**psi\_fix**

Documentation

**Content**

**Table of Contents**

[1 Introduction 4](#_Toc515285533)

[2 Tipps & Tricks 4](#_Toc515285534)

[2.1 Library Setup 4](#_Toc515285535)

[2.2 Heavy Pipelining 4](#_Toc515285536)

[3 RTL Descriptions 8](#_Toc515285537)

[3.1 psi\_fix\_bin\_div 8](#_Toc515285538)

[3.2 psi\_fix\_cic\_dec\_fix\_1ch 10](#_Toc515285539)

[3.3 psi\_fix\_cic\_int\_fix\_1ch 12](#_Toc515285540)

[3.4 psi\_fix\_cordic\_abs\_pl 14](#_Toc515285541)

[3.5 psi\_fix\_fir\_dec\_ser\_nch\_chpar\_conf 16](#_Toc515285542)

[3.6 psi\_fix\_fir\_dec\_ser\_nch\_chtdm\_conf 19](#_Toc515285543)

[3.7 psi\_fix\_lin\_approx\_<function> 22](#_Toc515285544)

[3.8 psi\_fix\_dds\_18b 24](#_Toc515285545)

[3.9 psi\_fix\_lowpass\_iir\_order1 26](#_Toc515285546)

[3.10 psi\_fix\_complex\_mult 28](#_Toc515285547)

[3.11 psi\_fix\_mov\_avg 30](#_Toc515285548)

[3.12 psi\_fix\_demod\_real2cplx 32](#_Toc515285549)

[3.13 psi\_fix\_cordic\_vect 35](#_Toc515285550)

[3.14 psi\_fix\_cordic\_rot 37](#_Toc515285551)

**Figures**

[Figure 1: Heavy Pipelining, Problem Description 5](#_Toc515285552)

[Figure 2: Heavy Pipelining, Retiming, Implementation without retiming 5](#_Toc515285553)

[Figure 3: Heavy Pipelining, Retiming, Implementation with retiming 6](#_Toc515285554)

[Figure 4: Heavy Pipelining, Manual Splitting 7](#_Toc515285555)

[Figure 5: psi\_fix\_bin\_div Architecture 9](#_Toc515285556)

[Figure 6: psi\_fix\_cic\_dec\_fix\_1ch Architecture 11](#_Toc515285557)

[Figure 7: psi\_fix\_cic\_int\_fix\_1ch Architecture 13](#_Toc515285558)

[Figure 8: psi\_fix\_fix\_dec\_ser\_nch\_chpar\_conf Architecture 18](#_Toc515285559)

[Figure 9: psi\_fix\_fix\_dec\_ser\_nch\_chtdm\_conf Architecture 21](#_Toc515285560)

[Figure 10: psi\_fix\_lin\_approx Interpolation Principle 23](#_Toc515285561)

[Figure 11: psi\_fix\_lin\_approx Architecture 23](#_Toc515285562)

[Figure 12: psi\_fix\_dds\_18b Spectrum for PhaseStep=0.12345 24](#_Toc515285563)

[Figure 13: psi\_fix\_dds\_18b Architecture 25](#_Toc515285564)

[Figure 14: psi\_fix\_lowpass\_iir\_order1 Architecture 27](#_Toc515285565)

[Figure 15: psi\_fix\_complex\_mult Architecture – Pipeline\_g = 0 (left) Pipeline\_g = 1 29](#_Toc515285566)

[Figure 16: psi\_fix\_mov\_avg Architecture 31](#_Toc515285567)

[Figure 17: psi\_fix\_demod\_real2cplx Architecture 34](#_Toc515285568)

[Figure 18: psi\_fix\_coric\_vect Architecture 36](#_Toc515285569)

[Figure 18: psi\_fix\_coric\_rot Architecture 38](#_Toc515285570)

# 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.

# Tipps & Tricks

## 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>*.

## Heavy Pipelining

### 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

### 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.

### 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.

# RTL Descriptions

## psi\_fix\_bin\_div

### Description

This component implements a fixed point binary divider.

### 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)

### 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.

### 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

## psi\_fix\_cic\_dec\_fix\_1ch

### 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.

### 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

### 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

### 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:

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

Some of the most common formulas are given below.

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

Since this formula evaluates to 1.0 for the case (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.

## psi\_fix\_cic\_int\_fix\_1ch

### 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.

### 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

### 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.

### 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:

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

Some of the most common formulas are given below.

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

Since this formula evaluates to 1.0 for the case (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 ) 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.

## psi\_fix\_cordic\_abs\_pl

### 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.

### 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)

### 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.

### Architecture

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

The algorithm only works for , therefore the absolute value of is calculated prior to executing the algorithm.

The CORDIC gain can be calculated by the formula below:

Where:  
 Cordic Gain  
 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.

## psi\_fix\_fir\_dec\_ser\_nch\_chpar\_conf

### 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.

### 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.

### 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 |  | 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 |  | Output data in parallel (see *InData*) |
| ***Configuration*** | | | |
| Ratio | Input |  | 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 |  | 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 |  | 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.

### 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 ( ) 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 , the input sample rate and the decimation ratio .

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

## psi\_fix\_fir\_dec\_ser\_nch\_chtdm\_conf

### 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.

### 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.

### 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 |  | 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 |  | 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 |  | 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.

### 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 ( ) 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 clock frequency , the input sample rate and the decimation ratio .

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.

## psi\_fix\_lin\_approx\_<function>

### 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()*.

### Generics

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

### 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.

### 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.

## psi\_fix\_dds\_18b

### 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

### 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 resp. one fully sine period.

### 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 (0.5 corresponds to ). 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 (0.5 corresponds to ). 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.

### Architecture

The figure below shows the implementation of the DDS.



Figure 13: psi\_fix\_dds\_18b Architecture

## psi\_fix\_lowpass\_iir\_order1

### 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.

### 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)

### 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** |
| data\_i | Input | InFmt\_g | Data input |
| ***Output*** | | | |
| str\_o | Output | 1 | Output strobe (same as *Vld*) |
| data\_o | Output | OutFmt\_g | Data output |

### 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

## psi\_fix\_complex\_mult

### 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:

The multiplication result comes:

*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:

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)

### 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

### 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](#_Description) |
| i2\_i | Input | CoefFmt\_g | Please refer to calculation description above [§2.9.1](#_Description) |
| q1\_i | Input | CoefFmt\_g | Please refer to calculation description above [§2.9.1](#_Description) |
| q2\_i | Input | CoefFmt\_g | Please refer to calculation description above [§2.9.1](#_Description) |
| ***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) |

### Architecture



Figure 15: psi\_fix\_complex\_mult Architecture – Pipeline\_g = 0 (left) Pipeline\_g = 1

## psi\_fix\_mov\_avg

### 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:

### 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

### 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 |

### 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.

## psi\_fix\_demod\_real2cplx

### 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 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.



### Generics

**RstPol\_g** Reset polarity (‘1’ = high active)  
**InFmt\_g** Input format  
**OutFmt\_g** Output format  
**CoefBits\_g** Number of bits to use for coefficients (including sign). With 25x18 multipliers either 25 or 18   
 (depending on the width of the input).  
**Ratio\_g** Ratio between sample frequency and carrier frequency

### 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 |
| ***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*) |

### Architecture

The figure below shows the implementation of the demodulator.



Figure 17: psi\_fix\_demod\_real2cplx Architecture

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.

## psi\_fix\_cordic\_vect

### Description

This entity implements the CORDIC algorithm for Cartesian to Polar conversion.

The CORDIC gain can optionally be compensated. If the gain is compensated externally, it is important to know the exact gain. Therefore the formula for calculating the CORDIC gain is given:

For the internal gain compensation it is recommended to choose an *InternalFmt\_g* in a way that it can be processed with one multiplier (e.g. for 7-series max. 25 bits).

### Generics

**InFmt\_g** Input format of the X/Y components (must be signed)  
**OutFmt\_g** Output format for the amplitude (must be unsigned)  
**InternalFmt\_g** Internal calculation format for the X/Y components. (must be signed)  
 The more fractional bits, the more precise the calculation gets.  
 Choose enough integer bits to ensure that no overflows happen.   
 For inputs in the form (1,0,x) that are always within the unit circle, (1,1,y) can be used.  
 For inputs in the form (1,0,x) that can contain arbitrary values for X and Y, (1,2,y) can be   
 used.  
**AngleFmt\_g** Angle output format (must be unsigned)  
**AngleIntFmt\_g** Internal calculation format for angles (must be signed).  
 The more fractional bits, the more precise the calculation gets.  
**Iterations\_g** Number of CORDIC iterations  
**GainComp\_g** True The CORDIC gain (~1.62) is compensated internally with a multiplier  
 False The CORDIC gain is not compensated.  
**Round\_g** Rounding mode at the output (use truncation for high clock speeds)  
**Sat\_g** Saturation mode at the output (use wrapping for high clock speeds)  
**Mode\_g** “PIPELINED” One pipeline stage per CORDIC iteration, can take one sample every   
 clock cycle.  
 “SERIAL” One clock cycle per iteration, less logic utilitzation

### Interfaces

|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Direction | Width | Description |
| ***Control Signals*** | | | |
| Clk | Input | 1 | Clock |
| Rst | Input | 1 | Reset |
| ***Input*** | | | |
| InVld | Input | 1 | AXI-S handshaking signal |
| InRdy | Input | 1 | AXI-S handshaking signal (only required for “SERIAL”) |
| InI | Input | InFmt\_g | Real part of the input signal |
| InQ | Input | InFmt\_g | Imaginary part of the input signal |
| ***Output*** | | | |
| OutVld | Output | 1 | AXI-S handshaking signal |
| OutAbs | Output | OutFmt\_g | Absolute value of the output signal |
| OutAng | Output | AngleFmt\_g | Angle of the output signal (in 2π 🡪 0.5 = π = 180°) |

### Architecture

The figure below shows the implementation of the vectoring CORDIC. The algorithm only works correctly in quadrant zero (where I and Q are positive). Therefore the input is mapped into this quadrant by sign swapping and the effect of this mapping is compensated at the output.



Figure 18: psi\_fix\_coric\_vect Architecture

## psi\_fix\_cordic\_rot

### Description

This entity implements the CORDIC algorithm for Polar to Cartesian conversion.

The CORDIC gain can optionally be compensated. If the gain is compensated externally, it is important to know the exact gain. Therefore the formula for calculating the CORDIC gain is given:

For the internal gain compensation it is recommended to choose an *InternalFmt\_g* in a way that it can be processed with one multiplier (e.g. for 7-series max. 25 bits).

### Generics

**InAbsFmt\_g** Format of the absolute (=amplitude) input (must be unsigned)  
**InAngleFmt\_g** Format of the angle input (must be unsigned), usually (1,0,x)  
**OutFmt\_g** Output format for I/Q outputs, usually signed  
**InternalFmt\_g** Internal calculation format for the X/Y components. (must be signed)  
 The more fractional bits, the more precise the calculation gets.  
 Choose enough integer bits to ensure that no overflows happen.   
 For inputs with an amplitude <= 1.0, (1,1,y) can be used..  
**AngleIntFmt\_g** Internal calculation format for angles (must be signed).  
 The more fractional bits, the more precise the calculation gets.  
**Iterations\_g** Number of CORDIC iterations  
**GainComp\_g** True The CORDIC gain (~1.62) is compensated internally with a multiplier  
 False The CORDIC gain is not compensated.  
**Round\_g** Rounding mode at the output (use truncation for high clock speeds)  
**Sat\_g** Saturation mode at the output (use wrapping for high clock speeds)  
**Mode\_g** “PIPELINED” One pipeline stage per CORDIC iteration, can take one sample every   
 clock cycle.  
 “SERIAL” One clock cycle per iteration, less logic utilitzation

### Interfaces

|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Direction | Width | Description |
| ***Control Signals*** | | | |
| Clk | Input | 1 | Clock |
| Rst | Input | 1 | Reset |
| ***Input*** | | | |
| InVld | Input | 1 | AXI-S handshaking signal |
| InRdy | Input | 1 | AXI-S handshaking signal (only required for “SERIAL”) |
| InAbs | Input | InAbsFmt\_g | Amplitude input |
| InAng | Input | InAngleFmt\_g | Angle input (in 2π 🡪 0.5 = π = 180°) |
| ***Output*** | | | |
| OutVld | Output | 1 | AXI-S handshaking signal |
| OutI | Output | OutFmt\_g | In-phase part of the output signal (X component) |
| OutQ | Output | OutFmt \_g | Quadrature-phase of the output signal (Y component) |

### Architecture

The figure below shows the implementation of the vectoring CORDIC. The algorithm only works correctly in quadrant zero (where I and Q are positive). Therefore the input is mapped into this quadrant by sign swapping and the effect of this mapping is compensated at the output.



Figure 19: psi\_fix\_coric\_rot Architecture