Table-Based versus Shift-And-Add constant multipliers for FPGAs

Florent de Dinechin, Silviu-Ioan Filip, Luc Forget, Martin Kumm

Table-Based versus Shift-And-Add constant multipliers for FPGAs

Florent de Dinechin*, Silviu-Ioan Filip†, Luc Forget* and Martin Kumm‡
*Univ Lyon, INSA Lyon, Inria, CITI, Lyon, France
†Univ Rennes, Inria, CNRS, IRISA, Rennes, France
‡University of Applied Science, Fulda, Germany

florent.de-dinechin@insa-lyon.fr, silviu.filip@inria.fr, luc.forget@insa-lyon.fr, martin.kumm@ai.hs-fulda.de

***Abstract*—The multiplication by a constant is a frequently used operation. To implement it on Field Programmable Gate Arrays (FPGAs), the state of the art offers two completely different methods: one relying on bit shifts and additions/subtractions, and another one using look-up tables and additions. So far, it was unclear which method performs best for a given constant and input/output data types. The main contribution of this work is a thorough comparison of both methods in the main application contexts of constant multiplication: filters, signal-processing transforms, and elementary functions. Most of the previous state of the art addresses multiplication by an integer constant. This work shows that, in most of these application contexts, a formulation of the problem as the multiplication by a real constant allows for more efficient architectures. Another contribution is a novel extension of the shift-and-add method to real constants. For that, an integer linear programming (ILP) formulation is proposed, which truncates each component in the shift-and-add network to a minimum necessary word size that is aligned with the approximation error of the coefficient. All methods are implemented within the open-source FloPoCo framework.**

I. Introduction

When building hardware circuits, multiplication by a constant is a pervasive operation. For instance, the present work is motivated by two application domains: a/ the implementation of elementary functions, where range reduction involves constants such as $\log(2)$ and π, and b/ digital signal processing, where constant multiplication is the central operation of many filters and other kernels, such as Fourier transforms. For low-throughput operation, one may use a standard multiplier in a sequential way and read the constant from a table. If a high throughput is needed, parallelism can be exploited by building specific multipliers specialized for each constant. The construction of such constant multipliers is the subject of the present article.

A general technique for implementing constant multiplication is to build an optimized shift-and-add tree. It has been widely studied for software, for ASICs, and for FPGAs. When targeting software, the function to optimize is the number of additions [1]–[5]. When targeting ASICs or FPGAs, the functions to optimize are the overall area at the level of full-adders [1], [6], [7] or even gates [8] as well as the overall speed [9] or throughput [10]. These are only correlated to the number of additions and depth of the tree, as different additions in the tree will be of different sizes, hence different area and delay.

For FPGAs, whose elementary logic cells are based on look-up tables (LUTs) with 4 to 6 inputs depending on the FPGA family, there is another technique, based on table lookups and addition. It is called KCM after the name of its inventor, Ken Chapman [11], and has been refined by Wirthlin [12] and recently by Walters [13].

Which method performs best strongly depends on the parameters of the problem: the input and output formats, and the constant itself. The two methods are reviewed in detail in Section II, the first objective of this work being too explicit which technique works best in which context. We are aware of only two earlier comparisons of KCM and shift-and-add [14], [15], both limited to the case of multiple constant multiplication (MCM) of small integer constants.

One may remark that most of the state of the art addresses the multiplication by an integer constant, but most applications require a multiplication by a *real* constant, such as the previous $\log(2)$ or π examples. The mainstream approach for this is to round the constant C to a fixed-point value $\widetilde{C}$ which is a scaled integer, then use an optimized multiplier by this scaled integer and finally round the result of this multiplication to the result format. There, most of the literature assumes the same number of bits for the input, the constant, and the output. However, it is often not what applications require, as shown by the following two examples. 1/ In a floating-point exponential [16], we need two constant multipliers: the first by $1/\log(2)$ whose input size is larger than its output size, followed by a multiplier of a small integer (a floating-point exponent) by $\log(2)$, to a high accuracy: *e.g.* in double-precision, 11 input bits and 68 output bits[1]. 2/ Accurate filter implementations [17] are based on a sum of products by constants. The rounding errors of each multiplier add up. Therefore, "guard" bits must be added to the internal datapath to absorb these errors and their amplification by the filter. Here again, the output precision of each multiplier is systematically higher than that of the input.

Section III of this work therefore defines the problem of multiplying a fixed-point input by a real number with all the

[1]With FloPoCo version 4.1.2, try the command `flopoco FPExp we=11 wf=53` and look in the produced VHDL for the signal `absKLog2`.

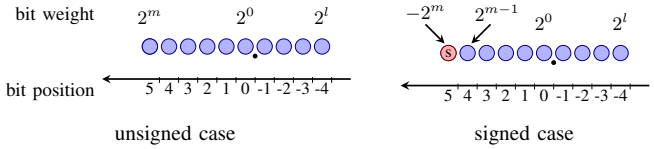


Fig. 1. The bits of a fixed-point format, here $(m, l) = (5, -4)$.

generality needed by applications. This enables us to refine both techniques (shift-and-add, and KCM) on this relevant problem. The evaluation in Section IV shows that up to 40% resources can be saved, compared to the previous three-step solution, for the same application-level accuracy.

Section V concludes and lists future works.

Common definitions, notations and conventions

Throughout the paper we denote the constant with C and the input with X. The sign of the constant is trivial to manage. The input X can be either unsigned, or signed in two's complement. From the application point of view, both are equally relevant. For instance, signals are usually signed, while floating-point mantissas are usually unsigned. Therefore, we must consider both. The signedness of the input has some architectural impact (need for sign extensions in some cases). However, it is always possible to minimize the corresponding overhead, using techniques exposed in the sequel. Thus, in practice, the costs of a signed and unsigned multiplier are very close. Therefore, for clarity, we chose to report only results obtained with signed multipliers.

Integers being a special case of fixed-point numbers, we use the following definition (inspired by the VHDL `sfixed` standard) for fixed-point formats. As illustrated by Fig. 1, a fixed-point format is fully specified by two integers (m, l) that denote the positions of the most and least significant bits (MSB and LSB) respectively. Both m and l can be negative if the format includes fractional bits. The weight of the bit at position i is always 2^i, except for the bit at position m if the format is signed: its weight is -2^m due to two's complement representation. The LSB position l denotes the *absolute precision* of the format. The MSB position m denotes its *range*. The total word size is $m - l + 1$. Other notations used in this article are gathered in Table I.

II. State of the Art in Integer Single-Constant Multiplication

In this section we review the major work that has been done regarding integer constant multiplication over the years.

A. Shift-and-add multipliers

Most of the literature addresses single-constant multiplication (SCM) by a combination of shift-and-add operations. For instance, the multiplication by 17 can be computed as $17X = X + 2^4X$, where the multiplication by 2^4 is a bit shift that (in hardware) costs nothing. The multiplication by 2228241 can be implemented in two additions only if one remarks that $2228241 = 17 \cdot 2^{17} + 17$: first compute $T = 17X$ (one addition), then compute $2228241X = (T \cdot 2^{17} + T)$ (another addition).

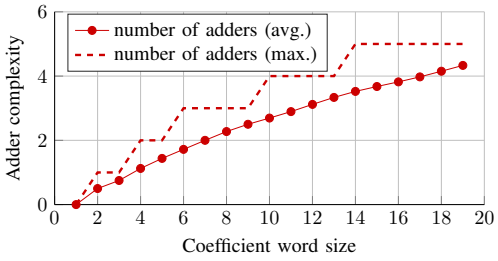


Fig. 2. Number of adders versus word size for optimal-adder adder graphs.

Given a constant C of w_C bits, the problem of finding an implementation in a minimal number of additions is an NP-complete optimization problem [18]. Booth recoding of the constant naturally leads to an implementation in $\lceil w_C/3 \rceil$ adders, but this is not optimal: the optimal number of additions is asymptotically sublinear in w_C [19]. Methods based on exhaustive enumerations of adder graphs have shown [1], [3] that not more than 5 adders are sufficient to realize SCM with any 19-bit number. Another optimal approach based on an exhaustive search indicates that 6 additions are sufficient for integer constants up to 32 bits [4]. This optimal number of adders will be denoted as the "adder complexity" of the constant. The trend of the adder complexity for constants up to 19 bits is shown in Fig. 2. Beyond 32 bits, good heuristics exist which match the optimal complexity or typically require not more than one additional adder [2].

So far we have evaluated the cost in terms of number of additions. This is indeed the relevant metric for software. However, in hardware, not all adders have the same cost [1]. Firstly, the intermediate results grow in the adder graph from input to outputs, and so do the adder sizes. Secondly, in a shift-and-add operation with a left shift of s bits, the s lower bits can be simply transferred from input to output as long as they are not negated (like in subtractions). If the shift is large enough compared to the data word size, the addition corresponds to a concatenation, at no cost. For these reasons, the relevant cost metric in hardware is the number of full adders [1], which we

TABLE I
Notations and Abbreviations Used in This Article

C	a (real or integer) constant
X	fixed-point input to a constant multiplier
$w_C \in \mathbb{N}$	word size of the integer constant in bits
$m_X \in \mathbb{Z}$	weight of the most significant bit of the input fixed-point format
$l_X \in \mathbb{Z}$	weight of the least significant bit of the input fixed-point format
$w_X \in \mathbb{N}$	size of the fixed-point input in bits: $w_X = m_X - l_X + 1$
R, m_R, l_R, w_R	the same, for the fixed-point output R
$\alpha \in \mathbb{N}$	number of inputs of an FPGA's architectural LUT
$\circ(x)$	rounding to the nearest integer (*i.e.*, $\circ(x) = \lfloor x + 1/2 \rfloor$)

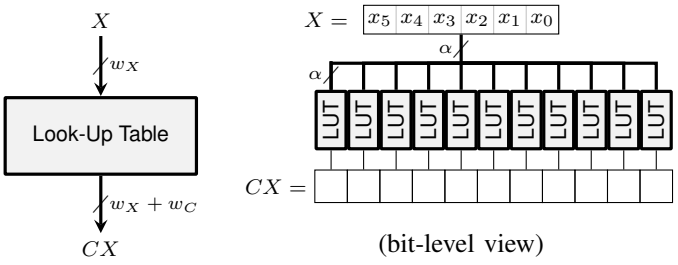


Fig. 3. Storing in a look-up table the product by C of all the possible values of X ($w_X = \alpha = 6$, $w_C = 5$).

will also denote as bit-level cost. Two shift-and-add trees with the same number of adders may have different bit-level costs, and algorithms that optimize for this bit-level cost have also been extensively studied [1], [6]–[8].

All this is relevant both for ASIC synthesis and for FPGA implementations, since integer addition is efficiently supported by the hardware structure of most FPGAs: the full-adder cost can directly be translated to the number of LUTs.

Some FPGAs support ternary adders at almost the same cost as binary adders. Shift-and-add graphs based on such ternary adders have also been studied [20], [21].

B. KCM multipliers

Most mainstream FPGAs are based on small look-up tables (LUTs). Let α denote the number of inputs of the architectural LUT of the target FPGA. For instance, on current FPGAs, α ranges from 4 to 6. Specific table-and-addition methods exploiting these LUTs have been developed. The simplest technique is, for small values of w_X, to pre-compute and tabulate the product of the constant by all the possible values of the input. At run-time, the SCM operation resumes to a table read, and for input sizes up to α, the cost is one FPGA LUT per output bit, as illustrated by Fig. 3.

For larger input sizes, the complexity of this naive approach becomes exponential with the input size w_X, but a technique introduced by Chapman [11] allows for architectures whose cost is linear with w_X. The principle is to decompose the input X in sub-words of α bits (or to view it as a radix-2^α number, hexadecimal being a popular example when $\alpha = 4$) :

$$X = \sum_{i=0}^{\lceil \frac{w_X}{\alpha} \rceil} X_i \cdot 2^{\alpha i} \qquad \text{(see Fig. 4).}$$

Now the product becomes $CX = \sum CX_i \cdot 2^{\alpha i}$ and we have a sum of (shifted) products CX_i, each of which can be computed by a look-up table with α input bits (back to Fig. 3). The cost of each table is $\alpha + w_C$ FPGA LUTs.

There are quite a few ways of implementing the summation, from a simple rake to a compressor tree. As a rule of thumb, the area cost of the summation is comparable to that of the tables. Wirthlin [12] then Walters [13] showed that in a rake, tables and additions could be merged, under the conditions that the tables input only $\alpha - 1$ bits (*i.e.,* using a radix-$2^{\alpha-1}$ decomposition of the input), which means more tables.

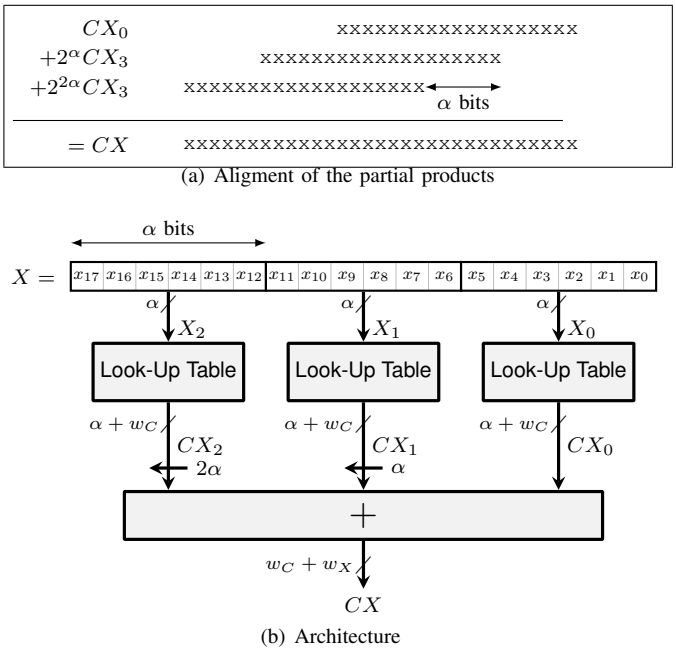


Fig. 4. KCM multiplication of a 18-bit constant C by an 18-bit input X split in 3 chunks of α =6 bits. Each subproduct CX_i is of size 24 bits.

Overall, this technique reduces the area by about one third, but due to the rake structure the latency is larger than with tree summations [13].

C. Comparison and discussion

To sum up, the cost of KCM depends on the size of the constant but not[2] on its "adder complexity" as defined above. Conversely, the cost of shift-and-add mostly depends on the complexity of the constant. At a first approximation, the bit-level cost should be linear in w_X. However, the addition sizing at the bit level strongly depends on w_X. To take an extreme example, multiplying a $w_X = 4$-bit unsigned number by 2228241 has cost zero, since all the additions can be done via concatenation. Multiplying the same constant by a $w_X = 16$-bit unsigned number costs one single 16-bit addition.

For all these reasons, for a given constant and a given w_X, there is probably no best way to select the best technique other than "try both, evaluate their costs, keep the best". The good news is that, although there is no simple closed formula giving the cost of a solution, it is easy to program in both cases very accurate bit-level cost models: such programs input C, w_X and the signedness of the input, and can predict the full-adder cost (or LUT cost) very accurately.

Work is therefore in progress for integrating the state of the art for integer SCM in a single FloPoCo operator that performs this design-space exploration. This also includes SCM based on ternary adders [21].

[2]Of course, synthesis tools will optimize out the table content when they find opportunities to do so, which happens *e.g.* for trivial constants (powers of two), but also for periodic constants [22].

III. Multiplication by a real constant

So far we have discussed multiplication of an integer input X by an integer constant C (possibly with some fixed-point scaling). The result is then a number R of size $w_R = w_C + w_X$ bits. However, most applications actually involve multiplications by a *real* constant, such as the previous $\log(2)$ or π. These are irrational constants with an infinite number of bits: they must be rounded first to a finite-size constant $\widetilde{C}$ of size w_C. If the precision w_R of the desired result is the same as the precision of the input: $w_R = w_X$, then one will typically round the constant to $w_C = w_R = w_X$. One may then use one of the previous fixed-point techniques. However, the result is a $2w_R$-bit number, and has to be rounded again. This is inefficient for two reasons. Firstly, there are two rounding errors in this process (rounding the constant, then rounding the product). Secondly, half of the computed bits are discarded in the final rounding.

A. Problem formulation

In this work, we define the *single real constant multiplication* (SRCM) operation as follows. Given a maximum error bound $\bar{\epsilon} > 0$, an input fix-point format (m_X, l_X), an output fix-point format (m_R, l_R), and a real constant C, an SRCM operator computes for each fix-point input X a fix-point output R such that

$$R = X \times C + \epsilon \tag{1}$$

with $|\epsilon| < \bar{\epsilon}$.

The maximum absolute error bound $\bar{\epsilon}$ cannot, by definition, be smaller than 2^{l_R-1} (one half ulp of the result format): this corresponds to rounding to the nearest number in the result format. The cost of achieving this accuracy depends on the constant, and can be arbitrarily large[3] [23]. However, for small input sizes, the plain tabulation offers correct rounding as shown in Fig. 5. On FPGAs, this is perfectly acceptable for input sizes up to $w_c = \alpha + 2$, or 8 bits in current technology. A correctly rounded 8-bit in, 8-bit out multiplier will cost 32 LUTs and have unit delay.

Fixing the error bound to 2^{l_R} corresponds to what is commonly known as faithful rounding: the operator will return one of the two fixed-point numbers surrounding the exact (unrounded) result. Besides, as the constraint $|\epsilon| < \bar{\epsilon}$ is strict, if the exact product happens to be representable in the result format, then the operator will return this exact result.

[3]Consider for instance the multiplication of 8-bit integers by the constant $C = 1.97058823529412$, with the result rounded to an integer ($l_R = 0$). For the value $X = 17$, the correctly rounded result is $\circ(CX) = 34$. However, the value of ϵ in this case is $\epsilon = 34 - CX = 0.49999999999996$: it is very close to a half-ulp – this example was constructed for this purpose. This makes it tricky to compute the correct rounding to the nearest. For illustration, if in the process of building a constant multiplier we use a rounded *down* value $\widetilde{C}$ of the constant, for instance to the decimal value $C = 1.97$ (with an error smaller than $6 \cdot 10^{-4}$), or to the nearest 8-bit number, or even to the nearest 32-bit number, or even to $C = 1.9705882352941$ (removing only the last decimal digit of C), then the computation $\circ(\widetilde{C}X)$ will yield the wrong value (33 instead of 34). Given a constant, it is a non-trivial problem to determine the accuracy of the intermediate computations that will guarantee correct rounding to the nearest *for all the possible inputs*. It has been solved elsewhere [23].

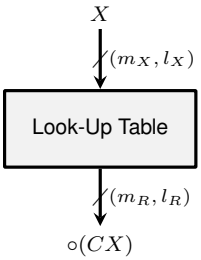


Fig. 5. Storing in a look-up table the rounded product by C of all the possible values of X.

On FPGAs, we have the choice of the formats, especially if the result of the multiplier is an intermediate data hidden inside a filter or a function. In this case, it is in general more economical to achieve the same accuracy 2^l using faithful rounding to a format of LSB weight 2^{l+1} than using correct rounding to a format of LSB 2^l.

For this reason, the remainder of this article focuses on faithful rounding.

B. Naive three-step approach

It is always possible to achieve faithful rounding using the naive three-step technique introduced above, with the following error decomposition (given here first in the case of unsigned fixed-point inputs, such that $0 \le X < 2^{m_X+1}$). Noting $\circ_l(x) = 2^{-l} \cdot \circ(2^l x)$, let

$$\widetilde{C} = \circ_{l_R - m_X - 1}(C) \quad \text{hence } \epsilon_C = \widetilde{C} - C \le 2^{l_R - m_X - 2} \tag{2}$$

$$R = \circ_{l_R}(\widetilde{C} \times X) \quad \text{hence } \epsilon_\times = R - \widetilde{C} \times X \le 2^{l_R - 1} \tag{3}$$

Then, the total absolute error due to the operator can be described as

$$\epsilon = R - CX \tag{4}$$

$$= R - \widetilde{C}X \ + \ \widetilde{C}X - CX \tag{5}$$

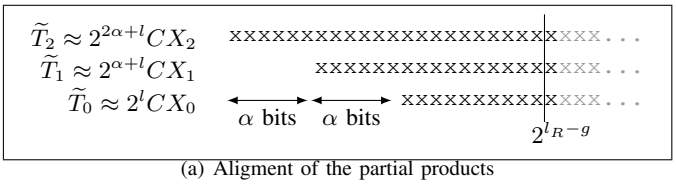
$$= \epsilon_\times + \epsilon_C X \tag{6}$$

The triangle inequality yields $|\epsilon| \le |\epsilon_\times| + |\epsilon_C| \cdot |X|$, and from the bound on the input $0 \le X < 2^{m_X+1}$ we conclude that $|\epsilon| < 2^{l_R}$, hence R is CX faithfully rounded.

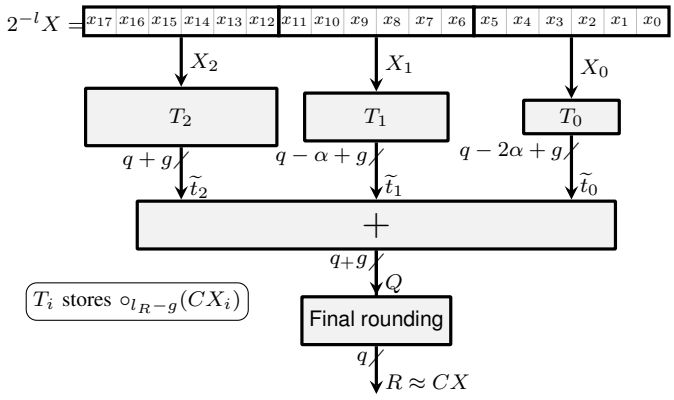
The extension to signed input is straightforward, with one subtlety: as m_x is the weight of the sign bit, we have $-2^{m_X} \le X < 2^{m_X}$. There is the opportunity to save one bit in the rounding of the constant, using $\widetilde{C} = \circ_{l_R - m_X}(C)$, but only if the inequality $\epsilon_C = \widetilde{C} - C \le 2^{l_R - m_X - 1}$ is strict, to achieve a strict inequality $|\epsilon| < 2^{l_R}$. This is the most common case, but it has to be checked nevertheless.

C. Fix×real constant multiplication using KCM

This operator has been described in detail in [17]. It is illustrated in Figs. 5–6. It is radically different from the three-step approach, since there is no rounding of C to some $\widetilde{C}$: the exact, real value of C is used to compute each table entry. Each entry is rounded directly to the target accuracy that will ensure faithful rounding, defined by a number of guard bits g whose value is computed as follows. Each table entails a rounding



(a) Aligment of the partial products



(b) Architecture

Fig. 6. FixRealKCM with an 18-bit input X split in 3 chunks of 6 bits.

error of at most 2^{l_R-g-1} (correct rounding of each table entry), so the overall rounding error is $\lceil w_X/\alpha \rceil \cdot 2^{l_R-g-1}$. Hence, g has to be selected such that $\lceil w_X/\alpha \rceil \cdot 2^{l_R-g-1} < 2^{l_R-1}$ holds which is given for any $g > \log_2(\lceil w_X/\alpha \rceil)$. The "Final rounding" box in Fig. 6(b) is then a simple truncation, provided that a rounding bit 2^{l_R-1} has been added to all the entries of one of the tables. Then this truncation becomes the rounding to the nearest ($\lfloor x+1/2 \rfloor = \circ(x)$) of the intermediate sum Q with its g guard bits to the final format, with an error smaller than 2^{l_R-1}. The sum of all these errors ensures faithful rounding.

The original integer KCM could be viewed as a special case when C is an integer. In this case, the tables contain LSB zeroes that the synthesis tools will in principle optimize out. However, it should still be handled as a special case, in particular because the integer KCM is exact and therefore does not need the g guard bits and the rounding bit.

D. Fix×real constant multiplication using shift-and-add

In contrast to the KCM case of the previous section, we have to find a proper rounding of C to some $\widetilde{C}$. However, we do not have to decompose the error as described in Section III-B. There is a large range from which valid coefficients can be selected. Taking the error definition

$$\epsilon = XC - X\widetilde{C} = X\Delta C < X_{\max}\Delta C_{\max} = \overline{\epsilon} \tag{7}$$

with $|X| < 2^{m_X+1}$ in the unsigned case and $|X| < 2^{m_X}$ in the signed case we get

$$\Delta C_{\max} = \begin{cases} \overline{\epsilon}/2^{m_X+1} & \text{for } X \text{ unsigned} \\ \overline{\epsilon}/2^{m_X} & \text{for } X \text{ signed} \end{cases} \tag{8}$$

Clearly, there will be one fixed-point constant $\widetilde{C}_{\min w}$ with minimal word size $w_{C,\min}$ that will fulfill the above condition.

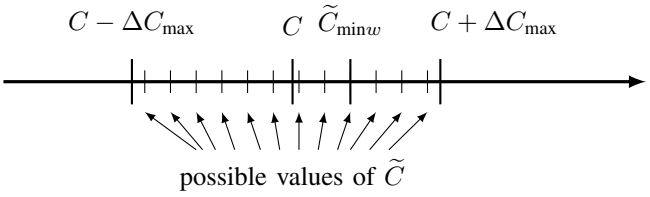


Fig. 7. Possible values of fixed-point coefficient $\widetilde{C}$.

TABLE II
EXAMPLE COEFFICIENTS $\widetilde{C}$ FOR $C = \pi/4$ FOR $\overline{\epsilon} = 2^{-25} \simeq 2.98 \cdot 10^{-8}$, THEIR CORRESPONDING ERROR DISTRIBUTION AND FULL ADDER COST (#FA) BEFORE AND AFTER TRUNCATION OF THE ADDER GRAPH

k	$\widetilde{C}$	$\lvert\overline{\epsilon}_C\rvert$	$\lvert\overline{\epsilon}_\times\rvert$	#add	#FA	#FA trunc
-8	$210828707/2^{28}$	$2.65 \cdot 10^{-8}$	$3.22 \cdot 10^{-9}$	7	226	175
-7	$52707177/2^{26}$	$2.28 \cdot 10^{-8}$	$6.95 \cdot 10^{-9}$	8	237	185
-6	$210828709/2^{28}$	$1.91 \cdot 10^{-8}$	$1.06 \cdot 10^{-8}$	6	210	152
-5	$105414355/2^{27}$	$1.53 \cdot 10^{-8}$	$1.44 \cdot 10^{-8}$	7	229	174
-4	$210828711/2^{28}$	$1.16 \cdot 10^{-8}$	$1.81 \cdot 10^{-8}$	7	238	174
-3	$26353589/2^{25}$	$7.94 \cdot 10^{-9}$	$2.18 \cdot 10^{-8}$	6	182	151
-2	$210828713/2^{28}$	$4.22 \cdot 10^{-9}$	$2.55 \cdot 10^{-8}$	7	225	165
-1	$105414357/2^{27}$	$4.96 \cdot 10^{-10}$	$2.93 \cdot 10^{-8}$	7	225	167
0	$210828715/2^{28}$	$3.22 \cdot 10^{-9}$	$2.65 \cdot 10^{-8}$	6	197	141
1	$52707179/2^{26}$	$6.95 \cdot 10^{-9}$	$2.28 \cdot 10^{-8}$	7	218	173
2	$210828717/2^{28}$	$1.06 \cdot 10^{-8}$	$1.91 \cdot 10^{-8}$	6	204	132
3	$105414359/2^{27}$	$1.44 \cdot 10^{-8}$	$1.53 \cdot 10^{-8}$	5	156	125
4	$210828719/2^{28}$	$1.81 \cdot 10^{-8}$	$1.16 \cdot 10^{-8}$	6	182	151
5	$13176795/2^{24}$	$2.18 \cdot 10^{-8}$	$7.94 \cdot 10^{-9}$	6	176	136
6	$210828721/2^{28}$	$2.55 \cdot 10^{-8}$	$4.22 \cdot 10^{-9}$	6	194	148
7	$105414361/2^{27}$	$2.93 \cdot 10^{-8}$	$4.96 \cdot 10^{-10}$	7	230	190

As the shift-and-add constant multiplication strongly depends on the constant, the idea here is to take a slightly larger coefficient word size $w_{C,\min}+q$ and to evaluate the complexity of all the fixed-point coefficients that lie in the valid range $C - \Delta C_{\max} < \widetilde{C} < C + \Delta C_{\max}$. We then choose the one with least complexity. This is illustrated in Fig. 7. The procedure follows a concept previously presented as addition-aware quantization [24]. However, in our case, each valid fixed-point coefficient is not only defined by its complexity, but also gives a remaining error margin that can be used for truncations in the shift-and-add computations. Hence, $\widetilde{C}$ candidates close to C result in a small $\overline{\epsilon}_C$ allowing a larger error for truncations in the multiplier $\overline{\epsilon}_\times$ and vice versa. The multiplier error $\overline{\epsilon}_\times$ can be tuned by a novel truncation method introduced in Section III-E.

An example for the constant multiplication with $C = \pi/4$ with a maximum overall error of $\overline{\epsilon} = 2^{-25}$ when evaluating $2^q = 16$ different fixed-point coefficients is given in Table II. The table shows the different fixed-point coefficients, their error distributions, the number of adders (#add) obtained by the RPAG algorithm [10] as well as the number of full adders (#FA) before and after the truncation (#FA trunc). The coefficient for $k = -1$ is the closest to $\widetilde{C}$ and provides the smallest $|\overline{\epsilon}_C|$ which allows the highest $|\overline{\epsilon}_\times|$ for truncating the multiplier. However, the coefficient that can be realized with

the minimum number of adders is found for $k = 3$. It also requires the least number of full adders before and after the truncation. Note also that it is a better choice compared to the minimum word size coefficient which is obtained for $k = 5$.

E. Truncated SCM

The goal of truncation is to minimize the required number of full adders in an SCM instance with a given adder graph solution, such that the output error is upper bounded by a maximum tolerable value $\overline{\epsilon}_\times$.

To do so, we have devised a method that models the problem using integer linear programming (ILP). The objective is the maximization of the number of FAs saved by allowing truncation. It is given in ILP Formulation 1. The complexity model used for counting the FAs is based on [6], which is extended to also take truncation into account. Note that the method works for generic adder graphs which includes multiple constant multiplication (MCM) or even constant matrix multiplications (CMM) operations based on shift-and-add networks.

The binary variables $t_{a,w}$ are used to encode the position where truncation takes place on each edge a, which is why there is only one active $t_{a,w}$ per edge (constraint C1). Every value ϵ_a represents an upper bound on the error propagated in the adder graph until edge a due to truncation, *before* the shift by s_a bits is applied. It is defined by constraint C2 as the maximum between the sum of the appropriately shifted error bounds on the input edges of the adder with output edge a (denoted with ϵ_{a_ℓ} and ϵ_{a_r} for the left and right operand edges a_ℓ and a_r) and the truncation error et_a on the output a. If a is not the output edge of any adder (*i.e.,* it starts from the input X), then $\epsilon_{a_\ell} = \epsilon_{a_r} = 0$ and $\epsilon_a = \text{et}_a$. At the end of the error propagation, the (shifted) error bounds on the output edges are bounded by $\overline{\epsilon}_\times$ (constraint C3).

The g_a variables defined by constraint C4 correspond to the number of FAs that can be saved in the adder graph due to truncation for each adder/subtractor. These values are computed with respect to a fully naive implementation in which no savings due to shifts and truncations are considered. They are the equivalents of the cases exemplified in [6, Sec. 2].

Considering that only odd fundamentals are computed at each adder node with output edge a, there are two possibilities for the values of the shifts s_ℓ and s_r: (a) $s_\ell \geq 0, s_r = 0$ or (b) $s_\ell = s_r < 0$. Both (a) and (b) can be further decomposed based on the signs of the left and right operands $(\text{sign}(a_\ell), \text{sign}(a_r)) \in \{(1,1), (-1,1), (1,-1)\}$ (the case where both operands are negative does not appear in common adder graphs as it is more hardware demanding to compute).

If we are in (a) with $(\text{sign}(a_\ell), \text{sign}(a_r)) \in \{(1,1), (-1,1)\}$ then one can save in terms of FAs the maximum between the sums of the shifted and truncated bits on each of the two operands (the first case in constraint C4). This is exemplified (without truncations) in [6, Fig. 5 (a)–(b)]. If on the other hand, $\text{sign}(a_r) = -1$, then the most FAs we can save are the number of truncated bits on the right operand (the second case in C4). An example (without truncation) is [6, Fig. 5 (c)].

ILP Formulation 1 ILP formulation for the truncated SCM/MCM/CMM problem for a given adder graph.

$$\max \sum_{a \in N_A} g_a$$

subject to

$$\text{C1:} \quad \sum_{w=0}^{W_a-1} t_{a,w} = 1, \forall a \in E$$

$$\text{C2:} \quad \epsilon_a = \max\left(2^{s_{a_\ell}}\epsilon_{a_\ell} + 2^{s_{a_r}}\epsilon_{a_r}, \text{et}_a\right), \forall a \in E$$

$$\text{C3:} \quad 2^{s_a}\epsilon_a \leq \overline{\epsilon}_\times, \forall a \in O$$

$$\text{C4:} \quad g_a = \begin{cases} \max\left(t_{a_\ell} + s_{a_\ell}, t_{a_r} + s_{a_r}\right) & \text{if } s_{a_\ell} \geq 0 \text{ and } \text{sign}(a_r) = 1 \\ t_{a_r} & \text{if } s_{a_\ell} \geq 0 \text{ and } \text{sign}(a_r) = -1 \\ 0 & \text{otherwise} \end{cases},$$

$$\forall a \in N_A$$

where $t_{a,w} \in \{0,1\}, \epsilon_a \in \{0,1,\ldots,2^{W_a}-1\}$,

$$\text{et}_a = \sum_{w=0}^{W_a-1} (2^w - 1) t_{a,w},$$

and

$$t_a = \sum_{w=0}^{W_a-1} w t_{a,w}.$$

Constant	Meaning
$W_a \in \mathbb{N}$	Word size on the edge a
$\overline{\epsilon}_\times$	Maximum tolerable error
s_a	Bits shifted on edge a
a_ℓ and a_r	Input edges for adder with output edge a
E	Set of edges
$O \subseteq E$	Set of output edges
$N_A \subseteq E$	Set containing one output edge for each adder in the graph
Variable	**Meaning**
$t_{a,w} \in \{0,1\}$	true, if edge a is truncated at position w
$\epsilon_a \in \mathbb{N}$	Maximum error on edge a after truncation

For situation (b), which seldom occurs in practice, we make the same simplifying assumption as in [6, Fig. 6] and consider that no FAs are saved (third case of constraint C4).

The gain in number of FAs is essentially limited by how tight the error bounds in the C2 constraints will be. Nevertheless, the results we obtain in practice compare well with other approaches found in the literature. Take for instance the

TABLE III
COMPARISON OF SYNTHESIS RESULTS FOR MULTIPLYING WITH $\pi/4$

$w_X = w_C$	FixFixKCM [13]		FixRealKCM [17]		FixFixShiftAdd [10]		FixRealShiftAdd (this work)	
	LUTs	delay [ns]	LUTs	delay [ns]	LUTs	delay [ns]	LUTs	delay [ns]
8	17	1.5	14	3.6	27	3.7	25	3.6
10	22	1.5	14	3.6	42	3.7	38	3.7
12	40	2.3	30	3.6	56	3.8	39	3.6
14	47	2.5	37	3.7	86	3.9	59	3.9
16	63	2.3	46	3.8	79	3.8	68	3.9
20	87	3.1	71	3.8	119	3.9	81	4.7
24	129	4.0	104	4.1	199	4.2	125	5.7

TABLE IV
THE MULTIPLIERS USED IN SINGLE AND DOUBLE-PRECISION FLOATING-POINT EXPONENTIAL

C	(m_X, l_X, l_R)	FixRealKCM [17] LUTs	FixRealKCM [17] delay	FixRealShiftAdd LUTs	FixRealShiftAdd delay
$1/\log(2)$	$(6, -3, 0)$	14	3.6 ns	32	3.6 ns
$\log(2)$	$(7, 0, -26)$	59	3.8 ns	92	5.4 ns
$1/\log(2)$	$(9, -3, 0)$	25	3.7 ns	50	4.1 ns
$\log(2)$	$(10, 0, -56)$	187	4.3 ns	–	–

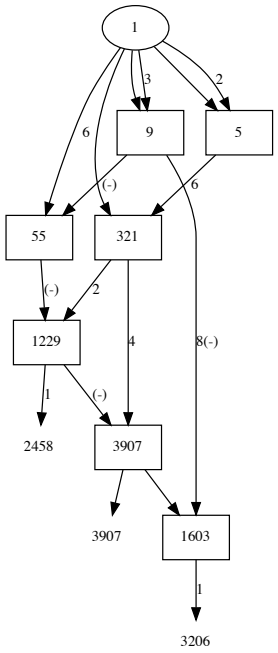


Fig. 8. Adder graph used for the MCM implementation with constants $2458, 3206$ and 3907 from [25, Sec. II-B].

heuristic pattern modification technique for truncated MCM in the context of finite impulse response filter implementations [25] used on the adder graph from Fig. 8. Using their algorithm with $\bar{\epsilon}_\times = 2^{10}$, the reduction in terms of FAs with respect to the fully naive implementation is 45, whereas with our ILP model we obtain a slightly better reduction of 46 FAs.

IV. RESULTS

Several synthesis experiments have been performed to compare the state of the art using fixed-point by fixed-point constant multiplication (FixFix) as described in Section III-B as well as the fixed-point by real constant multiplication (FixReal) using the KCM and the shift-and-add techniques described in Section III. All synthesis results were obtained using Vivado 2018.2 targeting a Xilinx Kintex7 FPGA (7k70tfbv484-3) with timing constraints for a 10 ns clock period. Table III summarizes the synthesis results.

For the FixFix cases, we used the integer constants approximating $\pi/4$ from [13], while for the FixReal case we used the actual real number. Note that some of the integer numbers from [13] are not exactly the rounding of $\pi/4$ to the nearest number on w_C bits but the closest odd number, "to avoid obvious optimizations" [13]. From an application context this choice is doubly disputable since 1/ optimizations are welcome and 2/ it increases the error. We nevertheless follow it here for the purpose of comparison with [13].

Column 'FixFixKCM' describes the results for the latest work on KCM using fixed point numbers by Walters as reported in [13]. Column 'FixRealKCM' shows results from recent work by Volkova et al. [17]. The shift-and-add results are given in column 'FixFixShiftAdd' by using the RPAG algorithm [10] for obtaining the adder graphs for the same integers as in [13]. Finally, column 'FixRealShiftAdd' presents results obtained from the method introduced in Section III-D.

Two important results can be observed from this experiment. First, the common approach of just quantizing the constant shows a much higher resource utilization than considering the constant as a real number. Second, the KCM approaches always outperform the shift-and-add methods, independent of the FixFix or FixReal case.

This is confirmed by Table IV which compares the two FixReal variants on the multipliers needed inside a floating-point exponential [16]. No results could be obtained for the high precision $\log(2)$ constant using FixRealShiftAdd as the current RPAG implementation is limited to 64 bits.

It is also confirmed by Figure 9 which compares the variation of the area (in LUTs) of the operators with input and output size. One can observe that the plot for FixRealKCM is smoother: for FixRealShiftAdd, adding one bit may introduce subgraph sharing opportunities that reduce the total size.

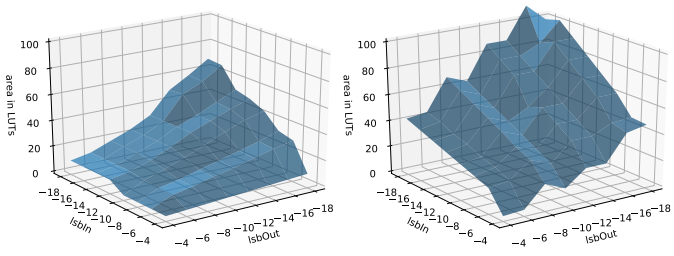


Fig. 9. Area of FixRealKCM(left) and FixRealShiftAdd (right) for $\log(2)$

V. Conclusion and Future Work

This work is a comparison of the two mainstream methods for constant multiplication on FPGAs: KCM and shift-and-add. The integer multiplication problem is reviewed in details, but this work also claims that the most general problem is the multiplication of a fixed-point number by a real-valued constant. This work introduces a new ILP-based optimization technique for truncating shift-and-add constant multipliers under a strict accuracy constraint. This technique is compared to existing work in the KCM space.

A general conclusion of this study is that KCM is more efficient, even for large sizes. This can be understood considering the large LUTs of modern FPGAs. However, it was not expected, as previous work showed that the shift-and-add approach was better for input word sizes larger than 8 bit in the case of multiple constant multiplication [15]. This can be explained by the fact that in this case, much more sharing is possible when multiplying with several constants. Besides, in this case, the truncation method introduced in Section III-E will help.

However, there exist combinations of constants and input sizes for which the shift-and-add method wins. The real conclusion is therefore that the best way to address this comparison is on a case-by-case basis, using a versatile tool that tracks the state of the art in both methods, provides the same general interface to them (including the option to input signed or unsigned numbers), and performs the comparison using accurate cost models. The present work introduces such a tool in the FloPoCo framework.

A shortcoming of the current truncation optimization phase from Section III-E is that it assumes a given adder graph. The next step is to investigate if the choice of adder graph can be directly integrated in the optimization phase effectively, potentially offering greater savings in terms of adder cost.

References

[1] A. Dempster and M. Macleod, "Constant integer multiplication using minimum adders," *Circuits, Devices and Systems*, vol. 141, no. 5, pp. 407–413, 1994.

[2] Y. Voronenko and M. Püschel, "Multiplierless Multiple Constant Multiplication," *ACM Transactions on Algorithms*, vol. 3, no. 2, pp. 1–38, 2007.

[3] O. Gustafsson, A. Dempster, K. Johansson, M. Macleod, and L. Wanhammar, "Simplified Design of Constant Coefficient Multipliers," *Circuits, Systems, and Signal Processing*, vol. 25, no. 2, pp. 225–251, 2006.

[4] 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.

[5] M. Kumm, "Optimal Constant Multiplication using Integer Linear Programming," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 5, pp. 567–571, 2018.

[6] K. Johansson, O. Gustafsson, and L. Wanhammar, "A detailed complexity model for multiple constant multiplication and an algorithm to minimize the complexity," in *Circuit Theory and Design*, 2005, pp. 465–468.

[7] ——, "Bit-Level Optimization of Shift-and-Add Based FIR Filters," in *IEEE International Conference on Electronics, Circuits and Systems, (ICECS)*. IEEE, 2007, pp. 713–716.

[8] L. Aksoy, E. Costa, P. Flores, and J. Monteiro, "Optimization of area in digital FIR filters using gate-level metrics," in *Design Automation Conference*, 2007, pp. 420–423.

[9] O. Gustafsson and L. Wanhammar, "Low-Complexity and High-Speed Constant Multiplications for Digital Filters Using Carry-Save Arithmetic," in *Digital Filters*. InTech, Apr. 2011.

[10] M. Kumm, P. Zipf, M. Faust, and C.-H. Chang, "Pipelined Adder Graph Optimization for High Speed Multiple Constant Multiplication," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2012, pp. 49–52.

[11] K. Chapman, "Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner)," *EDN magazine*, no. 10, p. 80, May 1993.

[12] M. Wirthlin, "Constant coefficient multiplication using look-up tables," *Journal of VLSI Signal Processing*, vol. 36, no. 1, pp. 7–15, 2004.

[13] E. G. Walters, "Reduced-area constant-coefficient and multiple-constant multipliers for Xilinx FPGAs with 6-input LUTs," *Electronics*, vol. 6, no. 101, 2017.

[14] M. Faust and C.-H. Chang, "Bit-parallel Multiple Constant Multiplication using Look-Up Tables on FPGA," *IEEE International Symposium of Circuits and Systems (ISCAS)*, pp. 657–660, 2011.

[15] M. Kumm, D. Fanghänel, K. Möller, P. Zipf, and U. Meyer-Baese, "FIR Filter Optimization for Video Processing on FPGAs," *Springer EURASIP Journal on Advances in Signal Processing*, pp. 1–18, 2013.

[16] F. de Dinechin and B. Pasca, "Floating-point exponential functions for DSP-enabled FPGAs," in *Field Programmable Technologies*, Dec. 2010, pp. 110–117, best paper candidate.

[17] A. Volkova, M. Istoan, F. de Dinechin, and T. Hilaire, "Towards hardware IIR filters computing just right: Direct form I case study," *IEEE Transactions on Computers*, Dec. 2018, to appear. [Online]. Available: http://hal.upmc.fr/hal-01561052

[18] P. Cappello and K. Steiglitz, "Some Complexity Issues in Digital Signal Processing," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 32, no. 5, pp. 1037–1041, Oct. 1984.

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

[20] M. Kumm, M. Hardieck, J. Willkomm, P. Zipf, and U. Meyer-Baese, "Multiple Constant Multiplication with Ternary Adders," in *IEEE International Conference on Field Programmable Logic and Application (FPL)*, 2013, pp. 1–8.

[21] M. Kumm, O. Gustafsson, M. Garrido, and P. Zipf, "Optimal Single Constant Multiplication using Ternary Adders," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 7, pp. 928–932, 2018.

[22] F. de Dinechin, "Multiplication by rational constants," *IEEE Transactions on Circuits and Systems, II*, vol. 52, no. 2, pp. 98–102, Feb. 2012.

[23] 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.

[24] 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.

[25] R. Guo, L. S. DeBrunner, and K. Johansson, "Truncated MCM using pattern modification for FIR filter implementation," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2010, pp. 3881–3884.