Architecture synthesis for linear time-invariant filters

Antoine Martinet

Master 2 Internship Report

at CITI lab
INRIA's SOCRATE Team

under the supervision of

Florent de Dinechin

2 February - 31 Jully,
2015

Contents

Introduction

Filters are nowadays essential tools for designing responsive systems. In signal processing, filters are used in radio signal coding and decoding, image and sound processing, and so on. In addition, they are also used in many control applications.

For most applications, filters can be computed in software. However, for performance reasons hardware implementations are needed in many applications. There is an interest in FPGAs implementations because they can help for prototyping a software defined radio (SDR).

SDR is the main research subject of the SOCRATE team, at the CITI lab, where this work has been conducted. The goal of this internship was to design a parametric architecture generator for Linear Time Invariant (LTI) filters. This work has been integrated to FloPoCo tool, whose purpose is to generate architecture cores computing just right.

This report first presents the context of LTI filters, SIF and their implementation in part 1. Part 2 will then describe the details of the implementation brought in arbitrary precision. Finally, an overview of the future work will be presented in part 3.

1 Tools for expressing filters and signal processing

1.1 Fundamentals about signal processsing

1.1.1 Definition of a signal

Lopez's PhD [6] gives a good presentation of the state of the art. Many notations and definitions are kept from this PhD.

Note that a bold symbol (for example $\boldsymbol{h}$) denotes a matrix, that may be a vector. On the contrary, a normal symbol (for example $\varepsilon_{t_1}(k)$) denotes a single value

Definition 1. *(Signal) Generally, a signal is a temporal variable, which takes a value from $\mathbb{R}$ at each time t. $x(t)$ denotes the value of the signal x at the instant t. When dealing with discrete time events, the time will be represented by k. The notation will then be $x(k)$, which is said to be a sample. $\{x(k)\}_{k\geq 0}$ denotes all the values possible for the signal x. The rest of this report adresses vectors of signals $\boldsymbol{x}$, where $\boldsymbol{x}(k) \in \mathbb{R}^n$*

1.1.2 l^∞ norm

Definition 2. *(ℓ^∞ norm) The ℓ^∞ norm of a scalar signal x, denoted $\|x\|_{\ell^\infty}$, is the smallest upper bound among all values (absolute values) possible for the signal x, that is:*

$$\|x\|_{\ell^\infty} = \sup_{k\in\mathbb{N}} |x(k)| \tag{1.1.1}$$

1.1.3 Linear Time Invariant Filters (LTI filters)

A filter, denoted by its transfer function $\mathcal{H}$, is an application which transforms a signal vector $\boldsymbol{u}$ (with $dim(\boldsymbol{u}) = n_u$) into a signal vector $\boldsymbol{y} = \mathcal{H}(\boldsymbol{u})$, of size $dim(\boldsymbol{y}) = n_y$. The case where $n_u = n_y = 1$ is referred as Single Input Single Output (SISO) filters. Other casesare referred as Multiple Input Multiple Output (MIMO) filters.

Definition 3. *(Linear Time Invariant Filter)*

Linearity:

$$\mathcal{H}(\alpha \cdot \boldsymbol{u}_1 + \beta \cdot \boldsymbol{u}_2) = \alpha \cdot \mathcal{H}(\boldsymbol{u}_1) + \beta \cdot \mathcal{H}(\boldsymbol{u}_2)$$

Time invariance:

$$\{\mathcal{H}(\boldsymbol{u})(k - k_0)\}_{k\geq 0} = \mathcal{H}(\{\boldsymbol{u}(k - k_0)\}_{k\geq 0})$$

1.1.4 Impulse response

Definition 4. *(Impulse Response) A SISO filter may be defined by its impulse response, denoted h. h is the impulse response of H to the impulsion of Dirac. Indeed each input signal can be described as a sum of Dirac impulsions:*

$$u = \sum_{i \geq 0} u(l)\delta_l$$

where δ_l is a Dirac impulsion centered in l, that is:

$$\delta_l(k) = \begin{cases} 1 & when \ \ k = l \\ 0 & else \end{cases} \tag{1.1.2}$$

The linearity condition of $\mathcal{H}$ implies: $\mathcal{H}(u) = \sum_{l \geq 0} u(l)\mathcal{H}(\delta_l)$. Time invariance gives: $\mathcal{H}(\delta_l)(k) = h(k-l)$. Then the computation from inputs to outputs takes this form:

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

This corresponds with the convolution product definition of u by h, denoted $y = h * u$. Dealing with MIMO filters, $\boldsymbol{h}(k) \in \mathbb{R}^{n_y \times n_u}$ is the impulse response of $\mathcal{H}$. $\boldsymbol{h}_{i,j}(k)$ is the response on the ith output to the Dirac implusion on the j-th input. The precedent equation becomes:

$$y_i(k) = \sum_{j=1}^{n_u} \sum_{l=0}^{k} u_j(l)h_{i,j}(k-l), \ \ \forall 1 \leq i \leq n_y$$

1.1.5 Worst-Case Peak Gain (WCPG) of a Filter: the maximum possible amplification

Definition 5. *(Worst-Case Peak Gain) The worst case peak gain is defined as the maximum amplification possible over all potential inputs through the filter.*

$$\|\mathcal{H}\|_{wcpg} = \sup_{u \neq 0} \frac{\|h * u\|_{l^\infty}}{\|u\|_{l^\infty}}$$

with h the impulse response of $\mathcal{H}$, u the input signal, and $h * u$ the convolution product of h by u (output of the filter).

1.2 FIR and IIR: two filters families

There are two types of LTI filters: *Finite impulse response* (FIR) and *Infinite impulse response* (IIR) filters. Formally, the impulse response is finite when:

$$\exists n \in \mathbb{N} | \forall k \geq n, h(k) = 0 \tag{1.2.1}$$

The smallest n verifying 1.2.1 is referred as the order of the filter. So a n-order FIR can be described by the following equation:

$$y(k) = \sum_{i=0}^{n} b_i u(k-i) \tag{1.2.2}$$

An IIR will be described as following:

$$y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=0}^{n} a_i y(k-i) \tag{1.2.3}$$

Here one can observe that the output at time k depends also on all previous n outputs (loopback). Also remark that a FIR can be seen as an IIR with $\forall i \in [0, n], a_i = 0$ The impulse response can then be deduced from 1.2.3 by resolving the recurrence relation:

$$h(k) = \begin{cases} 0 & when \ \ k < l \\ b_k - \sum_{l=1}^{n} a_l h(k-l) & when \ \ 0 \leq k \leq n \\ \sum_{l=1}^{n} a_l h(k-l) & when \ \ n < k \end{cases} \tag{1.2.4}$$

1.3 Different realizations: how to compute the output of a filter?

Definition 6. *(realization) A realization can be defined as an algorithm describing how to compute outputs from inputs. However, a realization does not describes the details of basic operations (format, size, rounding, etc...)*

It is important to know that all realizations of a filter are mathematically equivalent to each other (infinite precision). But in finite precision, rounding aspects have a huge impact on the correctness of results. Most of the time in this report, realizations will be referred as forms.

1.3.1 Direct and transposed forms

Direct and transposed forms are classic realizations. A good description of these forms can be found in Lopez' and Hilaire's PhDs [6] [5]. The direct form has been implemented in the FoPoCo project, The hardware implementation of anIIR can be seen on figure 1

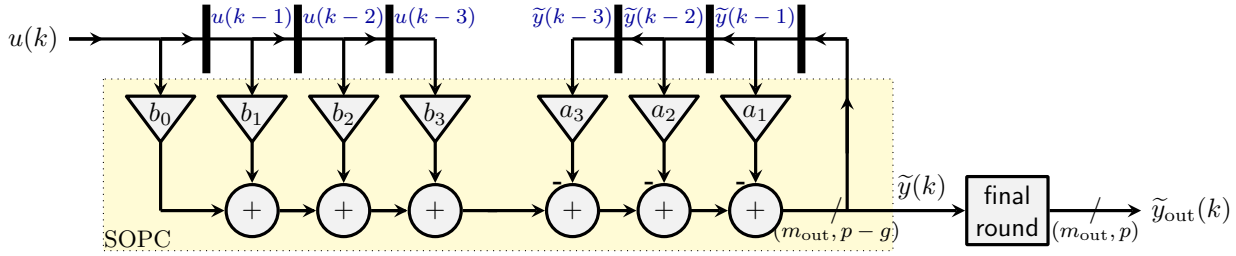

Figure 1: Abstract architecture for the direct form realization of an LTI filter

1.3.2 State-space representation: a recurrence to define an infinite response

This type of realization consists in expressing the evolution of a system considering its state at time k. In continuous time, it is described by differential equations at first order. In discret time (in which we are interested in), it is described by a simple recurrence:

$$\begin{cases} \boldsymbol{x}(k+1) = \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k) \\ \boldsymbol{y}(k+1) = \boldsymbol{C}\boldsymbol{x}(k) + \boldsymbol{D}\boldsymbol{u}(k) \end{cases} \tag{1.3.1}$$

Where $\boldsymbol{x}(k) \in \mathbb{R}^{n_x}$ is the state vector, $\boldsymbol{u}(k) \in \mathbb{R}^{n_u}$ is the input vector and $\boldsymbol{y}(k) \in \mathbb{R}^{n_y}$ is the output vector, at time k. The matrices $\boldsymbol{A} \in \mathbb{R}^{n_x \times n_x}$, $\boldsymbol{B} \in \mathbb{R}^{n_x \times n_u}$, $\boldsymbol{C} \in \mathbb{R}^{n_y \times n_x}$, and $\boldsymbol{D} \in \mathbb{R}^{n_y \times n_u}$, with $\boldsymbol{x}(0)$ are sufficient to describe an LTI filter, with convention $\boldsymbol{x}(k) = \boldsymbol{u}(k) = 0 \ , \ \forall k < 0$.

1.4 The Specialized Implicit Form (SIF): a unified representation

1.4.1 Definition

The classical state-space representation is intuitive, but it doesn't take into account the reality of implementation. The specialized implicit form (SIF) was introduced in [8], is well detailed in Hilaire's PhD and associated

papers [5, 7], and is a good answer to this problem. Indeed, dealing with finite precision and error amplification, the order in which operations are done becomes crucial. Another motivation is to have a unique representation for any realization of LTI filters, that allows to compute every degradation measures instead of redevelopping them for each new realization. This form distinguishes computations done at one time from computations done the other times. As well as in a state-space, $\boldsymbol{x}$-coordinates are state variables, but in addition to that, $\boldsymbol{t}$-coordinates are intermediate variables. The SIF is described as following:

$$\begin{pmatrix} \boldsymbol{J} & \boldsymbol{0} & \boldsymbol{0} \\ -\boldsymbol{K} & \boldsymbol{I}_{n_x} & \boldsymbol{0} \\ -\boldsymbol{L} & \boldsymbol{0} & \boldsymbol{I}_{n_y} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k+1) \\ \boldsymbol{x}(k+1) \\ \boldsymbol{y}(k) \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{0} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{0} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k) \\ \boldsymbol{x}(k) \\ \boldsymbol{u}(k) \end{pmatrix} \tag{1.4.1}$$

With n_t, n_x, n_y and n_u the sizes of $\boldsymbol{t}$, $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{u}$, respectively. $\boldsymbol{J}$, is a lower triangular matrix, with diagonal entries equal to 1. The previous matrices have the following dimensions:

$$\begin{aligned} \boldsymbol{J} \in \mathbb{R}^{n_t \times n_t}, \boldsymbol{M} \in \mathbb{R}^{n_t \times n_x}, \boldsymbol{N} \in \mathbb{R}^{n_t \times n_u}, \\ \boldsymbol{K} \in \mathbb{R}^{n_x \times n_t}, \boldsymbol{P} \in \mathbb{R}^{n_x \times n_x}, \boldsymbol{Q} \in \mathbb{R}^{n_x \times n_u}, \\ \boldsymbol{L} \in \mathbb{R}^{n_y \times n_t}, \boldsymbol{R} \in \mathbb{R}^{n_y \times n_x}, \boldsymbol{S} \in \mathbb{R}^{n_y \times n_u}, \end{aligned} \tag{1.4.2}$$

(1.4.3)

The best way to understand the SIF may be to see it as an algorithm, each line of the equation 1.4.1 corresponding to a sequential step of the computation. The algorithm results as follows:

for *int* $i = 0$; $i \leq n_t$; i++ **do**
  $\boldsymbol{t}_i(k+1) \leftarrow -\sum_{j<i} \boldsymbol{J}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{M}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{N}_{ij}\boldsymbol{u}_j(k)$
end
for *int* $i = 0$; $i \leq n_x$; i++ **do**
  $\boldsymbol{x}_i(k+1) \leftarrow -\sum_{j=1}^{n_t} \boldsymbol{K}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{P}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{Q}_{ij}\boldsymbol{u}_j(k)$
end
for *int* $i = 0$; $i \leq ny$; i++ **do**
  $\boldsymbol{y}_i(k) \leftarrow -\sum_{j=1}^{n_t} \boldsymbol{L}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{R}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{S}_{ij}\boldsymbol{u}_j(k)$
end

Algorithm 1: Computation of SIF outputs from inputs

Here, it is important to see that the form of $\boldsymbol{J}$ allows to compute the $\boldsymbol{t}_i$ sequentially. The algorithm can then be described as follows:

for *int* $i = 0$; $i \leq n_t$; i++ **do**
  $\boldsymbol{t}_i(k+1) \leftarrow -\boldsymbol{J'}_i\boldsymbol{t}(k+1) + \boldsymbol{M}_i\boldsymbol{x}(k) + \boldsymbol{N}_i\boldsymbol{u}(k)$
end
for *int* $i = 0$; $i \leq n_x$; i++ **do**
  $\boldsymbol{x}_i(k+1) \leftarrow -\boldsymbol{K}_i\boldsymbol{t}(k+1) + \boldsymbol{P}_i\boldsymbol{x}(k) + \boldsymbol{Q}_i\boldsymbol{u}(k)$
end
for *int* $i = 0$; $i \leq ny$; i++ **do**
  $\boldsymbol{y}_i(k) \leftarrow -\boldsymbol{L}_i\boldsymbol{t}(k+1) + \boldsymbol{R}_i\boldsymbol{x}(k) + \boldsymbol{S}_i\boldsymbol{u}(k)$
end

Algorithm 2: Simplified matricial algorithm

With $\boldsymbol{J'} = \boldsymbol{J} - I_{n_t}$.

Values of the vector $\boldsymbol{t}(k+1)$ are computed and used at the same iterations, so they are not kept in memory. As the equation 1.4.1 is mostly full of zeros, it is more convenient to use it's compressed formulation, wich is denoted $\boldsymbol{Z}$:

$$\boldsymbol{Z} = \begin{pmatrix} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} \tag{1.4.4}$$

The community usually takes $-\boldsymbol{J}$ for simplicity within further computations.

The SIF can of course be transformed into an equivalent ABCD (classic state-space) form, which gives:

$$\begin{aligned} \boldsymbol{A_Z} = \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{P}, \;\; \boldsymbol{B_Z} = \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{Q}, \\ \boldsymbol{C_Z} = \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{R}, \;\; \boldsymbol{D_Z} = \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{S}, \end{aligned} \tag{1.4.5}$$

with:

$$\begin{aligned} \boldsymbol{A_Z} \in \mathbb{R}^{n_x \times n_x}, \boldsymbol{B_Z} \in \mathbb{R}^{n_x \times n_u}, \\ \boldsymbol{C_Z} \in \mathbb{R}^{n_y \times n_x}, \boldsymbol{D_Z} \in \mathbb{R}^{n_y \times n_u}, \end{aligned} \tag{1.4.6}$$

1.4.2 Workflow for filter implementation

The choice of the SIF realization is not concerned by this work. This choice is up to the user, who may have a lot of good reasons not to design a realization that seem intuitive for us. The choice of the realization is a job done at LIP6 in the PEQUAN team, under the metalibm ANR project. The present work is just a hardware backend for the choice part. Of course, optimizations are available at every level. The expected use of this work is shown on figure 2.

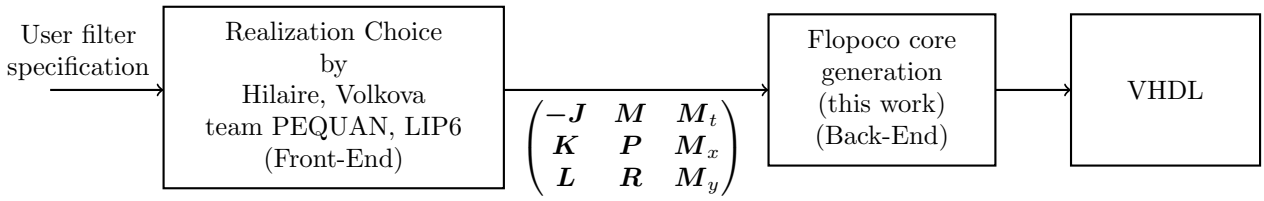

Figure 2: Workflow overview of tools usage

2 Architecture generation

2.1 Implementation: what can be done in practise?

2.1.1 The FloPoCo Framework: computing just right

FloPoCo is a C++ framework [4] which first purpose is to generate floating point cores in VHDL. It targets the configuration of FPGAs as computing units, and could also be useful dealing with ASIC specification. Indeed, FPGAs are well used for embedded computing, because they offer full reconfigurability and a relatively good balance between computation power and price.

More information is available at http://flopoco.gforge.inria.fr/

This work proposes to implement SIFs using the SOPC arithmetical core from FloPoCo. SOPC stands for Sum of Products by Constants, and is a KCM-multiplier based architecture. The details of the SOPC architecture has been described in [3].

2.1.2 KCM multipliers: an architecture to efficiently compute products on FPGAs

To be quick, a KCM multiplier is designed to fully use the power of look-up tables (LUTs) of an FPGA. To do so, it partitions the constant into chunks which match the size of a LUT. Then, tabulating the multiplications ends up with a final sum to compute the product. This has been first described by K.Chapman in [1]. The detail of implementation is visible on figure 3

2.1.3 SOPCs: exploiting full parallel architecures

The SOPC architecture keeps the same idea. The difference resides in the mutualization of the summation architecture through the n_c products. This summation is performed within a bit-heap based architecture implemented by FloPoCo. The detail is visible on figure 4.

To generate such cores, FloPoCo needs a number format specification, that is, the most significant bit (msb) and last significant bit (lsb) for the inputs and output of each operator to generate.

2.1.4 Precision concerns

In SOPCs architectures, accuracy is deducible from the inputs/output specifications and the size of the constants. This is described in [3].

Dealing with feedback inputs to build an SOPC-based filter, the question of precision is more complicated. Indeed, when outputs loop back on inputs, as soon as the problem is in finite precision, the error is amplified by a certain amount, depending on the coefficients, at each pass through the filter.

In this case the solution in the industry is to build an equivalent FIR filter by resolving the state-space recurrence. This leads to build an accurate hardware, but at the price of a huge waste of logic. The present work tries to answer this problem, trying to compute just right, keeping the recurrence and saving hardware.

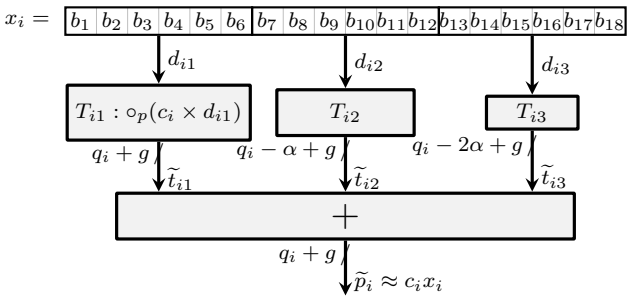

Figure 3: The FixRealKCM method when x_i is split in 3 chunks

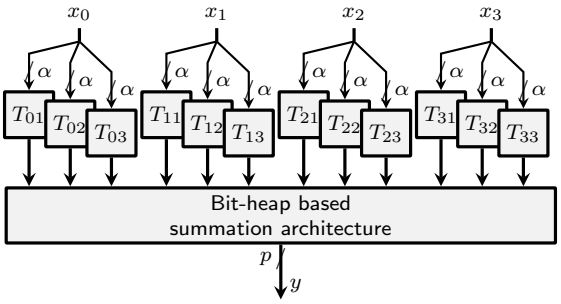

Figure 4: KCM-based SPOC architecture for $n_c = 4$, each input being split into 3 chunks

The main idea to dimension filters is to consider the total error as a single filter. The result of this filter is then added to the "perfect" filter to get the final output.

The two incoming parts are calculations from Lopez's PhD [6], with adaptations in the architecture context. Indeed, the original calculations are done for fixed precision, because they are expected to be done in software, with fixed word sizes. The challenge here is to produce architecture with arbitrary precision, that is, choosing the best msbs and lsbs for each basic operation.

2.1.5 Size computation in finite precision

When there are many potential realizations for a single LTI filter, the choice of one realization among the others is very related to the error analysis in output. The precision of our computations can then be defined so that the following condition is satisfied:

$$\varepsilon_{y_i} < 2^{-lsb_{y_i}} \quad \forall i \in [1, n_y] \tag{2.1.1}$$

With lsb_{y_i} the last significant bit of the output y_i, and ε_{y_i} the error on the computation of the output y_i

Following the same model, an error term for each SOPC is involved in the computation. All these error terms will be functions of the most significant and last significant bits (respectively msb and lsb) of every input, output and intermediate signal. So there is a need for computing every $\{msb_{t_i}, lsb_{t_i}\}$, $\{msb_{x_i}, lsb_{x_i}\}$ and $\{msb_{y_i}, lsb_{y_i}\}$,

This leads to the definition of errors introduced by the SOPCs architectures, based on the SIF computation algorithm:

$$\boldsymbol{\varepsilon}_t(k) = -\boldsymbol{J't}^*(k+1) + \boldsymbol{M}\boldsymbol{x}^*(k) + \boldsymbol{N}\boldsymbol{u}(k) - SOPC(\boldsymbol{J'}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{t}^*(k), \boldsymbol{x}^*(k), \boldsymbol{u}(k)), \tag{2.1.2}$$

$$\boldsymbol{\varepsilon}_x(k) = \boldsymbol{K}\boldsymbol{t}^*(k+1) + \boldsymbol{P}\boldsymbol{x}^*(k) + \boldsymbol{Q}\boldsymbol{u}(k) - SOPC(\boldsymbol{K}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{t}^*(k), \boldsymbol{x}^*(k), \boldsymbol{u}(k)), \tag{2.1.3}$$

$$\boldsymbol{\varepsilon}_y(k) = \boldsymbol{L}\boldsymbol{t}^*(k+1) + \boldsymbol{R}\boldsymbol{x}^*(k) + \boldsymbol{S}\boldsymbol{u}(k) - SOPC(\boldsymbol{L}, \boldsymbol{R}, \boldsymbol{S}, \boldsymbol{t}^*(k), \boldsymbol{x}^*(k), \boldsymbol{u}(k)), \tag{2.1.4}$$

where $SOPC(\boldsymbol{L}, \boldsymbol{R}, \boldsymbol{S}, \boldsymbol{t}^*(k), \boldsymbol{x}^*(k), \boldsymbol{u}(k))$ is the effective computation (with errors) of the sum of products.

Let's define the following vectors:

$$\boldsymbol{\varepsilon}(k) = \begin{pmatrix} \boldsymbol{\varepsilon_t}(k) \\ \boldsymbol{\varepsilon_x}(k) \\ \boldsymbol{\varepsilon_y}(k) \end{pmatrix} = \begin{pmatrix} \varepsilon_{t_1}(k) \\ \varepsilon_{t_2}(k) \\ \vdots \\ \varepsilon_{t_{n_t}}(k) \\ \\ \varepsilon_{x_1}(k) \\ \varepsilon_{x_2}(k) \\ \vdots \\ \varepsilon_{x_{n_x}}(k) \\ \\ \varepsilon_{y_1}(k) \\ \varepsilon_{y_2}(k) \\ \vdots \\ \varepsilon_{y_{n_y}}(k) \end{pmatrix} \tag{2.1.5}$$

2.1.6 Error analysis

Considering a filter $\mathcal{H}$ and it's SIF, following the algorithm 2, in finite precision, leads to the following equations:

$$\boldsymbol{t}^*(k+1) = -\boldsymbol{J'}\boldsymbol{t}^*(k+1) + \boldsymbol{M}\boldsymbol{x}^*(k) + \boldsymbol{N}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_t}(k) \tag{2.1.6}$$
$$\boldsymbol{x}^*(k+1) = \boldsymbol{K}\boldsymbol{t}^*(k+1) + \boldsymbol{P}\boldsymbol{x}^*(k) + \boldsymbol{Q}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_x}(k) \tag{2.1.7}$$
$$\boldsymbol{y}^*(k) = \boldsymbol{L}\boldsymbol{t}^*(k+1) + \boldsymbol{R}\boldsymbol{x}^*(k) + \boldsymbol{S}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_y}(k) \tag{2.1.8}$$

Here $\boldsymbol{t}^*$, $\boldsymbol{x}^*$ and $\boldsymbol{y}^*$ are the computed vectors, so with computations errors.

As $\boldsymbol{\varepsilon}$-errors come only from the SOPCs cores, another error term will symbolize the total error. Let's denote:

$$\boldsymbol{\varepsilon}^*_{t_i}(k+1) = \boldsymbol{t}^*_i(k) - \boldsymbol{t}_i(k) \tag{2.1.9}$$
$$\boldsymbol{\varepsilon}^*_{x_i}(k+1) = \boldsymbol{x}^*_i(k) - \boldsymbol{x}_i(k) \tag{2.1.10}$$
$$\boldsymbol{\varepsilon}^*_{y_i}(k) = \boldsymbol{y}^*_i(k) - \boldsymbol{y}_i(k) \tag{2.1.11}$$

the total error at instant k, considering computations errors and loopback, for $\boldsymbol{t}$, $\boldsymbol{x}$ and $\boldsymbol{y}$. The equations, corresponding to an algorithm, become:

$$\boldsymbol{\varepsilon}^*_t(k+1) = -\boldsymbol{J'}\boldsymbol{\varepsilon}^*_t(k+1) + \boldsymbol{M}\boldsymbol{\varepsilon}^*_x(k) + \boldsymbol{\varepsilon_t}(k) \tag{2.1.12}$$
$$\boldsymbol{\varepsilon}^*_x(k+1) = \boldsymbol{K}\boldsymbol{\varepsilon}^*_t(k+1) + \boldsymbol{P}\boldsymbol{\varepsilon}^*_x(k) + \boldsymbol{\varepsilon_x}(k) \tag{2.1.13}$$
$$\boldsymbol{\varepsilon}^*_y(k) = \boldsymbol{L}\boldsymbol{\varepsilon}^*_t(k+1) + \boldsymbol{R}\boldsymbol{\varepsilon}^*_x(k) + \boldsymbol{\varepsilon_y}(k) \tag{2.1.14}$$

This new algorithm corresponds here to the algorithm of the SIF of a filter $\mathcal{H}_{\boldsymbol{\varepsilon}}$, which describes the behaviour of computation errors at time k on the output. The linearity condition allows to decompose the real $\mathcal{H}^*$ filter in two distinct filters:

- $\mathcal{H}$ the absolute filter in infinite precision
- $\mathcal{H}_{\boldsymbol{\varepsilon}}$ the error filter

According to 1.4.4:

$$\boldsymbol{Z_\varepsilon} = \begin{pmatrix} \boldsymbol{-J} & \boldsymbol{M} & \boldsymbol{M}_t \\ \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{M}_x \\ \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{M}_y \end{pmatrix} \tag{2.1.15}$$

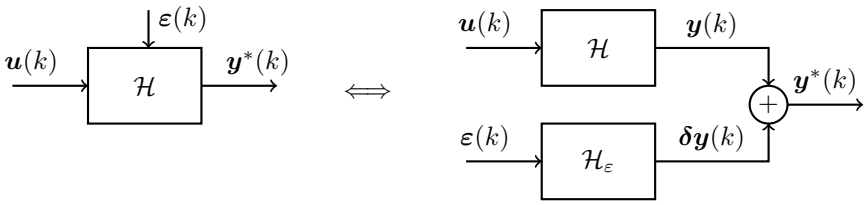

Figure 5: A signal view of the error propagation with respect to the ideal filter

with:

$$\boldsymbol{M}_t = (\boldsymbol{I}_{n_t} \;\; \boldsymbol{0}_{n_t \times n_x} \;\; \boldsymbol{0}_{n_t \times n_y}), \tag{2.1.16}$$
$$\boldsymbol{M}_x = (\boldsymbol{0}_{n_x \times n_t} \;\; \boldsymbol{I}_{n_x} \;\; \boldsymbol{0}_{n_x \times n_y}), \tag{2.1.17}$$
$$\boldsymbol{M}_y = (\boldsymbol{0}_{n_y \times n_t} \;\; \boldsymbol{0}_{n_y \times n_x} \;\; \boldsymbol{I}_{n_y}), \tag{2.1.18}$$

$\mathcal{H}_{\boldsymbol{\varepsilon}}$ is a filter with $(n_t + n_x + n_u)$ inputs and n_y outputs.

Proposition 1. The transfert function of filter $\mathcal{H}_{\boldsymbol{\varepsilon}}$, denoted $\boldsymbol{H}_\varepsilon$, is defined as follows:

$$\boldsymbol{H}_\varepsilon :\rightarrow \boldsymbol{C_Z}(z\boldsymbol{I}_n - \boldsymbol{A_Z})^{-1}\boldsymbol{M}_1 + \boldsymbol{M}_2 \;\; \forall z \in \mathbb{C} \tag{2.1.19}$$

with $\boldsymbol{A_Z}$ and $\boldsymbol{C_Z}$ the matrices defined by 1.4.5 and

$$\boldsymbol{M_1} = (\boldsymbol{K}\boldsymbol{J}^{-1} \quad \boldsymbol{I}_{n_x} \quad \boldsymbol{0}), \;\; \boldsymbol{M_2} = (\boldsymbol{L}\boldsymbol{J}^{-1} \quad \boldsymbol{0} \quad \boldsymbol{I}_{n_y}), \tag{2.1.20}$$

The demonstration is well detailed in Lopez' PhD [6].

Corollary 1. Considering a filter $\mathcal{H}$, $\boldsymbol{\varepsilon}(k)$ the vector of computation errors at time k in the finite precision of $\mathcal{H}$, and $\mathcal{H}\boldsymbol{\varepsilon}$ the error filter associated to $\mathcal{H}$. The behaviour of error can be described from $\boldsymbol{\varepsilon}(k)$ and $\mathcal{H}\boldsymbol{\varepsilon}$. The error is considered as an interval vector, denoted by its center and radius $\langle \boldsymbol{\varepsilon}_m, \boldsymbol{\varepsilon}_r \rangle$ and the interval vector of global error $\boldsymbol{\varepsilon}^*_y$, denoted $\langle \boldsymbol{\varepsilon}^*_{y_m}, \boldsymbol{\varepsilon}^*_{y_r} \rangle$.

In practise, all inputs in our case are centered around zero, which is not the case in the control community, where this notation makes sense. So, $\boldsymbol{\varepsilon}_m = 0$. The results are the following:

$$\boldsymbol{\varepsilon}^*_{y_m} = 0 \tag{2.1.21}$$
$$\boldsymbol{\varepsilon}^*_{y_r} = \langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle_{wcpg} \cdot \boldsymbol{\varepsilon}_r \tag{2.1.22}$$

In the following $\boldsymbol{\varepsilon}^*_{y_m}$ won't be taken into account.
Let's define:

$$n' = n_t + n_x + n_y$$

and:

$$\boldsymbol{v'} = \begin{pmatrix} \boldsymbol{t}(k+1) \\ \boldsymbol{x}(k+1) \\ \boldsymbol{y}(k) \end{pmatrix} \tag{2.1.23}$$

Then, following Lopez's computations, precisions are derived for every intermediate step:

$$|\boldsymbol{\varepsilon}^*_{y_i}| \leq \sum_{j=1}^{n'} |\langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle_{i,j}| \cdot \boldsymbol{2}^{\boldsymbol{lsb}_{\boldsymbol{v}'_j}} \tag{2.1.24}$$

To formalize with a matricial formulation:

$$|\boldsymbol{\varepsilon}^*_y| \leq |\langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle| \cdot \boldsymbol{2}^{\boldsymbol{lsb}_{\boldsymbol{v}'}} \tag{2.1.25}$$

To satisfy the condition 2.1.1, ξ is defined as the minimal error the user wants, which gives:

$$|\langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle| \cdot \mathbf{2}^{\boldsymbol{lsb_{v'}}} < \xi \tag{2.1.26}$$

That is,

$$\mathfrak{A} \cdot \mathbf{2}^{lsb_{v'} - msb_{v'} - 1} < \mathbf{1}_{n_y} \tag{2.1.27}$$

Where:

$$\mathfrak{A}_{i,j} = |\langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle_{i,j}| \cdot \frac{\mathbf{2}^{\boldsymbol{msb_{v'_j}+1}}}{\xi_i} \tag{2.1.28}$$

So, all lsb_{v_i} from all msb_{v_i} can be deduced from . As msb_{v_i} can be deduced directly from $\langle\langle \mathcal{H} \rangle\rangle_{wcpg}$, choosing and computing each operator precision is possible.

2.1.7 Worst-case peak gain computation

Computing the worst-case peak gain accurately in finite precision is a non-trivial problem. A state of the art implementation is being developped at LIP6 by Anastatsia Lozanova Volkova from Hilaire's team (PEQUAN) [9]. This work will use it as soon as it is available.

2.2 Architecture generation algorithm

computePrecisions($[msbs, lsbs][][]$) //get the matrix of msbs lsbs, functions of the wcpg.
for *i=1; i=Z.size(); i++* **do**
  row[ ]= Z[i][ ] //pick first row of Z
  for *j=1; j=1; j=Z.size() j++* **do**
    assign(SOPC[i], row[j], "T","X","U",[msbs,lsbs][i][j])
  end
  Second pass for wiring.
end

An example of implementation for a real-life case is given on figure 6.
On the figure 6, the colors in the matrix represent the different steps of computaions:

- blue for $\boldsymbol{t}$ computations
- green for $\boldsymbol{x}$ computations
- green for $\boldsymbol{y}$ computations

On the architecture scheme:

- grey background for x registers
- purple for loopback from the x registers
- orange for inputs

A small remark is needed here: most of coefficients of the $\boldsymbol{J}$ coefficient are null in the original case specification. They are just kept under this form to show how the algorithm works and how the architecture is built.

2.3 Particular Forms, degenerated cases

2.3.1 ABCD Form: still achievable

The ABCD Form can be considered as a degenerated form of the SIF, with $n_t = 0$. The algorithm will work in this case too.

$$Z = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & m_{1,1} & 0 & 0 & 0 & 0 & 0 \\
j_{2,1} & 1 & 0 & 0 & 0 & 0 & m_{2,2} & 0 & 0 & 0 & 0 \\
j_{3,1} & j_{3,2} & 1 & 0 & 0 & 0 & 0 & m_{3,3} & 0 & 0 & 0 \\
j_{4,1} & j_{4,2} & j_{4,3} & 1 & 0 & 0 & 0 & 0 & m_{4,4} & 0 & 0 \\
j_{5,1} & j_{5,2} & j_{5,3} & j_{5,4} & 1 & 0 & 0 & 0 & 0 & m_{5,5} & 0 \\
k_{1,1} & 1 & 0 & 0 & 0 & p_{1,1} & 0 & 0 & 0 & 0 & q_{1,1} \\
k_{2,1} & 0 & 1 & 0 & 0 & 0 & p_{2,2} & 0 & 0 & 0 & q_{2,1} \\
k_{3,1} & 0 & 0 & 1 & 0 & 0 & 0 & p_{3,3} & 0 & 0 & q_{3,1} \\
k_{4,1} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & p_{4,4} & 0 & q_{4,1} \\
k_{5,1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & p_{5,5} & q_{5,1} \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

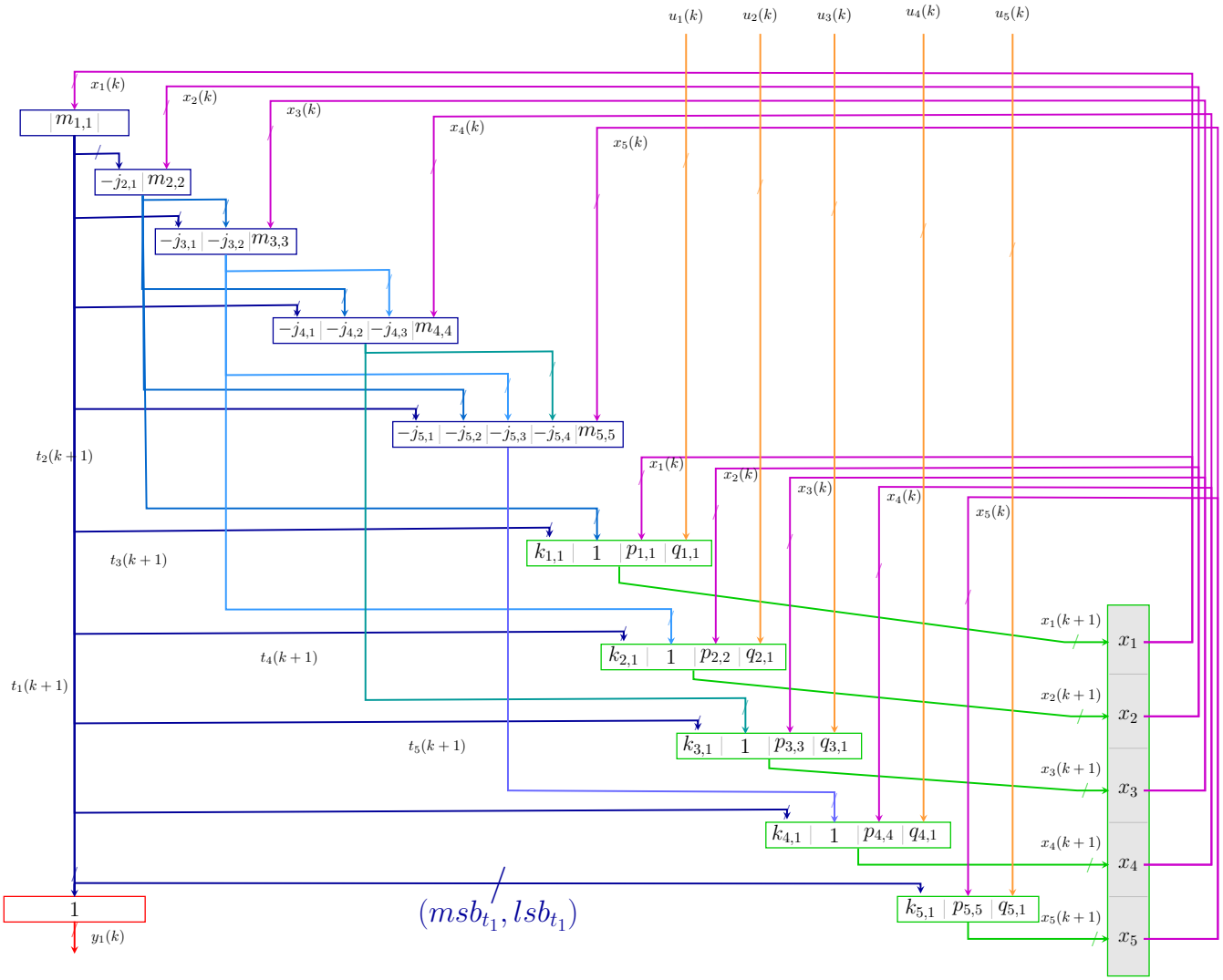

Figure 6: Architecture generation for implementing a SIF (example from a rho-DFII filter), with $n_t = 5$, $n_x = 5$ and $n_y = 1$

2.3.2 When $n_x = 0$

When $n_x = 0$, the interest of using a SIF is of course very limited. This case is equivalent to building a FIR, and no loopback is needed. Still, the algorithm will work, allocating only SOPCs operators.

2.4 Optimizations

2.4.1 Sparse matrices

The Z matrix of a SIF is most of the time sparse. So it is useful to remove zeros coefficients before allocating SOPCs. Indeed, it prevents useless inputs to be declared and can save a lot of hardware, although the HDL compiler might be able to optimize the hardware and remove "dead code". Anyway, it is healthy to keep a low compile time (either in FloPoCo or in the HDL compiler). Keeping the VHDL clean is always important, first for debugging issues, but also for comprehensiveness. This is already done in the current implementation.

2.4.2 Power of two coefficients

Coefficients equal to one or a power of two in the Z matrix can be interpreted as simple wires instead of multiplications in the SOPC. Entries equal to powers of two in SOPCs may be replaced by simple additions with the result of the SOPC. Here, the question of what solution is the best in terms of hardware consumption should be investigated (speed is not concerned here because the speed is determined by the length of the loop). But this work should in fact be done in KCM and SOPC, because they will be able to do the job without analysis and reporting precision feedback.

2.5 Implementation details

This implementation work has been done in the FixFilter section of the FloPoCo project. It is for now about 900 lines and contains:

- the implentation of the SIF algorithm
- the specification parser
- the testing framework (emulate method)

The testing framework consists on pre-computing expected results using the sollya lib [2]. Then the FloPoCo framework generates a VHDL testbench that embed the hardware architecture and the test of expected results. As an exhaustive testbench is impossible to generate ($2^{msb_{In}-lsb_{In}}{}^{2^m sbIn-lsbIn}$ values to test for each input), FloPoCo then generates a user-defined number of random tests.

2.6 Filter specification interface

To communicate with our SIF operator in FloPoCo, simply pass the coefficients in command line as it is done for the direct form is not affordable. So, a simple file format has been defined, in collaboration with our co-workers at LIP6, to store all the coefficients. The format is defined as follows:

```
X l c
x_1_1 x_1_2 ... x_1_c
x_2_1 x_2_2 ... x_2_c
.
.
.
x_l_1 x_l_2 ... x_l_c
```

Where X is the name of the matrix (X ∈ { J, K, L, M, N, P, Q, R, S, T}), x_i_j is the coefficient (with $i \in [1, l]$ and $j \in [1, c]$), l is the number of lines and c the numer of columns.

All the matrices are specified after each other in the file.

For now another file is used to specify sizes because we are still waiting for the wcpg-code.

3 Further work

3.1 Static analysis on coefficients

Detection of and removal of null coefficients is already implemented. Detecting power of two entries in the coefficients seem to be a quite simple improvement to implement. However, this part should be left to the implementation of SOPCs, and so KCM multipliers. Indeed, pushing this detection at the lowest level is easy and gives the opportunity to the lower operators to give a feedback about their actual accuracy. Detecting a power of two in a KCM multiplier with feedback on precision will help to build better calculations on msbs and lsbs. Moreover, doing the job in the KCM operator will prevent spending compile time analysing the list of coefficients.

3.2 Sub-filter detection

Detecting independent loops can be very interesting in the context of saving hardware. Considering a sub-filter with its own loop can permit to use its WCPG to compute the precisions for this sub-filter. This leads to another resolution with a precision specification that depends on the rest of the filter (logic after the sub-filter in the pipeline). However, this optimization can be done at the front-end step, which leads to an open question. In fact, detecting such loops has issues both in front-end and in backend, and is a non-trivial problem. So it remains as an open question for future investigations.

3.3 Precision calculations improvement

This work kept original calculations from Lopez's PhD [6], trying to adapt them to our context. However, lots of approximations are done through those calculations. Working on this part, going back over all the calculations could really improve the efficiency and size of all hardware implementations.

3.4 File format re-specification

Specifying new formats could help to improve the visibility and the usability of this new operator. The idea is to avoid specification output formatting operations for the user. In the future, the present implementation should recognize input matrices formated in the Python or Matlab formats.

Conclusion

This work tried to give an overview of LTI filters, and the considerations to take into account when trying to implement them. This report tried to adapt concepts and calculations from Lopez's and Hilaire's work to the context of generating architectures computing just right. This context is merging ideas from the communities of automatic, signal, and computer arithmetics, so a certain amount of time was spent on the unification of notations and conventions. Finally, the implementation of a parametric definition in the FloPoCo framework was started. This was permitted through the definition of a specification format and the integration of external code from LIP6. Further work will take into account many improvements and optimizations, such as:

- static analysis on coefficients and power of two removing
- sub-filter detection
- bounds and precision computation improvements
- format re-specification

References

[1] K.D. Chapman. Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner). *EDN magazine*, 39(10):80, May 1993.

[2] S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An environment for the development of numerical codes. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 28–31, Heidelberg, Germany, September 2010. Springer.

[3] Florent de Dinechin, Matei Istoan, and Abdelbassat Massouri. Sum-of-product architectures computing just right. In *Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2014.

[4] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, July 2011.

[5] Thibaut Hilaire. *Analyse et synthèse de l'implémentation de lois de contrôle-commande en précision finie*. PhD thesis, Université de Nantes, 2006.

[6] Benoit Lopez. *Implémentation optimale de filtres linéaires en arithmétique virgule fixe*. PhD thesis, Université Paris VI, 2014.

[7] P.Chevel T.Hilaire and J.F.Whidbornz. A unifying framework for finite wordlength realizations. In *IEEE*, 2007.

[8] P.Chevel T.Hilaire and Y.Trinquet. Implicit state-space representation: a unifying framework for FWL implementation of LTI systems. In *Proc. of the 16th IFAC World Congress.*, 2005.

[9] A. Volkova, T. Hilaire, and C. Lauter. Reliable evaluation of the Worst-Case Peak Gain matrix in multiple precision. In *IEEE Symposium on Computer Arithmetic*, 2015.