psi_fix

Documentation

Content

Table of Contents

Figures

1 Introduction

The purpose of this library is to provide HDL implementations for common fixed-point signal processing components along with bittrue Python models. The Python models are also callable from MATLAB.

This document serves as description of the RTL implementation for all components.

2 Tipps & Tricks

2.1 Library Setup

The *psi_fix* library refers to *psi_common* and *psi_tb* relatively and assumes the contents of these repositories are compiled into the same VHDL library.

There are two common ways of setting up projects without toubles:

1. *psi_fix*, *psi common* and *psi_tb* are compiled into a VHDL library called *psi_lib*. The project specific code is compiled to a different library and it refers to library elements using *psi_lib.<any_entity>.*
2. All code of the complete project including *psi_fix, psi common* and *psi_tb* is compiled into the same library. Independently of the name of that library, library elements can be referred to using *work.<any_entity>*.

2.2 Heavy Pipelining

2.2.1 Problem Description

The following code may lead to suboptimal results for very high clock frequencies because there are three operations in the same pipeline stage:

- The actual addition
- Rounding (adding of a rounding constant)
- Limitting

```
constant aFmt_c : PsiFixFmt_t := (1, 8, 8);
constant bFmt_c : PsiFixFmt_t := (1, 8, 8);
constant rFmt_c : PsiFixFmt_t := (1, 8, 0);

…

p : process(Clk)
begin
   if rising_edge(Clk) then
      r <= PsiFixAdd(a, aFmt_c, b, bFmt_c, rFmt_c, PsiFixRound, PsiFixSat);
   end if;
end process;
```

This leads to the implementation shown below.

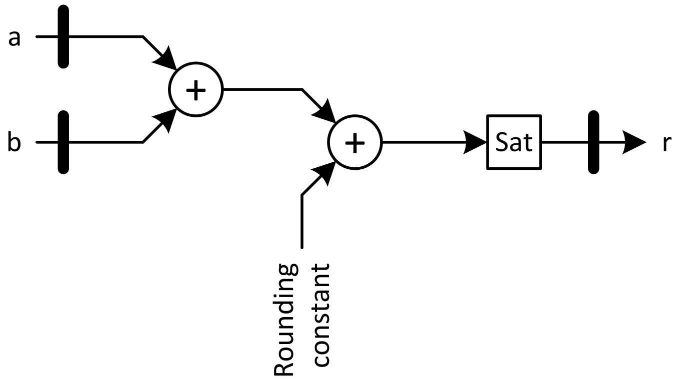


Figure 1: Heavy Pipelining, Problem Description

2.2.2 Solution 1: Register Retiming

Todays FPGA tools are quite good at register retiming. This means that the tools moves pipeline stages to optimize timing. ISE is also able to do retiming but it must be actively enabled in the project settings (synthesis).

Thanks to retiming, the user can just add a few pipeline stages at the output of the logic and the tool will move them into the logic to optimize timing.

```
constant aFmt_c : PsiFixFmt_t := (1, 8, 8);
constant bFmt_c : PsiFixFmt_t := (1, 8, 8);
constant rFmt_c : PsiFixFmt_t := (1, 8, 0);

…

p : process(Clk)
begin
   if rising_edge(Clk) then
      r1 <= PsiFixAdd(a, aFmt_c, b, bFmt_c, rFmt_c, PsiFixRound, PsiFixSat);
      r2 <= r1;
      r <= r2;
   end if;
end process;
```

The code above theoretically describes the following circuit which is not more timing-optimal than the original circuit:

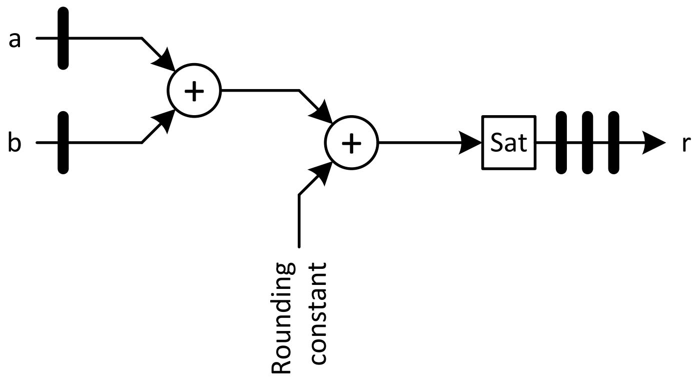


Figure 2: Heavy Pipelining, Retiming, Implementation without retiming

However, it register retiming is applied, the tool will convert the circuit into something as shown below. This is way more timing optimal and allows achieving higher clock frequencies.

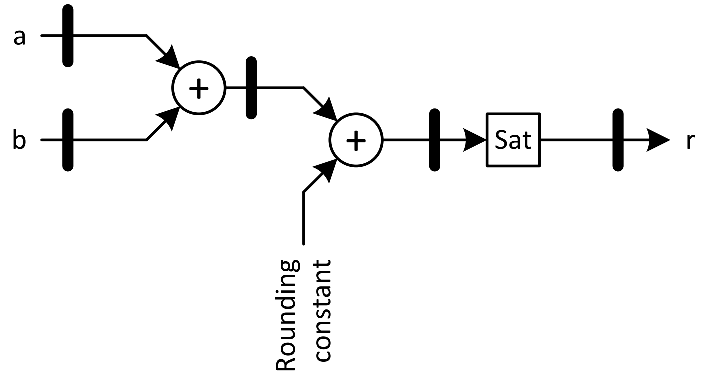


Figure 3: Heavy Pipelining, Retiming, Implementation with retiming

The advantage of the solution using retiming is, that the pipeline registers can be moved at a very fine-grained level (even finer than one VHDL code line) and the tool is free to move the pipeline stages to the optimal place.

The drawback is that this approach relies on the tool to recognize the timing problem and fix it by applying retiming. If the tool fails to do this for whatever reason, the design will not meet timing.

2.2.3 Solution 2: Manual Splitting

The operation can be split into multiple stages manually on VHDL level. This can be done by not doing all steps in one VHDL line but one after the other in multiple lines. Of course intermediate number formats must be chosen accordingly to ensure correct operation. An example is given below.

```
constant aFmt_c   : PsiFixFmt_t := (1, 8, 8);
constant bFmt_c   : PsiFixFmt_t := (1, 8, 8);
constant addFmt_c : PsiFixFmt_t := (1, 9, 8); -- + 1 Int-Bit for addition
constant rndFmt_c : PsiFixfmt_t := (1, 10, 8); -- + 1 Int-Bit for adding RC
constant rFmt_c   : PsiFixFmt_t := (1, 8, 0);


…

p : process(Clk)
begin
   if rising_edge(Clk) then
      -- addition only, no rounding or satturation
      add <= PsiFixAdd(a, aFmt_c, b, b_Fmt_c, addFmt_c, PsiFixTrunc, PsiFixWrap);
      -- rounding only
      rnd <= PsiFixResize(add, addFmt_c, rndFmt_c, PsiFixRound, PsiFixWrap);
      -- saturation ony
      r <= PsiFixResize(rnd, rndFmt_c, rFmt_c, PsiFixTrunc, PsiFixSat);
   end if;
end process;
```

This code directly leads to the implementation shown below and does not rely on the tools to do the retiming.

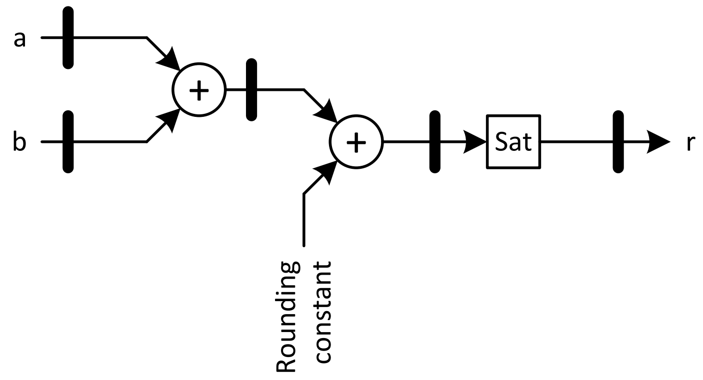


Figure 4: Heavy Pipelining, Manual Splitting

The advantage of this approach is that it does not rely on any tool-optimization.

The disadvantage is that slightly more code is required.

Or course the tools can still apply retiming to move the registers if required.

3 RTL Descriptions

3.1 psi_fix_bin_div

3.1.1 Description

This component implements a fixed point binary divider.

$$Quotient = \frac{Nomerator}{Denominator}$$

3.1.2 Generics

NumFmt_g	Numerator format
DenomFmt_g	Denominator format
QuotFmt_g	Quotient format
Round_g	Rounding mode at the output (round or truncate)
Sat_g	Saturation mode at the output (saturate of wrap)

3.1.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Input			
InVld	Input	1	AXI-S handshaking signal
InRdy	Output	1	AXI-S handshaking signal
InNum	Input	NumFmt_g	Numerator input
InDenom	Input	DenomFmt_g	Denominator input
Output			
OutVld	Output	1	AXI-S handshaking signal
OutQuot	Output	QuotFmt_g	Quotient output

At the input a handshaking for handling backpressure (incl. Rdy) is implemented since the binary divider is quite slow and may be the limiting component in offline data processing systems. At the output no handling for backpressure is implemented for simplicity reasons.

3.1.4 Architecture

The component converts numerator and denominator to unsigned numbers, so a standard binary divider can be implemented. At the output, the sign is restored correctly.

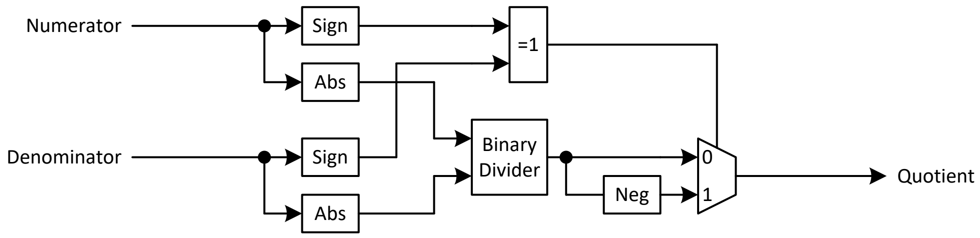


Figure 5: psi_fix_bin_div Architecture

3.2 psi_fix_cic_dec_fix_1ch

3.2.1 Description

This component implements a simple CIC decimator for a single channel. The decimation ratio must be known at compile time.

The CIC component always corrects the CIC gain roughly by shifting. As a result, the gain of the component is always between 0.5 and 1.0. Additionally a multiplier for exact gain adjustment can be added by setting the generic *AutoGainCorr_g* to true. In this case the gain is corrected to exactly 1.0.

3.2.2 Generics

Order_g Order of the CIC filter (number of integrator/comb pairs)
Ratio_g Decimation ratio
DiffDel_g Delay for the comb sections (1 or 2)
InFmt_g Input format
OutFmt_g Output format
AutoGainCorr_g True = compensate gain to 1.0, False = gain is between 0.5 and 1.0

3.2.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Input			
InVld	Input	1	AXI-S handshaking signal
InData	Input	InFmt_g	Denominator input
Output			
OutVld	Output	1	AXI-S handshaking signal
OutData	Output	InFmt_g	Quotient output

The CIC is able to process one input sample per clock cycle. Therefore no backpressure handling is implemented on the input.

CIC are most commonly used in streaming signal processing systems that require processing or storing the data at the full speed anyway. So no backpressure handling is implemented on the output side for simplicity

3.2.4 Architecture

The figure below shows the architecture of the CIC decimation filter.

Since the integrators are responsible for most of the CIC gain, the numbers are shifted and truncated after the integrator sections to the width required for producing less than 1 LSB error at the output. This allows saving some resources in the differentiator sections.

Note that the number format for the differentiator sections has one additional fractional bit (compared to the output format) per section. This results from the fact that depending on the signal frequency, the differentiators can have a gain up to two. This way the least significant bit at the input of the differentiators that can change the output by one LSB is preserved.

If the gain correction multiplier is used, signal path is chosen to be 25 bits wide and the gain correction coefficient is 17 bits (unsigned). For most implementations this design decisions are sufficient. If other requirements exist (e.g. very wide signal path), a project specific implementation of the CIC is required.

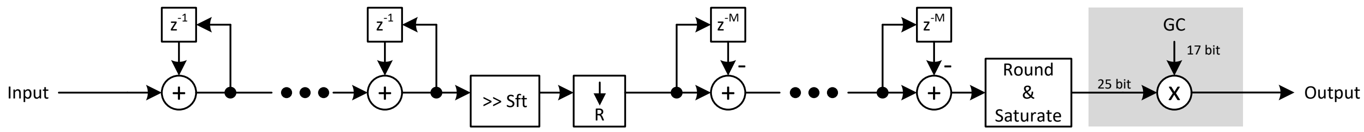


Figure 6: psi_fix_cic_dec_fix_1ch Architecture

The symbols are defined as follows:

R Decimation ratio
M Differential delay
N CIC order
Sft Number of bits to shift (to compensate overall gain to 0.5 < gain < 1.0)
GC Gain correction factor to compensate overall gain to 1.0

Some of the most common formulas are given below.

$$Gain_{CIC} = (R \cdot M)^N$$

$$Sft = ceil(\log_2(Gain_{CIC}))$$

For the case that the gain correction amplifier is disabled, the overall gain of the CIC is:

$$GainOverallNoGc = \frac{Gain_{CIC}}{2^{Sft}}$$

Since this formula evaluates to 1.0 for the case $R = x^2$ (decimation ratio is a power of two), the gain correction multiplier is not required in this case.

The optimal setting for the differential delay depends on the use case. Only the values 1 and 2 are supported. Other values are uncommon in real-life. Usually 1 is used if an FIR filter follows the CIC to further reduce the passband. If no FIR follows the CIC, a value 2 to is more optimal to avoid strong aliasing.

3.3 psi_fix_cic_int_fix_1ch

3.3.1 Description

This component implements a simple CIC interpolator for a single channel. The interpolation ratio must be known at compile time.

The CIC component always corrects the CIC gain roughly by shifting. As a result, the gain of the component is always between 0.5 and 1.0. Additionally a multiplier for exact gain adjustment can be added by setting the generic *AutoGainCorr_g* to true. In this case the gain is corrected to exactly 1.0.

3.3.2 Generics

Order_g	Order of the CIC filter (number of integrator/comb pairs)
Ratio_g	Interpolation ratio
DiffDel_g	Delay for the comb sections (1 or 2)
InFmt_g	Input format
OutFmt_g	Output format
AutoGainCorr_g	True = compensate gain to 1.0, False = gain is between 0.5 and 1.0

3.3.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Input			
InVld	Input	1	AXI-S handshaking signal
InRdy	Output	1	AXI-S handshaking signal
InData	Input	InFmt_g	Denominator input
Output			
OutVld	Output	1	AXI-S handshaking signal
OutRdy	Input	1	AXI-S handshaking signal
OutData	Output	InFmt_g	Quotient output

The CIC interpolator requires full handshaking including the handling of back-pressure at the input since it can only take one sample every N clock cycles. As a result, the *InRdy* signal is required to signal when an input sample was processed.

Full handshaking at the output side was implemented mainly to allow equally spaced output samples (in time). By nature the filter calculates multiple output samples back-to-back after an input sample arrived. For output rates lower than the clock-speed, this leads to a bursting behavior which is often (but not always) undesirable. By controlling the *OutRdy* signal, the user can control the output sample-rate and –spacing exactly.

3.3.4 Architecture

The figure below shows the architecture of the CIC interpolation filter.

Note that the number format for the differentiator sections has one additional integer bit (compared to the input format) per section. This results from the fact that depending on the signal frequency, the differentiators can have a gain up to two.

If the gain correction multiplier is used, signal path is chosen to be 25 bits wide and the gain correction coefficient is 17 bits (unsigned). For most implementations this design decisions are sufficient. If other requirements exist (e.g. very wide signal path), a project specific implementation of the CIC is required.

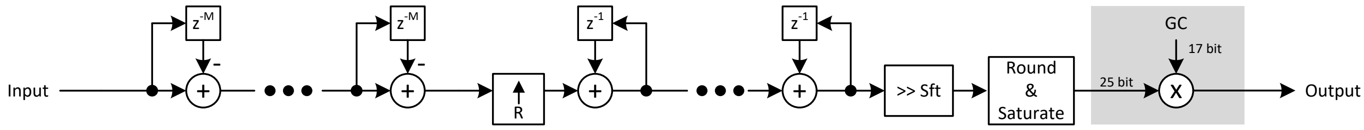


Figure 7: psi_fix_cic_int_fix_1ch Architecture

The symbols are defined as follows:

R Interpolation ratio
M Differential delay
N CIC order
Sft Number of bits to shift (to compensate overall gain to 0.5 < gain < 1.0)
GC Gain correction factor to compensate overall gain to 1.0

Some of the most common formulas are given below.

$$Gain_{CIC} = \frac{(R \cdot M)^N}{R}$$

$$Sft = ceil(\log_2(Gain_{CIC}))$$

For the case that the gain correction amplifier is disabled, the overall gain of the CIC is:

$$GainOverallNoGc = \frac{Gain_{CIC}}{2^{Sft}}$$

Since this formula evaluates to 1.0 for the case $R = x^2$ (interpolation ratio is a power of two), the gain correction multiplier is not required in this case.

The optimal setting for the differential delay depends on the use case. Only the values 1 and 2 are supported. Other values are uncommon in real-life. Usually 1 is used if the input signal is already oversampled (does not contain frequency components close to $\frac{fs}{2}$) and 2 is used otherwise.

Note that the CIC does not control timing on its own. This means by default, the CIC outputs one sample per clock cycle. If the input sample rate is slow, the output is bursting. If the time between two output samples has to be constant, the timing can be controlled by applying pulses at the desired frequency to the *OutRdy* handshaking signal. The reason for the CIC to not control any timing at the output is that this is a library component and it may also be used in offline processing algorithms.

3.4 psi_fix_cordic_abs_pl

3.4.1 Description

This component implements the absolute value calculation based on the CORDIC algorithm. Depending on the parameters, up to one pipeline stage per iteration can be implemented. This allows achieving even highest performance requirements.

Note that this component does not compensate the CORDIC gain. If this is required, the compensation of the CORDIC gain must be implemented externally.

3.4.2 Generics

InFmt_g Input format (must be signed)
OutFmt_g Output format (must be unsigned since this is an absolute value)
InternalFmt_g Number format used for all CORDIC calculations
Iterations_g Number of CORDIC iterations to execute
PipelineFactor_g A pipeline stage is implemented after every N iterations (1 = fully pipelined)
Round_g Rounding mode at the output (round or truncate)
Sat_g Saturation mode at the output (saturate of wrap)

3.4.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Input			
InVld	Input	1	AXI-S handshaking signal
InI	Input	InFmt_g	In-phase signal input
InQ	Input	InFmt_g	Quadrature-phase signal input
Output			
OutVld	Output	1	AXI-S handshaking signal
OutAbs	Output	OutFmt_g	Result output

The CORDIC implementation is fully pipelined. This means it can take one input sample every clock cycle. As a result the handling of backpressure was not implemented.

3.4.4 Architecture

The CORDIC algorithm for the calculation of the absolute value is defined by the formulas below.

$$x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i}$$

$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i}$$

$$d_i = +1\ if\ y_i < 0, else - 1$$

The algorithm only works for $x \geq 0$, therefore the absolute value of x is calculated prior to executing the algorithm.

The CORDIC gain can be calculated by the formula below:

$$G_{CORDIC} = \prod_{i=0}^{N-1} \sqrt{1 + 2^{-2i}}$$

Where:
G_{CORDIC} Cordic Gain
N Number of iterations

The formula converges towards 1.646760 with high numbers of iterations.

The amount of Pipelining to be implemented can be chosen using the generic *PipelineFactor_g*. However, the amount of logic (LUT) required does not change much with reduced pipelining. The main reason for reducing the amount of pipelining is latency reduction.

3.5 psi_fix_fir_dec_ser_nch_chpar_conf

3.5.1 Description

This entity was initially implemented as multi-channel filter with configurable coefficients. ***However, it can also be used efficiently for single-channel FIRs and for filters with fixed coefficients.***

This entity implements a multi-channel decimating FIR filter. All channels are processed in parallel (not TDM) but there is only one multiplier for each channel, so the taps of a channel are calculated one after the other. The filter coefficients, the order and the decimation rate are runtime configurable.

3.5.2 Generics

InFmt_g	Input format
OutFmt_g	Output format
CoefFmt_g	Coefficient format
Channels_g	Number of parallel channels
MaxRatio_g	Maximum decimation ratio supported
MaxTaps_g	Maximum number of taps supported
Rnd_g	Rounding mode at the output (round or truncate)
Sat_g	Saturation mode at the output (saturate of wrap)
UseFixCoefs_g	If true, fixed coefficients instead of configurable coefficients are implemented.
FixCoefs_g	Coefficients to use for *UseFixCoefs_g* = true.

3.5.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Input			
InVld	Input	1	AXI-S handshaking signal
InData	Input	$InFmt\_g \cdot Channels\_g$	Input data in parallel - Channel 0 [N-1:0] - Channel 1 [2*N-1:0] - …
Output			
OutVld	Output	1	AXI-S handshaking signal
OutAbs	Output	$OutFmt\_g \cdot Channels\_g$	Output data in parallel (see *InData*)
Configuration			
Ratio	Input	$ceil(\log_2(\mathrm{MaxRatio\_g}))$	Decimation ratio -1 0 → no decimation 1 → decimation by 2) This port is optional. If it is not connected, *MaxRatio_g* is used as fixed ratio.

Taps	Input	$ceil(\log_2(\text{MaxTaps\_g}))$	Taps – 1 0 → 1 Tap (order 0 filter) 63 → 64 Taps (order 63 filter) This port is optional. If it is not connected, *MaxTaps_g* is used as fixed tap count.
Coefficient Interface			
CoefClk	Input	1	Clock for the coefficient interface. This port can be left unconnected for fixed coefficient implementation (*UseFixCoefs_g* = true)
CoefWr	Input	1	Coefficient write enable signal This port can be left unconnected for fixed coefficient implementation (*UseFixCoefs_g* = true)
CoefAddr	Input	$ceil(\log_2(\text{MaxTaps\_g}))$	Address of the coefficient to access This port can be left unconnected for fixed coefficient implementation (*UseFixCoefs_g* = true)
CoefWrData	Input	CoefFmt_g	Coefficient value for write access (*CoefWr = 1)* This port can be left unconnected for fixed coefficient implementation (*UseFixCoefs_g* = true)
CoefRdData	Output	CoefFmt_g	Coefficient read data (valid 1 cycle after applying the address) This port can be left unconnected for fixed coefficient implementation (*UseFixCoefs_g* = true)

The coefficient interface has a separate clock since often the data processing clock is coupled to an ADC clock but the main bus system that configures the filter is running on a different clock.

The filter can continue taking new input data even if a calculation is ongoing. As a result, the handling of packpressure is not required as long as the processing power of the filter is sufficient to handle all input data. For the calculation, see below.

Note that the behavior of the filter is undefined if the maximum input rate that can be handles is exceeded.

3.5.4 Architecture

The figure below roughly shows the architecture of the FIR filter. Since the filter assumes all channels arrive in parallel with the same timing, the coefficient RAM is shared between all channels to save resources.

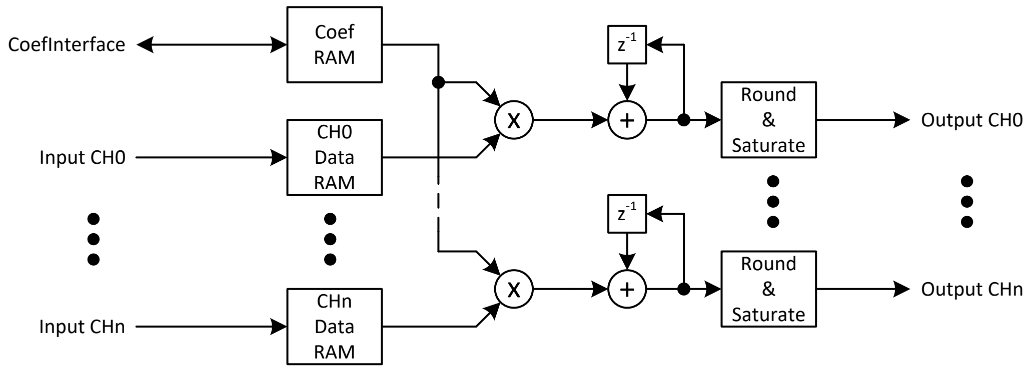


Figure 8: psi_fix_fix_dec_ser_nch_chpar_conf Architecture

A state machine (not shown in the figure for simplicity) starts a new calculation whenever all required input samples for the next calculation arrived.

The accumulation is executed at the full output precision of the multiplication. This matches the implementation of the DSP slices in Xilinx devices, so they can be fully utilized.

The accumulator contains one guard bit compared to the output format to detect overflows. However, the user (designer who integrates the filter) is responsible to choose coefficients in a way that the output format is never exceeded by more than a factor of two. This this is not possible the filter output format must be chosen large enough ($Range_{Output} \geq 0.5 \cdot MaximumOutput$) and saturated externally.

Obviously the architecture requires one clock cycle per tap calculation. As a result the maximum number of filter taps depends on the clock frequency F_{clk}, the input sample rate $F_{s,in}$ and the decimation ratio R.

$$Taps_{max} = \frac{F_{clk} \cdot R}{F_{s,in}}$$

In case of fixed coefficient implementation, the coefficient RAM is replaced by a ROM automatically.

3.6 psi_fix_fir_dec_ser_nch_chtdm_conf

3.6.1 Description

This entity was initially implemented as filter with configurable coefficients. ***However, it can also be used efficiently for filters with fixed coefficients.***

This component implements a multi-channel decimating FIR filter. All channels are processed TDM (one after the other). The multiplications are all executed using the same multiplier, so the taps of a channel are calculated one after the other. The filter coefficients, the order and the decimation rate are runtime configurable.

3.6.2 Generics

InFmt_g	Input format
OutFmt_g	Output format
CoefFmt_g	Coefficient format
Channels_g	Number of parallel channels (1 is not supported, must be >= 2)
MaxRatio_g	Maximum decimation ratio supported
MaxTaps_g	Maximum number of taps supported
Rnd_g	Rounding mode at the output (round or truncate)
Sat_g	Saturation mode at the output (saturate of wrap)
UseFixCoefs_g	If true, fixed coefficients instead of configurable coefficients are implemented.
FixCoefs_g	Coefficients to use for *UseFixCoefs_g* = true.

3.6.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Input			
InVld	Input	1	AXI-S handshaking signal
InData	Input	InFmt_g	Input data, one channel is passed after the other
Output			
OutVld	Output	1	AXI-S handshaking signal
OutAbs	Output	OutFmt_g	Output data, one channel is passed after the other
Configuration			
Ratio	Input	$ceil(\log_2(\text{MaxRatio\_g}))$	Decimation ratio -1 0 → no decimation 1 → decimation by 2) This port is optional. If it is not connected, *MaxRatio_g* is used as fixed ratio.

Taps	Input	$ceil(\log_2(\text{MaxTaps\_g}))$	Taps – 1 0 → 1 Tap (order 0 filter) 63 → 64 Taps (order 63 filter) This port is optional. If it is not connected, *MaxTaps_g* is used as fixed tap count.
Coefficient Interface			
CoefClk	Input	1	Clock for the coefficient interface This port can be left unconnected for fixed coefficient implementation (*UseFixCoefs_g* = true)
CoefWr	Input	1	Coefficient write enable signal This port can be left unconnected for fixed coefficient implementation (*UseFixCoefs_g* = true)
CoefAddr	Input	$ceil(\log_2(\text{MaxTaps\_g}))$	Address of the coefficient to access This port can be left unconnected for fixed coefficient implementation (*UseFixCoefs_g* = true)
CoefWrData	Input	CoefFmt_g	Coefficient value for write access (*CoefWr = 1)* This port can be left unconnected for fixed coefficient implementation (*UseFixCoefs_g* = true)
CoefRdData	Output	CoefFmt_g	Coefficient read data (valid 1 cycle after applying the address) This port can be left unconnected for fixed coefficient implementation (*UseFixCoefs_g* = true)

The coefficient interface has a separate clock since often the data processing clock is coupled to an ADC clock but the main bus system that configures the filter is running on a different clock.

The filter can continue taking new input data even if a calculation is ongoing. As a result, the handling of packpressure is not required as long as the processing power of the filter is sufficient to handle all input data. For the calculation, see below.

Note that the behavior of the filter is undefined if the maximum input rate that can be handles is exceeded.

3.6.4 Architecture

The figure below roughly shows the architecture of the FIR filter. Since the channels arrive one after the other, the one dual-port RAM is sufficient to store all data. The RAM is split into different regions (i.e. the higher address bits select the region reserved for a given channel).

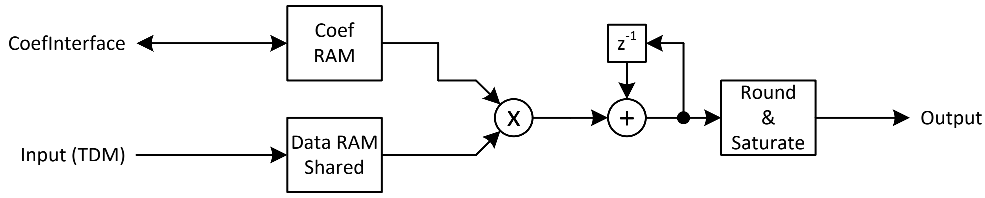


Figure 9: psi_fix_fix_dec_ser_nch_chtdm_conf Architecture

A state machine (not shown in the figure for simplicity) starts a new calculation whenever all required input samples for the next calculation arrived.

The accumulation is executed at the full output precision of the multiplication. This matches the implementation of the DSP slices in Xilinx devices, so they can be fully utilized.

The accumulator contains one guard bit compared to the output format to detect overflows. However, the user (designer who integrates the filter) is responsible to choose coefficients in a way that the output format is never exceeded by more than a factor of two. This this is not possible the filter output format must be chosen large enough ($Range_{Output} \geq 0.5 \cdot MaximumOutput$) and saturated externally.

Obviously the architecture requires one clock cycle per tap calculation of one channel. As a result the maximum number of filter taps depends on the number of channels N_{CH} clock frequency F_{clk}, the input sample rate $F_{s,in}$ and the decimation ratio R.

$$Taps_{max} = \frac{F_{clk} \cdot R}{F_{s,in} \cdot N_{CH}}$$

In case of fixed coefficient implementation, the coefficient RAM is replaced by a ROM automatically.

Important note: Changing the decimation rate and/or the filter order at runtime can temporarily lead to inconsistent settings because usually they are changed by register accesses that are executed one after the other. To avoid this problem, it is suggested to keep the filter in reset whenever the parameters are changed.

3.7 psi_fix_lin_approx_<function>

3.7.1 Description

This is actually not just one component but a whole family of components. They are all function approximations based on a table containing the function values for regularly spaced points and linear approximation between them.

All components are based on the same implementation of the approximation (*psi_fix_lin_approx_calc.vhd*) and they only vary in number formats and coefficient tables.

The code is not written by hand but generated from Python (*psi_fix_lin_approx.py)*. If a new function approximation shall be developed, it can first be designed using the function *psi_fix_lin_approx.Design()* that also helps finding the right settings. Afterwards VHDL code and a corresponding bittrueness testbench can be generated using *psi_fix_lin_approx.GenerateEntity()* and *psi_fix_lin_approx.GenerateTb()*.

3.7.2 Generics

Sinde each function approximation is built for an exact input range, precision and function, no parameters are required.

3.7.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Input			
InVld	Input	1	AXI-S handshaking signal
InData	Input	*	Signal input
Output			
OutVld	Output	1	AXI-S handshaking signal
OutData	Output	*	Result output

* The width of these ports depends on the specific function approximation.

The implementation of the linear approximation is fully pipelined. This means it can take one input sample every clock cycle. As a result the handling of backpressure was not implemented.

3.7.4 Architecture

The figure below shows the interpolation principle.

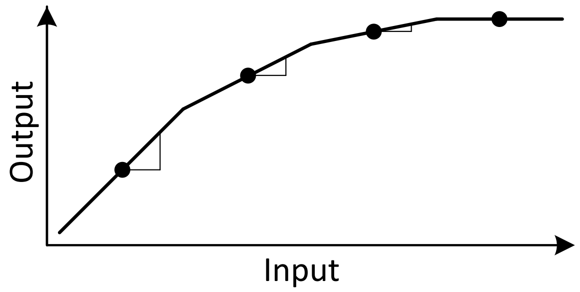


Figure 10: psi_fix_lin_approx Interpolation Principle

The complete range of the function is split into small sections. For each section the center point as well as the gradient are known and the output value is calculated from these two values (together with the difference between actual input and center point of the current segment).

The figure below shows the implementation of the approximation.

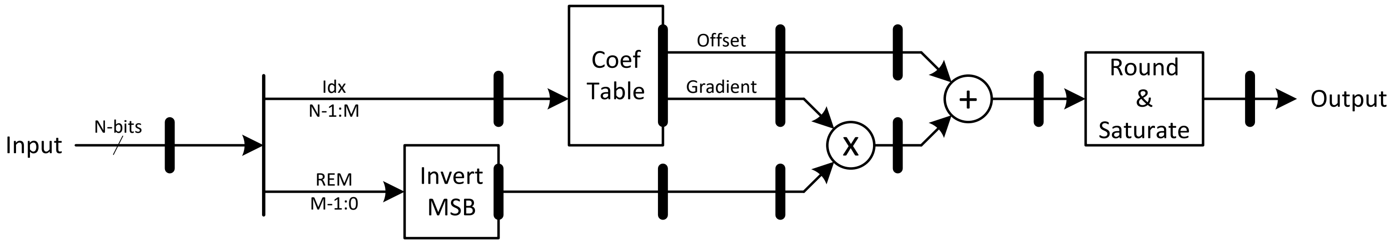


Figure 11: psi_fix_lin_approx Architecture

After splitting the input into index and reminder, the reminder is unsigned and related to the beginning of the segment. By inverting the MSB, the reminder is converted to the signed offset related to the center point of the segment.

The addition after the multiplication is executed at full precision and without rounding/truncation. This allows for the adder being implemented within a DSP slice. The rounding/truncation is then implemented in a separate pipeline stage.

3.8 psi_fix_dds_18b

3.8.1 Description

This entity implements an 18-bit DDS. The sine-wave is generated using the entity *psi_fix_lin_approx_sin_18b* and it has an error of less than one LSB for all values. As a result, there are no significant spurs in the generated spectrum (significant in terms of above the quantization noise floor) as shown in the figure below.

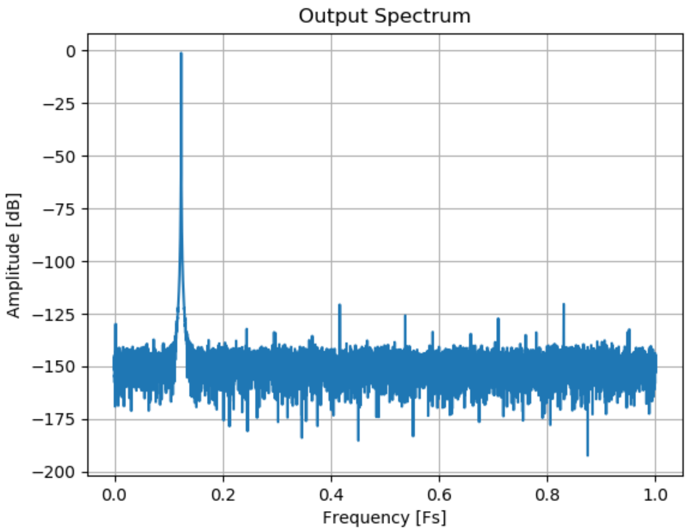


Figure 12: psi_fix_dds_18b Spectrum for PhaseStep=0.12345

3.8.2 Generics

PhaseFmt_g Phase accumulator format. This must be a number format with a range of 1.0 (either [0,0,x] or [1,-1,x]). A phase of 1.0 corresponds to 2π resp. one fully sine period.

3.8.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Configuration			
Restart	Input	1	This signal can be used to start the DDS again at the phase offset. This is useful if 100% reproducible outputs must be generated several times.
PhaseStep	Input	PhaseFmt_g	Phase step between two consecutive output samples. The phase step is given in 2π (0.5 corresponds to π). The phase step can be changed at runtime safely.
PhaseOffset	Input	PhaseFmt_g	Phase offset of the generated signal. The phase offset is given in 2π (0.5 corresponds to π). The phase offset can be changed at runtime safely.

Input			
InVld	Input	1	AXI-S handshaking signal that can be used to generate samples at any rate. For continuous operation (one sample per clock cycle) , the signal can be left unconnected.
Output			
OutVld	Output	1	AXI-S handshaking signal
OutSin	Output	18	Sine wave output in the format [1,0,17]
OutCos	Output	18	Cosine wave output in the format [1,0,17]

The total pipeline delay of the DDS is 10 clock cycles.

3.8.4 Architecture

The figure below shows the implementation of the DDS.

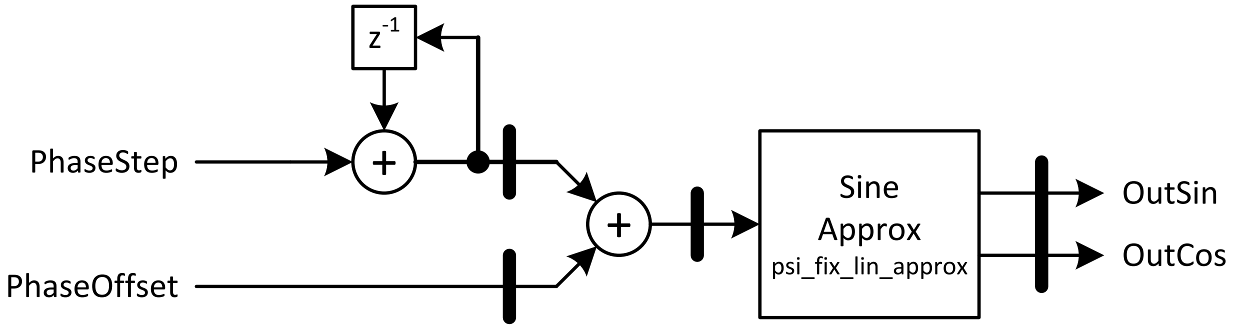


Figure 13: psi_fix_dds_18b Architecture

3.9 psi_fix_lowpass_iir_order1

3.9.1 Description

This entity implements a first order IIR lowpass with integrated coefficient calculation.

Note that the filter is targeted mainly to applications where the cutoff frequency is only one or two orders of magnitude lower than the sampling frequency.

For cases where the cutoff frequency is close to DC, the requirements for coefficient precision grow with this straight-forward filter structure. In this case a completely different structure especially targeted to low cutoff frequencies should be used instead of this standard component.

The filter requires that the coefficient format is passed as generic. Therefore the coefficient calculations are given below, so the user can evaluate the coefficients and decide on a format with acceptable quantization error.

$$\alpha = e^{-2\cdot\pi\cdot\frac{F_{cutoff}}{F_{sample}}}$$

$$\beta = 1 - alpha$$

3.9.2 Generics

FSampleHz_g	Sample frequency in Hz (strobe frequency)
FCutoffHz_g	Cutoff frequency in Hz (-3dB point)
InFmt_g	Input format
OutFmt_g	Output format
IntFmt_g	Format used for all internal calculations
CoefFmt_g	Coefficient format
Round_g	Rounding mode used everywhere in the filter (use *PsiFixTrunc* for highest clock speeds)
Sat_g	Saturation mode used everywhere in the filter (use *PsiFixWrap* for highest clock speeds, IIR filters of order 1 do not overshoot anyway, so saturation should not be required)
Pipeline_g	True → Highest clock frequencies but also higher latency False → Lowest latency but reduced clock speed
ResetPolarity_g	Polarity of the reset ('1' = high active)

3.9.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
clk_i	Input	1	Clock
rst_i	Input	1	Reset
Input			
str_i	Input	1	Input strobe (same as *Vld*). **The maximum allowed strobe rate is** $\frac{F_{clk}}{3}$
data_i	Input	InFmt_g	Data input
Output			
str_o	Output	1	Output strobe (same as *Vld*)
data_o	Output	OutFmt_g	Data output

3.9.4 Architecture

The figure below shows the implementation of the IIR filter. The pipeline stages in green are only present if *Pipeline_g = True*.

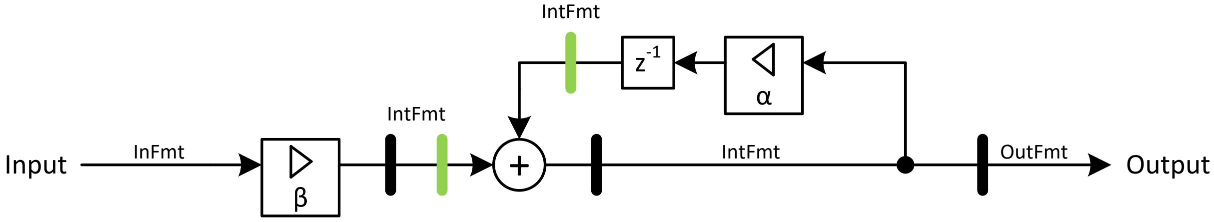


Figure 14: psi_fix_lowpass_iir_order1 Architecture

3.10 psi_fix_complex_mult

3.10.1 Description

The block performs multiplication on a complex number pair (*Inphase* & *Quadrature*, inputs of the block) or 2D matrix computation, let two complex numbers be:

$$x = (a + ib); y = (c + id)$$

The multiplication result comes:

$$x.y = (a + ib)(c + id) = (ac - bd) + i(ad + bc)$$

Where: In-phase input=a; Quadrature input=b; I1=c; I2=d; Q1=I2=d; Q2=I1=c

The block could be seen as well as 2D matrix multiplication, apart from the fact that a subtraction is hardcoded on the in-phase path and the given processing is equal as the one shown below:

$$\begin{bmatrix} Iout \\ Qout \end{bmatrix} = \begin{bmatrix} Inphase \\ Quadrature \end{bmatrix} \times \begin{bmatrix} I1 & -I2 \\ Q1 & Q2 \end{bmatrix} = \begin{matrix} Inphase \times I1 - Quadrature \times I2 \\ Inphase \times Q1 + Quadrature \times Q2 \end{matrix}$$

The total pipeline delay of the block is 3 clock cycles if no pipeline activation is set through generics, otherwise the pipeline is doubled (i.e. 6 stages)

3.10.2 Generics

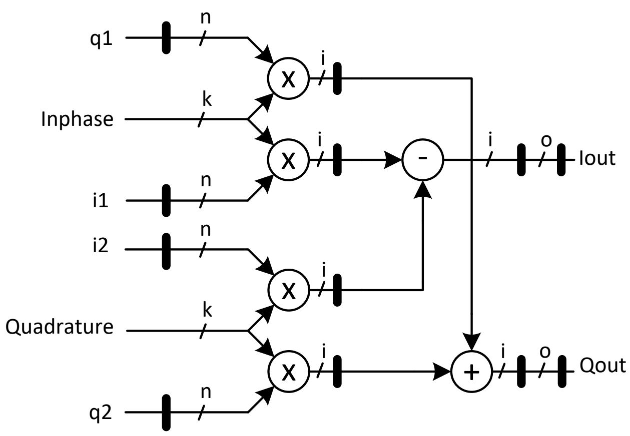
RstPol_g set the reset polarity
Pipeline_g Add internal register pipeline to get higher clock frequency synthesis result
InFixFmt_g Input format
InternalFmt_g Internal format
CoefFmt_g Coefficient format
OutFmtr_g Output format

3.10.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
clk_i	Input	1	Clock
rst_i	Input	1	Synchronous Reset
Input			
ipath_i	Input	InFixFmt_g	Real part of complex number input (in-phase data)
qpath_i	Input	InFixFmt_g	Imaginary part of complex number input (quadrature data)
vld_i	Input	1	Data strobe input
i1_i	Input	CoefFmt_g	Please refer to calculation description above §2.9.1
i2_i	Input	CoefFmt_g	Please refer to calculation description above §2.9.1
q1_i	Input	CoefFmt_g	Please refer to calculation description above §2.9.1

q2_i	Input	CoefFmt_g	Please refer to calculation description above §2.9.1
Output			
vld_o	Output	1	Data strobe output
iout_o	Output	OutFmt_g	Real part of complex number output (in-phase data)
out_o	Output	OutFmt_g	Imaginary part of complex number output (quadrature data)

3.10.4 Architecture



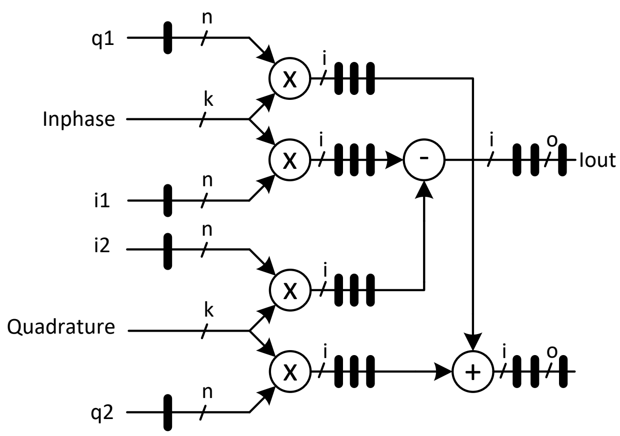


Figure 15: psi_fix_complex_mult Architecture – Pipeline_g = 0 (left) Pipeline_g = 1

3.11 psi_fix_mov_avg

3.11.1 Description

This entity implements a moving average implementation. It does not only calculate the moving sum but also compensate the gain from summing up multiple samples (either roughly by just shifting or exact by shifting and multiplication) if required.

The delay line is implemented using *psi_common_delay*, so the user can choose if SRLs or BRAMs shall be used or if the decision shall be taken automatically.

The gain of the filter including the compensation can be calculated by the formulas below:

$$G_{None} = Taps$$

$$G_{Rough} = \frac{Taps}{2^{ceil(log2(Taps))}}$$

$$G_{Exact} = 1.0$$

3.11.2 Generics

InFmt_g	Input format	
OutFmt_g	Output format	
Taps_g	Number of samples to do the moving average over	
GainCorr_g	“NONE”	The gain is not compensated
	“ROUGH”	The gain is roughly compensated by shifting (0.5 < gain < 1.0)
	“EXACT”	The gain is roughly compensated by shifting and then exactly adjusted using a multiplier. The resulting gain is 1.0 (with the precision of the 17-bit coefficient).
Round_g	Rounding mode at the output	
Sat_g	Saturation mode at the output	
OutRegs_g	Number of output register stages	

3.11.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Input			
InVld	Input	1	AXI-S handshaking signal
InData	Input	InFmt_g	Data input
Output			
OutVld	Output	1	AXI-S handshaking signal
OutData	Output	OutFmt_g	Data output

3.11.4 Architecture

The figure below shows the implementation of the moving average filter. All three gain correction implementations are shown in the figure while only the selected one is implemented of course.

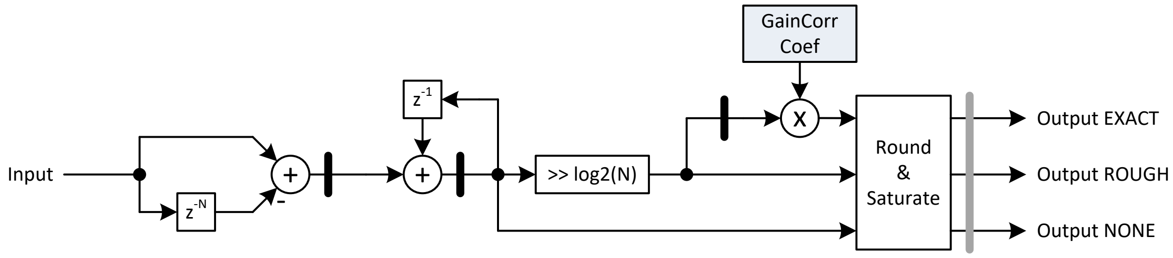


Figure 16: psi_fix_mov_avg Architecture

The number formats are not shown in the figure for simplicity since there are some calculations required. For details about the number formats, refer to the code. All number formats are automatically chosen in a way that no overflows occur internally.

The output register is shown in grey since the number of output registers is configurable.

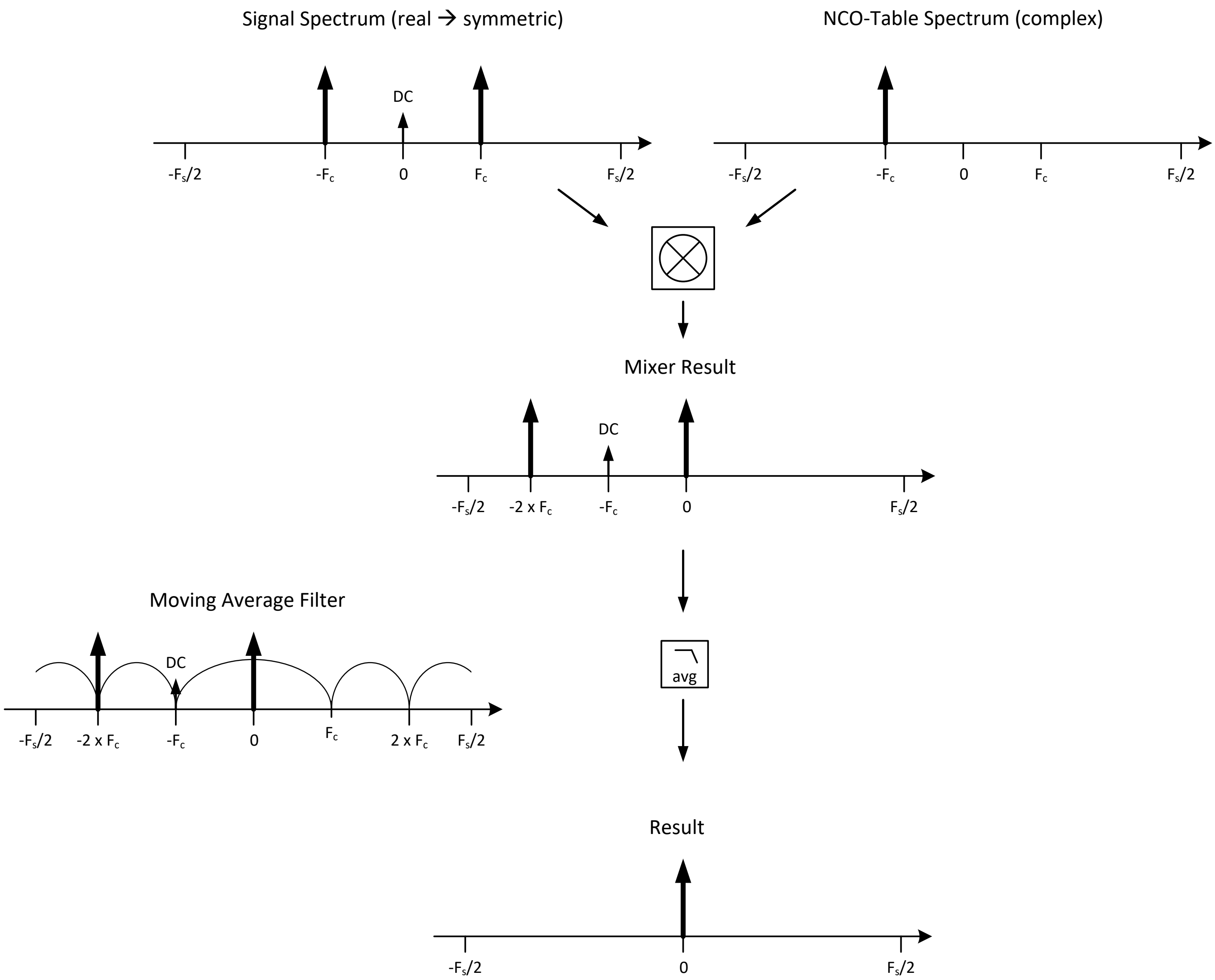
3.12 psi_fix_demod_real2cplx

3.12.1 Description

This entity implements a simple demodulator that takes a real input and produces a complex result. The demodulator first mixes the signal with the carrier frequency (generated internally in the demodulator using a table) and then filters the output with a moving-average filter (comb-filter) with $\frac{F_{sample}}{F_{carrier}}$ taps. This algorithm is illustrated in the figures at the end of this section.

The demodulator does only produce good quality results for very narrow-band signals with no significand out-of-band noise. If the signal has significant sidebands or noise, either additional filtering after the demodulator is required or a specialized demodulator must be written.

Another requirement of the demodulator is, that the carrier frequency is an integer fraction of the clock frequency.



3.12.2 Generics

RstPol_g Reset polarity ('1' = high active)
DataFmt_g Input, Output and Coefficient format
Ratio_g Ratio between sample frequency and carrier frequency
GainCorr_g Gain correction mode of the moving average filter (see *psi_fix_mov_avg*)
"NONE" The gain is not compensated
"ROUGH" The gain is roughly compensated by shifting (0.5 < gain < 1.0)
"EXACT" The gain is roughly compensated by shifting and then exactly adjusted using a multiplier. The resulting gain is 1.0 (with the precision of the 17-bit coefficient).

3.12.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
clk_i	Input	1	Clock
rst_i	Input	1	Synchronous Reset
Input			
str_i	Input	1	Input strobe (same as *Vld*).
data_i	Input	DataFmt_g	Data input
phi_offset_i	Input	log2(Ratio_g)	Phase offset of the mixer frequency in $\frac{2\pi}{Ratio_g}$
Output			
data_I_o	Output	DataFmt_g	Real part of the output signal
data_Q_o	Output	DataFmt_g	Imaginary part of the output signal
str_o	Output	1	Output strobe (same as *Vld*)

3.12.4 Architecture

The figure below shows the implementation of the demodulator.

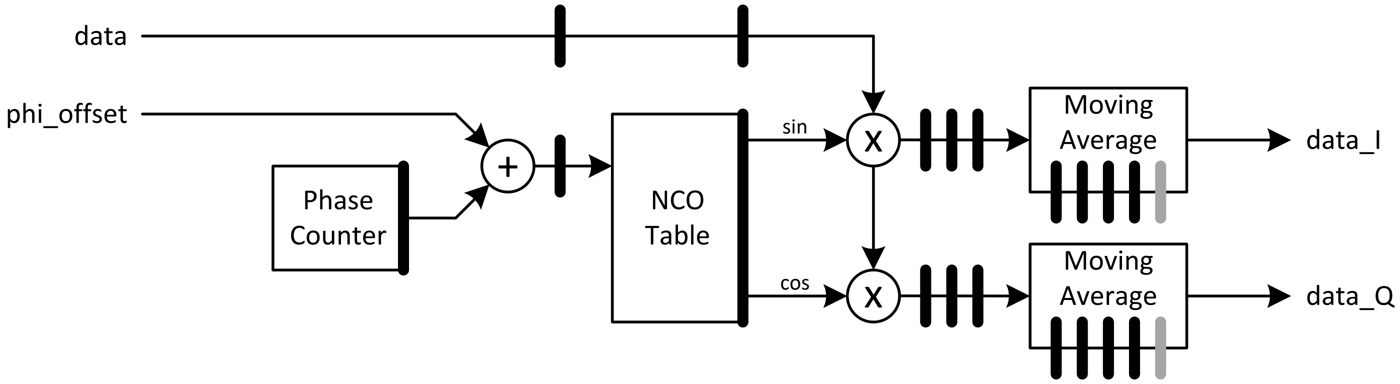


Figure 17: psi_fix_mov_avg Architecture

The pipeline stage shown in grey within the moving average filter is only present for *GainCorr_c = EXACT*.

The additional pipeline stage for the phase counter does not have to be compensated because the phase counter is incremented only after each sample and not before.