Integer and Floating-Point Constant Multipliers for FPGAs

Nicolas Brisebarre, Florent de Dinechin, Jean-Michel Muller[*]
LIP (CNRS/INRIA/ENS-Lyon/UCBL)
Université de Lyon
{Nicolas.Brisebarre, Florent.de.Dinechin, Jean-Michel.Muller}@ens-lyon.fr

Abstract

Reconfigurable circuits now have a capacity that allows them to be used as floating-point accelerators. They offer massive parallelism, but also the opportunity to design optimised floating-point hardware operators not available in microprocessors. Multiplication by a constant is an important example of such an operator. This article presents an architecture generator for the correctly rounded multiplication of a floating-point number by a constant. This constant can be a floating-point value, but also an arbitrary irrational number. The multiplication of the significands is an instance of the well-studied problem of constant integer multiplication, for which improvement to existing algorithms are also proposed and evaluated.

1 Introduction

FPGAs (for field-programmable gate arrays) are high-density VLSI chips which can be programmed to efficiently emulate arbitrary logic circuits. Where a microprocessor is programmed at the granularity of instructions operating on 32 or 64-bit data words, FPGAs are programmed at the bit and register level. This finer grain comes at a cost: a circuit implemented in an FPGA is typically ten times slower than the same circuit implemented as an ASIC (application-specific integrated circuit). Despite this intrinsic performance gap between FPGAs and ASIC, the former are often used as a replacement of the latter for applications which don't justify the non-recurring costs of an ASIC, or which have to adapt to evolving standards.

FPGAs have also been used as configurable accelerators in computing systems. They typically excel in computations which exhibit massive parallelism and require operations absent from the processor's instruction set.

The FloPoCo project[1] is an open-source C++ framework for the implementation of such non-standard operations, with a focus on floating-point [4]. This article is a survey of the issue of multiplication by a constant in this context.

State of the art and contributions

Multiplication by a constant is a pervasive operation. It often occurs in scientific computing codes, and is at the core of many signal-processing filters. It is also useful to build larger operators: previously published architectures for exponential, logarithm and trigonometric functions [8, 7] all involve multiplication by a constant. A single unoptimised multiplication by $4/\pi$ may account for about one third the area of a dual sine/cosine operator [7].

The present article essentially reconciles two research directions that were so far treated separately: on the one side, the optimisation of multiplication by an integer constant, addressed in Section 2, and on the other side the issue of correct rounding of multiplication or division by an arbitrary precision constant, addressed in Section 4.

Integer constant multiplication has been well studied, with many good heuristics published [3, 6, 13, 5, 1, 15]. Its theoretical complexity is still an open question: it was only recently proven sub-linear, although using an approach which is useless in practice [9, 15]. Our contribution in this domain is essentially a refinement of the objective function: where all previous works to our knowledge try to minimise the number of additions, we remark that these additions, measured in terms of *full adder* cells, have different sizes (up to a factor 4 for the large multiplier by $4/\pi$ of [7]), hence variable cost in reconfigurable logic. Trying to minimise the number of full adders, and looking for low-latency and easy to pipeline architectures, we suggest a surprisingly simple algorithm that, for constants up to 64 bits, outperforms the best known algorithms in terms of FPGA area usage and latency. Boullis and Tisserand [1] also tried to minimise adder size, but as a post-processing step, after an algorithm minimising the number of additions.

---

[1]www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo

Section 3 describes a multiplier by a floating-point constant of arbitrary size. The architecture is a straightforward specialisation of the usual floating-point multiplier. It is actually slightly simpler, because the normalisation of the result can be removed from the critical path.

Finally, Section 4 deals with the correct rounding of the multiplication by an arbitrary real constant. Previous work on the subject [2] has shown that this correct rounding requires a floating-point approximation of the constant whose typical size is twice the mantissa size of the input. This size actually depends on the real constant, and may be computed using a simple continued fractions algorithm. The other contribution of [2] is the proof of an algorithm which consists of two dependent fused-multiply-and-add operations. In the FPGA, the implementation will be much simpler, since it will suffice to instantiate a large enough FP constant multiplier. Of course, a multiplier by an arbitrary constant is also capable of computing the division by an arbitrary constant [14].

All these architectures are implemented in the FloPoCo framework.

2 Multiplication by an integer constant

Several recent papers [1, 9, 15] will provide the interested reader with a state of the art on this subject. Specific to FPGAs, the KCM algorithm [3] is also of interest, but it has been shown to always lead to larger architectures [5].

Let C be a positive integer constant, written in binary on k bits:

$$C = \sum_{i=0}^{k} c_i 2^i \quad \text{with } c_i \in \{0, 1\}.$$

Let X a p-bit integer. The product is written $CX = \sum_{i=0}^{k} 2^i c_i X$, and by only considering the non-zero c_i, it is expressed as a sum of $2^i X$. For instance, $17X = X + 2^4 X$. In the following, we will note this using the shift operator $\ll$, which has higher priority than $+$ and $-$. For instance $17X = X + X \ll 4$.

If we allow the digits of the constant to be negative ($c_i \in \{-1, 0, 1\}$) we obtain a redundant representation, for instance $15 = 01111 = 1000\bar{1}$ ($16 - 1$ written in signed binary). Among the representations of a given constant C, we may pick up one that minimises the number of non-zero bits, hence of additions/subtractions. The well-known *canonical signed digits* recoding (or CSD, also called Booth recoding [10]) guarantees that at most $k/2$ bits are non-zero, and in average $k/3$.

2.1 Parenthesing and architectures

The CSD recoding of a constant may be translated into a rectangular architecture [5], an example of which is given

by Figure 1. This architecture corresponds to the following parenthesing: $221X = X \ll 8 + (-X \ll 5 + (-X \ll 2 + X))$.
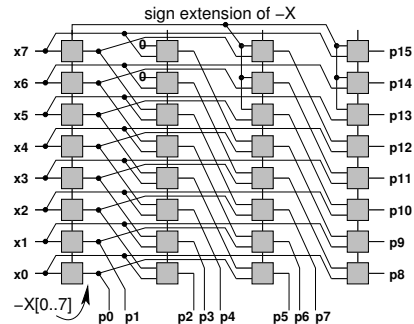

Figure 1. Linear multiplier of an 8-bit input by 221, using the recoding $100\bar{1}00\bar{1}01$

We introduce in this article a binary tree adder structure, constructed out of the CSD recoding of the constant as follows: non-zero bits are first grouped by 2, then by 4, etc. For instance, $221X = (X \ll 8 - X \ll 5) + (-X \ll 2 + X)$. A larger example is shown on Figure 2. This new parenthesing reduces the critical path: for k non-zero bits, it is now of $\lceil \log_2 k \rceil$ additions instead of k in the linear architecture of Figure 1.

Besides, shifts may also be reparenthesised: $221X = (X \ll 3 - X) \ll 5 + (-X \ll 2 + X)$. After doing this, the leaves of the tree are now multiplications by small constants: $3X, 5X, 7X, 9X...$ Such a smaller multiple will appear many times in a larger constant, but it may be computed only once: thus the tree is now a DAG (direct acyclic graph), and the number of additions is reduced.

Going from a tree to a DAG saves adders. Lefèvre [13] has generalised this idea to an algorithm that minimises the number of adders: it looks for maximal repeating bit patterns in the CSD representation, and generates them recursively. Lefèvre observed that the number of additions, on randomly generated constants of k bits, grows as
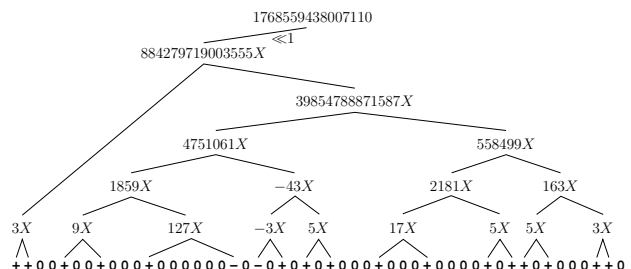

Figure 2. Binary DAG architecture for a multiplication by 1768559438007110 (the 50 first bits of the mantissa of π**).**

$O(k^{0.85})$. Here is an example of the sequence produced for the same constant 1768559438007110. This example was obtained thanks to the program `rigo.c`[2] written by Rigo and Lefèvre.

1:	u0 = x		9:	u1 = u1 + u3
2:	u3 = u0≪19 + u0		10:	u7 = u0≪21 - u0
3:	u3 = u3≪20		11:	u6 = u7≪18 + u0
4:	u3 = u3≪4 + u3		12:	u5 = u6≪4 - u0
5:	u7 = u0≪14 - u0		13:	u2 = u5≪5 + u0
6:	u6 = u7≪6 + u0		14:	u2 = u2≪1
7:	u5 = u6≪10 + u0		15:	u2 = u2≪2 - u2
8:	u1 = u5≪16		16:	u1 = u1 + u2

This code translates to a much more compact DAG than the one presented on Figure 2, because it looks for patterns in the full constant instead of just exploiting them when they accidentally appear, mostly at the leaves in our binary tree.

Still, Lefèvre's method may produce suboptimal results in our context. Firstly, it doesn't try to balance the DAG to minimise the latency. Secondly, it only minimises the number of additions, but not their actual hardware cost, which depends on their size. Let us now formalise this last issue.

2.2 DAG definition and cost analysis

Each intermediate variable in a DAG holds the result of the multiplication of X by an intermediate constant. To make things clearer, let us name an intermediate variable after this constant, for instance, V_{255} holds $255X$, and $V_1 = X$.

In the Rigo/Lefèvre code, each of these intermediate multiples is positive, and subtraction is allowed. Cost analysis is slightly simpler if we allow negative intermediate constants, but no subtraction. We then need unary negation to build negative constants. To minimise the use of negation, which has the same cost as addition on an FPGA, one may always transform a DAG into one with only one negation computing $V_{-1} = -X$.

To sum up, a DAG is built out of the following primitives:

$$
\begin{array}{lll}
\text{Shift:} & V_z \leftarrow V_i \ll s & (z = 2^s i), \\
\text{Neg:} & V_z \leftarrow -V_i & (z = -i), \\
\text{ShiftAdd:} & V_z \leftarrow V_i \ll s + V_j & (z = 2^s i + j).
\end{array}
$$

Each variable is a single assignment one, and it is possible to associate to it

- $|V_z|$, the maximal size in bits of the result it holds,

- $\text{cost}(V_z)$, the number of *full adder* involved.

[2]http://www.vinc17.org/research/mulbyconst/

Other cost functions are possible (e.g. delay). A DAG construction algorithm will maintain a list of the already computed variables, indexed by the constants.

The size $|V_z|$ is more or less the sum of the size of z and the size of X. If $z \geq 0$ then $|V_z| = |X| + \lfloor \log_2(z - 1) \rfloor$, where the -1 accounts for powers of 2. If $z < 0$ then $|z| = 1 + |X| + \lfloor \log_2(-z - 1) \rfloor$: one has to budget an additional sign bit for sign extension. This bit will actually be useful only for multiplying by $X = 0$, whose multiplication by a negative constant is nevertheless nonnegative. This detail is worth mentioning as it illustrates the asymmetry between negative constants and positive ones.

Computing the costs is easy once the $|V_z|$ have been computed:

- $\text{cost}(V_z \leftarrow V_i \ll s) = 0$. This is wiring only.

- $\text{cost}(V_z \leftarrow -V_i) = |V_z|$. Again, it is probably best to use this primitive only to compute $V_{-1} = -X$.

- $\text{cost}(V_z \leftarrow V_i \ll s + V_j) = |V_z| - s$. The lower bits of the result are those of V_j, so the actual addition is on $|V_z| - s$ bits only[3].

This cost function describes relatively acurately the cost of a combinatorial constant multiplier. It has to be extended to the case of pipelined multipliers: one has to add the overhead of the registers, essentially for the lower bits since a registered adder has the same cost as a combinatorial one in FPGAs. In principle, one pipeline stage may contain several DAG levels, at least for the lower levels.

2.3 Implementation and results

FloPoCo implements this DAG structure and cost analysis. It outputs VHDL for any DAG, but currently builds only the simple DAGs illustrated by Figure 1 and Figure 2, both in time linear with k (instantly in practice). Interested readers are invited to try it out.

Synthesis results are given in Table 1. These are FP multipliers, but their area and delay are largely dominated by the significand multiplication. For comparison, two sequences produced by `rigo.c`, for the significands of $\pi/2 \times 2^{50}$ and $\pi/2 \times 2^{107}$, were hand-translated into FloPoCo DAGs. For the 50-bit constant, although the number of additions is smaller, the final area is larger, as many of these additions are very large. This justifies the introduction

[3]There is one exception: if V_i and V_j do not overlap, *i.e.* if $|V_j| < s$, then the addition is free if j is positive: the higher bits are those of V_i and the lower bits of V_j. If j is negative, one needs to sign-extend V_j, and the cost is again $|V_z| - s$. This situation may only happen if the size of the constant is at least twice that of X, which indeed happens in several applications, for example the high-precision polynomial evaluation [12] that motivated Lefèvre, and the trigonometric argument reduction of [7].

of a new cost function, and illustrates that our binary DAG approach provides very good implementations for constants up to 64 bits. This should content a vast majority of applications.

Still, for the 107-bit constant, the final area is smaller, which tends to show that the asymptotic cost of Lefèvre's approach is better than that of our binary DAG, which remain mostly linear. More work is needed to adapt Lefèvre's approach to our cost function.

3 Multiplication by a floating-point constant

For the needs of this article, an FP number is written $(-1)^s \cdot 2^E \cdot 1.F$ where $1.F \in [1, 2)$ is a significand and E is a signed exponent. We shall note w_E and w_F the respective sizes of E and F, and $\mathbb{F}(w_E, w_F)$ the set of FP numbers in a format defined by (w_E, w_F). We want to allow for different values of w_E and w_F for the input X and the output R:

$$
\begin{aligned}
X &= (-1)^{s_X} \cdot 2^{E_X} \cdot 1.F_X &&\in \mathbb{F}(w_{E_X}, w_{F_X}) \\
R &= (-1)^{s_R} \cdot 2^{E_R} \cdot 1.F_R &&\in \mathbb{F}(w_{E_R}, w_{F_R})
\end{aligned}
$$

In all the following, the real value of the constant will be noted C, possibly an irrational number, and we define

$$
C = (-1)^{s_C} \cdot 2^{E_C} \cdot 1.F_C
$$

the unique floating-point[4] representation of C such that $1.F_C \in [1, 2)$. Here F_C may have an infinite binary representation. We note C_k the approximation of C rounded to the nearest on $w_{F_C} = k$ fraction bits:

$$
C_k = (-1)^{s_C} \cdot 2^{E_C} \cdot 1.F_{C_k} \quad .
$$

Finally, we also define the real number

$$
1.F_{\text{cut}} = \frac{2}{1.F_C} \quad \in [1, 2) \quad .
$$

We now describe a multiplier that computes the correct rounding R_k of $C_k \times X$. Then, Section 4 will compute the minimal k ensuring that $\forall X \in \mathbb{F}(w_{E_X}, w_{F_X})$, R_k is the correct rounding of $C \times X$.

Of course, if C is already a p-bit-significand FP number, it will be $k = p$.

The architecture given by Figure 3, and implemented as the FPConstMult class in FloPoCo, is essentially a simplification of the standard FP multiplier. The main modification is that rounding is simpler. In the standard multiplier, the product of two significands, each in $[1, 2)$, belongs to $[1, 4)$. Its normalisation and rounding is decided by looking at the product. In a constant multiplier, it is possible to predict if

---

[4]As the exponent is constant, the point doesn't actually float at all.

the result will be larger or smaller than 2 just by comparing F_X with F_{cut} – in practice, with F_{cut} truncated to w_{F_X} bits. This is also slightly faster, as the rounding decision is moved off the critical path.

Exponent computation consists in adding the constant exponent, possibly augmented by 1 if $F_X > F_{cut}$. Sign computation is also straightforward. Exceptional case handling is also slighly simpler. For instance, if the constant has a negative exponent, one knows that an overflow will never occur. Likewise, if it is positive or zero, underflow (flush to zero) cannot happen.
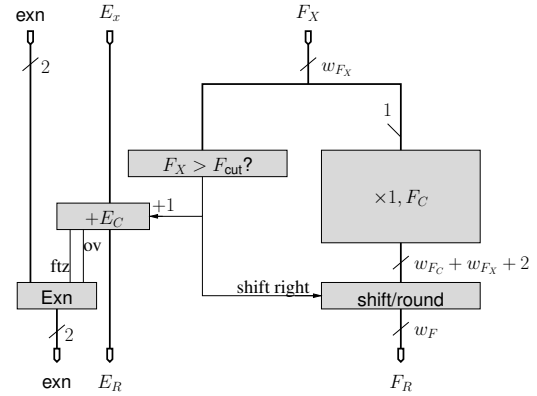

Figure 3. Multiplier by an FP constant

4 Correct rounding of the multiplication by a real constant

This section proposes a method for computing the minimal value of $k = w_{F_C}$ allowing for correct rounding (noted $\circ$) of the product of any input X by C. First, for a given k, we show how to build a predicate telling if there exist values of X such that $R_k = \circ(C_k X) \neq \circ(CX)$. This allows us to look for the minimal k verifying this predicate, knowing that it is expected to be close to $2w_{F_X}$ [2].

4.1 Looking for X such that $\circ(C_k X) \neq \circ(CX)$

The FP multiplier guarantees the correct rounding of the result of the multiplication by C_k, that is to say,

$$
\begin{aligned}
\forall X \in \mathbb{F}(w_{E_X}, w_{F_X}), \quad &|R_k - C_k X| \\
&\leq \frac{1}{2}\,\text{ulp}\,(C_k X) \leq \frac{1}{2}\,\text{ulp}\,(R_k),
\end{aligned}
$$

in which "ulp(t)" (*unit in the last place*) is the weight of the least significant bit of t.

	FloPoCo linear (Fig. 1)			FloPoCo binary DAG (Fig. 2)			Lefèvre/Rigo		
Precision of X and $\pi/2$	+	LUTs	delay	+	LUTs	delay	+	LUTs	delay
$w_{E_X} = 8, w_{F_X} = w_{F_R} = 23, w_{F_C} = 50$	19	435	30 ns	15	467	14 ns	12	645	16 ns
$w_{E_X} = 11, w_{F_X} = w_{F_R} = 52, w_{F_C} = 107$	38	2018	68 ns	26	1628	21 ns	22	1508	18 ns

Table 1. Synthesis results for floating-point multipliers (on Virtex4, speedgrade -12, using ISE 9.1)
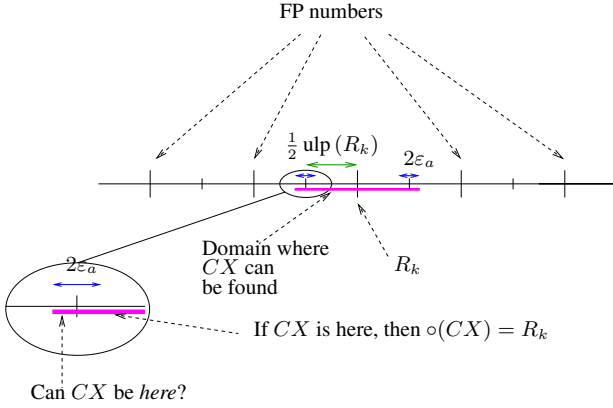

Figure 4. If CX is at a distance greater than 1/2 ulp of R_k, then it is at a distance lesser than $2\varepsilon_a$ from the middle of two consecutive FP numbers.

Moreover, C_k is also the rounded-to-nearest value of C. Let $\varepsilon_a = |C_k - C|$, we have

$$\forall X \in \mathbb{F}(w_{E_X}, w_{F_X}), \quad |R_k - CX| \leq \frac{1}{2} \text{ulp}(R_k) + X \cdot \varepsilon_a.$$

We may assume, without any loss of generality, that X and C belong to $[1, 2)$, i.e. $E_X = E_C = 0$. Then we have

$$\forall X \in \mathbb{F}(w_{E_X}, w_{F_X}), 1 \leq X < 2,$$
$$|R_k - CX| < \frac{1}{2} \text{ulp}(R_k) + 2\varepsilon_a \quad (1)$$

If we can prove that for all X, $|R_k - CX| \leq \frac{1}{2} \text{ulp}(R_k)$, then R_k will always be the closest FP number to CX, which is the required property. As shown in Figure 4.1, if CX satisfies to (1) and is at a distance greater than $\frac{1}{2} \text{ulp}(R_k)$ from R_k, it is necessarily at a distance lesser than $2\varepsilon_a$ from the middle of two consecutive FP numbers. Such a point is a rational number of the form $(2A + 1)/(2q)$, with $2^{w_{F_R}} \leq A \leq 2^{w_{F_R}+1} - 1$ and $q = 2^{w_{F_R}+t}$, where t is equal to 1 if CX has the same exponent as X (if $F_X \leq F_{cut}$), and is equal to 0 otherwise.

Therefore, to determine if an input X is such that R_k is not the correct rounding of CX, one can check first if there

exists an approximation to CX by a rational number of the form $(2A+1)/(2q)$, such that $|CX-(2A+1)/(2q)| \leq 2\varepsilon_a$.

The mathematical tool for solving this kind of rational approximation issues is continued fractions [11]. Using them, one can design several methods [2] that make it possible either to guarantee that CX will not be at a distance lesser than $2\varepsilon_a$ from the middle of two consecutive FP numbers (hence one can guarantee that the correct rounding of CX is always returned) or to compute all counter-examples, that is to say values of X such that $C_k X$ rounded to nearest is not the correct rounding of CX. In the latter case, one can derive from each counter-example the value by which we should increment k in order to get a correct rounding.

4.2 A predicate for k

We assume in the sequel that $F_X < F_{cut}$ (the case $F_X > F_{cut}$ is similar). We then have $CX \in [1, 2)$. Let M_X be the integer mantissa of X, i.e. $M_X = 2^{w_{F_X}} X$. We search for the integers $M_X \in \mathbb{Z}$ such that

$$\left| \frac{M_X}{2^{w_{F_X}}} C - \frac{2A+1}{2^{w_{F_R}+1}} \right| \leq 2\varepsilon_a.$$

Depending on the relative values of w_{F_R} and w_{F_X}, we face two situations:

4.2.1 Case where $w_{F_R} + 1 \geq w_{F_X}$

We assume in this case that $k = w_{F_R} + w_{F_X} + 3$. We search for the integers $M_X \in \mathbb{Z}$ such that

$$\left| M_X 2^{w_{F_R}-w_{F_X}+1} C - 2A - 1 \right| \leq 2^{2+w_{F_R}} \varepsilon_a.$$

Since $\varepsilon_a = |C_k - C| \leq 2^{-k-1}$, we have

$$\left| M_X 2^{w_{F_R}-w_{F_X}+1} C - 2A - 1 \right| \leq 2^{w_{F_R}-k+1}.$$

Note that $2^{w_{F_R}-k+1} < 1/(2M_X)$ iff $2^{w_{F_R}-k+2}M_X < 1$. As $M_X < 2^{w_{F_X}+1}$ and $2^{w_{F_R}+w_{F_X}-k+3} \leq 1$ since we assumed $w_{F_R} + w_{F_X} + 3 = k$, we have $2^{w_{F_R}-k+1} < 1/(2M_X)$ for all $X \in [1, 2)$: we compute the continued fraction expansion of $2^{w_{F_R}-w_{F_X}+1}C$ that yields all the candidate values X that may possibly satisfy $\circ(CX) \neq \circ(C_k X)$. For all such input X, we first check exhaustively if those rounded values actually differ and we collect all such X in a list L. Then, we compute the minimal value

η of $\left| \frac{M_X}{2^{w_{F_X}}} C - \frac{2A+1}{2^{w_{F_R}+1}} \right|$ when X ranges the list L of all counter-examples and we set $k = \max(w_{F_R} + w_{F_X} + 3, \lceil -\log_2(\eta) \rceil + 1)$. The inequality $k \geq \lceil -\log_2(\eta) \rceil + 1$ implies $k > -\log_2(\eta)$ that yields $2\varepsilon_a \leq 2^{-k} < \eta$, which guarantees that all inputs X will satisfy $\circ(CX) = \circ(C_k X)$.

4.2.2   Case where $w_{F_R} + 2 \leq w_{F_X}$

We assume in this case that $k = 2w_{F_X} + 2$. We search for the integers $M_X \in \mathbb{Z}$ such that

$$\left| M_X C - (2A+1)2^{w_{F_X} - w_{F_R} - 1} \right| \leq 2^{1+w_{F_X}} \varepsilon_a.$$

Here, again, we use $\varepsilon_a = |C_k - C| \leq 2^{-k-1}$ to infer

$$\left| M_X C - (2A+1)2^{w_{F_X} - w_{F_R} - 1} \right| \leq 2^{w_{F_X} - k}.$$

Here again, from the hypothesis $2w_{F_X} + 2 \leq k$, we infer $2^{w_{F_X} - k} < 1/(2M_X)$: the computation of the continued fraction expansion of C provides a complete list of values X candidate for satisfying $\circ(CX) \neq \circ(C_k X)$. We check exhaustively if those rounded values actually differ and we collect again all such X in a list L. Let η be the minimal value of $\left| \frac{M_X}{2^{w_{F_X}}} C - \frac{2A+1}{2^{w_{F_R}+1}} \right|$ when X ranges the list L of all counter-examples. We set $k = \max(w_{F_R} + w_{F_X} + 3, \lceil -\log_2(\eta) \rceil + 1)$. That value of k will ensure that $\circ(CX) = \circ(C_k X)$ for all input X.

4.3   The price of correct rounding

To sum up, the price of correct rounding, for a multiplication by an irrational constant like π or $\log 2$, will be a typical doubling of the number of bits of the constant used in significand multiplication. As the cost of such a multiplication is sublinear in the constant size [9], the price of correct rounding should actually be less than this factor 2 in area. The delay overhead will be much smaller, due to the binary tree architecture. This is confirmed by the following table, obtained using the binary DAG approach:

mult. by $\pi/2$, $w_{F_X} = 52$	$+$	LUTs	delay
standard ($w_{F_C} = 52$)	16	866	20 ns
correct rounding ($w_{F_C} = 107$)	26	1628	21 ns

5   Conclusion and perspectives

One may argue that multiplication by a constant is too anecdotical to justify so much effort. Yet it illustrates what we believe is the future of floating-point on FPGAs: thanks to their flexibility, they may accomodate non-standard optimised operators, for example a correctly rounded multiplication by an irrational constant. Such non-standard operators cannot be offered as off-the-shelf libraries, they have to be optimised for each application-specific context. This is the object of the FloPoCo project, an open C++ framework for arithmetic operator generation. With this work, FloPoCo builds multipliers by a constant which are both small and fast for constants of usual sizes. However our results suggest that there is still room for improvement. To this purpose, the present article defines a pertinent design space and offers an open implementation of VHDL generation for constant multiplier DAGs. Current work also focusses on extending this framework to automatically pipeline the generated operators.

References

[1]   N. Boullis and A. Tisserand. Some optimizations of hardware multiplication by constant matrices. *IEEE Transactions on Computers*, 54(10):1271–1282, 2005.

[2]   N. Brisebarre and J.-M. Muller. Correctly rounded multiplication by arbitrary precision constants. *IEEE Transactions on Computers*, 57(2):165–174, 2008.

[3]   K. Chapman. Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner). *EDN magazine*, May 1994.

[4]   F. de Dinechin, J. Detrey, I. Trestian, O. Creţ, and R. Tudoran. When FPGAs are better at floating-point than microprocessors. Technical Report ensl-00174627, École Normale Supérieure de Lyon, 2007. http://prunel.ccsd.cnrs.fr/ensl-00174627.

[5]   F. de Dinechin and V. Lefèvre. Constant multipliers for FPGAs. In *Parallel and Distributed Processing Techniques and Applications*, pages 167–173, 2000.

[6]   A. Dempster and M. Macleod. Constant integer multiplication using minimum adders. *Circuits, Devices and Systems, IEE Proceedings*, 141(5):407–413, 1994.

[7]   J. Detrey and F. de Dinechin. Floating-point trigonometric functions for FPGAs. In *Field-Programmable Logic and Applications*, pages 29–34. IEEE, 2007.

[8]   J. Detrey, F. de Dinechin, and X. Pujol. Return of the hardware floating-point elementary function. In *18th Symposium on Computer Arithmetic*, pages 161–168. IEEE, 2007.

[9]   V. Dimitrov, L. Imbert, and A. Zakaluzny. Multiplication by a constant is sublinear. In *18th Symposium on Computer Arithmetic*, pages 261–268. IEEE, 2007.

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

[11]   A. Y. Khinchin. *Continued Fractions*. Dover, 1997.

[12]   V. Lefèvre. An algorithm that computes a lower bound on the distance between a segment and $\mathbb{Z}^2$. In *Developments in Reliable Computing*, pages 203–212. Kluwer, 1999.

[13]   V. Lefèvre. Multiplication by an integer constant. Technical Report RR1999-06, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 1999.

[14]   J.-M. Muller, A. Tisserand, B. D. de Dinechin, and C. Monat. Division by constant for the ST100 DSP microprocessor. In *17th Symposium on Computer Arithmetic*, pages 124–130. IEEE Computer Society, 2005.

[15]   Y. Voronenko and M. Püschel. Multiplierless multiple constant multiplication. *ACM Trans. Algorithms*, 3(2), 2007.