ray tracing using interval arithmetic

surface S in 3D is defined as the set of solutions of an equation

$$f(x, y, z) = 0$$

where $f : \Omega \subseteq \mathbb{R}^3 \rightarrow \mathbb{R}$. For our purposes, assume this function is defined by any analytical expression. In ray tracing, we seek the intersection of a ray under the following form:

$$\vec{P}(t) = \vec{O} + t\vec{D}$$

with this surface S. By simple substitution of these position coordinates, we derive a unidimensional expression

$$f_t(t) = f(O_x + tD_x, O_y + tD_y, O_z + tD_z)$$

and solve where $f_t(t) = 0$ for the smallest $t > 0$.

Ray tracing is a global illumination based rendering method. It traces rays of light from the eye back through the image plane into the scene. Then the rays are tested against all objects in the scene to determine if they intersect any objects. If the ray misses all objects, then that pixel is shaded to the background color. Ray tracing handles shadows, multiple specular reflections, and texture mapping in a very easy straight-forward manner.

The basic idea behind ray tracing using interval arithmetic relies on the fundamental properly of inclusion of interval arithmetic. This property can be used in ray tracing for identifying and skipping empty regions of space. In the case of the 3D space, the inclusion property states that any function $f : \Omega \subseteq \mathbb{R}^3 \rightarrow \mathbb{R}$ (where Ω is an open subset of R^3) and a domain box $B = X \times Y \times Z \subseteq \Omega$ the corresponding interval extension $F : B \rightarrow F(B)$ is an inclusion function of f , in that

$$F(B) \subseteq f(B) = f(x, y, z)|(x, y, z) \in B$$

Evaluating F by interval arithmetic gives a simple and reliable rejection test for the box B not intersecting S. The overestimation introduced by interval arithmetic may cause multiple iterations to remove a box B, if the ray is not intersecting it.

Technical details of the algorithm are omitted here (see details in [18]). Two of the implicit functions which can be represented by using the interval arithmetic operators are presented in table 3. As it can be observed, the equations contain only shifts, power, addition and subtraction operators. This operations are fast on FPGA. Moreover, the application permits to experience with different precisions using both the infimum-supremum representation or the midpoint-radius representation. Furthermore, the trivially parallel description of the algorithm makes it suited for the acceleration using FPGAs. It might nevertheless be interesting to find the limits of the perceived quality for a certain function. This would offer a compromise between rendering quality and speed.

Additionally, this is an application for which the problem semantic is more suited for the midpoint-radius representation. It feels more comfortable to define a box in a 3D space by it's center and it's radius. Midpoint-radius representation often requires more computations due to the rounding errors appearing when computing the center. We can simplify the operator architectures by overestimating the result. Consequently, we can experience with reducing the area of the operators and thus overestimating more the result at the benefit of increased parallelism due to smaller operator size.

5 Conclusions and future work

A few months ago, FloPoCo consisted of a small framework with one operator (FPConstMultiplier). During the past months it has grown much faster than the initial roadmap forecasted. Although not all operators are finished, core operators like IntAdder, IntMultiplier, FPMultiplier are reaching maturity while others like FPAdder, Karatsuba multiplication, Leading Zero Counter, DotProduct still need some attention.

We have proved that the operators generated by FloPoCo surpass the generic ones present in generic floating-point libraries. We have also shown that FloPoCo can compete Xilinx CoreGen, a state of the art core generator for Xilinx FPGAs. Moreover, FloPoCo offers what no other products on the market do: different input/output precision for all floating-point operators together with frequency driven architecture generation. The latter is possible due to the innovative *automatic pipelining* feature present in FloPoCo.

The knowledge gained while implementing and integrating these operators in FloPoCo determined us to conduct a study on the current opportunities for interval arithmetic on FPGAs. We chose the two most common interval representations: infimum-supremum and midpoint-radius. We analyzed arithmetic operators in the context where, for each representation, the set of input/output intervals was a subset of $\mathbb{IF}$. We designed the architectures for each of the basic operations: addition, subtraction, multiplication and division. We provided a reality check on the implementation and architectural requirements of each operator. However, the proposed architectures remain to be implemented and tested in this context. Therefore, we have proposed a practical application for that. Ray tracing appears often in computer graphics for rendering complex scenes. It requires massive floating-point computations. The parallel nature of the problem makes it an ideal candidate for accelerating it using FPGAs. This is partially due to the fact that the *implicit functions* which are to be evaluated contain only basic operations which can be implemented at high frequencies within FPGAs.

Future work include completing this study with an application implementation. We will also work on other operators, especially in the dot product family.

References

[1] J. Detrey and F. Dinechin, "A tool for unbiased comparison between logarithmic and floating-point arithmetic," *J. VLSI Signal Process. Syst.*, vol. 49, no. 1, pp. 161–175, 2007.

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

[3] C. Bobda, *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*. Springer, 2007.

[4] N. S. Voros and K. Masselos, *System Level Design of Reconfigurable Systems-on-Chip*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[5] B. Fagin and C. Renard, "Field programmable gate arrays and floating point arithmetic," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, pp. 365–367, Sep 1994.

[6] N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic on FPGA based custom computing machine," in *FPGAs for Custom Computing Machines*, pp. 155–162, IEEE, 1995.

[7] 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.

[8] K. Underwood, "FPGAs vs. CPUs: Trends in peak floating-point performance," in *ACM/SIGDA Field-Programmable Gate Arrays*, ACM Press, 2004.

[9] 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*, vol. 2438 of *LNCS*, pp. 523–533, LNCS 2438, Sept. 2002.

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

[11] P. Belanović and M. Leeser, "A library of parameterized floating-point modules and their use," in *Field Programmable Logic and Applications*, vol. 2438 of *LNCS*, pp. 657–666, Springer, Sept. 2002.

[12] G. Govindu, R. Scrofano, and V. Prasanna, "A library of parameterizable floating-point cores for fpgas and their application to scientific computing," *International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2005.

[13] W. Ligon, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. Underwood, "A re-evaluation of the practicality of floating-point operations on FPGAs," in *IEEE Symposium on FPGAs for Custom Computing Machines*, (Napa Valley, USA), 1998.

[14] J. Detrey and F. de Dinechin, "A parameterized floating-point exponential function for FPGAs," in *IEEE International Conference on Field-Programmable Technology (FPT'05)* (G. Brebner, S. Chakraborty, and W.-F. Wong, eds.), (Singapore), pp. 27–34, IEEE, Dec. 2005.

[15] J. Detrey and F. de Dinechin, "A parameterizable floating-point logarithm operator for FPGAs," in *39th Asilomar Conference on Signals, Systems & Computers*, (Pacific Grove, CA, USA), pp. 1186–1190, IEEE Signal Processing Society, Nov. 2005.

[16] R. Kirchner, "Hardware support for interval arithmetic," *Reliable Computing*, vol. 12, pp. 225–237(13), June 2006.

[17] S. M. Rump, "Fast and parallel interval arithmetic," *BIT Numerical Mathematics*, vol. 39, no. 3, pp. 534–554, 1999.

[18] Knoll, "Interactive ray tracing of arbitrary implicits with simd interval arithmetic," in *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*.