Architecture synthesis for linear time-invariant filters

Antoine Martinet

June 10, 2015

Contents

Introduction

Filters are nowadays essential tools for designing responsive systems. As they appear in both signal processing and command communities, we can see that the use of filters is now spreading in many domains, related to them, such as radio signal coding and decoding, image and sound processing, and so on.

Although most of all the potential filters can be computed in software, some interesting applications in hardware could help to accelerate such computations, that might be very long in software in particular cases. Moreover, it allows to design and reconfigure specific parts of the hardware when working with FPGAs.

My work at the CITI lab in the SOCRATE team was to design a parametric architecture generator for LTI filters, which will be very useful to design a software-defined radio. This work has been integrated to FloPoCo tool, whose main goal is to generate architecture cores for computing just right.

In this report, I will first present reminders about signal processing. Then I will tell a little more about LTI filters and their software implementation in finite precision. Finally, I will present the main aspects of the hardware implementation of such filters.

1 Motivations

The main motivation of this work is to be able to compute efficiently LTI Filters in hardware. A more precise problem is the loopback problem, that is, when a filter uses its own outputs as inputs. Indeed, as the output is infinite, the problems of precision induce infinitely amplified errors, which cause the result to be unusable. In this case, the solution retained in the industry is for now to unroll such loops, building a very heavy hardware and wasting logic. The purpose of this work is to be able to build a hardware computing just right, avoiding logic waste.

2 Tools for expressing filters and signal processing

2.0.1 Definition of a signal

The state of the art comes mainly from Lopez's PhD [4] and Hilaire's PhD [3].

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

2.0.2 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 $\boldsymbol{y} = \mathcal{H}(\boldsymbol{u})$, of size $dim(\boldsymbol{y}) = n_y$. When $n_u = n_y = 1$, we speak about Single Input Single Output (SISO) filters. In other cases, we speak about Multiple Input Multiple Output (MIMO) filters.

Definition 2. *(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:

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

2.0.3 Impulse response

Definition 3. *(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 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:

$$\delta(k) = \begin{cases} 1 & when\ \ k = l \\ 0 & else \end{cases} \tag{2.0.1}$$

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, we have $\boldsymbol{h} \in \mathbb{R}^{n_y \times n_u}$ as the impulse response of $\mathcal{H}$. $\boldsymbol{h}_{i,j}$ is the response on the ith output to the Dirac implusion on the j-th input. The precedent equation becomes:

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

2.0.4 Worst-Case Peak Gain (WCPG) of a Filter

Definition 4. *(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).

2.1 FIR and IIR

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

$$\exists n \in \mathbb{N} | \forall k \geq n, h(k) = 0 \tag{2.1.1}$$

The smallest n verifying 2.1.1 is referred as the order of the filter. So a n-order FIR can be described by the following equation:

$$y(k) = \sum_{i=0}^{n} b_i u(k-i) \tag{2.1.2}$$

An IIR will be described as following:

$$y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=0}^{n} a_i y(k-i) \tag{2.1.3}$$

Here one can observe that the output at time k depends also on all previous n outputs (loopback). One can also see 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 2.1.3 by resolving the recurrence relation:

$$h(k) = \begin{cases} 0 & whenk < l \\ b_k - \sum_{l=1}^{n} a_l h(k-l) & when0 \leq k \leq n \\ \sum_{l=1}^{n} a_l h(k-l) & whenn < k \end{cases} \tag{2.1.4}$$

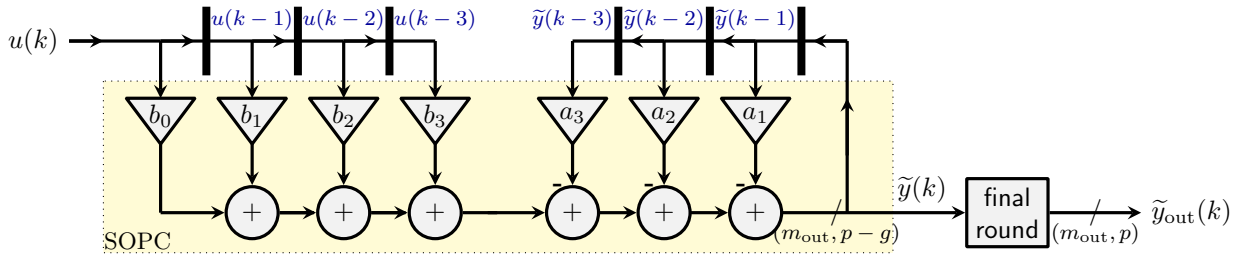

Figure 1: Abstract architecture for the direct form realization of an LTI filter

2.2 Realizations

Definition 5. *(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, order, 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, we will describe realizations as forms.

2.2.1 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 [4] [3]. The direct form has been implemented and in the FoPoCo project, you can see the hardware implementation an IIR on figure 2.2.1

2.2.2 State-space representation

This type of realization consists in expressing the evolution of a system considering it's 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:

$$\begin{cases} \boldsymbol{x}(k+1) = \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k) \\ \boldsymbol{y}(k+1) = \boldsymbol{C}\boldsymbol{x}(k) + \boldsymbol{D}\boldsymbol{u}(k) \end{cases} \tag{2.2.1}$$

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 $\boldsymbol{x}(k) = \boldsymbol{u}(k) = 0 \mid \forall k < 0$.

2.2.3 FloPoCo

FloPoCo is a C++ framework which first purpose is to generate floating point cores in VHDL. Although it may have some pertinence in the ASIC design market, it main target is the configuration of FPGAs as computing units. Indeed, FPGAs are a rising market 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/

2.3 Specialized Implicit Form (SIF)

he specialized implicit form (SIF) was introduced in [6] and is well detailed in [3, 5]. The classical state-space representation is intuitive, but it doesn't takes into account the reality of implementation. Indeed, dealing with finite precision and error amplification, the order in which operations are done becomes crucial. Another idea 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, we have $\boldsymbol{x}_i$ as state variables, but in addition to that, $\boldsymbol{t}_i$ are intermediate variables. The SIF is described as following:

$$\begin{pmatrix} \boldsymbol{J} & \boldsymbol{0} & \boldsymbol{0} \\ -\boldsymbol{K} & \boldsymbol{I}_{n_x} & \boldsymbol{0} \\ -\boldsymbol{L} & \boldsymbol{0} & \boldsymbol{I}_{n_y} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k+1) \\ \boldsymbol{x}(k+1) \\ \boldsymbol{y}(k) \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{0} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{0} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k) \\ \boldsymbol{x}(k) \\ \boldsymbol{u}(k) \end{pmatrix} \tag{2.3.1}$$

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. Then we have the following dimensions for the previous matrices:

$$\begin{aligned} &\boldsymbol{J} \in \mathbb{R}^{n_t \times n_t}, \boldsymbol{M} \in \mathbb{R}^{n_t \times n_x}, \boldsymbol{N} \in \mathbb{R}^{n_t \times n_u}, \\ &\boldsymbol{K} \in \mathbb{R}^{n_x \times n_t}, \boldsymbol{P} \in \mathbb{R}^{n_x \times n_x}, \boldsymbol{Q} \in \mathbb{R}^{n_x \times n_u}, \\ &\boldsymbol{L} \in \mathbb{R}^{n_y \times n_t}, \boldsymbol{R} \in \mathbb{R}^{n_y \times n_x}, \boldsymbol{S} \in \mathbb{R}^{n_y \times n_u}, \end{aligned} \tag{2.3.2}$$

(2.3.3)

The best way to understand the SIF may be to see it as an algorithm, each line of the equation 2.3.1 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**
 | $\boldsymbol{y}_i(k+1) \leftarrow -\sum_{j=1}^{n_t} \boldsymbol{L}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{R}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{S}_{ij}\boldsymbol{u}_j(k)$
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**
 | $\boldsymbol{y}_i(k+1) \leftarrow -\boldsymbol{L}_i\boldsymbol{t}(k+1) + \boldsymbol{R}_i\boldsymbol{x}(k) + \boldsymbol{S}_i\boldsymbol{u}(k)$
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 is mostly full of zeros, it is more convenient to use it's compressed formulation, wich is denoted as the $\boldsymbol{Z}$:

$$\boldsymbol{Z} = \begin{pmatrix} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} \tag{2.3.4}$$

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

We can of course bind the SIF with the ABCD (classic state-space) form, which gives:

$$\begin{aligned} \boldsymbol{A_Z} &= \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{P}, \;\; \boldsymbol{B_Z} = \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{Q}, \\ \boldsymbol{C_Z} &= \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{R}, \;\; \boldsymbol{D_Z} = \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{S}, \end{aligned} \tag{2.3.5}$$

With:

$$\begin{aligned} \boldsymbol{A_Z} \in \mathbb{R}^{n_x \times n_x}, \boldsymbol{B_Z} \in \mathbb{R}^{n_x \times n_u}, \\ \boldsymbol{C_Z} \in \mathbb{R}^{n_y \times n_x}, \boldsymbol{D_Z} \in \mathbb{R}^{n_x \times n_u}, \end{aligned} \tag{2.3.6}$$

2.4 Implementation

In this work, we propose to implements SIFs using the SOPCs arithmetical core from flopoco. SOPC stands for Sum of Producs by Constants, and is a KCM-multiplier based architecture. The details of the SOPC architecture has been described in [2]. To be quick, a KCM multiplier is designed to fully use the power of look-up tables (LUTs) in an FPGA. To do so, it partitions the constant into chunks which match the size of a LUT. Then, tabulating the multiplications, we end 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 2

The SOPC architecture keeps the same idea. The difference is, the summation architecture is mutualized through the n_c products in a bit-heap based architecture implemented by flopoco. The detail is visible on figure 3.

2.5 Size computation in finite precision

When there is many potential realizations for a single LTI filter, we can see that 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:

$$\varepsilon_{y_i} < 2^{-lsb_{y_i}} \;\; \forall i \in [1, n_y] \tag{2.5.1}$$

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, we define an error quantum for each SOPC involved in the computation. All these error quantas will be functions of the most significant and last significant bits (respectively msb and lsb) of every input, output and intermediate signal.

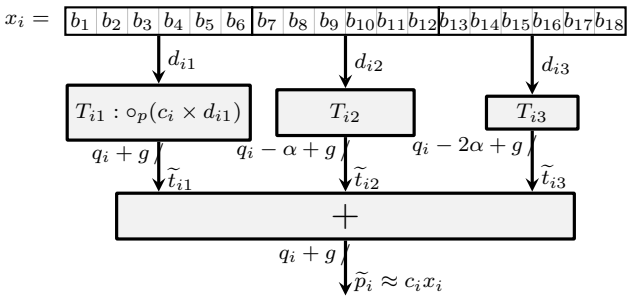

Figure 2: The FixRealKCM method when x_i is split in 3 chunks

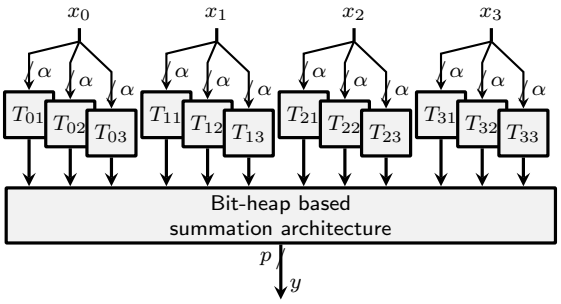

Figure 3: KCM-based SPOC architecture for $n_c = 4$, each input being split into 3 chunks

We define the following vectors:

$$\boldsymbol{\varepsilon}(k) = \begin{pmatrix} \boldsymbol{\varepsilon_t}(k) \\ \boldsymbol{\varepsilon_x}(k) \\ \boldsymbol{\varepsilon_y}(k) \end{pmatrix} = \begin{pmatrix} \varepsilon_{t_1}(k) \\ \varepsilon_{t_2}(k) \\ \vdots \\ \varepsilon_{t_{n_t}}(k) \\ \\ \varepsilon_{x_1}(k) \\ \varepsilon_{x_2}(k) \\ \vdots \\ \varepsilon_{x_{n_x}}(k) \\ \\ \varepsilon_{y_1}(k) \\ \varepsilon_{y_2}(k) \\ \vdots \\ \varepsilon_{y_{n_y}}(k) \end{pmatrix} \tag{2.5.2}$$

2.5.1 Error analysis

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

$$\boldsymbol{t}^*(k+1) = -\boldsymbol{J'}\boldsymbol{t}^*(k+1) + \boldsymbol{M}\boldsymbol{x}^*(k) + \boldsymbol{N}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_t}(k) \tag{2.5.3}$$
$$\boldsymbol{x}^*(k+1) = \boldsymbol{K}\boldsymbol{t}^*(k+1) + \boldsymbol{P}\boldsymbol{x}^*(k) + \boldsymbol{Q_i}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_x}(k) \tag{2.5.4}$$
$$\boldsymbol{y}^*(k+1) = \boldsymbol{L}\boldsymbol{t}^*(k+1) + \boldsymbol{R}\boldsymbol{x}^*(k) + \boldsymbol{S_i}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_y}(k) \tag{2.5.5}$$

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, we have another error quantum that will symbolize the total error. We denote

$$\boldsymbol{\delta t}(k+1) = \boldsymbol{t}_i^*(k) - \boldsymbol{t}_i(k) \tag{2.5.6}$$
$$\boldsymbol{\delta x}(k+1) = \boldsymbol{x}_i^*(k) - \boldsymbol{x}_i(k) \tag{2.5.7}$$
$$\boldsymbol{\delta y}(k+1) = \boldsymbol{y}_i^*(k) - \boldsymbol{y}_i(k) \tag{2.5.8}$$

the total error at instant k, considering computations errors and loopback, for $\boldsymbol{t}$, $\boldsymbol{x}$ and $\boldsymbol{y}$. We get then:

$$\boldsymbol{\delta t}(k+1) = -\boldsymbol{J'}\boldsymbol{\delta t}(k+1) + \boldsymbol{M}\boldsymbol{\delta x}(k) + \boldsymbol{\varepsilon_t}(k) \tag{2.5.9}$$
$$\boldsymbol{\delta x}(k+1) = \boldsymbol{K}\boldsymbol{\delta t}(k+1) + \boldsymbol{P}\boldsymbol{\delta x}(k) + \boldsymbol{\varepsilon_x}(k) \tag{2.5.10}$$
$$\boldsymbol{\delta y}(k+1) = \boldsymbol{L}\boldsymbol{\delta t}(k+1) + \boldsymbol{R}\boldsymbol{\delta x}(k) + \boldsymbol{\varepsilon_y}(k) \tag{2.5.11}$$

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

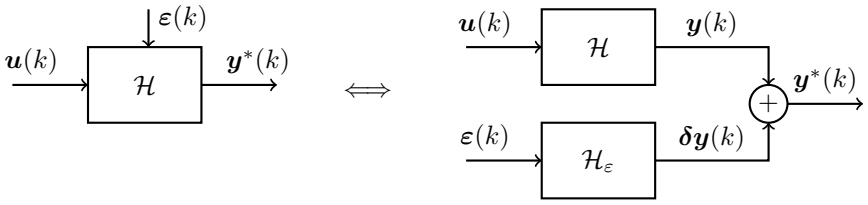

Figure 4: A signal view of the error propagation with respect to the ideal filter

According to 2.3.4, we have:

$$\boldsymbol{Z}_{\boldsymbol{\varepsilon}} = \begin{pmatrix} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{M}_t \\ \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{M}_x \\ \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{M}_y \end{pmatrix} \tag{2.5.12}$$

with:

$$\boldsymbol{M}_t = (\boldsymbol{I}_{n_t} \boldsymbol{0}_{n_t \times n_x} \boldsymbol{0}_{n_t \times n_y}), \tag{2.5.13}$$
$$\boldsymbol{M}_x = (\boldsymbol{0}_{n_x \times n_x} \boldsymbol{I}_{n_x} \boldsymbol{0}_{n_x \times n_y}), \tag{2.5.14}$$
$$\boldsymbol{M}_y = (\boldsymbol{0}_{n_y \times n_t} \boldsymbol{0}_{n_y \times n_x} \boldsymbol{I}_{n_y}), \tag{2.5.15}$$

$\mathcal{H}_{\boldsymbol{\varepsilon}}$ 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:

$$\boldsymbol{H}_{\varepsilon} :\to \boldsymbol{C}_{\boldsymbol{Z}}(z\boldsymbol{I}_n - \boldsymbol{A}_{\boldsymbol{Z}})^{-1}\boldsymbol{M}_1 + \boldsymbol{M}_2 \ \ \forall z \in \mathbb{C} \tag{2.5.16}$$

with $\boldsymbol{A}_{\boldsymbol{Z}}$ and $\boldsymbol{C}_{\boldsymbol{Z}}$ the matrices defined by 2.3.5 and

$$\boldsymbol{M}_{\boldsymbol{1}} = (\boldsymbol{K}\boldsymbol{J}^{-1} \quad \boldsymbol{I}_{n_x} \quad \boldsymbol{0}), \ \ \boldsymbol{M}_{\boldsymbol{2}} = (\boldsymbol{L}\boldsymbol{J}^{-1} \quad \boldsymbol{0} \quad \boldsymbol{I}_{n_y}), \tag{2.5.17}$$

The demonstration is well detailed in Lopez' PhD [4].

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 $\mathcal{H}\boldsymbol{\varepsilon}$. We consider the error 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 $\boldsymbol{\delta y}$, denoted $\langle \boldsymbol{\delta y}_m, \boldsymbol{\delta y}_r \rangle$. In practise, all inputs are centered around zero, which is not the case in the command community, where this notation makes sense. So we consider that $\boldsymbol{\varepsilon}_m = 0$. The results are the following:

$$\boldsymbol{\delta y}_m = 0 \tag{2.5.18}$$
$$\boldsymbol{\delta y}_r = \langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle_{wcpg} \cdot \boldsymbol{\varepsilon}_r \tag{2.5.19}$$

In the future we will then get rid of $bold symbol \delta y_m$.
Let's define:

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

and:

$$\boldsymbol{v'} = \begin{pmatrix} \boldsymbol{t}(k+1) \\ \boldsymbol{x}(k+1) \\ \boldsymbol{y}(k) \end{pmatrix} \tag{2.5.20}$$

Then, following Lopez's computations, we can derive precisions for every intermediate step:

$$|\boldsymbol{\delta y}_i| \leq \sum_{j=1}^{n'} |\langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle_{i,j}| \cdot \mathbf{2}^{\boldsymbol{l}_{\boldsymbol{v'_j}}} \tag{2.5.21}$$

To formalize with a matricial formulation, we get:

$$|\boldsymbol{\delta y}| \leq |\langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle| \cdot \mathbf{2}^{\boldsymbol{lsb}_{\boldsymbol{v'}}} \tag{2.5.22}$$

To satisfy the condition 2.5.1, we can simply define ξ as the minimal error the user wants. We can then state:

$$|\langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle| \cdot \mathbf{2}^{\boldsymbol{lsb}_{\boldsymbol{v'}}} < \xi \tag{2.5.23}$$

That is,

$$\mathfrak{A} \cdot \mathbf{2}^{lsb_{v'} - msb_{v'} - 1} < \mathbf{1}_{n_y} \tag{2.5.24}$$

Where:

$$\mathfrak{A}_{i,j} = |\langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle_{i,j}| \cdot \frac{\mathbf{2}^{\boldsymbol{msb}_{\boldsymbol{v'_j}} + \mathbf{1}}}{\xi_i} \tag{2.5.25}$$

$$\tag{2.5.26}$$

TODO: check if this step is sufficient.

2.5.2 A working solution to this problem

A solution poposed in Lopez's PhD [4], is to consider the following reformulation of constraint 2.5.25. We can give a stronger majoration, as follows:

$$\frac{\mathfrak{A}_{i1}}{\mathbf{2}^{msb_{v'_1} - lsb_{v'_1} + 1}} + \frac{\mathfrak{A}_{i2}}{\mathbf{2}^{msb_{v'_2} - lsb_{v'_2} + 1}} + \cdots + \frac{\mathfrak{A}_{in'}}{\mathbf{2}^{msb_{v'_{n'}} - lsb_{v'_{n'}} + 1}} < 1, \ \ \forall 1 \leq i \leq n_y \tag{2.5.27}$$

2.6 Worst-case peak gain computation

Computing the worst-case peak gain accurately in finite precision is a very complex problem. As we can't afford doing such a work, we are about to integrate code developped by Anastasia Lozanova Volkova from Hilaire's team at LIP6 that will be published soon.

2.7 Communication

To communicate with our SIF operator in FloPoCo, we can't simply pass the coefficients in command line as it is done for the first order of direct form. So we defined a simple file format 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 do not have not the integration of the "wcpg-code" done.

3 Dimensionment

3.1 Implementation

In SOPCs architectures, the accuracy is deducible from the inputs/outputs specifications and the size of the constants.

This is described in

Dealing with feedback inputs, the question of precision is more complicated. Indeed, when results loop back to inputs, as soon as we are in finite precision, the error is amplified by a certain amount, depending on the coefficients, at each pass through the filter.

The main idea to dimension such 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.

In finite precision, sizes are constrained to be all the same. The demonstration of the size computation has been described in Lopez' PhD [4]. The idea now is to see what we can do in arbitrary precision, trying to save a maximum of logic while keeping a right result on the precision required by the user.

Here we have to compute each size at each step of the computation. Indeed, the WCPG is not useful for the first part of a FIR, as it has no loop. So we just need the WCPG for the second part, because it is just in this part of the circuit that there is a potential error amplification.

Direct and transposed forms are not directly transposable into SIF, but this problem is secondary.

3.2 Algorithm

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], TXU) //where TXU, is the indicator of the signal (this is just determined by the position of the coefficient)
  end
  Second pass for wiring.
end

An example of implementation for a real-life case is given on figure 5.

On the figure 5, the colors in the matrix represent the different steps of computaions:

- blue for t computations
- green for x computations
- green for y computations

On the architecture scheme, we have:

- the grey background for indicating x registers
- purple for loopback form the x registers
- orange for inputs

A small precision is needed here. Here, most of coefficients of the $\boldsymbol{J}$ coefficient are null. We just kept them under this form to show how the algorithm works and how the architecture is built.

3.3 Particular Forms

3.4 ABCD Form

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.

$$\left(\begin{array}{cccccccccc|c}
1 & 0 & 0 & 0 & 0 & m_{1,1} & 0 & 0 & 0 & 0 & 0 \\
j_{2,1} & 1 & 0 & 0 & 0 & 0 & m_{2,2} & 0 & 0 & 0 & 0 \\
j_{3,1} & j_{3,2} & 1 & 0 & 0 & 0 & 0 & m_{3,3} & 0 & 0 & 0 \\
j_{4,1} & j_{4,2} & j_{4,3} & 1 & 0 & 0 & 0 & 0 & m_{4,4} & 0 & 0 \\
j_{5,1} & j_{5,2} & j_{5,3} & j_{5,4} & 1 & 0 & 0 & 0 & 0 & m_{5,5} & 0 \\
k_{1,1} & 1 & 0 & 0 & 0 & p_{1,1} & 0 & 0 & 0 & 0 & q_{1,1} \\
k_{2,1} & 0 & 1 & 0 & 0 & 0 & p_{2,2} & 0 & 0 & 0 & q_{2,1} \\
k_{3,1} & 0 & 0 & 1 & 0 & 0 & 0 & p_{3,3} & 0 & 0 & q_{3,1} \\
k_{4,1} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & p_{4,4} & 0 & q_{4,1} \\
k_{5,1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & p_{5,5} & q_{5,1} \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right)$$

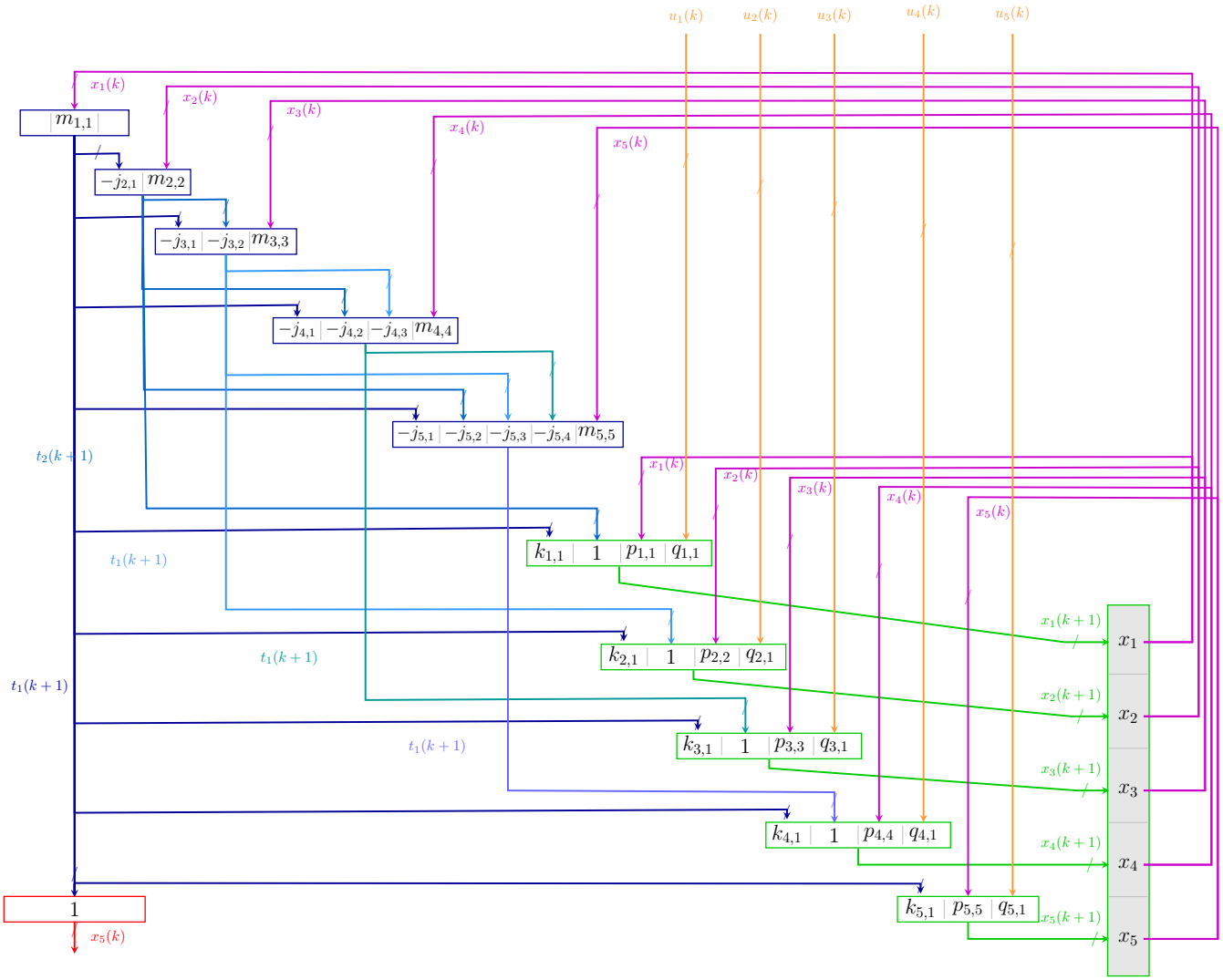

Figure 5: Architecture generation for implementing a SIF (example from a rho-DFII filter), with $n_t = 5$, $n_x = 5$ and $n_y = 1$

3.5 When $n_x = 0$

When $n_x = 0$, the interest of using implicit form 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. The computation of precisions are then equivalent to the FIR precision compuation, described in [].

3.6 Optimizations

3.6.1 Sparse matrices

The Z matrix of a SIF might be sparse in some degenerated cases. 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 more important, first for debugging issues, but also for comprehensiveness.

3.6.2 One entries

One entries in the Z matrix can be interpreted as simple wires instead of multiplications in the SOPC. So, we could eventually replace entries in SOPCs by simple additions with the result of the SOPC. Here, we should investigate to see what solution is the best in terms of hardware consumption (speed is not concerned here because the speed is determined by the length of the loop).

Conclusion

In this work, I tried to give an overview of LTI filters, and the considerations to take into account when trying to implement them. I tried to adapt concepts and computations 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. Finally, I began the implementation of a parametric definition in the FloPoCo framework. This was permitted through the definition of a specification language and the integration of external code from LIP6.

References

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

[2] Abdelbassat Massouri Florent de Dinechin, Matei Istoan. Sum-of-product architectures computing just right. In *IEEE*, 2007.

[3] Thibaut Hilaire. Analyse et synthèse de l'implémentation de lois de contrôle-commande en précision finie. In *PhD*, 2006.

[4] Benoit Lopez. Implémentation optimale de filtre linéaire en arithmétique virgule fixe. In *PhD*, 2014.

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

[6] 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.

Appendix: Reminders about signal processing

LTI filters in general are usually defined as sums of products. Several quantities are useful to understand their characteristics

3.7 Notations

During the entire report we will use several notations and conventions:

- in signal processing, t is the common notation for continue time and k the notation for discrete time. This report keeps this convention.
- $\langle\langle \mathcal{H} \rangle\rangle_{WCPG}$ is the worst case peak gain of the filter $\mathcal{H}$
- y is an output variable
- x is a state variable
- t is an intermediate variable
-
-

3.8 Remainders about signal processing

Definition 6. *(Signal) Generally, a signal is a temporal variable, which takes a value from $\mathbb{R}$ at each time t. We denote x(t) the value of the signal x at the instant t. When dealing with discrete time events, the time will be represented by k. Then we talk about x(k), which is said to be a sample. $\{x(k)\}_{k\geq 0}$ denotes all the values possible for the signal x. In the rest of this report, we will talk about vectors of signals $\boldsymbol{x}$, where $\boldsymbol{x}(k) \in \mathbb{R}^n$*

Definition 7. *(ℓ^1 norm) The ℓ^1 norm of a scalar signal x, denoted $\|x\|_{\ell^1}$, is the sum of absolute values of $x(k)$ at each instant k:*

$$\|x\|_{\ell^1} = \sum_{k=0}^{+\infty} |x(k)| \tag{3.8.1}$$

This norm exists only if x is ℓ^1-sommable, that is, if and only if the equation 3.8.1 converges.

Definition 8. *(ℓ^2 norm) The ℓ^2 norm of a scalar signal x, denoted $\|x\|_{\ell^2}$, is defined as follows:*

$$\|x\|_{\ell^2} = \sqrt{\sum_{k=0}^{+\infty} x(k)^2} \tag{3.8.2}$$

This norm exists only if x is square-sommable, that is, if and only if the equation 3.8.2 converges.

Definition 9. *(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:*

$$\|x\|_{\ell^\infty} = \sup_{k\in\mathbb{N}} |x(k)| \tag{3.8.3}$$

3.8.1 Impulse response

Definition 10. *(Impulse Response) A SISO filter may be defined by it's impulse response, denoted h. h is the impulse response of $\mathcal{H}$ to the impulsion of Dirac:*

$$\delta(k) = \begin{cases} 1 & when\ k = 0 \\ 0 & else \end{cases} \tag{3.8.4}$$

Indeed each input can be described as a sum of Dirac impulsions:

$$u = \sum_{l \geq 0} u(l)\delta_l \tag{3.8.5}$$

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

$$\delta(k) = \begin{cases} 1 & when\ k = l \\ 0 & else \end{cases} \tag{3.8.6}$$

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:

$$y(k) = \sum_{l \geq 0} u(l)h(k-l) = \sum_{l=0}^{k} u(k)h(k-l) \tag{3.8.7}$$

This corresponds with the convolution product definition of u by h, denoted $y = h * u$.

Dealing with MIMO filters, we have $\boldsymbol{h} \in \mathbb{R}^{n_y \times n_u}$ as the impulse response of $\mathcal{H}$. $\boldsymbol{h}_{i,j}$ is the response on the ith output to the Dirac implusion on th jth input.

The precedent equation becomes:

$$y_i(k) = \sum_{j=1}^{n_u} \sum_{l \geq 0}^{k} u_j(l)h_{i,j}(k-l) \qquad \forall 1 \leq i \leq n_y \tag{3.8.8}$$