# VIIRF

Versatile IIR Filter

## 1 License

# 2 Introduction

This project comprises the hardware description (VHDL) and configuration tools (*Python*) of a flexible IIR filter implementation. No vendor-specific language constructs or bus interfaces are used, such that the system description can easily be used in all hardware development environments.

The IIR filter is incorporated by cascaded (series-connected) second-order sections (SOS), or biquads. This enables very high-order filters while maintaining their numerical stability and integrity.

So far, the SOS are implemented using the direct-form I (DF1), which is robust against internal overflows.

The *config* folder contains scripts (*Python* and *Matlab*) that help configuring the desired filter:

- *IIR_Config.py*
  This is a *Python* script that takes the SOS and gain coefficients of a cascaded SOS filter and performs numerous sanity checks on the coefficients and possible hardware configurations. It also simulates the filter (both floating-point and fixed-point) and plots the step response. It also creates stimuli and reference outputs for the VHDL testbenches (which are stored in the folder *FilterStimuliData*).

- *IIR_SOS_Filt_df1.py*
  This contains two *Python* classes that implement two cascaded direct-form I filters (one floating-point, one quantized), which are used by *IIR_Config.py* to simulate the filter's response.

- *export_sos_python_lists.m*
  This *Matlab* function exports SOS and G-matrices generated by *Matlab* (e.g., *fdatool*) to a text-file with a syntax that is compatible with *Python*/*IIR_Config.py.*

## 2.A Biquad Coefficients

The SOS/biquad implements the following transfer function:

$$G(z)=\frac{b_0+b_1 z^{-1}+b_2 z^{-2}}{1+a_1 z^{-1}+a_2 z^{-2}},$$

which, using the direct-form 1 implementation, and adding a separate output gain *g*, results in this difference equation:

$$y[n]=g\cdot(b_0 x[n]+b_1 x[n-1]+b_2 x[n-2]-a_1 y[n-1]-a_2 y[n-2]).$$

The filter coefficients are named identically throughout this project.

The coefficients are provided to the configuration script (*IIR_Config.py*) as floating-point values and are quantized according to the Q-notation defined by the VHDL-generics *W_COEF* and

*W_FRAC*, whereas *W_COEF* is the total length (bits) of the coefficient, and *W_FRAC* is the length of the fraction.

For example, a Q-notation of Q1.15 is selected with *W_COEF* = 16 and *W_FRAC* = 15.

The quantized coefficients are calculated according to:

$$Cq = \mathrm{round}\left(C \cdot 2^{\mathrm{W\_FRAC}}\right)$$

The SOS-coefficients are (e.g., in *Matlab*), provided as a matrix, where each row contains the coefficients of one SOS in this form:

$$[b_{0,} b_{1,} b_{2,} 1, a_{1,} a_2].$$

This is also the form utilized by *IIR_Config.py*. Note that the 4th coefficient is always unity.

Each SOS can have an output gain *g* or not. The gains are provided by the vector G, where each entry corresponds to the gain of an SOS.
In the hardware implementation, the parameter *SOSGAIN_EN* controls the generation of the section's output multiplication hardware. Depending on the coefficient scaling (see usage example below), the section-gains can often all be set to unity and these multiplication steps can be omitted.

Furthermore, the filter implementation also features a final output gain that is applied after all cascaded sections (see hardware description). This gain is also provided by the G-matrix, as the last entry (see usage example below).

## 2.B Usage Example

In the following, the filter design workflow is illustrated.

1. Design a transfer function or IIR filter and obtain the SOS-representation of it. With *Python*, this can be achieved e.g., by using the *scipy* package. *Matlab's fdatool* is also a very convenient way to design digital filters. The process for *fdatool* is as such:

    a) With *fdatool*, design the desired filter (select the IIR topology).

    b) Edit – Convert Structure – Direct-Form I, SOS

    c) Edit – Reorder and Scale Second-Order Sections
    Apply desired reordering and scaling methods. For low-noise / high-precision filters, put the scale-slider towards "Highest SNR". Then (not exclusively) the SOS-gains are all unity, which also reduces resource utilization and latency.

    d) File – Export
    Export SOS and G to the workspace. We do not need to quantize the coefficients, as we do this ourselves.

    e) In *Matlab*, run the function e*xport_sos_python_lists.m* (part of this project) on the exported SOS and G matrices. This creates a text-file with SOS and G as python lists.

2. Insert the SOS and G lists into *IIR_Config.py*

3. Configure the remaining parameters of the filter, e.g., if all section gains are unity, *SOSGAIN_EN* can be set to false and the unity-coefficients can be deleted. Note that the shape of the G-list depends on *SOSGAIN_EN* and *FINALGAIN_EN*. See *IIR_Config.py.* Choose a filter coefficient quantization. This can be dependent of the type of multipliers available in the target hardware. E.g., *Xilinx's DSP48E1* features a 25x18 multiplier. Hence, it is advantageous if *W_SECT_DAT* = 25 and *W_COEF* = 18. Select *W_FRAC* as high as possible, but such that it still covers the coefficient range (*IIR_Config.py* will raise errors if the selected value is unfit). Higher data/coefficient widths are of course possible, but then the synthesis tool has to use more multipliers / logic.

4. Run *IIR_Config.py*. If it succeeds, it creates various files, most in a directory (*FILENAME_METADATA* and D*IRNAME_STIMULIDATA*).
   There might be console outputs indicating that filter data has been saturated. This is OK, and the effects are visible in the step-response. If the response is unsatisfactory, increase *NUM_BITS_SECT_DAT_EXT_MANUAL* until the response is good (this increases *W_SECT_DAT*). This is likely the case if *SOSGAIN_EN* is set to true and the SOS-gains are small. Another approach is to rescale the filter coefficients such that all SOS-gains are unity, and only use the filter's final gain (if needed). Additionally, an increase of the coefficient width can also improve the filter's performance/precision.

5. If the filter's performance is satisfactory, configure *sos_cascaded_top_tb.vhd* according to the contents in the filter metadata-file (e.g., *FilterMetaData.txt*), that has been created by the *Python* script.

6. Copy the contents of the stimuli-directory (e.g., *FilterStimuliData*), that has been created by the *Python* script, to the folder *testbench_stimuli*. This is where the testbench reads the files from (This step can be omitted by pointing the testbench to the *FilterStimuliData*-folder).
   NOTE: The testbench might need absolute file paths, depending on the utilized system. See/ configure the paths in *sos_cascaded_top_tb.vhd*.

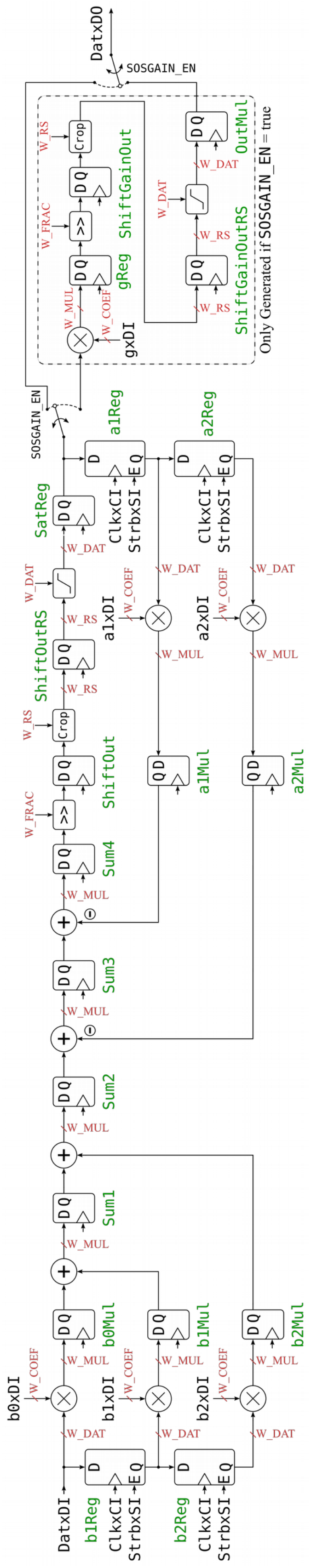
7. Run the testbench. This simulates the filter and compares the output to what the python script generated. No errors should be raised during the testbench execution (except if it runs out of stimuli data – see error reports in the simulator console).

8. Configure *sos_cascaded_top.vhd* also according to the metadata-file (e.g., *FilterMetaData.txt*).

9. Synthesize / utilize *sos_cascaded_top.vhd* as desired.

# 3 Hardware Description

## 3.A Direct-Form 1 Biquad Core



The figure on the left illustrates the hardware implementation of the direct-form 1 biquad (the figure is also available in the *doc* folder of this project). This implementation is heavily pipelined such that there is as little logic as possible between registers in order to enable a higher maximum clock rate.

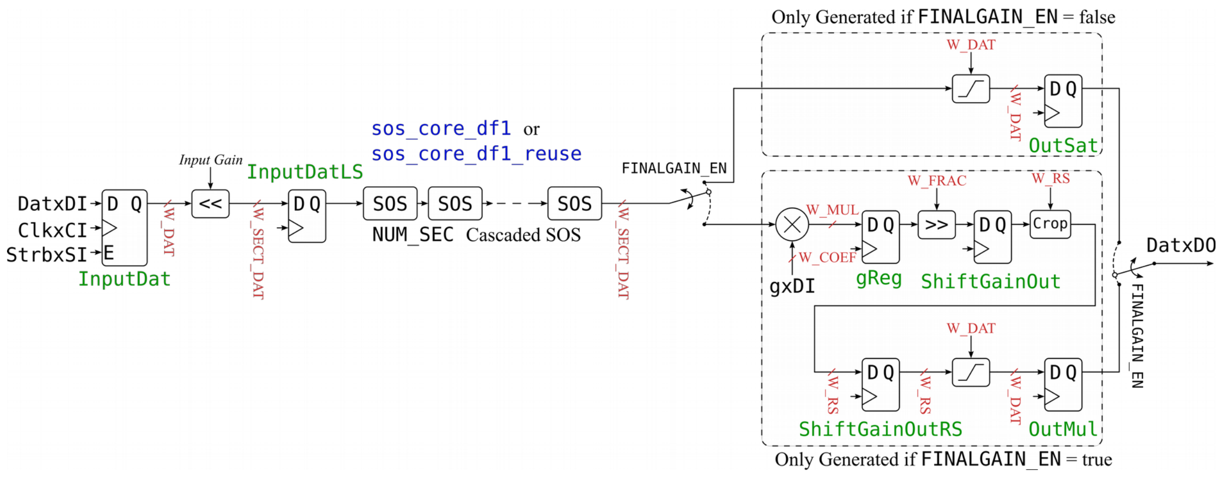
This structure requires 5 (or 6, if *SOSGAIN_EN* is true) multipliers.

This structure is implemented in *sos_core_df1.vhd*

There is a second implementation of this filter core provided by *sos_core_df1_reuse.vhd*, which uses a state-machine in order to provide the filter data sequentially to a single multiplier. This can not run at very high clock rates due to the additional logic required to feed the multiplier with the correct data, but it uses only one multiplier.

## 3.B Top-Level Module: Cascading of the SOS Cores

The following figure illustrates how, in *sos_cascaded_top.vhd*, the SOS are cascaded and how the final filter output gain is applied.



## 3.C Configuration

The filter's top-level file is *sos_cascaded_top.vhd*. It can be configured with the following VHDL generics. These generics are provided by the *Python* script *Config_IIR.py*

- NUM_SEC (Integer)
  Number of cascaded SOS sections. Maximum value: 255
- W_DAT_INPUT (Integer)
  Width of the (signed) filter input data (bits)
- GAIN_INPUT (Integer, must be power of 2)
  This gain is applied to the input signal before it is passed through the filter. Usable to increase the filter's precision (more bits available due to the left-shift operation of this gain).
- W_SECT_DAT (Integer)
  Width of the cascaded SOS data-path (bits). Due to the input gain, and potentially due to the output gain and SOS-gains, the SOS require wider data vectors in order not to lose data.
- W_COEF (Integer)
  Width of the (quantized) coefficients (bits)
- W_FRAC (Integer)
  Fraction width (in bits) of the quantized coefficients. Together with W_COEF, this defines the Q-notation of the filter coefficient quantization.
- SOSGAIN_EN (Bool)
  If set to true, each biquad has a gain-stage at its output. If set to false, this additional multiplication-hardware is omitted / not generated.

- FINALGAIN_EN (Bool)
  If set to true, there is a gain-stage at the output of the filter (at output of last SOS in the cascade). If set to false, this additional multiplication hardware is omitted  / not generated.

- W_DAT_OUTPUT (Integer)
  Width of the (signed) filter output data (bits). The output of the filter is saturated to this signed data width, there is no overflow.

- W_DAT_INTF (Integer)
  Width of the data/address signals of the generic filter coefficient interface.

- USE_PIPELINE_CORE (Bool)
  If set to true, the pipelined implementation of the filter is used (*sos_core_df1.vhd*; uses more multipliers, but can operate at higher clock rates). If set to false, the multiplier of each SOS is reused (*sos_core_df1_reuse.vhd*), hence only one multiplier per SOS is required (uses less multipliers, but cannot operate at higher clock rates due to the additional logic required to feed the multiplier with the correct data).

## 3.D Coefficient Data Interface

The coefficients are provided to the filter via a simple data interface that can easily be adapted to interfaces like *Wishbone* or *Axi*. Its internal working can also be deduced from the testbench. There is some pseudo-code below.
The signals of the interface are:

- *SectAddrxDI*
  Address of the SOS for which the coefficients are provided. Range: 0 to *NUM_SEC*.
  Note: When *SectAddrxDI* = *NUM_SEC* and *CoeffAddrxDI* = 6, the final output gain of the whole filter (*FinalGain*) is addressed (see also below).

- *CoeffAddrxDI*
  Address of the biquad coefficient. Range: 0-6. Coefficient mapping:

| *CoeffAddrxDI* | SOS Coefficient |
|---|---|
| 0 | *b0* |
| 1 | *b1* |
| 2 | *b2* |
| 3 | *a1* |
| 4 | *a2* |
| 5 | *g* (Section output gain) |
| 6 | *FinalGain* (SectAddrxDI must be = NUM_SEC to write this coefficient) |

- *CoeffDatxDI*
  Coefficient data

- *CoeffValidxSI*
  Coefficients are only written to the filter if this valid-signal is logic high.

## Pseudo-Code for Coefficient Data Interface

In a microprocessor that communicates with the FPGA (where the filter is implemented) through some bus structure, the coefficients might be stored as follows (C code, assuming SOSGAIN_EN=False, hence only 5 coefficients required per section, and FINALGAIN_EN=True):

```
#define IIR_NUMSEC       7
static const int32_t IIR_Coeff[7][5] = {
{8388608,-16261805,8388608,-16553618,8553470},
{8388608,-16163323,8388608,-15554455,7559856},
{8388608,-16669864,8388608,-16936749,8256172},
{8388608,-16839813,8388608,-16755940,7996490},
{8388608,-16669445,8388608,-16264385,8263768},
{8388608,-16735613,8388608,-16935940,7948290},
{8388608,-16632613,8388608,-16485940,7974190}
};
static const int32_t IIR_OutGain = 81558;
```

The corresponding pseude-code to transmit this data to the filter might look as follows:

```
CoeffValidxSI = 0; // Set to zero at beginning. Make sure this is written to the hardware/bus/memory (i.e., make
sure it reaches the filter core, e.g., use volatile-statements…). This is also necessary for all other accesses
regarding the coefficient interface below! (marked “critical access” below)

// Iterate over all filter sections:
for i = 0; i<IIR_NUMSEC; i++

        SectAddrxDI = i; // critical access

                // Iterate over all coefficients; note that SOSGAIN_EN=False, hence only 5 coeffs needed (in
                this example)
                for k = 0; k<5; k++

                        CoeffAddrxDI = k; // critical access
                        CoeffDatxDI = IIR_Coeff[i][k]; // critical access
                        // Data is read by the filter-hardware upon the cycling of the valid-bit:
                        CoeffValidxSI = 1; // critical access
                        CoeffValidxSI = 0; // critical access


// Set the final gain:
 SectAddrxDI = IIR_NUMSEC; // Write the final gain. critical access
 CoeffAddrxDI = 6; // Write the final gain. critical access
 CoeffDatxDI = IIR_OutGain; // critical access
 CoeffValidxSI = 1; // critical access
CoeffValidxSI = 0; // critical access
```

## 4 Revision History

| | | |
|---|---|---|
| Initial project setup and commit | Mario Mauerer (MM) | Zürich, 5. June 2017 |
| Added pseudo-code for coefficient interface | MM | 24. May 2018 |