Towards Hardware IIR Filters Computing Just Right: Direct Form I Case Study

Anastasia Volkova, Matei Istoan, Florent de Dinechin, Thibault Hilaire

Towards Hardware IIR Filters Computing Just Right: Direct Form I Case Study

Anastasia Volkova, Matei Istoan, Florent de Dinechin, Thibault Hilaire

Abstract—Linear Time Invariant (LTI) filters are often specified and simulated using high-precision software, before being implemented in low-precision fixed-point 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 converted to the low-precision output format with an error smaller than its least significant bit. This simple specification guarantees the numerical quality of the hardware, even for critical LTI systems. Besides, it is possible to derive the optimal values of all the internal data formats that ensure that the specification is met. This requires a detailed error analysis that captures not only the quantization and rounding errors, but also their infinite accumulation in recursive filters. This generic methodology is detailed for the case of low-precision LTI filters in the Direct Form I implemented in FPGA logic. It is demonstrated by a fully automated and open-source architecture generator tool, and validated on a range of Infinite Impulse Response filters.

Index Terms—digital filters, computer arithmetic, fixed-point, error analysis, constant multiplication, FPGA

1 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 in the frequency domain as a transfer function $\mathcal{H}(z)$:

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

where $n_a \geq n_b$ and n_a is the order of the filter.

In the time domain, a filter specified with $\mathcal{H}(z)$ may be evaluated with various algorithms [1]. In this work we consider the Direct Form I (DFI) [1] algorithm that relates the output signal $y(k)$ and the input signal $u(k)$ in the following way:

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

The DFI algorithm is the simplest algorithm for the evaluation of Infinite Impulse Response (IIR) filters, since it directly uses the coefficients of the transfer function. Small-order DFIs are often used as basic building blocks for sections of second-order algorithms [1]. Due to its high numerical sensitivity, DFI is rarely used for high order filters. A contribution of this article is a methodology that protects designers from such sensitivity issues. This methodology is not restricted to DFI and could be applied to other filter algorithms (cascade and/or parallel decomposition, state-space, ρ-operator forms, etc). This article focuses on DFI because of its simplicity.

Equation (1) or (2), along with a definition of each coefficient a_i and b_i, constitute the *mathematical specification* of the filter. This article deals with the *implementation* of such a specification as fixed-point hardware operating on low- to moderate-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 architectural 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 sensitive filters, the result can become very inaccurate. Design tools for filter synthesis such as Matlab's *fdatool* let the designer chose an extended internal precision. The risk is then to obtain an architecture that internally computes more accurately than it can output, thus wasting area, time and power.

The goal of the present work is to relieve designers from such low-level architectural decisions, allowing them to focus on the relevant parameters shown in Figure 1: the (real) coefficients, the fixed-point format of the input, and the precision expected for the output. For this, the precision

- *A. Volkova is with Univ Lyon, Inria, CNRS, ENS de Lyon, Université Claude Bernard Lyon 1, LIP UMR 5668, France. E-mail: see www.avolkova.org*
- *M. Istoan is with Imperial College London, Electrical and Electronic Engineering Department, United Kingdom. E-mail: m.istoan@imperial.ac.uk*
- *F. de Dinechin is with Univ Lyon, INSA Lyon, INRIA, CITI, France. E-mail: florent.de-dinechin@insa-lyon.fr*
- *T. Hilaire is with Sorbonne Université, Paris, France, and also with Inria and LRI Université Paris-Saclay, Orsay, Fance. E-mail: thibault.hilaire@lip6.fr*

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

Manuscript received xxx, revised yyy.

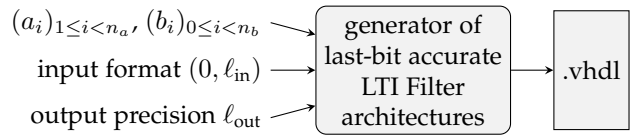


Fig. 1. Interface of the proposed tool.

of the output must also specify its accuracy. In this work, the following specification is used: *An implemented filter must behave as if each result was computed with infinite accuracy with respect to* (2)*, then converted only once to the low-precision output format, with an error smaller than the weight of its least significant bit (LSB)*.

The main contribution of this article is to show that for such *last-bit accurate* implementations, near-optimal values of architectural parameters such as coefficient quantization and datapath bit-widths can be automated.

This is achieved thanks to a complete and rigorous analysis of the rounding errors and their amplification through the feedback loop, building upon recent work on reliably bounding error propagation through filters [2], [3].

This methodology is demonstrated in an open source tool that builds DFI implementations for a particular hardware target: FPGAs based on Look-Up Tables (LUTs). Built upon the FloPoCo project[1], this tool has the interface shown on Figure 1. It also incorporates several architectural novelties. The constant multipliers are built using an evolution of the KCM algorithm [4], [5] that manages the multiplication by a real constant without needing to quantize it first [6]. The summation is efficiently performed thanks to the BitHeap framework recently introduced in FloPoCo [7], [8]. These technical choices lead to logic-only architectures suited even to low-end FPGAs. However, the same philosophy could be used to build other architecture generators, for instance exploiting embedded multipliers and DSP blocks.

After a survey of the state of the art in Section 2, Section 3 provides some prerequisites on the target arithmetic and error models. A complete methodology on the rigorous error analysis is presented in the Section 4. Section 5 gives details on the architecture of the arithmetic units and their rounding error analysis. Finally, Section 6 demonstrates implementation results, and Section 7 discusses the perspectives of this work.

2 Previous works and state of the art

The problem addressed in this article is an instance of the word-length optimization (WLO) problem [9] [10] [11] [12] [13] [14]. The target here is hardware, where each signal may use a different data format (in software, the available formats are constrained by the processor's capabilities). WLO addresses two issues: 1/ *range analysis* consists in studying the data ranges, and choosing the data formats of each signal to avoid overflow. In fixed point, range analysis constrains the most significant bit (MSB) of each signal. 2/ *precision analysis* consists in studying the impact of rounding errors, and choosing the formats to avoid underflows and rounding errors. In fixed point, precision analysis constrains the least significant bit (LSB).

WLO techniques can be based on simulations (in which case it is difficult to provide guarantees beyond the datasets used for the simulation), or analytical, as in the present work. Analytical methods are faster and can provide guarantees, however these guarantees can be pessimistic.

Some analytical methods [9], [14] model output errors as additive noise with statistical hypotheses. In this case, the goal of WLO is to maximize the Signal Quantization to Noise Ratio (SQNR), which is the ratio of the variance of the output signal by the variance of the output error. SQNR does not give any guarantee on the accuracy of the output signal, it merely provides an idea on the typical number of meaningful bits in the output. Other analytical methods, including the one presented here, use worst-case bounds for the rounding and approximation errors. This enables strict guarantees on the function of an architecture, for instance last-bit accuracy. Note that it is easy to evaluate a last-bit accurate implementation in terms of SQNR: it behaves as if the exact computation was rounded once; with the usual statistical hypothesis that this rounding is uniformly distributed, the output error variance can classically be evaluated as $2^{2\ell_{\text{out}}}/12$.

Worst-case analytical WLO methods in the literature have used interval arithmetic (IA), affine arithmetic (AA) [10], [13], [14], its generalization to higher-order error polynomials [15] and even SAT-modulo theory (SMT) [11]. IA and AA are mostly used for precision analysis, and SMT and IA for range analysis.

Previous worst-case analytical methods sometimes do not support feedback loops [13]. Generic techniques such as abstract interpretation [16] combined with IA or AA, may provide guarantees on programs with loops [17], but these guarantees will be very pessimistic for sensitive IIR filters.

This work presents a worst-case analytical WLO approach that provides tight (not uselessly pessimistic) and strong (independent of the sensitivity) guarantees on the results of LTI filters with feedback loop.

To the best of our knowledge, the state of the art here is [14] which attempts to solve exactly this problem. The present work improves on [14] in the following respects:

1/ We claim to provide a better problem formulation. In particular, although [14] correctly compute the bitwidths of the data on the feedback loops, they must output all these bits. The output of their filters therefore includes bits that only hold noise. As Section 6 will show, this easily doubles the output size for low-precision, high-sensitivity filters.

2/ We are more rigorous in the implementation. In LTI filters, both range and errors are infinite series, which in [14] are truncated in a non-reliable way that may underestimate the number of terms for sensitive filters. In particular, the series is evaluated by a floating-point Matlab computation, and the accumulation of rounding errors in this computation does not seem to be taken into account. The present work is very rigorous in this respect.

3/ We merge the fine-tuning of constant multiplier architectures with the WLO problem. For a real constant coefficient C and an input X, [14] will classically quantize C to some C_q, then quantize (round) the product C_qX: this involves two error terms. The present work uses an architec-

1. http://flopoco.gforge.inria.fr/, version 4.1.3.

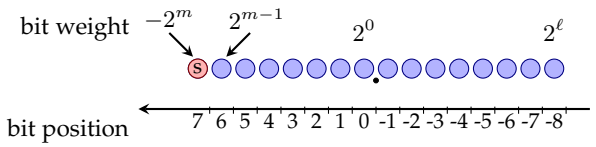


Fig. 2. The bits of a two's complement fixed-point format, here $(m, \ell) = (7, -8)$.

ture that directly quantizes the product CX, with only one error term. With fewer error terms, the error analysis will be less pessimistic. It also saves hardware (roughly speaking, it avoids computing the lower bits of C_qX that will be later lost to the rounding).

4/ This work is supported by a complete VHDL generator integrated in a mature open-source project.

3 Definitions and Notations

3.1 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 numbers, classically represented in two's complement. As illustrated by Figure 2, a fixed-point format is then 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, whose weight is -2^m (due to two's complement representation). The LSB position ℓ denotes the *precision* of the format. The MSB position m denotes its *range*. The word-length of the fixed-point number in the format (m, ℓ) is $m - \ell + 1$. Note that the range of numbers that can be represented with the format (m, ℓ) is $[-2^m, 2^m - 2^\ell]$. For instance, for a signed fixed-point format representing numbers in $(-1, 1)$ on 16 bits, we have one "sign bit" to the left of the point and 15 bits to the right, so $(m, \ell) = (0, -15)$. Then, we can represent numbers in $[-1, 1 - 2^{-15}]$ with a step of 2^{-15}.

3.2 Approximations and errors

Due to the finite precision issues, the exact filter $\mathcal{H}$, with exact output y, cannot usually be implemented. An actually implemented filter will produce a finite precision output $\widetilde{y}_{\text{out}}$ (see Figure 3). The overall error of such an implementation, denoted ε_{out}, is defined as the difference between the computed value and its mathematical specification:

$$\varepsilon_{\text{out}}(k) \quad = \quad \widetilde{y}_{\text{out}}(k) - y(k). \tag{3}$$

More generally, throughout the article, we denote ε (with some subscript) an error, 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 (2) 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.

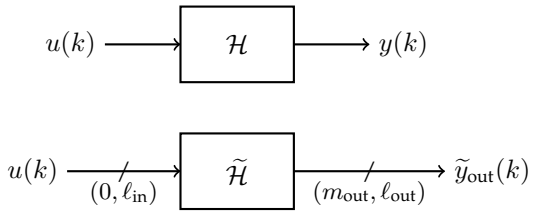


Fig. 3. The ideal filter (top) and its implementation (bottom)

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

3.3 Perfect rounding vs. last-bit accuracy

The rounding of the ideal, real-valued output y to the nearest fixed-point number of precision ℓ is denoted $\circ_\ell(y)$. In the worst case, it entails an error $|\circ_\ell(y(k)) - y(k)| \leq 2^{\ell-1}, \quad \forall k$. For instance, rounding a real to the nearest integer ($\ell = 0$) 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 (2) with a precision-ℓ output, is a *perfectly rounded* computation with an error bound $\overline{\varepsilon}_{\text{out}} = 2^{\ell-1}$.

Unfortunately, reaching perfect rounding accuracy may require arbitrary intermediate precision, for reasons that will be detailed in Section 4.1. We therefore impose a slightly relaxed constraint: $\overline{\varepsilon}_{\text{out}} < 2^\ell$. 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 for the error bound $\overline{\varepsilon}_{\text{out}}$ that $\overline{\varepsilon}_{\text{out}} \geq 2^{\ell-1}$, it is still a tight specification. In particular, if the exact y happens to be a representable precision-ℓ number, then a last-bit accurate architecture will return exactly this value. As will be shown in the sequel, last-bit accuracy can be reached with very limited hardware overhead.

The main conclusion of this discussion is the following: specifying the LSB of the output format (ℓ_{out} in Figure 1) is enough to also specify the accuracy of the implementation.

This is a clear interface improvement over classical approaches, such as the various Matlab toolboxes that generate hardware filters. In such approaches, one must provide ℓ_{out} and various other parameters that impact the accuracy, then measure the resulting accuracy somehow. Not only is the proposed interface simpler, it also enables architecture optimization under a strict accuracy constraint: An optimal architecture will be an architecture that is accurate enough, but no more.

3.4 Worst-case peak gain of an LTI filter

To determine the MSB position of the output (m_{out}) and to perform the roundoff analysis, we need to capture the amplification of a signal by an LTI filter. This amplification can be computed using the so-called *Worst-Case Peak-Gain* (WCPG) [18], [19] through the following theorem.

Theorem 1 (Worst-Case Peak Gain). *Let $\mathcal{H}$ be a stable[2] single-input single-output LTI filter. If for all possible $k \geq 0$ an input*

2. i.e. poles of $H(z)$ are strictly in the unit circle.

signal $u(k)$ is bounded in magnitude by $\bar{u}$, then the output $y(k)$ is bounded:

$$\forall k, \quad |y(k)| \leq \bar{y} = \langle\!\langle \mathcal{H} \rangle\!\rangle \, \bar{u} \tag{4}$$

where $\langle\!\langle \mathcal{H} \rangle\!\rangle$ is the Worst-Case Peak Gain [18], [19] of the system. It can be computed as the ℓ_1-norm of the system's impulse response $h(k)$:

$$\langle\!\langle \mathcal{H} \rangle\!\rangle = ||h||_1 = \sum_{k=0}^{\infty} |h(k)| \tag{5}$$

The bound $\langle\!\langle \mathcal{H} \rangle\!\rangle \, \bar{u}$ on the output is quite conservative in practice, but for any filter it is possible to construct a finite input signal $\{u(k)\}_{0 \leq k \leq K}$ that yields an output that approaches the $\langle\!\langle \mathcal{H} \rangle\!\rangle \, \bar{u}$ up to any arbitrarily small distance. Indeed, if the output is expressed through convolution,

$$|y(k)| = \left| \sum_{l=0}^{k} h(l) u(k-l) \right| \leq \bar{u} \sum_{k=0}^{\infty} |h(k)| \tag{6}$$

we obtain the equality if the input $u(k)$ is such that

$$u(k) = \bar{u} \cdot \mathrm{sign}(h(K-k)), \tag{7}$$

where $\mathrm{sign}(x)$ returns ± 1 or 0 depending on the sign of x.

This work computes the WCPGs with arbitrary precision using the reliable algorithm presented in [2] and its fast but rigorous implementation[3]. It also builds worst-case signals implementing (7) to test the resulting architectures.

4 Error analysis of direct-form LTI filter implementations

This section shows how to obtain a last-bit accurate fixed-point implementation of the mathematical definition (2).

Since the considered filters are linear, we may assume without loss of generality that the input MSB is 0. Based on Theorem 1, the MSB of the output m_{out} is defined by $m_{\text{out}} = \lceil \log_2 \langle\!\langle \mathcal{H} \rangle\!\rangle \rceil$. Technically, it may happen, rarely, that rounding errors propagate all the way to the MSB. Since these errors will be bounded by $2^{\ell_{\text{out}}}$, the formula to be used is actually

$$m_{\text{out}} = \left\lceil \log_2 \left(\langle\!\langle \mathcal{H} \rangle\!\rangle + 2^{\ell_{\text{out}}} \right) \right\rceil \tag{8}$$

In addition, the algorithm from [3] guarantees that the computed MSB position is never underestimated.

In fixed-point arithmetic, instead of computing output $y(k)$, we will compute an approximation $\widetilde{y}(k)$ of the involved Sum of Product by Constants (SOPC), using internal fixed-point formats yet to be determined:

$$\widetilde{y}(k) \approx \sum_{i=0}^{n_b} b_i u(k-i) - \sum_{i=1}^{n_a} a_i \widetilde{y}(k-i) \tag{9}$$

and the final output $\widetilde{y}_{\text{out}}(k)$ will be some rounding of this intermediate value $\widetilde{y}(k)$.

This computational scheme is summed up by the abstract architecture of Figure 4, whose overall evaluation error is defined as

$$\varepsilon_{\text{out}}(k) \quad = \quad \widetilde{y}_{\text{out}}(k) - y(k) \tag{10}$$

3. https://scm.gforge.inria.fr/anonscm/git/metalibm/wcpg.git

Our goal is to detect all sources of error and express them in terms of the choices of LSB positions. Then, under the constraint of last-bit accuracy, i.e. $\varepsilon_{\text{out}}(k) < 2^{\ell_{\text{out}}}$, we will look for the optimal internal format.

Let us first decompose this error into its sources.

4.1 Internal format and final rounding

The architecture needs to internally use a fixed-point format that offers extended precision with respect to the input/output format. We note this extended format $(m_{\text{out}}, \ell_{\text{ext}})$.

Its MSB is the MSB of the result (m_{out}). Indeed, overflows may occur during the internal computation, but this computation is performed modulo $2^{m_{\text{out}}}$, therefore the final result will be correct.

This internal format also offers additional LSB bits (sometimes called *guard bits*) in which rounding errors may accumulate without touching the output bits – the sequel will show more formally how to compute this extended format.

Eventually, the intermediate result this extended format $(m_{\text{out}}, \ell_{\text{ext}})$ needs to be rounded to the output format (by the "final round" box on Figure 4). This entails an additional error ε_{f}, formally defined as

$$\varepsilon_{\text{f}}(k) = \widetilde{y}_{\text{out}}(k) - \widetilde{y}(k) \quad . \tag{11}$$

This error may be bounded by $\overline{\varepsilon}_{\text{f}} = 2^{\ell_{\text{out}}-1}$, and can be attained as soon as the exact result is a *midpoint*, i.e. exactly between two representable numbers. It is the reason why perfect rounding, mentioned in Section 3.3 may require arbitrary intermediate precision: perfect rounding imposes an error budget of $2^{\ell_{\text{out}}-1}$ which is entirely consumed by the final rounding. To achieve perfect rounding, one must 1) build a computational datapath that is *exact* for the input signals leading to a midpoint output, and 2) for non-midpoint outputs, compute the error budget once final rounding has been taken into account, and dimension the datapath to remain within this budget. If an (infinite) input stream can come arbitrarily close to a mid-point, then the internal computation will require arbitrary precision. To our knowledge, this question has not been studied, and studying it is out of the scope of this article.

Fortunately, last-bit accuracy is much easier to achieve: even with $\overline{\varepsilon}_{\text{f}} = 2^{\ell_{\text{out}}-1}$, the overall error budget of $2^{\ell_{\text{out}}}$ leaves an error budget of $2^{\ell_{\text{out}}-1}$ for the internal computation.

Remark that we feed back the intermediate result $\widetilde{y}(k)$ (on the extended format), 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 last-bit accuracy.

4.2 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 5. For now, we may summarize all these errors in a single term $\varepsilon_{\text{r}}(k)$ mathematically defined as

$$\varepsilon_{\text{r}}(k) \quad = \quad \widetilde{y}(k) - \left(\sum_{i=0}^{n_b} b_i u(k-i) - \sum_{i=1}^{n_a} a_i \widetilde{y}(k-i) \right) \tag{12}$$

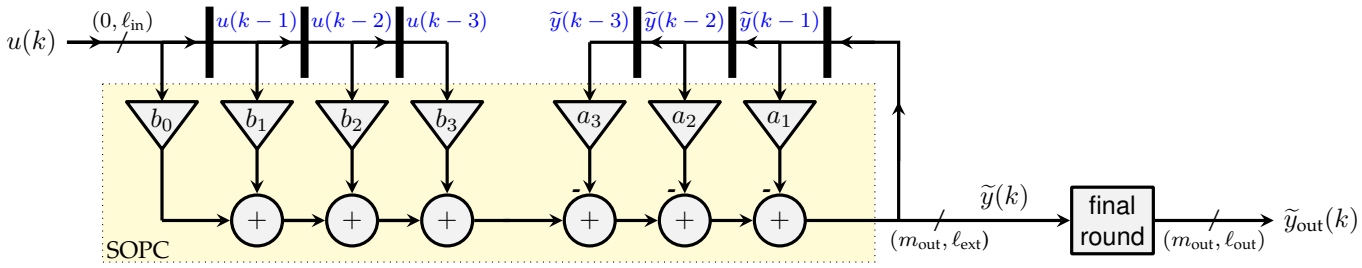


Fig. 4. Abstract architecture for the direct form realization of an LTI filter

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 (finite precision) inputs $u(k-i)$ and $\widetilde{y}(k-i)$ as the architecture.

4.3 Error amplification 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 $\widetilde{y}(k)$ (see Figure 4) differs from the ideal $y(k)$. Let us define $\varepsilon_{\text{t}}(k)$ as the error of $\widetilde{y}(k)$ with respect to $y(k)$:

$$\varepsilon_{\text{t}}(k) \;= \widetilde{y}(k) - y(k). \tag{13}$$

This error is potentially amplified by the architecture.

Let us rewrite $\widetilde{y}(k-i)$ in the right-hand side of (12):

$$\begin{aligned}
\varepsilon_{\text{r}}(k) \;&=\; \widetilde{y}(k) - \sum_{i=0}^{n_b} b_i u(k-i) + \sum_{i=1}^{n_a} a_i y(k-i) \\
&\quad + \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \qquad \text{(using (13))} \\
&=\; \widetilde{y}(k) - y(k) + \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \quad \text{(using (2))} \\
&=\; \varepsilon_{\text{t}}(k) + \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \qquad \text{(using (13))}.
\end{aligned} \tag{14}$$

If we rewrite equation (14) as

$$\varepsilon_{\text{t}}(k) \;=\; \varepsilon_{\text{r}}(k) - \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \tag{15}$$

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

$$\mathcal{H}_\varepsilon(z) = \frac{1}{1 + \sum_{i=1}^{n_a} a_i z^{-i}}\,. \tag{16}$$

Figure 5 illustrates this relationship between the ideal output y, the implemented output $\widetilde{y}_{\text{out}}$ and the different error terms.

We can now apply the Worst-Case Peak-Gain theorem to $\mathcal{H}_\varepsilon$ with input ε_{r} in order to bound ε_{t} by

$$\overline{\varepsilon}_{\text{t}} = \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle\, \overline{\varepsilon}_{\text{r}}\,. \tag{17}$$

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

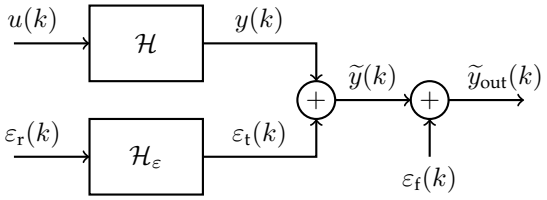


Fig. 5. A signal view of the error propagation with respect to the ideal filter

4.4 Putting it all together

Using the above considerations, we can put all errors together and rewrite (10) as

$$\begin{aligned}
\varepsilon_{\text{out}}(k) \;&= \widetilde{y}_{\text{out}}(k) - \widetilde{y}(k) + \widetilde{y}(k) - y(k) \\
&= \varepsilon_{\text{f}}(k) + \varepsilon_{\text{t}}(k).
\end{aligned} \tag{18}$$

Hence,

$$\overline{\varepsilon}_{\text{out}} \;= \overline{\varepsilon}_{\text{f}} + \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle\, \overline{\varepsilon}_{\text{r}} \tag{19}$$

The objective of the last-bit accuracy of the architecture translates into the constraint $\overline{\varepsilon}_{\text{out}} < 2^{\ell_{\text{out}}}$. Taking into account the final rounding (which implies the error $\overline{\varepsilon}_{\text{f}} = 2^{\ell_{\text{out}}-1}$), we obtain the constraint on the error $\overline{\varepsilon}_{\text{r}}$ of the SOPCs that is required to satisfy the last-bit accuracy of the overall architecture:

$$\overline{\varepsilon}_{\text{r}} < \frac{2^{\ell_{\text{out}}-1}}{\langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle}\quad. \tag{20}$$

This constraint finally translates to the LSB ℓ_{ext} of the intermediate result as follows. We assume that we may build an SOPC last-bit accurate to any value of ℓ_{ext} (this will be the object of Section 5). Such an SOPC will ensure $\overline{\varepsilon}_{\text{r}} < 2^{\ell_{\text{ext}}}$. Using (20), we obtain that the optimal value of ℓ_{ext} that ensures this constraint is

$$\ell_{\text{ext}} = \ell_{\text{out}} - 1 - \lceil \log_2 \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle \rceil \quad. \tag{21}$$

In other words, the internal format adds $1 + \lceil \log_2 \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle \rceil$ LSB guard bits to the output format.

The implementation of this error analysis actually uses a guaranteed overestimation of $\langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle$ [2]. This ensures that rounding errors in the the computation of $\langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle$ itself do not jeopardize the accuracy. Because of this overestimation, very rarely, we might compute on one bit more than what was required by (21). This rare one-bit overestimation of the datapath size has no impact in practice.

5 Sum of products computing just right

5.1 Problem statement

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

$$r = \sum_{i=1}^{N} c_i x_i \tag{22}$$

accurate to 2^{ℓ_r}, for a set of real constants c_i, and a set of fixed-point inputs x_i.

In a previous work [20], all the x_i shared the same format, as is the case for a FIR filter. In the context of an IIR filter, this is no longer true: in Figure 4, we have a single SOPC where the c_i may be a_i or b_i, and the x_i may be either some delayed $u(k)$, or some delayed $\widetilde{y}(k)$. The format of the $\widetilde{y}(k)$, as determined in the previous section, is in general different from that of the $u(k)$. Therefore, the present work uses a more generic interface to the SOPC generator, where the format of each input may be specified independently. This interface is shown in Figure 6. The input LSBs are provided as ℓ_i. For the input MSBs, instead of m_i, the interface uses the maximum absolute value $\overline{x_i}$ of each x_i, which provides a finer information that will be exploited in the sequel. In the context of an IIR filter, the output precision will be $\ell_r = \ell_{\text{ext}}$, this value being defined by the error analysis of previous sections.

Another difference with [20] is that the output MSB m_r is input to the generator. An overestimation of m_r could be computed out of the c_i and the input formats, as in [20]. However, the worst case peak gain of an IIR filter provides a finer value of m_r, and in this case we want to provide this value to the SOPC generator.

Here again, the weight ℓ_r of the LSB of the SOPC output also specifies the accuracy of this SOPC: the present section shows how to build an SOPC accurate to 2^{ℓ_r}. This is what was assumed in the previous section with $\ell_r = \ell_{\text{ext}}$.

5.2 Error analysis for a last-bit accurate SOPC

The fixed-point summation of the various terms $c_i x_i$ is depicted in Figure 7. For this figure, we take as an example the 4-input SOPC of an IIR filter of order 2 with arbitrary coefficients: it is a smaller version of the one depicted in Figure 4, where x_0 and x_1 are, respectively, $u(k)$ and $u(k-1)$, while x_2 and x_3 are respectively $\widetilde{y}(k-1)$ and $\widetilde{y}(k-2)$. The output r will become $\widetilde{y}(k)$.

As shown in the figure, a real c_i may have an infinite number of bits. Therefore, even though the x_i are finite, each product $c_i x_i$ potentially has an infinite number of bits.

The MSB of each product $c_i x_i$ is easily determined out of the value of c_i itself and $\overline{x_i}$: $|x_i| \leq \overline{x_i}$, therefore $|c_i x_i| \leq c_i \overline{x_i}$,

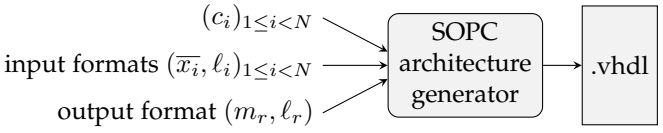


Fig. 6. Interface to a sum-of-product-by-constant generator

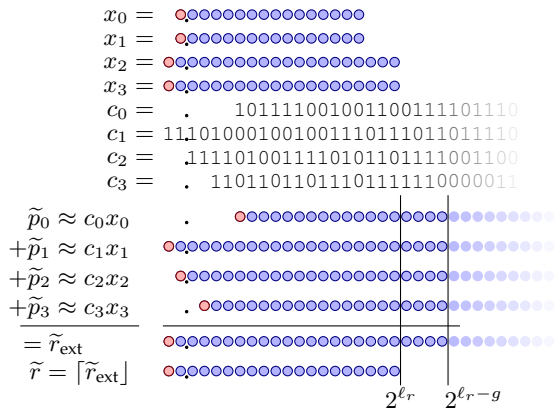


Fig. 7. Alignment of the $c_i x_i$ for fixed-point x_i and real c_i

so the MSB of $c_i x_i$ will be $\lceil \log_2(|c_i \overline{x_i}|) \rceil$. Here, using $\overline{x_i}$ instead of an MSB specification for x_i can save one bit. As done previously, in order to anticipate possible overflows due to rounding, the implementation must add, before taking the $\log_2$, an upper bound of its rounding error. This bound will be detailed in the sequel.

Negative $c_i x_i$ must have their sign extended to the MSB of the sum, so it could seem that Figure 7 only shows the cases when all the $c_i x_i$ are positive. Here we must explain another technicality. The sign extension `ss...ssxxxxxxx` of a signed number `sxxxx`, where `s` is the sign bit, may be performed as follows [21]:

```
    00...0s̄xxxxxxx
+   11...110000000
=   ss...ssxxxxxxx
```

Here $\overline{\texttt{s}}$ is the boolean complement of `s`. The reader may check this equation in the two cases, $\texttt{s} = 0$ and $\texttt{s} = 1$. Now the variable part $\overline{\texttt{s}}$`xxxxxxx` has the same MSB as in the positive case, and this is what Figure 7 shows.

This transformation is not for free: we need to add the constant `11...110000000`. 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 the following, we will use another trick to merge this addition for free in the computations of one of the $c_i x_i$.

Performing all the internal computations to the output precision ℓ_r would in general not allow last-bit accuracy to precision ℓ_r, due to the accumulation of rounding errors. The solution is, as previously, to use a slightly extended precision $\ell_r - g$ for the internal computation: g is a number of "guard" bits. As this extended precision will require more hardware, we now discuss how to compute the extended precision that will minimize this hardware overhead.

We assume that we are able to build hardware constant multipliers that compute some approximation

$$\widetilde{p}_i = c_i x_i + \varepsilon_i(g) \tag{23}$$

of the mathematical product $c_i x_i$ such that the LSB of each $\widetilde{p}_i$ is $\ell_r - g$ (see Figure 7), and we assume that the rounding error ε_i of each of these multipliers is bounded by some $\overline{\varepsilon}_i(g)$:

$$|\varepsilon_i(g)| < \overline{\varepsilon}_i(g) \quad . \tag{24}$$

The value of $\overline{\varepsilon}_i(g)$ depends on the constant: multiplication by zero will be exact, as will be, under some conditions, multiplications by powers of two and by other constants that can be written in binary on few bits. In the general case where c_i is real, the multiplier will entail a rounding error which depends on the multiplier technique used (a detailed example will be shown in the sequel). However, whatever the technique, this error bound can be made as small as needed by increasing g (in other words, by computing more accurately).

The output value $\widetilde{r}$ is computed in an architecture as the sum of the $\widetilde{p}_i$. This summation, as long as it is performed with adders of the proper size, will entail no error (Figure 7). Indeed, fixed-point addition of numbers of the same format may entail overflows (these have been taken care of), but no rounding error. This enables us to write

$$\widetilde{r}_{\text{ext}} \quad = \quad \sum_{i=0}^{N-1} \widetilde{p}_i, \tag{25}$$

therefore the total rounding error of the sum of product is defined as

$$\varepsilon_{\text{SOPC}} \quad = \quad \sum_{i=0}^{N-1} \widetilde{p}_i - \sum_{i=0}^{N-1} c_i x_i \quad = \quad \sum_{i=0}^{N-1} \varepsilon_i(g) \tag{26}$$

and thanks to (24) can be bounded as follows:

$$\overline{\varepsilon}_{\text{SOPC}} \quad < \quad \sum_{i=0}^{N-1} \overline{\varepsilon}_i(g) \quad . \tag{27}$$

As each $\overline{\varepsilon}_i(g)$ can be made arbitrarily small by increasing g, there exists some g such that

$$\sum_{i=0}^{N-1} \overline{\varepsilon}_i(g) \quad < \quad 2^{\ell_r - 1} \quad . \tag{28}$$

The intermediate result now has g more bits at its LSB than we need (Figure 7). It therefore needs itself to be rounded to the target format. This is easy, using the identity $\lfloor x \rceil = \lfloor x + \frac{1}{2} \rfloor$, scaled by 2^{ℓ_r}: rounding to precision ℓ_r is obtained by first adding $2^{\ell_r - 1}$ (this is a single bit) then discarding bits lower than 2^{ℓ_r}. In the worst case, this will entail an error $\varepsilon_{\text{final rounding}}$ of at most $2^{\ell_r - 1}$.

To sum up, the overall error of a last-bit accurate SOPC architecture is:

$$\widetilde{r} - \sum_{i=0}^{N-1} c_i x_i \quad = \quad \varepsilon_{\text{final rounding}} + \varepsilon_{\text{SOPC}} \tag{29}$$

$$< \quad 2^{\ell_r - 1} + 2^{\ell_r - 1} \quad = \quad 2^{\ell_r} \quad . \tag{30}$$

All the previous is quite independent of the target technology. However, the actual computation of the optimal g out of constraint (28) will depend on the multiplier technique chosen. This is the reason why we do not give a generic formula providing g.

However, for illustration and completeteness, the remainder of this section focusses on a particular technology: LUT-based SOPC architectures for FPGAs. It explores architectural means to reach last-bit accuracy at the smallest possible cost on this technology.

5.3 Tabulated perfectly rounded constant multipliers

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

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 the 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 cost exactly one 6-input LUT. For 8-bit inputs, each bit costs only 4 LUTs. In general, for $(6+k)$-bit inputs, each output bit costs 2^k 6-LUTs: this approach scales poorly to larger inputs.

Perfect rounding to precision $\ell_r - g$ means a maximum error smaller than a half-LSB: $\overline{\varepsilon}_i = 2^{\ell_r - g - 1}$. Note that for real-valued c_i, this is more accurate than rounding the result of a multiplier inputting $\circ_{\ell_r}(c_i)$: the latter would accumulate two successive rounding errors.

5.4 Table-based constant multipliers for FPGAs

For larger precisions, we may use a variation of the KCM technique, due to Chapman [4] and further studied by Wirthlin [5]. The original KCM method addresses the multiplication by an integer constant. We here present a variation called FixRealKCM that performs the multiplication by a *real* constant.

This method consists in breaking down the binary representation of an input x_i into D_i chunks d_{ik} of α bits. With the input size being $m_i - \ell_i + 1$, we have

$$D_i = \lceil (m_i - \ell_i + 1)/\alpha \rceil \tag{31}$$

(see Figure 8). Mathematically, this is written

$$x_i \quad = \quad \sum_{k=1}^{D_i} 2^{-k\alpha} d_{ik} \quad \text{where} \quad d_{ik} \in \{0, ..., 2^\alpha - 1\} \quad . \tag{32}$$

Another point of view is that the input x_i is considered as a radix-2^α number, the d_{ik}s being its digits. For instance with $\alpha = 4$ we obtain the classical hexadecimal writing of x_i.

The product becomes

$$c_i x_i \quad = \quad \sum_{k=1}^{D_i} 2^{-k\alpha} c_i d_{ik} \quad . \tag{33}$$

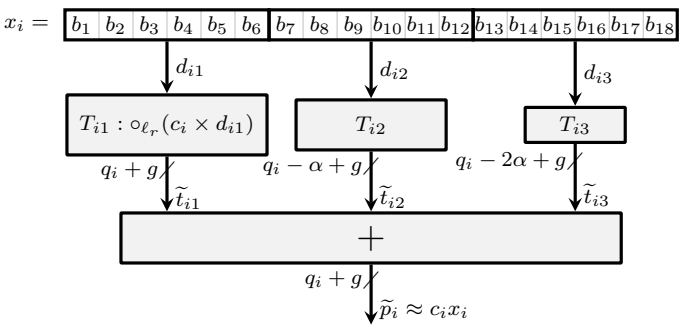


Fig. 8. FixRealKCM with x_i split in $D_i = 3$ chunks of $\alpha = 6$ bits

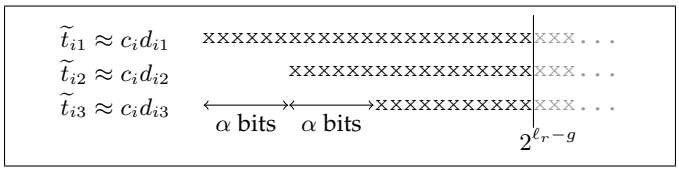


Fig. 9. Alignment of the terms in the KCM method

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 in Figure 8. Of course, $c_i d_{ik}$ has an infinite number of bits in the general case: as previously, we will round it to precision $\ell_r - g$. In all the following, we define $\widetilde{t}_{ik} = \circ_{\ell_r - g}(c_i d_{ik})$ this rounded value (see Figure 9).

Contrary to classical (integer) KCM, the tables do not consume the same amount of resources. The factor $2^{-k\alpha}$ in (33) shifts the MSB of the table output $\widetilde{t}_{ik}$, as illustrated by Figure 9.

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

$$\varepsilon_i < D_i \times 2^{\ell_r - g - 1} \quad . \tag{34}$$

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

5.5 Accumulating the products

Instead of considering each KCM in isolation, it is better to consider the summation at the SOPC level. Indeed, our SOPC result is now obtained by computing a double sum:

$$\widetilde{y} = \circ_p \left(\sum_{i=0}^{N-1} \sum_{k=1}^{D_i} 2^{-k\alpha} \widetilde{t}_{ik} \right) \tag{35}$$

Here, the errors of each $\widetilde{t}_{ik}$ add up into an overall SOPC error, out of which the value of g can be computed.

Before that, let us also observe that it is often possible to use a finer bound than (34). Indeed, some constant multipliers entail no error: it is for instance the case for multiplication by 0 and by 1. Such trivial cases will happen quite often if the proposed SOPC generator is used as a backend for a larger architecture generator, as is the case in the present article. Besides, such trivial cases deserve specific treatment since their implementation is much simpler than the generic case.

Therefore, the implementation first invokes, for each constant, a method that returns the maximum error that will be entailed by a multiplier by this constant. This error is expressed in units in the last place (ulp), and therefore does not depend on the (yet unknown) value of g. The implementation sums these errors, then uses this sum to compute the value of g that will enable last-bit accuracy. Once this g has been determined, the generator may proceed with the actual construction of the multipliers.

Here is the list of cases currently managed by the implementation:

- if $c_i = 0$, then $\varepsilon_i = 0$.
- if $|c_i| = 1$ or more generally if $|c_i| = 2^k$, then $\varepsilon_i = 0$ if $k + \ell_i \geq \ell_r$ (shift of x_i such that all the bits will be kept), otherwise $\varepsilon_i = 1$ (shift to the right, losing some bits due to truncation). Here we may overestimate the error, because the test should be if $k + \ell_i \geq \ell_r - g$, but we don't know g yet.
- In the general case when we use the generic KCM architecture, $\varepsilon_i = D_i/2$ (we have D_i tables, each entailing one half-ulp of error).

One final technicality: we have so far assumed that the number of tables D_i is computed out of the input size, using (31). However, for small constants, it may happen that the contribution of the lower tables can be neglected. To understand this, consider Figure 9: each table output is shifted right if c_i is small. Therefore, the implementation will not generate a table if its MSB is smaller than $\ell_r - g - 1$. The error analysis remains valid in this case, although the source of the error is no longer the rounding of the table, but it's being neglected altogether. If more than one table is fully neglected, this error analysis was slightly pessimistic (we could have a single half-ulp for all the neglected tables), but it remains safe.

5.6 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, using the associativity of exact fixed-point addition, this summation can be implemented very efficiently using compression techniques developed for multipliers [21] and more recently applied to sums of products [22], [23]. This work uses the *bit heap* framework introduced in [7]. 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. It also manages two's complement numbers efficiently – the interested reader is referred to [7] for details.

Figure 10 shows an example of bit heap obtained by the proposed method. In this figure (generated by the tool), we have binary weights on the horizontal axis, and the various terms to add on the vertical axis. Roughly speaking, the height of a bit heap is proportional to the number of non-zero coefficients. The width shows the needed internal

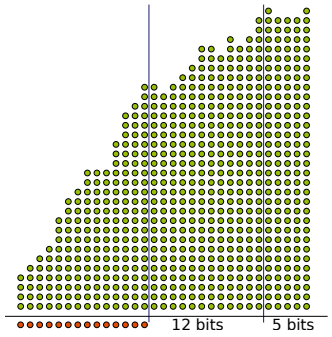


Fig. 10. The bit heap of a narrowband 12-bit filter (example 1 of section 6 with $\omega_p = 0.5$). In this case the tool has computed $-\log_2 \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle - 1 = 12$ and $g = 5$, so the internal datapth must be 29-bit wide for last-bit accuracy.

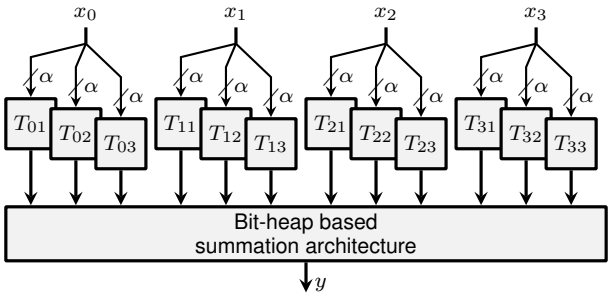


Fig. 11. KCM-based SOPC architecture for $N = 4$, each input being split into 3 chunks

precision computed by the tool to ensure both filter stability and last-bit accuracy. Vertical lines mark the bit weights ℓ_{out} and ℓ_{ext}: they illustrate the $1+\lceil \log_2 \langle\langle \mathcal{H}_\varepsilon \rangle\rangle \rceil$ extra bits needed to manage the error amplification in the feedback loop (see equation (21)), as well as the g extra bits needed to absorb the rounding errors in each KCM table.

Figure 10 only shows the initial bit heap. FloPoCo also generates an architecture that computes the sum of all these bits. The current version still uses the greedy heuristic of [7]. The state of the art is to compute an optimal architecture using Integer Linear Programming techniques [8], and will be used soon.

5.7 Final rounding by truncation

There is one more term to add to the summation of (35): the rounding bit $2^{\ell_r - 1}$, necessary for the final rounding by truncation. Its value is added (for free) to one of the tables.

Finally, the typical architecture generated by our tool is depicted by Figure 11.

6 Implementation and Results

This section demonstrates on several examples the versatility of the tool and how it can be used for filter design-space exploration. It also analyzes the cost of last-bit accuracy, and evaluates how much the methodology potentially overestimates the word-lengths.

6.1 Experimental setup

The method described in this paper is implemented as the `FixIIR` operator of FloPoCo (version 4.1.3 or above). `FixIIR` offers the interface shown on Figure 1, and inputs the coefficients a_i and b_i as arbitrary-precision numbers or mathematical expressions. `FixIIR`, like most FloPoCo operators, was designed with a testbench generator [24]. Operators reported here have been checked for last-bit accuracy by extensive simulations using worst-case signals constructed thanks to (7). The results below were obtained after place and route for Kintex-7 (7k70tfbv484-3) using Vivado 2016.4. The reported timings don't include the IBUF/OBUF delay.

6.2 Example 1: sliding passband

This example shows how the proposed tool may quantify the difficulty of implementing narrow band filters. Such filters are frequently used in Software Defined Radio [25],

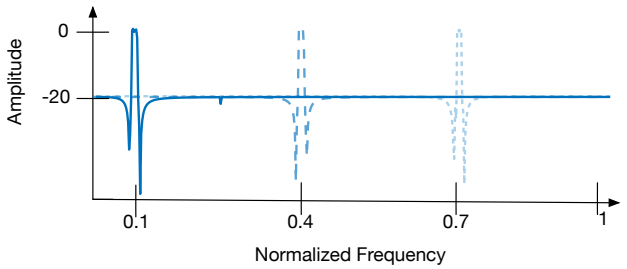


Fig. 12. Example 1, three transfer functions of the family of narrow-passband filters with sliding ω_p.

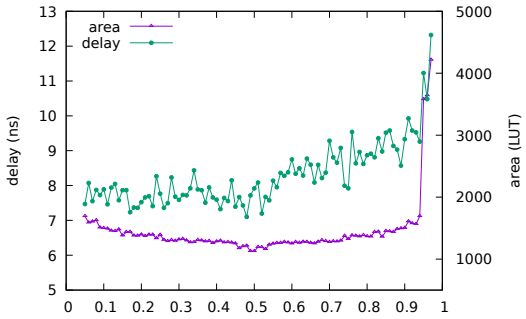


Fig. 13. Example 1, area and delay of implementations as a function of the parameter ω_p.

biomedical circuit design [26], digital television, etc. Classically, the implementation of such bandpass filters is more complicated when the passband is near the bounds of the frequency domain [1]. To illustrate this effect, we fix the widths of the passband and transition band to 0.01 and "slide" the passband through the normalized frequency interval $[0, 1]$. Formally, we set

- passband $[\omega_p, \omega_p + 0.01]$ with passband ripple 1dB
- stopbands in $[0, \omega_p - 0.01]$ and $[\omega_p + 0.02, 1]$ with minimum attenuation 20dB

where $\omega_p = 0.05 + 0.01k$, $k = 0, 1, \ldots, 94$. Each filter in this family is an elliptic filter designed with Python Scipy. All filters are of the order 4, except for $k \in [92, 94]$ where the order increases to 6.

Figure 13 gives the area and delay with respect to the position of $\omega_p \in [0.05, 0.97]$. It can be clearly observed that narrow bandpass filters have lower complexity when the normalized passband frequency ω_p approaches 0.5. The bit heap corresponding to the implementation with $\omega_p = 0.5$ is given in Figure 10.

Interestingly, the last three filters ($k \in [92, 94]$) are considerably more sensitive to rounding errors (i.e. $\langle\langle \mathcal{H}_\varepsilon \rangle\rangle$ increases for them), thus require more area to ensure the output is accurate to 12 fractional bits.

Such phenomena for the narrow-band bandpass filters is well known. However, in the general case, overcoming difficulties in the implementation of the recursive filters is usually not straightforward and might require specific knowledge from the designer. The proposed tool automatically provides accurate implementations of even the most sensitive filters at the minimal cost, and the designer is informed of the cost of this accuracy.

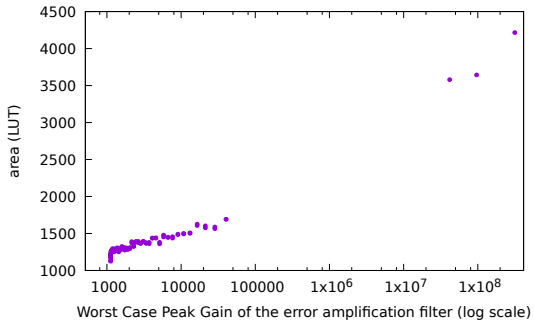


Fig. 14. Example 1, dependency of area on $\langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle$.

TABLE 1
Coefficients of the IIR4 filter

i	b_i	a_i
0	0.01	-
1	−0.00736448	−3.24671
2	0.016119471	4.3113278648
3	−0.007364485	−2.731124776
4	−0.01	0.6968113152

6.3 Example 2: filters from the WLO literature

We also attempted to reproduce the following filters.

IIR4 is a 4^{th} order filter mentioned in [9], [27], whose coefficients (Table 1) were manually extracted from software code in [27]. For this filter, [27] reports SQNR between 49.3dB and 78.5dB, which corresponds to approximately 8 to 13 meaningful output bits. For this, they use 16-bit multipliers, with intermediate additions on 16 to 32 bits. For the same transfer function and 8 (resp. 13) correct output bits, our tool generates an implementation where the internal feedback signal (which is the input to our multipliers) is 18 (resp. 23) bits. These results illustrate that last-bit accurate implementations are competitive with implementations offering only statistical confidence.

In [13], authors fix the filter order to 8 but do not give the specifications. Obviously, for a given order, depending on the frequency specifications, the filter might be more or less sensitive to the finite wordlength effects. We therefore chose two IIR8 specifications, both designed with Matlab: **IIR8a** is an 8^{th}order butterworth filter with sampling frequency $F_s = 48$ kHz, cutoff frequency $F_c = 10.8$ kHz and 3 dB attenuation. Minimum distance from its poles to the unit circle is $1.77e-01$. **IIR8b** is an 8^{th} order elliptic lowpass filter with normalized cutoff frequency 0.3, passband ripple 1 dB and minimum 50 dB attenuation in the stopband. This filter if more sensitive to quantization, distance form poles to the unit circle is $7.56e-03$.

Figure 15 illustrates the evolution of the architectural cost with input/output precision for IIR4, IIR8a and IIR8b. The cost of each multiplier is expected to grow quadratically with input precision, which explains the trend for the area curves. The delay is always that of the feedback loop.

Finally, **IIR2** is the most sensitive filter from [14]. Its coefficients are $(b_0, b_1, b_2) = (101.8, -203.4, 101.6)$ and $(a_1, a_2) = (-1.967, 0.968)$. Results in [14] are reported for an input in $[-100, 100]$ and an output error bound of 0.1. By linearity, for inputs in $[-1, 1]$ this corresponds to $\overline{\varepsilon_{\text{out}}} = 10^{-3}$, which in our tool is ensured by $\ell_{\text{out}} = -10$. The feedback WCPG is found to be $\langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle \approx 1368$, therefore our tool computes that $\lceil \log_2 \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle \rceil + 1 = 12$ bits must be added to the datapath to manage the amplification of errors: it finds that $\ell_{\text{ext}} = -22$ is needed on the feedback loop. Comparatively, [14] reports 18 fractional bits when inputs are in $[-100, 100]$ which would translate to $\ell_{\text{ext}} = -25$ by linearity. This is, in [14], the precision of the feedback loop, but also of the output: note that 15 of these output bits hold useless noise. Also, the coefficients are rounded in [14] to an LSB of weight -26. An architecture multiplying such a feedback signal by such a constant will compute product bits extending down to weight $-26-25 = -51$, then round them out. Comparatively, our approach computes that it needs to add $g = 5$ guard bits inside the SOPC: it will compute no bit with weight smaller than -27.

TABLE 2
Using the proposed tool on some IIR filters (all with $\ell_{\text{in}} = \ell_{\text{out}}$)

filter specification		guard bits		synthesis results	
name	$-\ell_{out}$	$\lceil \log_2 \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle \rceil$	g	Area	delay
IIR2 [14]	10	11	5	832 LUT	9.1ns
IIR4 [9], [27]	8	8	5	710 LUT	6.8ns
	12		5	958 LUT	7.5ns
IIR8a [13]	8	4	6	831 LUT	6.8ns
	32		7	4974 LUT	10.3ns
IIR8b [13]	8	14	6	2262 LUT	8.3ns
	32		7	8265 LUT	11.6ns

6.4 The cost of last-bit accuracy

The previous example shows that the proposed methodology leads to more efficient architectures than the state of the art. Table 2 gives the cost of last-bit accuracy in terms of guard bits for other examples.

In Table 2, $\langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle$ does not depend on the input/output formats. One can observe that g has a moderate dependency input precision (only $+1$ bit from 8 to 32 bits for instance). Indeed, g is essentially the $\log_2$ of the number of KCM tables, which is itself proportional to $-\ell_{out} - \log_2 \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle - 1$, not just $-\ell_{out}$.

6.5 Evaluating the pessimism of the method

As the proposed methodology combines worst-case error bounds, it is necessary pessimistic. To quantitatively evaluate this, we attempted to reduce the internal datapath (from the values computed by the tool) and exercised the obtained filter on a signal built to trigger the worst-case error amplification (see (7) in Section 3.4). This remains a heuristic, as this test signal does not guarantee the occurence of worst-case rounding errors. Still, we find that most filters fail this test as soon as more than 2 bits are removed from their internal datapath. This shows that the internal sizes computed by the methodology are quite tight.

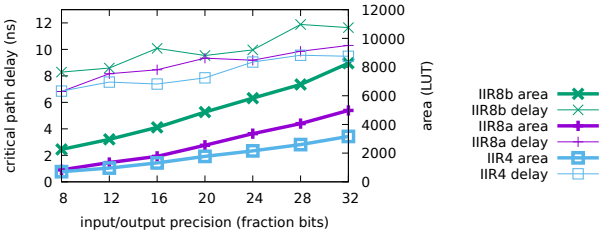


Fig. 15. Evolution of area (in LUT) and delay (ns) with input/output precisions for the IIR4 and IIR8 filters.

7 Conclusion and Perspectives

This article advocates a very simple specification for low-precision architectures of digital filters: whatever the inputs, the difference between the results computed by the architecture and the results computed by an infinite precision machine should be less than the weight of the least significant bit of the output. This specification brings safety to designers who can trust that the architecture behaves as, e.g., a double-precision Matlab simulation. It also defines a universal rule of the game by which different architectures can be fairly compared.

This work demonstrates this approach through an end-to-end open-source tool that generates the VHDL code of a Direct Form I implementation from its mathematical coefficients. It explores architectural choices in the family of Direct Form I filters, and selects the architecture that has minimal-size internal formats for which we can guarantee the above specification.

An important observation is that even for simple, low-order filters, the intermediate format needs to be significantly larger than the input/output format: in our experiments, we never need less than 4 extra bits for the internal format to achieve the specification. For higher-order or unstable filters, several tens of extra bits may be needed, even if the output format is only 8, 12 or 16 bits. This overhead depends on the worst-case peak gain of the error filter: it is mostly independent of the output format.

This work, however, does not claim to close the subject of digital filter architecture generation.

Future work first includes several technical improvements to the current implementation, such as the exploitation of symmetries in the coefficients, or a better bit heap compression algorithm. Also, we chose in this work to have a single SOPC for both the a_i and the b_i multipliers. This ensures a global optimization of the bit-heap compression, hence better area. However, frequency could be improved by using two SOPCs, one for $\sum b_i u(k-i)$ (taken out of the loop), the other for $\sum a_i y(k-i)$. This would in principle come at the expense of larger area. However the exact impact is unclear, since we will now have two different values of g.

Beyond these refinements, this work is a solid foundation on which to build future research. Here are some directions.

In this work we have assumed real coefficients. We may now address the issue of *coefficient quantization* from a new point of view. If a designer manages to quantize coefficients on very few bits and still obtain a stable filter with acceptable transfer function, the proposed technique will work without change: quantized coefficients are also real coefficients. Therefore, the proposed approach enables a clear decoupling of the issue of coefficient quantization from the issue of intermediate rounding in the architecture.

However, quantized coefficients will also enable further optimizations. Indeed, the product of a quantized coefficient by an input (or by a subword of the input in the case of KCM multipliers) will have a finite number of bits. This potentially improves both error analysis and architecture generation: if these bits are all within the internal bit range determined by the tool, the product becomes exact, while the corresponding table output becomes smaller than in the case of a real coefficient, also leading to smaller bit heaps. In addition, with quantized coefficients, it also becomes relevant to compare with shift-and-add implementations of constant multipliers [28].

The next step is to explore filter structures that are more interesting than the DFI, for instance the decomposition into second-order sections [1]. In this structure a filter is decomposed into a cascade of biquad filters, each of which can be implemented with a DFI algorithm. An advantage of such a decomposition is smaller sensitivity towards rounding errors. Quantitatively, each section of the cascaded system may be analyzed independently using the worst-case peak gain and requires a different number of additional bits ℓ_{ext}. However, it is not completely straightforward how to choose the rounding strategy: on the one hand, rounding the output of each section to some $\tilde{y}_{\text{out}_i}$ increases the overall output error; on the other hand, direct propagation of $\tilde{y}(k)$ without rounding will increase the size of subsequent SOPCs. A trade-off might be achieved by a clever choice of section ordering [29].

It would also be interesting to consider other filter structures, such as Direct Form II (transposed or not), state-spaces, cascade and/or parallel decomposition, ρ-operator based structures, Lattice Wave Digital filters [30], etc. These algorithms are less sensitive to finite precision effects [30] (coefficient quantization and roundoff errors) and, even if some of them require more computations than the DFI, the total area (LUT) to achieve *just right* computing may be reduced. This work will allow one to quantify this, although it will not be completely straightforward: these structures have many intermediate formats, whose sizes also have to be determined and minimized. A unifying filter representation called SIF [31] could be the key to generalize the approach of the present article.

References

[1] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, 3rd ed. NJ, USA: Prentice Hall Press, 2009.

[2] A. Volkova, T. Hilaire, and C. Lauter, "Reliable evaluation of the worst-case peak gain matrix in multiple precision," in *22nd IEEE Symposium on Computer Arithmetic*, 2015.

[3] ——, "Determining fixed-point formats for a digital filter implementation using the worst-case peak gain measure," in *49th Asilomar Conference on Signals, Systems and Computers*, 2015.

[4] K. Chapman, "Fast integer multipliers fit in FPGAs," *EDN magazine*, no. 10, p. 80, 1993.

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

[6] F. de Dinechin, H. Takeugming, and J.-M. Tanguy, "A 128-tap complex FIR filter processing 20 giga-samples/s in a single FPGA," in *44th Asilomar Conference on Signals, Systems and Computers*, 2010.

[7] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, "Arithmetic core generation using bit heaps," in *Field-Programmable Logic and Applications*, 2013.

[8] M. Kumm and P. Zipf, "Pipelined compressor tree optimization using integer linear programming," in *Field Programmable Logic and Applications*, 2014.

[9] G. A. Constantinides, P. Y. Cheung, and W. Luk, *Synthesis and optimization of DSP algorithms*. Kluwer, 2004.

[10] D. U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, "Accuracy-guaranteed bit-width optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1990–2000, 2006.

[11] A. B. Kinsman and N. Nicolici, "Bit-width allocation for hardware accelerators for scientific computing using SAT-modulo theory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 3, pp. 405–413, 2010.

[12] D. Ménard, N. Hervé, O. Sentieys, and H.-N. Nguyen, "High-level synthesis under fixed-point accuracy constraint," *Hindawi Journal of Electrical and Computer Engineering*, 2012.

[13] S. Vakili, J. M. P. Langlois, and G. Bois, "Enhanced precision analysis for accuracy-aware bit-width optimization using affine arithmetic," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 12, pp. 1853–1865, 2013.

[14] O. Sarbishei, K. Radecka, and Z. Zilic, "Analytical optimization of bit-widths in fixed-point LTI systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 3, pp. 343–355, 2012.

[15] D. Boland and G. Constantinides, "Bounding variable values and round-off effects using Handelman representations," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 11, pp. 1691 –1704, 2011.

[16] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *4th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 1977, pp. 238–252.

[17] D. Boland and G. Constantinides, "Word-length optimization beyond straight line code," in *ACM Field Programmable Gate Arrays*. ACM, 2013.

[18] V. Balakrishnan and S. Boyd, "On computing the worst-case peak gain of linear systems," *Systems & Control Letters*, vol. 19, pp. 265–269, 1992.

[19] S. P. Boyd and J. Doyle, "Comparison of peak and RMS gains for discrete-time systems," *Systems & Control Letters*, vol. 9, no. 1, pp. 1–6, 1987.

[20] F. de Dinechin, M. Istoan, and A. Massouri, "Sum-of-product architectures computing just right," in *Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2014.

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

[22] H. Parendeh-Afshar, A. Neogy, P. Brisk, and P. Ienne, "Compressor tree synthesis on commercial high-performance FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, no. 4, 2011.

[23] R. Kumar, A. Mandal, and S. P. Khatri, "An efficient arithmetic sum-of-product (SOP) based multiplication approach for FIR filters and DFT," in *International Conference on Computer Design*. IEEE, 2012, pp. 195–200.

[24] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.

[25] L. Tsoeunyane, S. Winberg, and M. Inggs, "Software-defined radio FPGA cores: Building towards a domain-specific language," *International Journal of Reconfigurable Computing*, pp. 3 925 961:1–3 925 961:28, 2017.

[26] K. Limnuson, H. Lu, H. J. Chiel, and P. Mohseni, "FPGA implementation of an IIR temporal filtering technique for real-time stimulus artifact rejection," in *IEEE Biomedical Circuits and Systems Conference*, 2011, pp. 49–52.

[27] K.-I. Kum, J. Kang, and W. Sung, "AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors," *IEEE Transactions on Circuits and Systems II*, vol. 47, no. 9, pp. 840–848, 2000.

[28] K. Möller, M. Kumm, M. Kleinlein, and P. Zipf, "Pipelined reconfigurable multiplication with constants on FPGAs," in *Field Programmable Logic and Applications*, 2014, pp. 1–6.

[29] M. Lankarany and H. Marvi, "Noise reduction in digital iir filters by finding optimum arrangement of second-order sections," in *Canadian Conference on Electrical and Computer Engineering*, 2008, pp. 689–692.

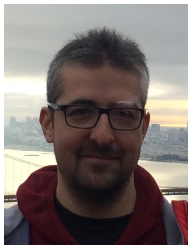
[30] A. Volkova and T. Hilaire, "Fixed-point implementation of lattice wave digital filter: comparison and error analysis," in *European Signal Processing Conference*, 2015.

[31] T. Hilaire, "Towards Tools and Methodology for the Fixed-Point Implementation of Linear Filters," in *Digital Signal Processing Workshop and IEEE Signal Processing Education Workshop (DSP/SPE)*, 2011, pp. 488–493.

Anastasia Volkova was born in Odessa, Ukraine in 1991. She obtained a master's degree from Odessa National I. I. Mechnikov University in 2014, then a PhD from Sorbonne Universités – University Pierre and Marie Curie in 2017. She is now a postdoctoral researcher in Inria at LIP Laboratory, ENS de Lyon (France). Her research interests include computer arithmetic, digital signal processing and arithmetic aspects of hardware/software implementations of signal processing algorithms.

Matei Istoan , born in Cluj-Napoca, Romania in 1988, obtained his MSc from ENS de Lyon in 2012, then a PhD from INSA de Lyon in 2017. He is now a research associate at Imperial College London. His research interests include circuit design for FPGAs, computer arithmetic and hardware implementations for function evaluation and signal processing.

Florent de Dinechin born 1970, obtained his PhD from Université of Rennes-1 in 1997. He was a postdoctoral fellow at Imperial College, London, then an assistant professor École Normale Supérieure de Lyon, and is now a professor at INSA-Lyon. His research interests include hardware and software computer arithmetic, FPGA arithmetic, floating-point, formal proofs for arithmetic algorithms, elementary function evaluation, and digital signal processing. Since 2008, he manages the FloPoCo software project.



Thibault Hilaire born in 1977, obtained a master's degree from École Centrale de Nantes in 2002, then a PhD from Université of Nantes in 2006. After two postdoctoral positions at Université of Rennes 1 and Technische Universität Wien, he joined Sorbonne Université – University of Pierre and Marie Curie as an associate professor. His research interests include fixed-point arithmetic (and more generally computer arithmetic), software and hardware implementation of signal processing and control algorithms.