# Internship report

Antoine Martinet

June 10, 2015

# Contents

**Introduction** **3**

**1 Motivations** **3**

**2 Tools for expressing filters and signal processing** **3**
2.1 Definition of a signal . . . 3
2.1.1 Linear Time Invariant Filters (LTI filters) . . . 3
2.1.2 Worst-Case Peak Gain (WCPG) of a Filter . . . 3
2.2 FIR and IIR . . . 3
2.3 Realizations . . . 4
2.3.1 Direct and transposed forms . . . 4
2.3.2 State-space representation . . . 4
2.4 Specialized Implicit Form (SIF) . . . 5
2.5 Implementation . . . 5
2.6 Size computation in finite precision . . . 6
2.6.1 Error analysis . . . 7
2.7 Adjustments in arbitrary precision . . . 8

**3 Dimensionment** **9**
3.1 . . . 9
3.2 Algorithm . . . 9
3.3 Particular Forms . . . 9
3.4 ABCD Form . . . 9
3.5 $nx = 0$ . . . 9
3.6 Algorithm . . . 9
3.7 Optimizations . . . 9
3.7.1 Sparse matrices . . . 9
3.7.2 One entries . . . 10

**4 Implementation** **11**
4.1 First Sub-part . . . 11
4.2 Second Sub-part . . . 11
4.2.1 First Subsub-part . . . 11
4.3 Third Sub-part . . . 11
4.4 Fourth Sub-part . . . 11

**Conclusion** **11**

**Appendix: Remainders about signal processing** **11**
4.5 Notations . . . 12
4.6 Reminders about signal processing . . . 12
4.6.1 Impulse response . . . 12

# 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

# 2 Tools for expressing filters and signal processing

### 2.0.1 Linear Time Invariant Filters (LTI filters)

A filter, denoted by it's transfer function $\mathcal{H}$, is an application which transforms a signal $u$ (with $dim(u) = n_u$ ) in a signal $y = \mathcal{H}(u)$, of size $dim(y) = n_y$ . 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 1.** *(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:*

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

### 2.0.2 Impulse response

**Definition 2.** *(Impulse Response) A SISO filter may be defined by it's impulse response, denoted h. h is the impulse response of H to the impulsion of Dirac:*

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

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

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

*where $\delta_l$ is a Dirac impulsion centered in $l$, that is:*

$$\delta(k) = \begin{cases} 1 & when k = l \\ 0 & else \end{cases} \tag{2.0.2}$$

*The linearity condition of $\mathcal{H}$ implies: $\mathcal{H}(u) = \sum_{l\geq 0} u(l)\mathcal{H}(\delta_l)$. Timeinvariance gives: $\mathcal{H}(\delta_l)(k) = h(k - l)$*

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

*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:*

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

### 2.0.3 Worst-Case Peak Gain (WCPG) of a Filter

**Definition 3.** *(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 2.2.1 is reffered 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 2.2.3 by resolving the recurrence relation:

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

## 2.2 Realizations

**Definition 4.** *(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 every 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 described in Lopez' and Hilaire's phds. As they have already been implemented and published in the fopoco project, I will not detail them here. You can see a hardware implementation of the direct form for an IIR done in flopoco on figure 2.3.1

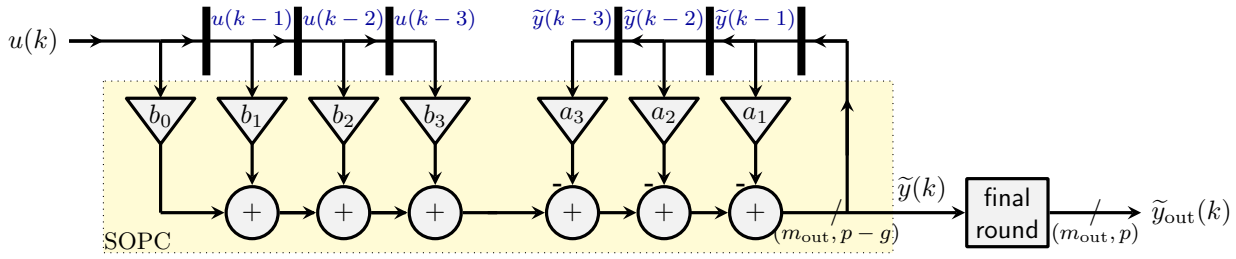


Figure 1: Abstract architecture for the direct form realization of an LTI filter

### 2.2.2 State-space representation

This type of realization consists in expressing the evolution of a system considering it's state at a time k. In continuous time, it is described by differentilal equations at first order. In the discret time case (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 $\boldsymbol{k}(k) = \boldsymbol{u}(k) = 0 \mid \forall k < 0$.

## 2.3 Specialized Implicit Form (SIF)

he specialized implicit form (SIF) was introduced in [] and is well detailed in []. It was built to permit the expression of any realization of LTI filters. The idea is to have a unique representation 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:

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

$$\tag{2.3.3}$$

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.

The SIF can also be seen as an algorithm, each line of the equation 2.4.1 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**
| $\boldsymbol{y}_i(k+1) \leftarrow -\sum_{j=1}^{n_t} \boldsymbol{L}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{R}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{S}_{ij}\boldsymbol{u}_j(k)$
**end**

**Algorithm 1:** Computation of SIF outputs from inputs

**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

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:

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

## 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. 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. The detail is visible on figure 2.5

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 onf figure **??**.

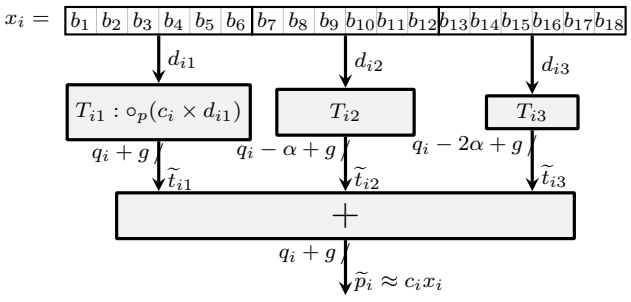


Figure 2: The FixRealKCM method when $x_i$ is split in 3 chunks

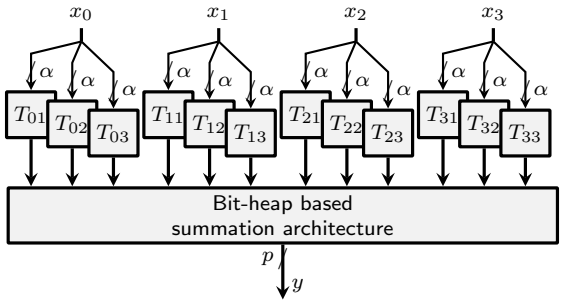


Figure 3: KCM-based SPOC architecture for $n_c = 4$, each input being split into 3 chunks

## 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:

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

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

Then, we can derive every errors, functions of the most significant and last significant bits (respectively msb and lsb)

We can 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:

$$\boldsymbol{t}^*(k+1) = -\boldsymbol{J'}\boldsymbol{t}^*(k+1) + \boldsymbol{M}\boldsymbol{x}^*(k) + \boldsymbol{N}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_t}(k) \tag{2.5.3}$$

$$\boldsymbol{x}^*(k+1) = \boldsymbol{K}\boldsymbol{t}^*(k+1) + \boldsymbol{P}\boldsymbol{x}^*(k) + \boldsymbol{Q_i}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_x}(k) \tag{2.5.4}$$

$$\boldsymbol{y}^*(k+1) = \boldsymbol{L}\boldsymbol{t}^*(k+1) + \boldsymbol{R}\boldsymbol{x}^*(k) + \boldsymbol{S_i}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_y}(k) \tag{2.5.5}$$

We denote $\boldsymbol{\delta t}(k+1) = \boldsymbol{z}_i^*(k) - \boldsymbol{z}_i(k)$ the error at instant k, considering computations errors and loopback,

for $\boldsymbol{z} \in \{\boldsymbol{t}, \boldsymbol{x}, \boldsymbol{y}\}$. We get then:

$$\boldsymbol{\delta t}^*(k+1) = -\boldsymbol{J'}\boldsymbol{\delta t}^*(k+1) + \boldsymbol{M}\boldsymbol{\delta x}^*(k) + \boldsymbol{\varepsilon_t}(k) \tag{2.5.6}$$
$$\boldsymbol{\delta x}^*(k+1) = \boldsymbol{K}\boldsymbol{\delta t}^*(k+1) + \boldsymbol{P}\boldsymbol{\delta x}^*(k) + \boldsymbol{\varepsilon_x}(k) \tag{2.5.7}$$
$$\boldsymbol{\delta y}^*(k+1) = \boldsymbol{L}\boldsymbol{\delta t}^*(k+1) + \boldsymbol{R}\boldsymbol{\delta x}^*(k) + \boldsymbol{\varepsilon_y}(k) \tag{2.5.8}$$

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

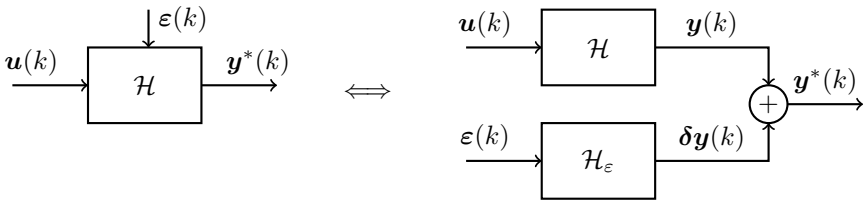


Figure 4: A signal view of the error propagation with respect to the ideal filter

According to 2.4.4, we have:

$$\boldsymbol{Z_\varepsilon} = \begin{pmatrix} \boldsymbol{-J} & \boldsymbol{M} & \boldsymbol{M}_t \\ \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{M}_x \\ \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{M}_y \end{pmatrix} \tag{2.5.9}$$

with:

$$\boldsymbol{M}_t = (\boldsymbol{I}_{n_t}\boldsymbol{0}_{n_t \times n_x}\boldsymbol{0}_{n_t \times n_y}), \tag{2.5.10}$$
$$\boldsymbol{M}_x = (\boldsymbol{0}_{n_x \times n_x}\boldsymbol{I}_{n_x}\boldsymbol{0}_{n_x \times n_y}), \tag{2.5.11}$$
$$\boldsymbol{M}_y = (\boldsymbol{0}_{n_y \times n_t}\boldsymbol{0}_{n_y \times n_x}\boldsymbol{I}_{n_y}), \tag{2.5.12}$$

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

$$\boldsymbol{H}_\varepsilon :\rightarrow \boldsymbol{C_Z}(z\boldsymbol{I}_n - \boldsymbol{A_Z})^{-1}\boldsymbol{M}_1 + \boldsymbol{M}_2 \ \ \forall z \in \mathbb{C} \tag{2.5.13}$$

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

$$\boldsymbol{M_1} = (\boldsymbol{K}\boldsymbol{J}^{-1} \quad \boldsymbol{I}_{n_x} \quad \boldsymbol{0}), \boldsymbol{M_2} = (\boldsymbol{L}\boldsymbol{J}^{-1} \quad \boldsymbol{0} \quad \boldsymbol{I}_{n_y}), \tag{2.5.14}$$

The demonstration is well detailed in Lopez' phd.

*Corollary* 1. Considering a filter $\mathcal{H}$, $\boldsymbol{\varepsilon}(k)$ the vector of computation errors at time k in the finite precision of $\mathbb{H}$, and $\mathcal{H}\boldsymbol{\varepsilon}$ the error filter associated to $\mathcal{H}$. The behaviour of error can be described from $\boldsymbol{\varepsilon}(k)$ and $\mathcal{H}\boldsymbol{\varepsilon}$. Considering the error as an interval vector, denoted $\langle \boldsymbol{\varepsilon}_m, \boldsymbol{\varepsilon}_r \rangle$, the interval vector of global error $\boldsymbol{\delta y}$, denoted $\langle \boldsymbol{\delta y}_m, \boldsymbol{\delta y}_r \rangle$, is given by:

$$\boldsymbol{\delta y}_m = \langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle_{DC} \cdot \boldsymbol{\varepsilon}_m \tag{2.5.15}$$
$$\boldsymbol{\delta y}_r = \langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle_{wcpg} \cdot \boldsymbol{\varepsilon}_r \tag{2.5.16}$$

In practise, the interval arithmetic comes from the command community. In signal community, and moreover in computer arithmetics, we are used to compute on intervals centered around zero. Then, we can reasonably deduce that DC-gains are null, so the previous solution gives:

$$\boldsymbol{\delta y}_m = 0 \tag{2.5.17}$$
$$\boldsymbol{\delta y}_r = \langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle_{wcpg} \cdot \boldsymbol{\varepsilon}_r \tag{2.5.18}$$

For the same reason (centered around zero), the interval becomes:

$$\boldsymbol{\delta y}_m = -\langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle_{wcpg} \cdot \boldsymbol{\varepsilon}_r \tag{2.5.19}$$
$$\boldsymbol{\delta y}_r = \langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle_{wcpg} \cdot \boldsymbol{\varepsilon}_r \tag{2.5.20}$$

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

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

To formalize with a matricial formulation, we get:

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

**TODO**: define or replace v
To satisfy the condition 2.6.1, we can simply define $\xi$ as the minimal error the user wants. We can say then:

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

That is,

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

Where:

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

$$\boldsymbol{D}_{i,j} = |\langle\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\rangle_{i,j}| \cdot \frac{\mathbf{2}^{\boldsymbol{msb_{v'_j}+1}}}{\xi_i} \tag{2.5.26}$$

## 2.6 Adjustments in arbitrary precision

# 3 Dimensionment

## 3.1

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 compliated. 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 deimension 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. The idea now is to see what we can do in arbitrary precision.

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**

## 3.3 Particular Forms

## 3.4 ABCD Form

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

## 3.5 $nx = 0$

When $nx = 0$, the interest of using implicit form is of course very limited. Still, the algorithm will work, allocating only SOPCs operators.

## 3.6 Algorithm

## 3.7 Optimizations

### 3.7.1 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.

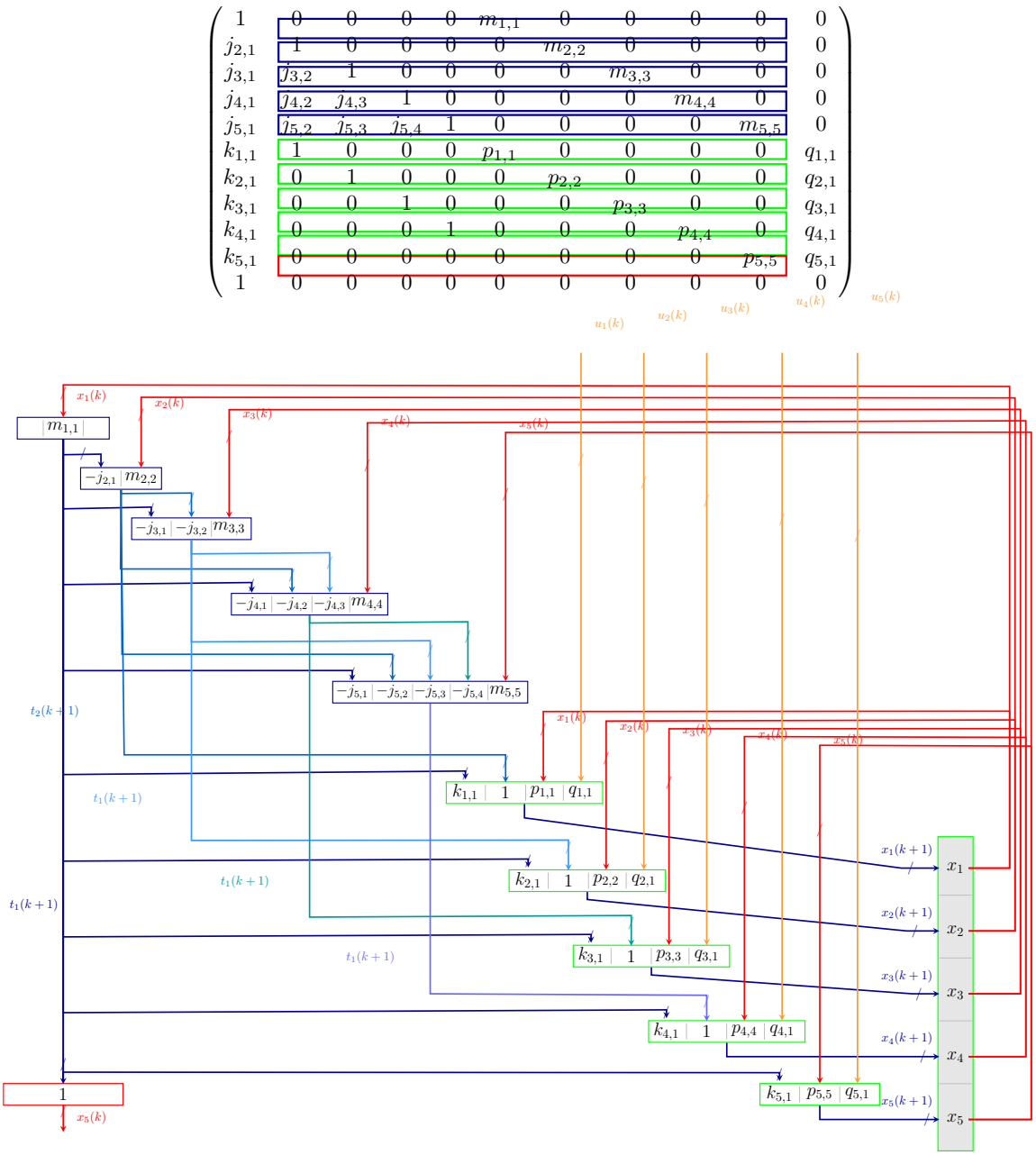


Figure 5: The FixRealKCM method when $x_i$ is split in 3 chunks

### 3.7.2 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).

# 4 Implementation

## 4.1 First Sub-part

## 4.2 Second Sub-part

### 4.2.1 First Subsub-part

## 4.3 Third Sub-part

## 4.4 Fourth Sub-part

# Conclusion

# References

[1] A.Morvan. Utilisation du modèle polyhédrique pour la synthèse d'architectures pipelinées. In *Utilisation du modèle polyhédrique pour la synthèse d'architectures pipelinées (Thesis)*, 2013.

[2] B.E.Nelson. The mythical ccm: In search of usable (and reusable) fpga-based general computing machines. In *IEEE 17th International Conference on Application-specific System, Architectures and Processors*, 2006.

[3] C.Click and K.D.Cooper. Combining analyses, combining optimizations. In *ACM Transactions on Programming Languages and Systems, 17(2)*, 1995.

[4] P. Coussy and eds. A. Morawiec. High-level synthesis: From algorithm to digital circuit. In *Springer*, 2008.

[5] J.P. Elliot. Understanding behavioral synthesis: A practical guide to high-level design. In *Kluwer Academic Publishers*, 1999.

[6] D.MacMillen et al. An industrial view of electronic design automation. In *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 2000.

[7] A.Darte F.Brandner, B.Boissinot, B.D.de Dinechin, and F.Rastello. Computing liveness sets for ssa form programs. In *Research report 7503, INRIA*, 2011.

[8] T.Leng J.C.Huang. Generalized loop-unrolling: a method for program speed-up. In *IEEE Symposium on Application-Specific System and Software Engineering Technology (ASSET99)*, 1999.

[9] J.Mermet. Fundamentals and standards in hardware description languages. In *Springer Verlag*, 1993.

[10] L.Thorczon K.D.Cooper. Engineering a compiler. In *MK-editions*, 2012.

[11] L.T.Simpson. Value-driven redundancy elimination. In *Value-Driven Redundancy Elimination (Technical Report), Rice University*, 1996.

[12] M.Meredith P.Coussy, D.DGajski and A.Takach. An introduction to high-level synthesis. In *IEEE Design and Test of Computers*, 2009.

[13] S.P.Jones P.Hudak, J.Hughes and P.Wadler. A history of haskell: being lazy with class. In *The Third ACM SIGPLAN History of Programming Languages Conference, San Diego, California*, 2007.

# Appendix: Reminders about signal processing

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

## 4.5 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
-
-

## 4.6 Reminders about signal processing

**Definition 5.** *($\ell^1$ norm) The $\ell^1$ norm of a scalar signal x, denoted $\|x\|_{\ell^1}$, is the sum of absolute values of $x(k)$ at each instant k:*

$$\|x\|_{\ell^1} = \sum_{k=0}^{+\infty} |x(k)| \tag{4.6.1}$$

*This norm exists only if $x$ is $\ell^1$-sommable, that is, if and only if the equation 4.6.1 converges.*

**Definition 6.** *($\ell^2$ norm) The $\ell^2$ norm of a scalar signal x, denoted $\|x\|_{\ell^2}$, is defined as follows:*

$$\|x\|_{\ell^2} = \sqrt{\sum_{k=0}^{+\infty} x(k)^2} \tag{4.6.2}$$

*This norm exists only if $x$ is square-sommable, that is, if and only if the equation 4.6.2 converges.*

**Definition 7.** *($\ell^\infty$ norm) The $\ell^\infty$ 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:*

$$\|x\|_{\ell^\infty} = \sup_{k \in \mathbb{N}} |x(k)| \tag{4.6.3}$$

### 4.6.1 Impulse response

**Definition 8.** *(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:*

$$\delta(k) = \begin{cases} 1 & when\ k = 0 \\ 0 & else \end{cases} \tag{4.6.4}$$

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

$$u = \sum_{l \geq 0} u(l)\delta_l \tag{4.6.5}$$

*where $\delta_l$ is a Dirac impulsion centered in $l$, that is:*

$$\delta(k) = \begin{cases} 1 & when\ k = l \\ 0 & else \end{cases} \tag{4.6.6}$$

*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:*

$$y(k) = \sum_{l \geq 0} u(l)h(k-l) = \sum_{l=0}^{k} u(k)h(k-l) \tag{4.6.7}$$

*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 $i$th output to the Dirac implusion on th $j$th input.*

*The precedent equation becomes:*

$$y_i(k) = \sum_{j=1}^{n_u} \sum_{l \geq 0}^{k} u_j(l)h_{i,j}(k-l) \qquad \forall 1 \leq i \leq n_y \tag{4.6.8}$$