Multiplication by rational constants
Florent de Dinechin

To cite this version:

HAL Id: ensl-00610328
https://ens-lyon.hal.science/ensl-00610328v3
Submitted on 15 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Multiplication by rational constants

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 logarithmetic 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 \( \frac{1}{9} \) appearing in stencil applications, but this work covers 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 FPConstMulRational\(^1\) 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 \( \frac{1}{3} \), \( \frac{1}{9} \) and \( \frac{7}{5} \), 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,
Lemma 1. Let us consider an irreducible fraction \( \frac{c}{d} \), where 2 divides neither \( c \) nor \( d \). If \( c < d \), then the binary representation of \( \frac{c}{d} \) is purely periodic, i.e., it starts with an occurrence 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 \( \frac{c}{d} = h + c'/d \). For the purpose of multiplying an input \( x \) by the constant \( \frac{c}{d} \), 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 \( \frac{c}{d} \) 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 occurrence of \( c/d \) exactly expresses the periodicity of the fraction \( c/d \).

**Examples:**
- \( \frac{1}{3} \) 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
  \[
  1/3 = 0.0(1)_2^\infty ;
  \]
- \( \frac{5}{9} \) 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
  \[
  5/9 = 0.0(100011)_2^\infty .
  \]

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

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

1: **procedure** \texttt{PERIODIC}\texttt{\_}\texttt{REPRESENTATION}(\( a, b \))
2: \( (c, d) \leftarrow \texttt{SIMPLIFY}(a, b) \)
3: \( e \leftarrow 0 \)
4: \textbf{while} \( c \mod 2 = 0 \) \textbf{do}
5: \( c \leftarrow c/2 \)
6: \( e \leftarrow e + 1 \)
7: \textbf{end while}
8: \textbf{while} \( d \mod 2 = 0 \) \textbf{do}
9: \( d \leftarrow d/2 \)
10: \( e \leftarrow e - 1 \)
11: \textbf{end while}
12: (Now \( a/b = 2^c c/d \) with both \( c \) and \( d \) odd)
13: \( h \leftarrow c/d \)
14: \( c \leftarrow c \mod d \) \hspace{1cm} (header)
15: \( p \leftarrow c/d \) \hspace{1cm} (periodic part)
16: \( (Now \ a/b = h + c/d \) with \( c < d \))
17: \( s \leftarrow 1 \) \hspace{1cm} (period size)
18: \textbf{if} \( d = 1 \) \textbf{then}
19: \( (a/b \) has a finite binary representation)\)
20: \( p \leftarrow 0 \) \hspace{1cm} (periodic pattern)
21: \textbf{else}
22: \( t \leftarrow 2^s \) \hspace{1cm} (Invariant of the loop below: \( t = 2^s \))
23: \textbf{while} \( t \mod d \neq 1 \) \textbf{do}
24: \( s \leftarrow s + 1 \)
25: \( t \leftarrow 2t \)
26: \textbf{end while}
27: \( p \leftarrow ct/d \) \hspace{1cm} (periodic pattern)
28: \textbf{end if}
29: \( \textbf{return} \ (e, h, p, s) \)
30: \textbf{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 \( w_p \), which is the number of bits of \( a/b \) that have to be considered for the multiplication. The value of \( w_p \) 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 $\pi_0$ 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

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

so that $\pi_i$ is the product of $x$ by an approximation of $c/d$ of size $2^i$ 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 $\pi_i$. 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 $\pi_j$ 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 $j \leq i$, $\pi_j$ is already computed, and the last stage may compute

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

(another option would be to compute $f = \pi_i + 2^{-2^s} \pi_j$, 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 parenthesings of the two final additions: $r = (hx + \pi_j) + 2^{-2^s} \pi_i$ or $r = hx + (\pi_j + 2^{-2^s} \pi_i)$. We may assume that the computation of $hx$ has a depth strictly smaller than that of $\pi_i$ (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.

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 $p = 01102$, but $\pi_0$ 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 precision 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 $aX/b$.

Note that this theorem covers the most useful case when the input and output precisions are identical.
and we have a sum of (shifted) products $CX_i$. Instead of computing these products, we read $CX_i$ from a table of pre-computed 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 overall 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 $a/bX$, requires of the order of $a + s$ LUTs: only the most significant $\alpha$ 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.

V. LUT-BASED METHODS FOR RATIONAL CONSTANTS

On most FPGAs, the basic logic element is the look-up-table, a small memory addressed by $\alpha$ bits. Once filled with the truth table of an arbitrary Boolean function of these $\alpha$ bits, it implements this function. The KCM algorithm due to Chapman [15] and further studied by Wirthin [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 $\alpha$ bits (see Figure 2):

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

The product of $X$ by a constant $C$ becomes

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

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 $M = \frac{2^q + 1}{2^{q+g}}$.

The distance between $\frac{a}{b}X$ and $M$ is therefore written:

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

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

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

This defines an “exclusion zone” around mid-points. Following a classical argument [14], if we compute $R \approx \frac{a}{b}X$ such that $|R - \frac{a}{b}X| < \frac{1}{2^{q+1}}$, rounding $R$ to $g$ bits is then equivalent to rounding $\frac{a}{b}X$. Truncating the infinite representation of $\frac{a}{b}X$ to the precision $2^{-q-1-[\log 2b]}$ 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 $\alpha$ bits. Once filled with the truth table of an arbitrary Boolean function of these $\alpha$ bits, it implements this function. The KCM algorithm due to Chapman [15] and further studied by Wirthin [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 $\alpha$ bits (see Figure 2):

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

The product of $X$ by a constant $C$ becomes

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

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

### 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


