Hardware LTI Filters Computing Just Right

Florent de Dinechin, Thibault Hilaire, Matei Istoan, Benoit Lopez, Abdelbassat Massouri
Université de Lyon, INRIA,
INSA-Lyon, CITI-INRIA, F-69621, Villeurbanne, France

***Abstract*—Linear Time Invariant (LTI) filters are often specified and simulated using high-precision software, before being implemented in low-precision hardware. A problem is that the hardware does not behave exactly as the simulation due to quantization and rounding issues. This article advocates the construction of LTI architectures that behave as if the computation was performed with infinite accuracy, then rounded only once to the low-precision output format. From this minimalist specification, it is possible to deduce the optimal values of many architectural parameters, including all the internal data formats. This requires a detailed error analysis that captures the rounding errors, but also their infinite accumulation in infinite impulse response filters. This error analysis may then guides the design of hardware satisfying the accuracy specification at the minimal hardware cost. This is illustrated on the case of low-precision LTI filters implemented in FPGA logic. This approach is fully automated in a generic, open-source architecture generator tool built upon the FloPoCo framework, and evaluated on a range of Finite and Infinite Impulse Response filters.**

I. Introduction

This article addresses the automatic implementation of Linear Time Invariant (LTI) digital filters. Such filters are ubiquitous in signal processing and control, and are typically defined as a transfer function in the frequency domain:

$$\mathcal{H}(z) = \frac{\sum_{i=0}^{n} b_i z^{-i}}{1 + \sum_{i=1}^{n} a_i z^{-i}} \quad \forall z \in \mathcal{C}. \tag{1}$$

Equivalently, the output signal $y(k)$ and the input signal $u(k)$ may also be related by the following equation:

$$y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=1}^{n} a_i y(k-i) \tag{2}$$

Equation (**??**) or (**??**), along with a mathematical definition of each coefficient a_i and b_i, constitute the *mathematical specification* of the problem. In this specification, the coefficients are considered as real numbers. They may even be given as explicit formulae, as for instance in textbook pulse-shaping filters (half-sine or root-raised cosine) used in wireless communication [**?**]. However, the coefficients may also be provided as high-precision floating-point numbers.

This article deals with the *implementation* of such a specification as fixed-point hardware operating on low-precision data (typically 8 to 24 bits).

To specify such an implementation, a designer needs to define several parameters on top of the mathematical specification. Obviously, he needs to define the finite-precision input and output formats. He also needs to make several

This work is partly supported by the MetaLibm project (ANR-13-INSE-0007) of the French *Agence Nationale de la Recherche*.

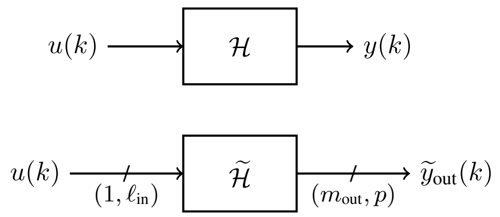


Fig. 1. The ideal filter (top) and its implementation (bottom)

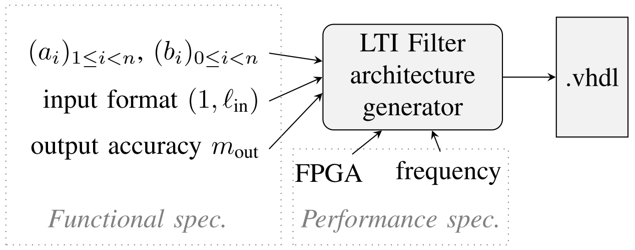

$(a_i)_{1\le i<n}$, $(b_i)_{0\le i<n}$
input format $(1, \ell_{\text{in}})$
output accuracy m_{out}
LTI Filter architecture generator
.vhdl
FPGA
frequency
Functional spec.
Performance spec.

Fig. 2. Interface to the proposed tool. The coefficients a_i and b_i are considered as real numbers: they may be provided as high-precision numbers from e.g. Matlab, or even as mathematical formulae such as `sin(3*pi/8)`. The integers m, l and p respectively denote the bit weights of the most significant and least significant bits of the input, and the least significant bit of the result. In the proposed approach, p specifies output precision, but also output accuracy.

architecture choices, some of which will impact the accuracy of the computation. For instance, each real-valued coefficient must be rounded to some internal machine format. A naive choice is to round the coefficients to the input/output format, but then for large values of n (the filter order), the result can become very inaccurate. Some design tools for filter synthesis let the designer chose an extended internal precision. The risk is then to obtain an architecture that wastes area, time and power by computing more accuracy than it can output.

The main contribution of this article is to show that such design decisions can be automated, based on the following simple claim:

An LTI architecture should be designed to compute results that are accurate to the last bit, but no more.

This claim is based on two common-sense observations. On the one hand, there is no point in designing an architecture that outputs bits which we know hold no useful information. On the other hand, there is no point in computing internally to an accuracy that we will not be able to express on the output.

Section **??** will show how this claim may be formalized and developed into a complete error analysis. This enables a very simple interface (Fig. **??**) to an LTI implementation tool. The designer may focus on those design parameters which are relevant: the (real) coefficients, and the input/output formats.

The construction of a minimal-cost architecture of proven last-bit accuracy can be fully automated out of this information.

As an illustration, an open-source tool demonstrates this implementation process for a particular hardware target: FPGAs based on Look-Up Tables (LUTs). Built upon the FloPoCo project[1], this tool automatically generates VHDL for LTI filters from the specification of Fig. **??**. It also benefits from the FloPoCo back-end framework: the generated architectures are optimized for a user-specified FPGA family, and a user-specified frequency (Fig. **??**).

This demonstrator also incorporates several architectural novelties. The constant multipliers are built using an evolution of the KCM algorithm [**?**], [**?**] that manages multiplications by a real constant without needing to truncate it first [**?**]. The summation is efficiently performed thanks to the BitHeap framework recently introduced in FloPoCo [**?**]. These technical choices lead to logic-only architectures suited even to low-end FPGAs, a choice motivated by work on implementing the ZigBee protocol standard [**?**] (some of the examples illustrating this article address this standard). However, the same philosophy could be used to build other architecture generators, for instance exploiting embedded multipliers and DSP blocks.

TODO meilleur exposé du plan quand il aura convergé.

II. Definitions and notations

A. Fixed-point formats

There are many standards for representing fixed-point data. The one we use in this work is inspired by the VHDL `sfixed` standard. For simplicity we only deal with signed fixed-point number, classically represented in two's complement. A fixed-point format is then fully specified by two integers (m, p) that respectively denote the position of the most significant and least significant bit (MSB and LSB) of the data. The respective values of the MSB and LSB are thus 2^m and 2^p. Sometimes we will call the LSB position p the *precision* of the format. The MSB position denotes its *range*. Both m or p can be negative if the format includes fractional bits. Besides, m is also the position of the sign bit. As both MSB and LSB are included, the size of a fixed-point number is $m - p + 1$. For instance, for a signed fixed-point format representing numbers in $(-1, 1)$ on 16-bit, we have one sign bit to the left of the point and 15 bits to the right, so $(m, p) = (0, -15)$.

In all the following, we note p (without a subscript) the precision of the output format, such that 2^p is the weight of the least significant bit (LSB) of $\widetilde{y}_{\text{out}}$.

B. Approximations and errors

The overall error, noted ε_{out}, of an architecture that outputs a fixed-point result $\widetilde{y}_{\text{out}}$ is defined as the difference between the computed value and its mathematical specification, noted y:

$$\varepsilon_{\text{out}} \quad = \quad \widetilde{y}_{\text{out}} - y \tag{3}$$

[1]http://flopoco.gforge.inria.fr/

More generally, in all the article, we denote ε (with some subscript) an error, which is always defined as the difference between a more accurate term and a less accurate one.

We also try to use tilded letters (e.g. $\widetilde{y}_{\text{out}}$ above) for approximate or rounded terms. This is but a convention, and the choice is not always obvious. For instance, the $u(k)$ in (**??**) are fixed-point inputs, and most certainly the result of some approximate measurement or computation. However, from the point of view of the architecture, inputs are given, so they are considered exact.

In all the following, we also note $\overline{\varepsilon}$ a bound on ε, *i.e.* the maximum value of $|\varepsilon|$.

C. Perfect and faithful rounding

The rounding of a real such as our ideal output y to the nearest fixed-point number of precision p is denoted $\circ_p(y)$. In the worst case, it entails an error $|\circ_p(y) - y| < 2^{p-1}$. For instance, rounding a real to the nearest integer ($p = 0$) may entail an error up to $0.5 = 2^{-1}$. This is a limitation of the format itself. Therefore, the best we can do, when implementing (**??**) with a precision-p output, is a *perfectly rounded* computation with an error bound $\overline{\varepsilon}_{\text{out}} = 2^{p-1}$.

Unfortunately, reaching perfect rounding accuracy may require arbitrary intermediate precision. This is not acceptable in an architecture. We therefore impose a slightly relaxed constraint: $\overline{\varepsilon}_{\text{out}} < 2^p$. We call this *last-bit accuracy*, because the error must be smaller than the value of the last (LSB) bit of the result. It is sometimes called *faithful rounding* in the literature.

Considering that the output format implies that $\overline{\varepsilon}_{\text{out}} \geq 2^{p-1}$, it is still a tight specification. For instance, if the exact y happens to be a representable precision-p number, then a last-bit accurate architecture will return exactly this value.

The main reason for chosing last-bit accuracy over perfect rounding is that, as will be shown in the sequel, it can be reached with very limited hardware overhead. Therefore, in terms of cost and efficiency, an architecture that is last-bit-accurate to p bits makes more sense than a perfectly rounded architecture to $p - 1$ bits, for the same accuracy bound 2^p.

In any case, the main conclusion of this discussion is the following: specifying the output precision (p on Fig. **??** is enough to also specify the accuracy of the implementation.

D. Worst-case peak gain of an LTI filter

The following lemma captures the amplification of a signal by an LTI filter.

Lemma II.1. *If the input $u(k)$ to an LTI filter $\mathcal{H}$ is bounded by $\overline{u}$ (*i.e.* $\forall k \quad |u(k)| \leq \overline{u}$), then the output $y(k)$ will be bounded iff the moduli of the poles of the transfer function $\mathcal{H}(z)$ are all strictly smaller than 1. This property is know as the Bounded Input Bounded Output (BIBO) stability [**?**].*

In this case, the Worst-Case Peak Gain (WCPG) $||\mathcal{H}||$ of the filter $\mathcal{H}$ is defined as the largest possible peak value of the output $y(k)$ over all the possible bounded input $u(k)$:

$$\overline{y} = ||\mathcal{H}||\overline{u} \ . \tag{4}$$

The WCPG can be computed as the ℓ_1-norm of the impulse response $h(k)$ of $\mathcal{H}$, i.e. $||\mathcal{H}|| = \sum_{k=0}^{\infty} |h(k)|$.

Proof. Due to the linearity of the filter $\mathcal{H}$ and the property of the impulse response, the output $y(k)$ is a convolution between the impulse response and the input:

$$y(k) = \sum_{l=0}^{k} h(l)u(k-l) \tag{5}$$

So

$$|y(k)| \leq \sum_{l=0}^{k} |h(l)|\overline{u} \tag{6}$$

with equality if $\forall 0 \leq l \leq k \quad u(k-l) = \text{sign}(h(l))\overline{u}$. □

The bound $\overline{y}$ is quite conservative in practice. However, it is always possible to exhibit an input $u(k)$ bounded by $\overline{u}$ such that the corresponding output is arbitrary close to its bound $||\mathcal{H}||\overline{u}$.

The value of the Worst-Case Peak Gain can be computed at any arbitrary precision [**?**].

III. Sum of products computing just right

In this section, we address the sub-problem of building a Sum of Product by Constants (SOPC) architecture, *i.e.* an architecture computing

$$y = \sum_{i=1}^{n} c_i x_i \tag{7}$$

for a set of real constants c_i, and a set of fixed-point inputs x_i, each with its fixed-point format (m_i, ℓ_i). The interface to this problem is shown on Figure **??**. In the context of an LTI filter (illustrated by Figure **??**), the c_i may be a_i or b_i, and the x_i may be either some delayed u_i, or some delayed y_i. The format of the y_i is, in general, different from that of the u_i.

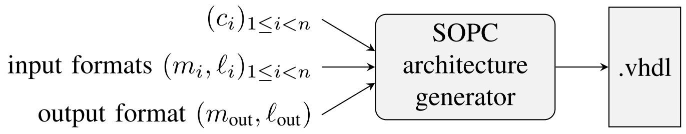

$(c_i)_{1 \leq i < n}$
input formats $(m_i, \ell_i)_{1 \leq i < n}$
output format $(m_{\text{out}}, \ell_{\text{out}})$
SOPC architecture generator
.vhdl

Fig. 3. Interface to a sum-of-product-by-constant generator

A. Datapath analysis

The fixed-point summation of the various terms $c_i x_i$ is depicted on Fig. **??**. For this figure, we take as an example a 4-input SOPC with arbitrary coefficients. As shown on the figure, the exact product of a real c_i by an input x_i may have an infinite number of bits.

1) Determining the most significant bit of $c_i x_i$: A first observation is that the MSB of the products $c_i x_i$ is completely determined by the c_i and the input fixed-point format. Indeed, we have fixed-point, hence bounded, inputs. If the domain of x_i is $x_i \in (-1, 1)$, the MSB of $c_i x_i$ is the MSB of $|c_i|$. This is illustrated by Figure **??**. In general, if $|x_i| < M$, the MSB of $c_i x_i$ will be $\lceil \log_2(|c_i|M) \rceil$.

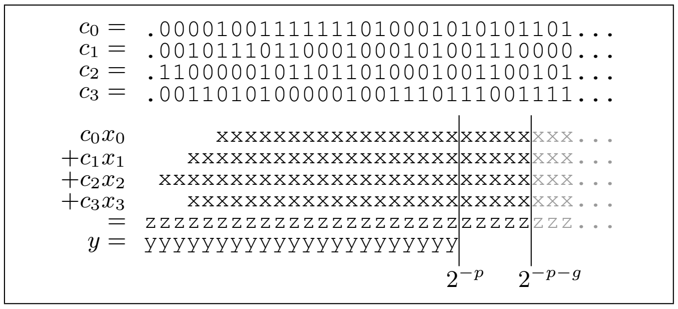

$c_0 =$ .000010011111110100010101011 01...
$c_1 =$ .00101110110001000101001110000...
$c_2 =$ .11000001011011010001001100101...
$c_3 =$ .00110101000001001110111001111...
c_0x_0 xxxxxxxxxxxxxxxxx xxxxx xxx...
$+c_1x_1$ xxxxxxxxxxxxxxxxxxx xxxxx xxx...
$+c_2x_2$ xxxxxxxxxxxxxxxxxxxxx xxxxx xxx...
$+c_3x_3$ xxxxxxxxxxxxxxxxxxx xxxxx xxx...
$=$ zzzzzzzzzzzzzzzzzzzzzz zzzzz zzz...
$y =$ yyyyyyyyyyyyyyyyyyyyyy
2^{-p}
2^{-p-g}

Fig. 4. The alignment of the $c_i x_i$ follows that of the c_i

This is obvious if $c_i x_i$ is positive. It is not true if $c_i x_i$ is negative, since in two's complement it needs to be sign-extended to the MSB of the result. However the sign extension of a signed number `sxxxx`, where `s` is the sign bit, may be performed as follows [**?**]:

$$\begin{array}{rl} & \texttt{00...0}\overline{\texttt{s}}\texttt{xxxxxxx} \\ + & \texttt{11...110000000} \\ = & \texttt{ss...ssxxxxxxx} \end{array}$$

Here $\overline{\texttt{s}}$ is the complement of `s`. The reader may check this equation in the two cases, `s` $= 0$ and `s` $= 1$. Now the variable part of the product has the same MSB as $|c_i|$, just as in the positive case.

This transformation is not for free: we need to add a constant. Fortunately, in the context of a summation, we may add in advance all these constants together. Thus the overhead cost of two's complement in a summation is limited to the addition of one single constant. In many cases, as we will see, this addition can even be merged for free in the computations of the $c_i x_i$.

2) Determining the least significant bit: error analysis for a sum of product : As we have seen, a single integer p, the weight of the least significant bit (LSB) of the output, also specifies the accuracy of the computation.

However, performing all the internal computations to this precision would not be accurate enough. Suppose for instance that we could build perfect hardware constant multipliers, returning the perfect rounding $\widetilde{p_i} = \circ_p(c_i x_i)$ of the mathematical product $c_i x_i$ to precision p. Even such a perfect multiplier would entail an error, defined as $\varepsilon_i = \widetilde{p_i} - c_i x_i$, and bounded by $\varepsilon_i < \overline{\varepsilon}_{\text{mult}} = 2^{-p-1}$. Again this rounding error is the consequence of the limited-precision output format. The problem is that in a SPC, such rounding errors add up. Let us analyze this.

The output value $\widetilde{y}$ is computed in an architecture as the sum of the $\widetilde{p_i}$. This summation, as soon as it is performed with adders of the proper size, will entail no error. Indeed, fixed-point addition of numbers of the same format may entail overflows (these have been taken care of above), but no rounding error. This enables us to write

$$\widetilde{y} \quad = \quad \sum_{i=0}^{N-1} \widetilde{p_i}, \tag{8}$$

therefore

$$\varepsilon \;=\; \sum_{i=0}^{N-1} \widetilde{p_i} - \sum_{i=0}^{N-1} c_i x_i \;=\; \sum_{i=0}^{N-1} \varepsilon_i \quad . \tag{9}$$

Unfortunately, in the worst case, the sum of the ε_i can come close to $N\overline{\varepsilon}_{\text{mult}}$, which, as soon as $N > 2$, is larger than 2^{-p}: this naive approach is not last-bit accurate.

The solution is, however, very simple. A slightly larger intermediate precision, with g additional bits ("guard" bits) may be used. The error of each multiplier is now bounded by $\overline{\varepsilon}_{\text{mult}} = 2^{-p-1-g}$. It can be made arbitrarily small by increasing g. Therefore, there exists some g such that $N\overline{\varepsilon}_{\text{mult}} < 2^{-p-1}$ Actually, this is true even if a less-than-perfect multiplier architecture is used, as soon as we may compute a bound $\overline{\varepsilon}_{\text{mult}}$ of its accuracy, and this bound is proportional to 2^{-g}.

However, we now have another issue: the intermediate result now has g more bits at its LSB than we need. It therefore needs itself to be rounded to the target format. This is easy, using the identity $\circ(x) = \lfloor x + \frac{1}{2} \rfloor$: rounding to precision 2^{-p} is obtained by first adding 2^{-p-1} (this is a single bit) then discarding bits lower than 2^{-p}. However, in the worst case, this will entail an error $\varepsilon_{\text{final rounding}}$ of at most 2^{-p-1}.

To sum up, the overall error of a faithful architecture SCP is therefore

$$\varepsilon \;=\; \varepsilon_{\text{final rounding}} + \sum_{i=0}^{N-1} \varepsilon_i \;<\; 2^{-p-1} + N\overline{\varepsilon}_{\text{mult}} \tag{10}$$

and this error can be made smaller than 2^{-p} as soon as we are able to build multipliers such that

$$N\overline{\varepsilon}_{\text{mult}} < 2^{-p-1} \quad . \tag{11}$$

B. LUT-based architectures computing just right for FPGAs

All the previous was quite independent of the target technology: it could apply to ASIC synthesis as well as FPGA. Also, a very similar analysis can be developed for an inner-product architecture where the c_i are not constant.

The remainder of this section, conversely, is more focused on a particular context: LUT-based SCP architectures for FPGAs. It explores architectural means to reach last-bit accuracy at the smallest possible cost ("computing just right").

On most FPGAs, the basic logic element is the look-up-table (LUT), a small memory addressed by α bits. On the current generation of FPGAs, $\alpha = 6$.

1) Perfectly rounded constant multipliers: As we have a finite number of possible values for x_i, it is possible to build a perfectly rounded multiplier by simply tabulating all the possible products. The precomputation of table values must be performed with large enough accuracy (using multiple-precision software) to ensure the correct rounding of each entry. This even makes perfect sense for small input precisions on recent FPGAs: if x_i is a 6-bit number, each output bit of the perfectly rounded product $c_i x_i$ will consume exactly one 6-input LUTs. For 8-bit inputs, each bit consumes only 4 LUTs. In general, for $(6 + k)$-bit inputs, each output bit consumes 2^k 6-LUTs: this approach scales poorly to larger inputs. However, perfect rounding to $p + g$ bits means a maximum error smaller than an half-LSB: $\overline{\varepsilon}_{\text{mult}} = 2^{-p-g-1}$. Note that for real-valued c_i, this is more accurate than using a perfectly rounded multiplier that inputs $\circ_p(c_i)$: this would accumulate two successive rounding errors.

2) Table-based constant multipliers for FPGAs: For larger precisions, we may use a variation of the KCM technique, due to Chapman [**?**] and further studied by Wirthlin [**?**]. The original KCM method addresses the multiplication by an integer constant. We here present a variation that performs the multiplication by a *real* constant.

This method consists in breaking down the binary decomposition of an input x_i into n chunks d_{ik} of α bits. With the input size being $m+f$, we have $n = \lceil (m+f)/\alpha \rceil$ such chunks ($n = 3$ on Figure **??**). Mathematically, this is written

$$x_i \;=\; \sum_{k=1}^{n} 2^{-k\alpha} d_{ik} \quad \text{where} \quad d_{ik} \in \{0, ..., 2^{\alpha} - 1\} \quad . \tag{12}$$

The product becomes

$$c_i x_i \;=\; \sum_{k=1}^{n} 2^{-k\alpha} c_i d_{ik} \quad . \tag{13}$$

Since each chunk d_{ik} consists of α bits, where α is the LUT input size, we may tabulate each product $c_i d_{ik}$ in a look-up table that will consume exactly one α-bit LUT per output bit. This is depicted on Figure **??**. Of course, $c_i d_{ik}$ has an infinite number of bit in the general case: as previously, we will round it to precision 2^{-p-g}. In all the following, we define $\widetilde{t_{ik}} = \circ_p(c_i d_{ik})$ this rounded value (see Figure **??**).

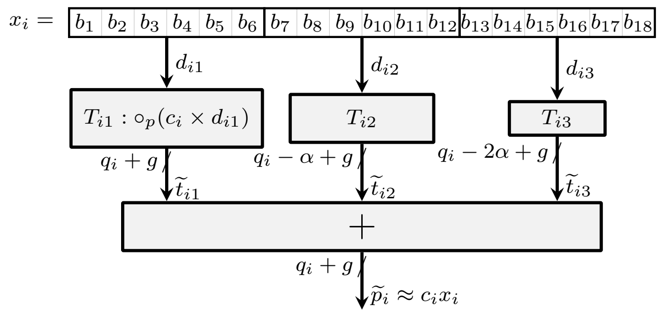
Contrary to classical (integer) KCM, all the tables do not consume the same amount of resources. The factor $2^{-k\alpha}$ in (**??**) shifts the MSB of the table output $\widetilde{t_{ik}}$, as illustrated by Figure **??**.

Here also, the fixed-point addition is errorless. The error of such a multiplier therefore is the sum of the errors of the n tables, each perfectly rounded:

$$\varepsilon_{\text{mult}} < n \times 2^{-p-g-1} \quad . \tag{14}$$

This error is proportional to 2^{-g}, so can made as small as needed by increasing g.

3) Computing the sum: In FPGAs, each bit of an adder also consumes one LUT. Therefore, in a KCM architecture, the LUT cost of the summation is expected to be roughly proportional to that of the tables. However, this can be



$x_i =$
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15} b_{16} b_{17} b_{18}
d_{i1}
d_{i2}
d_{i3}
$T_{i1} : \circ_p(c_i \times d_{i1})$
T_{i2}
T_{i3}
$q_i + g$
$q_i - \alpha + g$
$q_i - 2\alpha + g$
$\widetilde{t}_{i1}$
$\widetilde{t}_{i2}$
$\widetilde{t}_{i3}$
$+$
$q_i + g$
$\widetilde{p}_i \approx c_i x_i$

Fig. 5. The FixRealKCM method when x_i is split in 3 chunks

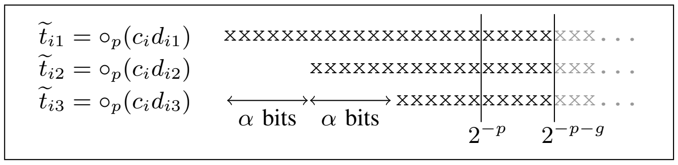

$\widetilde{t}_{i1} = \circ_p(c_i d_{i1})$
$\widetilde{t}_{i2} = \circ_p(c_i d_{i2})$
$\widetilde{t}_{i3} = \circ_p(c_i d_{i3})$
α bits
α bits
2^{-p}
2^{-p-g}

Fig. 6. Aligment of the terms in the KCM method

Half-Sine Root-Raised Cosine

Fig. 7. Bit heaps for two 8-tap, 12-bit FIR filters generated for Virtex-6

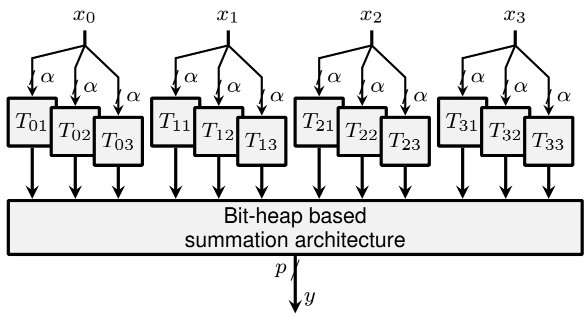

x_0
x_1
x_2
x_3
α
T_{01} T_{02} T_{03}
T_{11} T_{12} T_{13}
T_{21} T_{22} T_{23}
T_{31} T_{32} T_{33}
Bit-heap based summation architecture
p
y

Fig. 8. KCM-based SPC architecture for $N = 4$, each input being split into 3 chunks

improved by stepping back and considering the summation at the SPC level. Indeed, our faithful SPC result is now obtained by computing a double sum:

$$\widetilde{y} = \circ_p \left(\sum_{i=0}^{N-1} \sum_{k=1}^{n} 2^{-k\alpha} \widetilde{t}_{ik} \right) \tag{15}$$

Using the associativity of fixed-point addition, this summation can be implemented very efficiently using compression techniques developed for multipliers [?] and more recently applied to sums of products [?], [?]. In FloPoCo, we may use the bit-heap framework introduced in [?]. Each table throws its $\widetilde{t}_{ik}$ to a bit-heap that is in charge of performing the final summation. The bit-heap framework is naturally suited to adding terms with various MSBs, as is the case here.

This is illustrated on Fig. **??**, which shows the bit heaps for two classical filters from [?]. On these figures, we have binary weights on the horizontal axis, and the various terms to add on the vertical axis. These figure are generated by FloPoCo before bit heap compression.

We can see that the shape of the bit heap reflects the various MSBs of the c_i. One bit heap is higher than the other one, although they add the same number of product and each product should be decomposed into the same number of KCM tables. This is due to special coefficient values that lead to specific optimizations. For instance $c_i = 1$ or $c_i = 0.5$ lead to a single addition of x_i to the bit heap; $c_i = 0$ leads to nothing.

4) Managing the constant bits: Actually there are two more terms to add to the summation of Eq. (**??**): the rounding bit 2^{-p-1}, necessary for the final rounding-by-truncation as explained in Section **??**, and the sum of all the sign-extension constants introduced in Section **??**. These bits, especially the rounding bit, are visible on Fig. **??**.

There is a minor optimization to do here: these constant bits form a binary constant that can be added (at table-filling time) to all the values of one of the tables, for instance T_{00}. Then the rounding bit will be added for free. The sign extension constants may add a few MSB bits to the table output, hence increase its cost by a few LUTs. Still, with this trick, the summation to perform is indeed given by Eq. (**??**), and the final rounding, being a simple truncation, is for free. This optimization will be integrated to our implementation before publication.

Finally, the typical architecture generated by our tool is depicted by Figure **??**.

IV. Error analysis of direct-form LTI filter implementations

This section shows how to obtain an implementation of the mathematical definition (**??**) in fixed-point with last-bit accuracy on the computed result with respect to this mathematical definition. This computation is performed by the abstract architecture of Figure **??**.

The architecture of Figure **??** computes an approximate value $\widetilde{y}_{\text{out}}(k)$ that differs from the idead value defined by (**??**) because of several occurences of discretization and rounding. Formally, we refine the definition of the overall evaluation error as

$$\varepsilon_{\text{out}}(k) \quad = \quad \widetilde{y}_{\text{out}}(k) - y(k) \tag{16}$$

Let us now decompose this error into its sources.

A. Final rounding of the internal format

The architecture needs to internally use a fixed-point format that offers extended precision with respect to the input/output format. From another point of view, this internal format offers g additional LSB bits (usually called *guard bits*) in which rounding errors may accumulate without touching the output bits. The sequel will show more formally how to compute an optimal value of this architectural parameter g. Anyway we need to round the internal result from this extended format to the output format (in the "final round" box on Figure **??**). This entails an additional error ε_{f}, formally defined as

$$\varepsilon_{\text{f}}(k) = \widetilde{y}_{\text{out}}(k) - \widetilde{y}(k) \quad . \tag{17}$$

This error may be bounded by $\overline{\varepsilon}_{\text{f}} = 2^{p-1}$ (round to nearest).

Remark that we feed back the internal result $\widetilde{y}(k)$, not the output result $\widetilde{y}_{\text{out}}(k)$. This prevents an amplification of $\varepsilon_{\text{f}}(k)$ by the feedback loop that could compromise the goal of faithful rounding.

B. Rounding and quantization errors in the sum of products

As the coefficients a_i and b_i are real numbers, they must be rounded to some finite value (quantization) before the multiplication can take place. Then, the multiplication and the summation may themselves involve rounding errors. Managing all these rounding errors will be the subject of section **??**, which will show how to build an architecture that achieves a

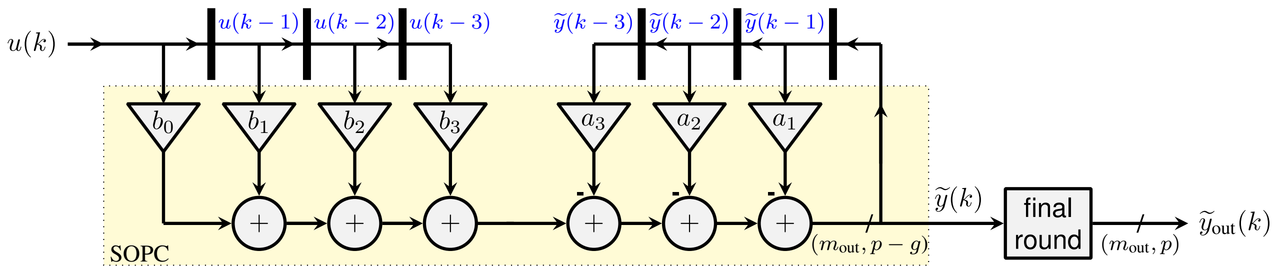


Fig. 9. Abstract architecture for the direct form realization of an LTI filter

given accuracy goal at the minimum cost. For now, we may summarize all these errors in a single term $\varepsilon_\text{r}(k)$ mathematically defined as

$$\varepsilon_\text{r}(k) \quad = \quad \widetilde{y}(k) - \left(\sum_{i=0}^{n} b_i u(k-i) - \sum_{i=1}^{n} a_i \widetilde{y}(k-i)\right) \tag{18}$$

This equation should be read as follows: $\varepsilon_\text{r}(k)$ measures how much a result $\widetilde{y}(k)$ computed by the SOPC architecture diverges from that computed by an ideal SOPC (that would use the infinitely accurate coefficients a_i and b_i, and be free of rounding errors), this ideal SOPC being applied on the same inputs $u(k-i)$ and $\widetilde{y}(k-i)$ as the architecture.

C. Error amplfication in the feedback loop

The input signal $u(k)$ can be considered exact, in the sense that whatever error it may carry is not due to the filter under consideration. However, the feedback signal $\widetilde{y}(k)$ that is input to the computation differs from the ideal $y(k)$. Let us now define $\varepsilon_\text{t}(k)$ as the error of $\widetilde{y}(k)$ with respect to $y(k)$:

$$\varepsilon_\text{t}(k) \quad = \quad \widetilde{y}(k) - y(k). \tag{19}$$

This error is potentially amplified by the architecture.

Using (**??**), let us rewrite $\widetilde{y}(k-i)$ in the right-hand side of (**??**):

$$\begin{aligned}\varepsilon_\text{r}(k) &= \widetilde{y}(k) - \sum_{i=0}^{n} b_i u(k-i) + \sum_{i=1}^{n} a_i y(k-i) \\ &\quad + \sum_{i=1}^{n} a_i \varepsilon_\text{t}(k-i) \\ &= \widetilde{y}(k) - y(k) + \sum_{i=1}^{n} a_i \varepsilon_\text{t}(k-i) \quad \text{using (??)} \\ &= \varepsilon_\text{t}(k) + \sum_{i=1}^{n} a_i \varepsilon_\text{t}(k-i) \qquad \text{using (??)}. \end{aligned} \tag{20}$$

If we rewrite (**??**) as

$$\varepsilon_\text{t}(k) \quad = \quad \varepsilon_\text{r}(k) - \sum_{i=1}^{n} a_i \varepsilon_\text{t}(k-i) \tag{21}$$

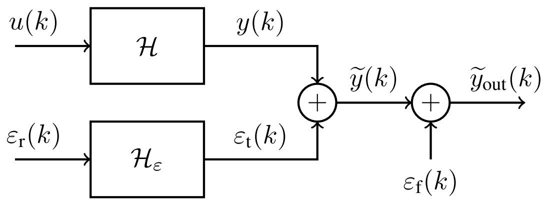
we obtain the equation of an LTI filter inputting $\varepsilon_\text{r}(k)$ and outputting $\varepsilon_\text{t}(k)$, whose transfer function is

$$\mathcal{H}_\varepsilon(z) = \frac{1}{1 + \sum_{i=1}^{n} a_i z^{-i}} \quad . \tag{22}$$

Finally, (**??**) can be rewritten

$$\widetilde{y}(k) \quad = \quad y(k) + \varepsilon_\text{t}(k) \tag{23}$$

which is illustrated by Figure **??**.



$u(k)$
$\mathcal{H}$
$y(k)$
$\widetilde{y}(k)$
$\widetilde{y}_{\text{out}}(k)$
$\varepsilon_\text{r}(k)$
$\mathcal{H}_\varepsilon$
$\varepsilon_\text{t}(k)$
$\varepsilon_\text{f}(k)$

Fig. 10. A signal view of the error propagation with respect to the ideal filter

We can now apply Lemma **??** to $\mathcal{H}_\varepsilon$ with input ε_r in order to bound ε_t by

$$\overline{\varepsilon}_\text{t} \quad = \quad ||\mathcal{H}_\varepsilon|| \overline{\varepsilon}_\text{r} \ . \tag{24}$$

Therefore, we can also keep $\overline{\varepsilon}_\text{t}$ as low as needed by increasing the internal precision to reduce $\overline{\varepsilon}_\text{r}$.

D. Putting it all together

We may now rewrite (**??**) as

$$\begin{aligned}\varepsilon_\text{out}(k) &= \widetilde{y}_\text{out}(k) - \widetilde{y}(k) + \widetilde{y}(k) - y(k) \\ &= \varepsilon_\text{f}(k) + \varepsilon_\text{t}(k)\end{aligned} \tag{25}$$

hence

$$\begin{aligned}\overline{\varepsilon}_\text{out} &= \overline{\varepsilon}_\text{f}(k) + \overline{\varepsilon}_\text{t} \\ &= \overline{\varepsilon}_\text{f}(k) + ||\mathcal{H}_\varepsilon|| \overline{\varepsilon}_\text{r}\end{aligned} \tag{26}$$

The accuracy target for faithful rounding is $\overline{\varepsilon}_\text{out} = 2^p$. The final rounding implies an error bounded by $\overline{\varepsilon}_\text{f} = 2^{p-1}$ To achieve faithful rounding, it therefore suffices that the error $\overline{\varepsilon}_\text{t}$ of the filter before final rounding is bounded by 2^{-p-1}. This leads to the following constraint on $\overline{\varepsilon}_\text{r}$:

$$\overline{\varepsilon}_\text{r} < \frac{2^{p-1}}{||\mathcal{H}_\varepsilon||} \quad . \tag{27}$$

This constraint finally translates to the LSB $p-g$ of the intermediate result as follows. Assuming we build an SOPC faithful to $p-g$, we will have $\overline{\varepsilon}_\text{r} < 2^{p-g}$. Therefore the smallest number of guard bits ensuring that the constraint (**??**) holds is

$$g = 1 + \lceil \log_2 ||\mathcal{H}_\varepsilon|| \rceil \quad . \tag{28}$$

Taps	I/O size	Speed	Area
8	12 bits	4.4 ns (227 Mhz) 1 cycle @ 344 Mhz	564 LUT 594 LUT 32 Reg.
8	18 bits	5.43ns (184 MHz) 2 cycles @ 318 MHz	1325 LUT 1342 LUT 92Reg.
16	12 bits	5.8 ns (172 Mhz) 1 cycle @ 289 Mhz	1261 LUT 1257 LUT 41 Reg.
16	18 bits	7.3 ns (137 MHz) 2 cycles @ 265 MHz	2863 LUT 2810 LUT 120 Reg.

Note: the register count does not include the input shift register.

TABLE I
SYNTHESIS RESULTS OF A ROOT-RAISED COSINE FIR.

Meanwhile, the MSB of the intermediate format (which is the same as that of the result) is completely determined by the constants:

$$m_{\text{out}} = \lceil \log_2 ||\mathcal{H}|| \rceil \ . \tag{29}$$

Some overflows may occur in the internal computation. However in which case the computation is performed modulo $2^{m_{\text{out}}}$, and the result will be correct.

V. IMPLEMENTATION AND RESULTS

The method described in this paper is implemented as the `FixFIR` operator of FloPoCo. `FixFIR` offers the interface shown on Fig. **??**, and inputs the c_i as arbitrary-precision numbers. On top of it, the operators `FixHS` and `FixRRCF` simply evaluate the c_i using the textbook formula for Half-Sine and Root-Raised Cosine respectively, and feed them to `FixFIR`. We expect many more such wrappers could be written in the future.

`FixFIR`, like most FloPoCo operators, was designed with a testbench generator [**?**]. All these operators reported here have been checked for last-bit accuracy by extensive simulation.

Table **??** reports synthesis results for architectures generated by `FixRRCF`, as of FloPoCo SVN revision 2666. All the results were obtained for Virtex-6 (6vhx380tff1923-3) using ISE 14.7. These results are expected to improve in the future, as the (still new) bit-heap framework is tuned in FloPoCo.

A. Analysis of the results

As expected, the dependency of the area to the number of taps is almost linear. On the one hand, more taps increase the number of needed guard bits. On the other hand, more taps mean a larger bit heap which exposes more opportunities for efficient compression. However, there is also the dependency of the c_i themselves to the number of taps.

Conversely, the dependency of area on precision is more than linear. It is actually expected to be almost quadratic, as can be observed by evaluating the number of bits tabulated for each multiplier (see Fig. **??**).

Interestingly, the dependency of the delay to the number of taps is clearly sub-linear. This is obvious from Fig. **??**. All the table accesses are performed in parallel, and the delay is dominated by the compression delay, which is logarithmic in the bit-heap height [**?**].

B. Comparison to a naive approach

In order to evaluate the cost of computing just right, we built a variant of the `FixFIR` operator following a more classical approach. In this variant, we first quantize all the coefficients to precision p (*i.e* such that their LSB has weight 2^{-p}). We obtain fixed-point constants, and we then build an architecture that multiplies these constants with the inputs using the classical (integer) KCM algorithm.

A simple error analysis shows that the mere quantization to precision p is already responsible for the loss of two bits of accuracy on $\widetilde{y}$. The exact error due to coefficient quantization depends on the coefficients themselves but space is missing here to detail that. Then, the multiplications are exact. However, as illustrated by Figure **??**, their exact results extend well below precision 2^p: they have to be rounded. We just truncate the output of each KCM operators to 2^{-p}. This more than doubles the overall error. It also allows the synthesis tools to optimize out the rightmost 6 bits of Fig. **??**. Then these truncated products are summed using a sequence of $N-1$ adders of precision p.

The results are visible in Table **??**. The naive approach leads to a much smaller, but slower design. The area difference doesn't come from the tables, if we compare Fig. **??** and Fig. **??**. It comes from the summation, which is performed on g more bits in the last-bit accurate approach. Besides, only the last-bit accurate approach uses a bit heap, whose compression heuristics are currently optimized for speed more than for area.

However, this comparison is essentially meaningless, since the two architectures are not functionally equivalent: The proposed approach is accurate to 12 bits, while the naive approach loses more than 3 bits to quantization and truncation. Therefore, a meaningful comparison is the proposed approach, but accurate to $p = 9$, which we also show in Table **??**. This version is better than the naive one in both area and speed.

Since we are comparing two designs by ourselves, it is always possible to dispute that we deliberately sabotaged our naive version. And in a sense, we did: We designed it as small as possible (no guard bits, truncation instead of rounding, etc). Short of designing a filter that returns always zero (very poor accuracy, but very cheap indeed), this is the best we can do. But all these choices are highly disputable, in particular because they impact accuracy.

All this merely illustrates the point we want to make: it only makes sense to compare designs of equivalent accuracies. And the only accuracy that makes sense is last-bit accuracy, for the reasons exposed in **??**. We indeed put our best effort in designing the last-bit accurate solution, and we indeed looked for the most economical way of achieving a given accuracy. This is comforted by this comparison. Focussing on last bit accuracy enables us to compute right, and just right.

For the sake of completeness, we also implemented a version using DSP blocks, reported in Table **??**. Considering the large internal precision of DSP blocks, it costs close to nothing to design this version as last-bit accurate for $p = 12$ or $p = 18$ (in the latter case, provided the constant is input with enough guard bits on the 24-bit input of the DSP block). This is very preliminary work, however. We should offer the choice between either an adder tree with a shorter delay, or a

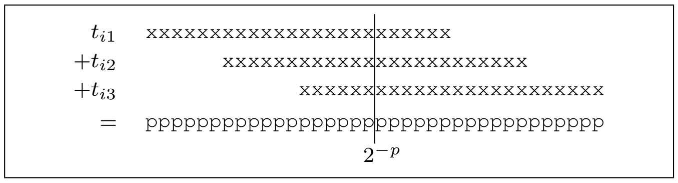

t_{i1}
xxxxxxxxxxxxxxxxxxxxxxxxxx
$+t_{i2}$
xxxxxxxxxxxxxxxxxxxxxxxxxx
$+t_{i3}$
xxxxxxxxxxxxxxxxxxxxxxxxxx
$=$
pppppppppppppppppppppppppppppppppppp
2^{-p}

Fig. 11. Using integer KCM: $t_{ik} = \circ_p(c_i) \times d_{ik}$. This multiplier is both wasteful, and not accurate enough.

Method	Speed	Area	accuracy
$p = 12$, proposed	4.4 ns (227 Mhz)	564 LUT	$\overline{\varepsilon} < 2^{-12}$
$p = 12$, naive	5.9 ns (170 Mhz)	444 LUT	$\overline{\varepsilon} > 2^{-9}$
$p = 9$, proposed	4.12 ns (243 Mhz)	380 LUT	$\overline{\varepsilon} < 2^{-9}$
$p = 12$, DSP-based	9.1 ns (110 Mhz)	153 LUT, 7 DSP	$\overline{\varepsilon} < 2^{-9}$

TABLE II
THE ACCURACY/PERFORMANCE TRADE-OFF ON 8-TAP, 12 BIT ROOT-RAISED COSINE FIR FILTERS.

long DSP chain which would be slow, but consume no LUT. We should also compare to Xilinx CoreGen FIR compiler, which generates only DSP-based architectures. Interestingly, this tool requires the user to quantize the coefficients and to take all sorts of decisions about the intermediate accuracies and rounding modes.

CONCLUSION

This paper claims that sum-of-product architectures should be last bit accurate, and demonstrates that this has two positive consequences: It gives a much clearer view on the trade-off between accuracy and performance, freeing the designer from several difficult choices. It actually leads to better solutions by enabling a "computing just right" philosophy. All this is demonstrated on an actual open-source tool that offers the highest-level interface.

Future work include several technical improvements to the current implementation, such as the exploitation of symmetries in the coefficients or optimization of the bit heap compression.

Beyond that, this work opens many perspectives.

As we have seen, fixed-point sum of products and sum of squares could be optimized for last-bit accuracy using the same approach.

We have only studied one small corner of the vast literature about filter architecture design. Many other successful approaches exist, in particular those based on multiple constant multiplication (MCM) using the transpose form (where the registers are on the output path) [?], [?], [?], [?], [?]. A technique called Distributed Arithmetic, which predates FPGA [?], can be considered a generalization of the KCM technique to the MCM problem. From the abstract of [?] that, among other things, compares these two approaches, "if the input word size is greater than approximately half the number of coefficients, the LUT based multiplication scheme needs less resources than the DA architecture and vice versa". Such a rule of thumb (which of course depends on the coefficients themselves) should be reassessed with architectures computing just right on each side. Most of this vast literature treats accuracy after the fact, as an issue orthogonal to architecture design.

A repository of FIR benchmarks exists, precisely for the purpose of comparing FIR implementations [?]. Unfortunately, the coefficients there are already quantized, which prevents a meaningful comparison with our approach. Few of the publications they mention report accuracy results. However, cooperation with this group should be sought to improve on this.

We have only considered here the implementation of a filter once the c_i are given. Approximation algorithms, such as Parks-McClellan, that compute these coefficients, essentially work in the real domain. The question they answer is "what is the best filter with real coefficients that matches this specification". It is legitimate to wonder if asking the question: "what is the best filter with low-precision coefficients" could not lead to a better result.

Still in filter design, the approach presented here should be extended to infinite impulse response (IIR) filters. There, a simple worst-case analysis (as we did for SPC) doesn't work due to the infinite accumulation of error terms. However, as soon as the filter is stable (i.e. its output doesn't diverge), it is possible to derive a bound on the accumulation of rounding errors [?]. This would be enough to design last-bit accurate IIR filters computing just right.