Multiplication by rational constants

LIP research report 2011-3

Florent de Dinechin
LIP, Université de Lyon (ENS-Lyon/CNRS/INRIA/UCBL)
46, allée d'Italie, 69364 Lyon Cedex 07
Florent.de.Dinechin@ens-lyon.fr

***Abstract*—Multiplications by simple rational constants often appear in fixed-point or floating-point application code, for instance in the form of division by an integer constant. The hardware implementation of such operations is of practical interest to FPGA-accelerated computing. It is well known that the binary representation of rational constants is eventually periodic. This article shows how this feature can be exploited to implement multiplication by a rational constant in a number of additions that is logarithmic in the precision. An open-source implementation of these techniques is provided, and is shown to be practically relevant for constants with small numerators and denominators, where it provides improvements of 20 to 40% in area with respect to the state of the art. It is also shown that for such constants, the additional cost for a correctly rounded result is very small, and that correct rounding very often comes for free in practice.**

***Index Terms*—multiplication by a constant, rational number, floating-point, reconfigurable computing**

I. Introduction

Multiplication by constants has received much attention in the literature, especially as many digital signal processing algorithms can be expressed as products by constant matrices. In such cases the constants are typically irrational (e.g. the square roots of unity in a Fast Fourier Transform).

The context of the present work is quite different. We are interested in porting applications into optimized architectures for FPGAs, and in this process we want to optimize the arithmetic operators when possible. Such applications often involve multiplications by rational constants, and this article studies how to implement them as efficiently as possible. Our initial motivation was floating-point divisions by 3 and by 9 appearing in stencil applications, but this work covers multiplications by arbitrary rational constants a/b. However, we should already point out that, at least in the hardware context, this study is relevant mostly for small values of a and b, "small" meaning here that they fit on few bits.

There are three contributions in this article. We first present an algorithm that builds multipliers by rational constants based on their periodic representation. The study of periodic representations has very old decimal roots [1], and we extend to arbitrary rational constants early works related to division by a constant, mostly in a software context [2], [3]. The interested reader will find a unifying survey on constant division in [4], including several other references specific to division by 10 needed for binary/decimal conversions. The generalization to arbitrary rational constants and the application to hardware is a systematic exploitation of an empirical observation made by Gustafsson and Qureshi in [5]. The multipliers we describe use a number of additions that is asymptotically logarithmic in the precision of the constant. This logarithmic complexity (which is optimal [6]) for a large class of constants is interesting in its own right, in a field where little is known in terms of theoretical complexity [7]. In practice, however, exploiting periodicity is only useful when the period is short. We show in Section II how to compute this period for an arbitrary rational, and in Section III how to construct optimal adder trees out of it. Here we build upon a wide body of literature on constant multiplication, minimizing the number of additions [8], [9], [10], [11], [12], and, for hardware, also minimizing the total size of these adders [9], [10], [13], [14].

The second contribution is to show in Section IV that correct rounding of the result (that is, obtaining the correctly rounded product of X by a/b, and not of X by some finite fixed-point or floating-point approximation to a/b) comes at very little overhead for our "small" rationals: it requires to consider only a few more bits of the binary representation of the constant. Actually, these bits often come for free in the architectures built in Section III.

The third contribution, in Section V, concerns constant multiplication algorithms based on look-up tables (LUTs), developed for FPGAs by Chapman [15] then Wirthlin [16]. We show how these algorithms also benefit from the periodicity of the constant.

The shift-and-add technique has been implemented as the FPConstMultRational[1] operator in the open-source FloPoCo arithmetic core generator [17], while the LUT-based technique is implemented in the FixRealKCM operator. Section VI provides some data for correctly rounded multipliers by 1/3, 1/9 and 7/5, comparing shift-and-add trees to LUT-based multipliers, and showing area savings up to 40% with respect to the previous state of the art.

II. On the periodicity of the binary representation of rational numbers

The most usual system of representing numbers is the position system, where a number is represented by a sequence of digits,

[1]This operator is included in the FloPoCo distribution, release 2.3 or later, available from http://flopoco.gforge.inria.fr/.

and each digit is weighted by powers of some radix, usually 10 (decimal system) or 2 (binary system).
In such a system, any rational number a/b has an eventually periodic representation. This is true in decimal ($1/3 = 0.3333\cdots$, $1/9 = 0.11111\cdots$) but also in binary ($1/3 = 0.01010101_2\cdots$, $1/9 = 0.000111000111000_2\cdots$). Numbers with an finite decimal representation can be viewed as as special case where the periodic pattern is composed of zeroes, for instance $0.5 = 0.50000\cdots$. A number with a finite decimal representation may have an infinite binary one, for instance $1/5 = 0.2_{10} = 0.001100110011\cdots$. The opposite is not true, due to the fact that two divides ten.
The following lemma tells us which numbers have a purely periodic binary representation:

Lemma 1. *Let us consider an irreducible fraction c/d, where 2 divides neither c nor d. If $c < d$, then the binary representation of c/d is purely periodic, i.e. it starts with an occurence of the periodic pattern.*

The condition that 2 divides neither c nor d is not a constraint, since powers of two correspond to shifts in binary. For most purposes, they may be handled separately in a trivial way.
If $c > d$ the Euclidean division of c by d gives us $c = hd + c'$, and we may rewrite $c/d = h + c'/d$. For the purpose of multiplying an input x by the constant c/d, we therefore have $cx/d = hx + c'x/d$. Existing literature addresses the multiplication by the finite integer constant h, so we may focus on the multiplication by the purely periodic constant c'/d.
The following lemma allows us to compute the periodic pattern:

Lemma 2. *Let c/d be an irreducible fraction, where $c < d$, and 2 divides neither c nor d. The size s of its period is the multiplicative order of 2 modulo d,* i.e. *the smallest integer such that $2^s \mod d = 1$. The periodic pattern is the integer $p = \lfloor 2^s c/d \rfloor$.*

Proof: By definition of s we have $2^s = kd + 1$ for some integer k. Therefore $p = \lfloor 2^s c/d \rfloor = \left\lfloor \frac{(kd+1)c}{d} \right\rfloor = \lfloor kc + c/d \rfloor = kc$ since $c/d < 1$. We deduce that

$$2^s c/d = p + c/d,$$

where the recursive occurence of c/d exactly expresses the periodicity of the fraction c/d. ∎

Examples:

- $1/3$ has period size $s = 2$ because $2^2 \mod 3 = 1$. The pattern is $\lfloor 1 \times 2^2/3 \rfloor = 1$, which we write on 2 bits 01, and we obtain that
$$1/3 = 0.(01_2)^\infty \ ;$$
- $5/9$ has period size $s = 6$ because $2^6 \mod 9 = 1$. The pattern is $\lfloor 5 \times 2^6/9 \rfloor = 35$, which we write on 6 bits 100011, and we obtain that
$$5/9 = 0.(100011_2)^\infty.$$

Thanks to these lemmas, algorithm 1 determines the periodic binary representation of a fractional number a/b. This representation consists of 4 integers:

- s the period size in bits, a positive integer,
- p the periodic pattern, a positive integer that we will usually write in binary,
- h the header, a positive integer, also typically written in binary,
- e the scaling factor exponent, or shift.

such as

$$a/b = 2^e \left(h + \sum_{i=1}^{+\infty} \frac{p}{2^{si}} \right).$$

Algorithm 1 Computing the periodic representation of a rational a/b as a tuple of integers (e, h, p, s).

```
1:  procedure PeriodicRepresentation(a, b)
2:      (c, d) ← Simplify(a, b)
3:      e ← 0
4:      while c mod 2 = 0 do
5:          c ← c/2
6:          e ← e + 1
7:      end while
8:      while d mod 2 = 0 do
9:          d ← d/2
10:         e ← e − 1
11:     end while
12:     (Now a/b = 2^e c/d with both c and d odd)
13:     h ← c/d                                   (header)
14:     c ← c mod d                               (periodic part)
15:
16:     (Now a/b = h + c/d with c < d)
17:
18:     s ← 1                                     (period size)
19:     if d = 1 then        (a/b has a finite binary representation)
20:         p ← 0                                 (periodic pattern)
21:     else                 (a/b has an infinite binary representation)
22:         t ← 2            (Invariant of the loop below: t = 2^s)
23:         while t mod d ≠ 1 do
24:             s ← s + 1
25:             t ← 2t
26:         end while
27:         p ← ct/d                              (periodic pattern)
28:     end if
29:     return (e, h, p, s)
30: end procedure
```

Let us now exploit this representation to build a multiplier of a variable x by a fraction a/b.

III. Periodical shift-and-add trees

In this section, we input the periodic representation of a rational constant a/b, and also a precision w_0, which is the number of bits of a/b that have to be considered for the multiplication. The value of w_0 typically expresses the accuracy requirements of the floating-point or fixed-point context. For instance, section IV will define the value of w_0 that ensures correct rounding for a given floating-point format.
In [5], Gustafsson and Qureshi suggested trying to represent a real constant on more than w_0 bits if it leads to a shift-and-add architecture with fewer additions. They indeed mention the

fact that, due to their periodic representation, rational constants are good candidates for exploiting this idea, without exploiting this idea systematically.

With the notations of previous section, let us define

$$\pi_0 = 2^{-s} px \ .$$

The 2^{-s} factor simply scales the integer p to an approximation of $c/d < 1$, so π_0 is an approximation of cx/d. We may then compute increasingly accurate approximations of cx/d as

$$\pi_1 = \pi_0 + 2^{-s}\pi_0 \ ,$$

$$\pi_2 = \pi_1 + 2^{-2s}\pi_1 \ ,$$

and in general

$$\pi_{i+1} = \pi_i + 2^{-2^i s}\pi_i$$

so that π_i is the product of x by an approximation of c/d of size $2^i s$ bits.

Therefore, a constant corresponding to 2^i repetitions of the period may be built in i additions, and this is optimal [6].

Let us now consider the details, including further optimizations. We note w_h the size in bits of the header h. We need to compute $hx + cx/d$, where cx/d is the periodic part. If w_0 is the precision to which $h + c/d$ must be represented, then c/d must be represented at least on $w_0 - w_h$ bits.

First, one of the existing methods is used to build px and, if $h \neq 0$, hx. As these two constants should be small for the method to be relevant, exhaustive exploration techniques [9], [10], [12] may be used to perform this step optimally. Such methods lead to less than 4 additions for any h or p of size smaller than 12 bits, and less than 5 additions for sizes smaller than 19 bits. The FloPoCo implementation currently uses the simpler heuristic presented in [14], which is better suited to hardware as it also minimizes the size of the adders and leads to minimum-depth adder trees. In our experiments, it consistently computes a minimal-adder-count architecture, probably due to the fact that both h and p are very small integers for the simple rationals that motivate this work.

Then we may compute the π_i. In this process, we may stop as soon as $2^i s \geq w_0 - w_h$. However, it is usually possible to implement a smaller last addition. Let i be such that $2^i s < w_0 \leq 2^{i+1} s$: we must compute the π_j for $0 \leq j \leq i$. Let j be the smallest integer such that $(2^i + 2^j)s \geq w_0 - w_h$. As $j \leq i$, π_j is already computed, and the last stage may compute

$$f = \pi_j + 2^{-2^i s}\pi_i$$

(another option would be to compute $f = \pi_i + 2^{-2^j s}\pi_j$, but this would lead to a larger adder [14]).

If $h = 0$, this is all. If $h \neq 0$, we still have to add hx. This product is itself computed using a classical constant multiplier, in parallel to the computation of the fractional product. There are two possible parenthesing of the two final additions: $r = (hx + \pi_j) + 2^{-2^i s}\pi_i$ or $r = hx + (\pi_j + 2^{-2^i s}\pi_i)$. We may assume that the computation of hx has a depth strictly smaller than that of π_i (which should be the case for "small" rationals). With this assumption, as soon as $j < i$, the first parenthesing leads to a shallower tree, and is therefore preferred. If $i = j$, the second parenthesing will be preferred when it leads to a smaller overall number of full adders.

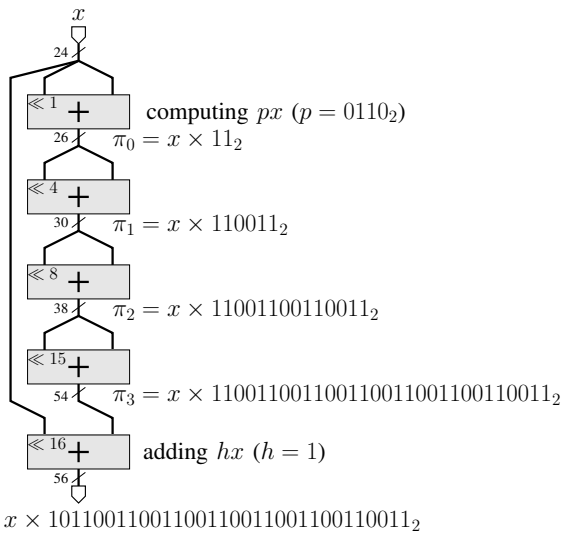

Fig. 1. Multiplication of a 24-bit mantissa by $7/5$ (period $p = 0110_2$, header $h = 1$)

Figure 1 illustrates the resulting architecture on the example of $a/b = 7/5$ for single precision, with a target precision $w_0 = 28$ (providing correct rounding according to Section IV). The smallest value of i such that $2^i s \geq 28$ is $i = 3$. We don't find in this case a smaller j such that $(2^{i-1} + 2^j)s \geq 28$. For this simple example the product px is computed in one addition only, while the product hx is computed in zero additions.

This figure also illustrates a small additional optimization: we trim, whenever possible, leading and trailing zeroes from the various sub-constants to minimize datapath width. For instance, for $a/b = 7/5$, the period is $p = 0110_2$, but π_0 is actually computed as $x \times 11_2$ and the two zeroes are added only when performing the shifts. The final result is actually one bit more accurate than it seems, since there is one more trailing zero to the truncated constant. These technical details are taken into account by the generator in FloPoCo.

IV. Correct rounding

It turns out that for the small rational constants for which this method is of interest ("small" meaning small a and small b), obtaining a correctly rounded result is also fairly cheap. More specifically, the following theorem holds. It is, in essence, a generalization of the "exclusion lemma" used to prove that some division algorithms are correctly rounded [18].

Theorem 1. *Let n be the precison of the input X and q be the precision of the result R, and assume $q \geq n$. If C is a constant obtained by truncating the binary representation of a/b to at least $q + 1 + \lceil \log_2 b \rceil$ bits, then rounding the product CX to the nearest floating-point number of precision q is equivalent to rounding the exact product $\frac{a}{b}X$.*

Note that this theorem covers the most useful case when the input and output precisions are identical.

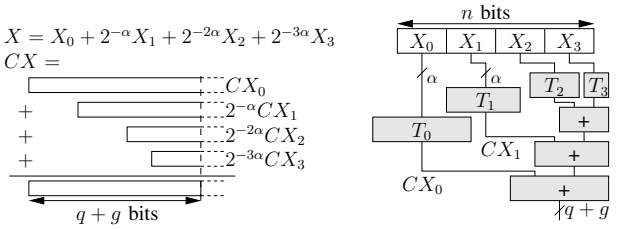

Fig. 2. The KCM LUT-based method

Proof: For reasons already invoked, we may assume without loss of generality that $X \in [1/2, 1)$, and that both a and b are odd. Let us use the integral significand representation of the input X: $X = \frac{I}{2^n}$ where I is an integer. We want to show that $\frac{a}{b}X$ cannot be too close to the mid-point between two floating-point numbers in the result format. Such a mid-point M can be written $M = \frac{2J+1}{2^{q+1}}$.
The distance between $\frac{a}{b}X$ and M is therefore written:

$$\begin{aligned}\left|\frac{a}{b}X - M\right| &= \left|\frac{a}{b}\frac{I}{2^n} - \frac{2J+1}{2^{q+1}}\right| \\ &= \left|\frac{2^{q+1-n}aI - (2J+1)b}{2^{q+1}b}\right|\end{aligned}$$

Here, $2^{q+1-n}aI$ is an even integer since $q \geq n$. On the other hand, $(2J+1)b$ is an odd integer, as the product of two odd integers. We deduce that their difference is at least one, hence

$$\left|\frac{a}{b}X - M\right| \geq \frac{1}{2^{q+1}b}$$

This defines an "exclusion zone" around mid-points. Following a classical argument [14], if we compute $R \approx \frac{a}{b}X$ such that $|R - \frac{a}{b}X| < \frac{1}{2^{q+1}b}$, rounding R to q bits is then equivalent to rounding $\frac{a}{b}X$. Truncating the infinite representation of $\frac{a}{b}$ to the precision $2^{-q-1-\lceil \log 2b \rceil}$ provides this accuracy.

■

V. LUT-BASED METHODS FOR RATIONAL CONSTANTS

On most FPGAs, the basic logic element is the look-up-table, a small memory addressed by α bits. Once filled with the truth table of an arbitrary Boolean function of these α bits, it implements this function. The KCM algorithm[2] due to Chapman [15] and further studied by Wirthlin [16] is an efficient way to use these LUTs to implement a multiplication by an integer constant. As we are interested in rational constants, we use here a variation introduced in [19] that multiplies a fixed-point input X by an arbitrary real constant C.
This algorithm consists in breaking down the binary decomposition of X into chunks of α bits (see Figure 2):

$$X = \sum_{i=0}^{\lceil \frac{n}{\alpha} \rceil - 1} X_i . 2^{-\alpha i} \quad \text{where} \quad X_i \in \{0, ..., 2^\alpha - 1\}$$

The product of X by a constant C becomes

$$CX = \sum_{i=0}^{\lceil \frac{n}{\alpha} \rceil} CX_i . 2^{-\alpha i}$$

[2]This historical acronym seems to mean constant (K) Coefficient Multipliers.

and we have a sum of (shifted) products CX_i. Instead of computing these products, we read CX_i from a table of precomputed values T_i, indexed by X_i.
The cost of each table is one FPGA LUT per output bit (on Figure 2, the corresponding boxes are sized accordingly). We also have to count the cost of computing the sum of these CX_i. The minimal cost is obtained with the sequential implementation of this sum depicted on Figure 2: it consists of $\lceil n/\alpha \rceil - 1$ adders of increasing sizes. However, an adder tree, reducing the latency for a slightly larger overal cost, is usually preferred. As the cost of an adder in FPGAs is typically one LUT per bit, the cost of the adders is roughly equivalent to the cost of the tables.
The FixRealKCM operator in FloPoCo implements this technique. In addition, to ensure last-bit accuracy of CX, the datapath has to be extended with g guard bits that will absorb the rounding errors performed when filling the tables [19]. The error analysis determining g is easily adapted to ensure correct rounding in the case of rational constants, using the results of Section IV.
We now remark that the periodicity of the constant also leads to an optimization of the KCM tables. To illustrate it, let us take as an example $C = 1/3$, and consider in Figure 3 a table holding $X_i/3$ for X_i on $\alpha = 4$ bits. Since each row, having the same denominator, is eventually periodic, the whole table is eventually periodic. This example 22-bit table can be implemented as 5 LUTs instead of 22. In general, for a constant $a/b < 1$ of period size s, the table for a/bX_i requires of the order of $\alpha + s$ LUTs: only the most significand α bits are not periodic.
This optimization only reduces the size of the tables, not the size of the adders, which limits its impact to a 50% improvement at most. However, it is discovered by synthesis tools, so we do not need to explicit it in the code.

VI. RESULTS AND COMPARISONS

Table I provides some results obtained thanks to FloPoCo for the proposed multipliers, compared to the previous implementation of [14], and (for FPGAs) to the KCM approach. In the latter case, one Logic Element (LE) may implement one 4-input LUT, or one Full Adder (FA): the costs reported on each line indeed use the same units.
In all the cases of Table I, the previous shift-and-add approach from [14] already builds a tree of optimal depth, but not of optimal size. Specifically, it does build sub-constants which are repetitions of the period, but the numbers of repetition are not always powers of two, which prevents reusing them optimally. The same holds for the implementation of [9] on spiral.net.

0/3	00000000000000000000
1/3	00001010101010101010
2/3	00010101010101010101
3/3	00100000000000000000
4/3	00101010101010101010
14/3	10010101010101010101
15/3	10100000000000000000

Fig. 3. For rational constants (here $1/3$), the KCM tables are periodic.

constant	$n = q$	Section III		using [14]		KCM
		p_c	+ (FA)	p_c	+ (FA)	+ (LE)
1/3	24	32	4 (118)	27	4 (190)	5 (127)
$h = 0$	53	64	5 (317)	56	5 (368)	12 (508)
$p = 01_2$	113	128	6 (792)	116	6 (1026)	27 (2088)
1/9	24	30	5 (132)	29	5 (131)	5 (167)
$h = 0$	53	60	6 (356)	58	6 (408)	12 (613)
$p = 000111_2$	113	120	7 (885)	118	7 (1116)	27 (2283)
7/5	24	33	5 (139)	28	5 (193)	5 (162)
$h = 1$	53	65	6 (366)	57	6 (595)	12 (594)
$p = 0110_2$	113	129	7 (900)	117	7 (1507)	27 (2267)

TABLE I
ADDER COUNT AND SIZE OF THE SIGNIFICAND MULTIPLIERS FOR SOME CORRECTLY ROUNDED RATIONAL CONSTANT MULTIPLIERS. THE PRECISIONS CHOSEN ARE THOSE OF THE IEEE754-2008 FORMATS.

The other generator of spiral.net [11] minimizes adder count but not adder size, leading to a larger overal number of FA than the periodic approach. These online generators are limited to constant sizes smaller than 25 bits, so they do not appear in the table.

As expected, the KCM approach leads to an adder count proportional to the input precision n and independent of the constant. However, most of these adders are smaller than the output precision q (see Figure 2), whereas the adders in the shift-and-add method are all larger than n (see Figure 1). Therefore, KCM is competitive for small n (typically smaller than 20) especially if $q > n$ [19].

The KCM results are given for $\alpha = 4$ for comparison with the literature (for instance [16] reports 308 LUTs for a KCM with a 24-bit input and a 24-bit constant – this would not even offer correct rounding as the operators of Table I). However, recent FPGAs have larger LUTs ($\alpha = 5$), which leads to a typical reduction of $4/5$ of the KCM cost. In our table, a KCM with $\alpha = 5$ would take the lead only for the first line (113 LE instead of 127 for KCM with $\alpha = 4$, and 118 LE for the periodic shift-and-add). For the other lines, the shift-and-add approach still wins, and KCM is only interesting for precisions smaller than 24 bits.

VII. CONCLUSION

The periodic binary representation of rational constants can be usefully exploited to build efficient hardware for the multiplication by such constants. An implementation of this idea in the open-source FloPoCo core generator is demonstrated. This technique is mostly relevant for constants a/b where both a and b are small integers, and in this case correct rounding of the multiplication by the infinitely accurate rational constant comes at a minor overhead: On most of the examples studied, correct rounding is for free. An important application of this technique is the implementation of divisions by small integers. A multiplier by $1/b$ using this approach is bit-for-bit equivalent to a correctly rounded divider by b.

It could be interesting to study if variations of this technique could not be used to implement division by small integers in software multiple-precision packages.

Acknowledgements

Thanks to B. Pasca and A. Plesco for bringing up this question, to N. Brisebarre for his lights on number theory, and to the anonymous reviewers for their insightful suggestions.

REFERENCES

[1] J. W. L. Glaisher, "Periods of reciprocals of integers prime to 10," *Proc. Cambridge Philos. Soc.*, vol. 3, pp. 185–206, 1878.
[2] E. Artzy, J. A. Hinds, and H. J. Saal, "A fast division technique for constant divisors," *Communications of the ACM*, vol. 19, pp. 98–101, Feb. 1976.
[3] S.-Y. R. Li, "Fast constant division routines," *IEEE Transactions on Computers*, vol. C-34, no. 9, pp. 866–869, Sep. 1985.
[4] P. Srinivasan and F. Petry, "Constant-division algorithms," *IEE Proc. Computers and Digital Techniques*, vol. 141, no. 6, pp. 334–340, Nov. 1994.
[5] O. Gustafsson and F. Qureshi, "Addition aware quantization for low complexity and high precision constant multiplication," *IEEE Signal Processing Letters*, vol. 17, no. 2, pp. 173–176, 2010.
[6] O. Gustafsson, "Lower bounds for constant multiplication problems," *IEEE Transactions On Circuits And Systems II: Express Briefs*, vol. 54, no. 11, pp. 974 – 978, Nov. 2007.
[7] V. Dimitrov, L. Imbert, and A. Zakaluzny, "Multiplication by a constant is sublinear," in *18th Symposium on Computer Arithmetic*. IEEE, 2007, pp. 261–268.
[8] R. Bernstein, "Multiplication by integer constants," *Software – Practice and Experience*, vol. 16, no. 7, pp. 641–652, 1986.
[9] A. Dempster and M. Macleod, "Constant integer multiplication using minimum adders," *Circuits, Devices and Systems, IEE Proceedings*, vol. 141, no. 5, pp. 407–413, 1994.
[10] O. Gustafsson, A. G. Dempster, K. Johansson, and M. D. Macleod, "Simplified design of constant coefficient multipliers," *Circuits, Systems, and Signal Processing*, vol. 25, no. 2, pp. 225–251, 2006.
[11] Y. Voronenko and M. Püschel, "Multiplierless multiple constant multiplication," *ACM Trans. Algorithms*, vol. 3, no. 2, 2007.
[12] J. Thong and N. Nicolici, "An optimal and practical approach to single constant multiplication," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 9, pp. 1373–1386, 2011.
[13] L. Aksoy, E. Costa, P. Flores, and J. Monteiro, "Exact and approximate algorithms for the optimization of area and delay in multiple constant multiplications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 6, pp. 1013–1026, 2008.
[14] N. Brisebarre, F. de Dinechin, and J.-M. Muller, "Integer and floating-point constant multipliers for FPGAs," in *Application-specific Systems, Architectures and Processors*. IEEE, 2008, pp. 239–244.
[15] K. Chapman, "Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner)," *EDN magazine*, May 1994.
[16] M. Wirthlin, "Constant coefficient multiplication using look-up tables," *Journal of VLSI Signal Processing*, vol. 36, no. 1, pp. 7–15, 2004.
[17] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, Aug. 2011.
[18] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhauser Boston, 2009.
[19] F. de Dinechin and B. Pasca, "Floating-point exponential functions for DSP-enabled FPGAs," in *Field Programmable Technologies*, Dec. 2010, pp. 110–117.