**psi\_fix**

Documentation

**Content**

**Table of Contents**

[1 Introduction 4](#_Toc519692025)

[1.1 Working Copy Structure 4](#_Toc519692026)

[1.2 External Dependencies 4](#_Toc519692027)

[1.3 VHDL Libraries 5](#_Toc519692028)

[1.4 Running Simulations 5](#_Toc519692029)

[1.5 Contribute to PSI VHDL Libraries 6](#_Toc519692030)

[1.6 Handshaking Signals 7](#_Toc519692031)

[2 Tipps & Tricks 8](#_Toc519692032)

[2.1 Library Setup 8](#_Toc519692033)

[2.2 Heavy Pipelining 8](#_Toc519692034)

[3 RTL Descriptions 11](#_Toc519692035)

[3.1 psi\_fix\_bin\_div 11](#_Toc519692036)

[3.2 psi\_fix\_cic\_dec\_fix\_1ch 13](#_Toc519692037)

[3.3 psi\_fix\_cic\_dec\_fix\_nch\_par\_tdm 15](#_Toc519692038)

[3.4 psi\_fix\_cic\_dec\_fix\_nch\_tdm\_tdm 17](#_Toc519692039)

[3.5 psi\_fix\_cic\_int\_fix\_1ch 19](#_Toc519692040)

[3.6 psi\_fix\_cordic\_abs\_pl 21](#_Toc519692041)

[3.7 psi\_fix\_fir\_dec\_ser\_nch\_chpar\_conf 23](#_Toc519692042)

[3.8 psi\_fix\_fir\_dec\_ser\_nch\_chtdm\_conf 26](#_Toc519692043)

[3.9 psi\_fix\_lin\_approx\_<function> 29](#_Toc519692044)

[3.10 psi\_fix\_dds\_18b 32](#_Toc519692045)

[3.11 psi\_fix\_lowpass\_iir\_order1 34](#_Toc519692046)

[3.12 psi\_fix\_complex\_mult 36](#_Toc519692047)

[3.13 psi\_fix\_mov\_avg 38](#_Toc519692048)

[3.14 psi\_fix\_demod\_real2cplx 40](#_Toc519692049)

[3.15 psi\_fix\_cordic\_vect 43](#_Toc519692050)

[3.16 psi\_fix\_cordic\_rot 45](#_Toc519692051)

[3.17 psi\_fix\_pol2cart\_approx 47](#_Toc519692052)

[3.18 psi\_fix\_mod\_cplx2real 49](#_Toc519692053)

**Figures**

[Figure 1: Working copy structure 4](#_Toc519692105)

[Figure 2: Handshaking signals 7](#_Toc519692106)

[Figure 3: Heavy Pipelining, Problem Description 8](#_Toc519692107)

[Figure 4: Heavy Pipelining, Retiming, Implementation without retiming 9](#_Toc519692108)

[Figure 5: Heavy Pipelining, Retiming, Implementation with retiming 9](#_Toc519692109)

[Figure 6: Heavy Pipelining, Manual Splitting 10](#_Toc519692110)

[Figure 7: psi\_fix\_bin\_div Architecture 12](#_Toc519692111)

[Figure 8: psi\_fix\_cic\_dec\_fix\_1ch Architecture 14](#_Toc519692112)

[Figure 9: psi\_fix\_cic\_dec\_fix\_nch\_par\_tdm Architecture 16](#_Toc519692113)

[Figure 10: psi\_fix\_cic\_dec\_fix\_nch\_tdm\_tdm Architecture 18](#_Toc519692114)

[Figure 11: psi\_fix\_cic\_int\_fix\_1ch Architecture 20](#_Toc519692115)

[Figure 12: psi\_fix\_fix\_dec\_ser\_nch\_chpar\_conf Architecture 25](#_Toc519692116)

[Figure 13: psi\_fix\_fix\_dec\_ser\_nch\_chtdm\_conf Architecture 28](#_Toc519692117)

[Figure 14: psi\_fix\_lin\_approx Interpolation Principle 30](#_Toc519692118)

[Figure 15: psi\_fix\_lin\_approx Architecture 30](#_Toc519692119)

[Figure 16: psi\_fix\_dds\_18b Spectrum for PhaseStep=0.12345 32](#_Toc519692120)

[Figure 17: psi\_fix\_dds\_18b Architecture 33](#_Toc519692121)

[Figure 18: psi\_fix\_lowpass\_iir\_order1 Architecture 35](#_Toc519692122)

[Figure 19: psi\_fix\_complex\_mult Architecture – Pipeline\_g = 0 (left) Pipeline\_g = 1 37](#_Toc519692123)

[Figure 20: psi\_fix\_mov\_avg Architecture 39](#_Toc519692124)

[Figure 21: psi\_fix\_demod\_real2cplx Architecture 42](#_Toc519692125)

[Figure 22: psi\_fix\_coric\_vect Architecture 44](#_Toc519692126)

[Figure 23: psi\_fix\_coric\_rot Architecture 46](#_Toc519692127)

[Figure 24: psi\_fix\_pol2cart\_approx Architecture 48](#_Toc519692128)

[Figure 25: psi\_fix\_mod\_cplx2real Archietcture 50](#_Toc519692129)

# 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.

## Working Copy Structure

If you just want to use some components out of the *psi\_fix* library, the only requirement is to checkout *psi\_common* into the same directory as *psi\_fix* (side-by-side). The reason for this is that *psi\_fix* uses some components from the library *psi\_common*.

If you want to also run simulations and/or modify the library, additional repositories are required (available from the same source as *psi\_fix*) and they must be checked out into the folder structure shown in the figure below since the repositories reference each-other relatively.



Figure 1: Working copy structure

It is not necessary but recommended to use the name *psi\_lib* as name for the *<Root>* folder.

## External Dependencies

* Python 3.5 or higher is required to run the bit-true models of the *psi\_fix* library (which implicitly happens during regression tests)
* Python 3 must be callable using “*python3”* from the commandline on your system. For Linux this is the default, for windows it is recommended to create a copy of *python.exe* that is named *python3.exe*. Additionally the path to the python directory must be added to the PATH environment variable.
* The following packages from *pip* must be installed (“*pip install <package>”*)
  + Scipy
  + numpy

## VHDL Libraries

The PSI VHDL libraries (including *psi\_fix*) require all files to be compiled into the same VHDL library.

There are two common ways of using VHDL libraries when using PSI VHDL libraries:

1. All files of the project (including project specific sources and PSI VHDL library sources) are compiled into the same library that may have any name.   
   In this case PSI library entities and packages are referenced by *work.psi\_<library>\_<xxx> (e.g. work.psi\_fix\_bin\_div* or *work.psi\_common\_array\_pkg.all*).
2. All code from PSI VHDL libraries is compiled into a separate VHDL library. It is recommended to use the name *psi\_lib*.  
   In this case PSI library entities and packages are referenced by *psi\_lib.psi\_<lib>\_<xxx> (e.g. psi\_lib.psi\_fix\_bin\_div* or *psi\_lib.psi\_common\_array\_pkg.all*).

## Running Simulations

Currently only Modelsim is is supported, support for GHDL is planned.

### Regression Test

To run the regression test, follow the steps below:

* Open Modelsim
* The TCL console, navigate to *<Root>/VHDL/psi\_fix/sim*
* Execute the command “*source ./run.tcl*”

All test benches are executed automatically and at the end of the regression test, the result is reported. Note that python is called during that process, so if you python is not installed correctly, errors will occur.

### Working Interactively

During work on library components, it is important to be able to control simulations interactively. To do so, it is suggested to follow the following flow:

* Open Modelsim
* The TCL console, navigate to *<Root>/VHDL/psi\_fix/sim*
* Execute the command “*source ./interactive.tcl”*
  + This will compile all files and initialize the PSI TCL framework
  + From this point on, all the commands from the PSI TCL framework are available, see documentation of *PsiSim*
* Most useful commands to recompile and simulate entities selectively are
  + *compile\_files –contains <string>*
  + *run\_tb –contains <string>*

## Contribute to PSI VHDL Libraries

To contribute to the PSI VHDL libraries, a few rules must be followed:

* Good Code Quality
  + There are not hard guidelines. However, your code shall be readable, understandable, correct and save. In other words: Only good code quality will be accepted.
* Configurability
  + If there are parameters that other users may have to modify at compile-time, provide generics. Only code that is written in a generic way and can easily be reused will be accepted.
* Bit-true model
  + A bit-true python model must be provided for *psi\_fix* components. Otherwise they will not be accepted.
* Self checking Test-benches
  + It is mandatory to provide a self-checking test-bench with your code.
  + The test-bench shall cover all features of your code
  + The test-bench shall automatically stop after it is completed (all processes halted, clock-generation stopped). See existing test-benches provided with the library for examples.
  + The test-bench shall only do reports of severity *error*, *failure* or even *fatal* if there is a real problem.
  + If an error occurs, the message reported shall start with “###ERROR###:”. This is required since the regression test script searches for this string in reports.
  + For *psi\_fix*, the test bench must call the python model and check if VHDL and python are bit-true
* Documentation
  + Extend this document with proper documentation of your code.
* New test-benches must be added to theregression test-script
  + Change */sim/config.tcl* accordingly
  + Test if the regression test really runs the new test-bench and exits without errors before doing any merge requests.

## Handshaking Signals

### General Information

The PSI library uses the AXI4-Stream handshaking protocol (herein after called AXI-S). Not all entities may implement all optional features of the AXI-S standard (e.g. backpressure may be omitted) but the features available are implemented according to AXI-S standard and follow these rules.

The full AXI-S specification can be downloaded from the ARM homepage:  
<https://developer.arm.com/docs/ihi0051/a>

The most important points of the specification are outlined below.

### Excerpt of the AXI-S Standard

A data transfer takes place during a clock cycle where TVALID and TREADY (if available) are high. The order in which they are asserted does not play any role.

* A master is not permitted to wait until TREADY is asserted before asserting TVALID.
* Once TVALID is asserted it must remain asserted until the handshake occurs.
* A slave is permitted to wait for TVALID to be asserted before asserting the corresponding TREADY.
* If a slave asserts TREADY, it is permitted to de-assert TREADY before TVALID is asserted.

An example an AXI handshaking waveform is given below. All the points where data is actually transferred are marked with dashed lines.



Figure 2: Handshaking signals

### Naming

The naming conventions of the AXI-S standard are not followed strictly. The most common synonyms that can be found within the PSI VHDL libraries are described below:

TDATA InData, OutData, Data, Sig, Signal, <application specific names>

TVALID Vld, InVld, OutVld, Valid, str, str\_i

TREADY Rdy, InRdy, OutRdy

Note that instead of one TDATA signal (as specified by AXI-S) the PSI VHDL Library sometimes has multiple data signals that are all related to the same set of handshaking signals. This helps with readability since different data can is represented by different signals instead of just one large vector.

# 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 3: 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 4: 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 5: 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 6: 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 7: 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* | Filter input |
| ***Output*** | | | |
| OutVld | Output | 1 | AXI-S handshaking signal |
| OutData | Output | *InFmt\_g* | Filter 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 8: 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\_dec\_fix\_nch\_par\_tdm

### Description

This component implements a decimating multi-channel CIC filter that takes all channels in parallel on the input side but delivers output in TDM fashion.

This filter is equal to the one described in 3.2, the only difference is that it supports multiple channels. So for details refer to 3.2.

### Generics

**Channels\_g** Number of channels (must be >= 2) **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\*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 |
| OutData | Output | *OutFmt\_g* | Output data in TDM fashion. The first output sample is Channel 0, then Channel 1, … |

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

For details on the filter mathematics, refer to 3.2.4. This section only describes how the multi channel filter is implemented.



Figure 9: psi\_fix\_cic\_dec\_fix\_nch\_par\_tdm Architecture

The symbols are defined as follows:

Decimation ratio  
 Differential delay  
 CIC order  
 Number of channels  
 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

## psi\_fix\_cic\_dec\_fix\_nch\_tdm\_tdm

### Description

This component implements a decimating multi-channel CIC filter that works in TDM fashion.

This filter is equal to the one described in 3.2, the only difference is that it supports multiple channels. So for details refer to 3.2.

### Generics

**Channels\_g** Number of channels (must be >= 2) **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* | Input data time-division-multiplexed.  The first sample is Channel 0, the second one Channel 1, … |
| ***Output*** | | | |
| OutVld | Output | 1 | AXI-S handshaking signal |
| OutData | Output | *OutFmt\_g* | Output data in TDM fashion. The first output sample is Channel 0, then Channel 1, … |

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

For details on the filter mathematics, refer to 3.2.4. This section only describes how the multi channel filter is implemented.



Figure 10: psi\_fix\_cic\_dec\_fix\_nch\_tdm\_tdm Architecture

The symbols are defined as follows:

Decimation ratio  
 Differential delay  
 CIC order  
 Number of channels  
 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

## 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 11: 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 12: 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 13: 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

Since 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 14: 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 15: 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.

### Function Details

#### Sin18b

**Function:** Sine  
**Input Range:** 0 … 1 (in 2 π)  
**Remarks:** The sine wave is scaled down by exactly one LSB to exclude +/-1.0 to prevent an   
 additional integer bit being required for only this case.

#### Sqrt18b

**Function:** Square root  
**Input Range:** 0.25 … 1  
**Remarks:** If the input signal can be below 0.25, it must be shifted into this range and the shift must be compensated at the output. This can be achieved by shifting the input by *2N* bits to the left   
 and shift the output by *N* bits to the right since *.*

## 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 16: 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 17: 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 18: 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:

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  
**InAFmt\_g** Input A format  
**InBFmt\_g** Input A format  
**InternalFmt\_g** Internal format  
**OutFmt\_g** Output format  
**Round\_g** Output rounding mode  
**Sat\_g** Output saturation mode

### Interfaces

|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Direction | Width | Description |
| ***Control Signals*** | | | |
| clk\_i | Input | 1 | Clock |
| rst\_i | Input | 1 | Synchronous Reset |
| ***Input*** | | | |
| ai\_i | Input | InAFmt\_g | Real part of input signal A |
| aq\_i | Input | InAFmt\_g | Imaginary part of input signal A |
| bi\_i | Input | InBFmt\_g | Real part of input signal B |
| bq\_i | Input | InBFmt\_g | Imaginary part of input signal B |
| vld\_i | Input | 1 | AXI-S Handshaking signal |
| ***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 19: psi\_fix\_complex\_mult Architecture – Pipeline\_g = false (left) Pipeline\_g = true

## 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 20: 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 21: 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  
**PlStgPerIter\_g** Number of pipeline stages per iteration (1 or 2).  
 This setting only has an effect on the pipelined implementation. For the serial   
 implementation it does not have any effect.

### 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 22: 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).

***Important Note:***

In most cases (especially for Signals < 18 bits), the entity *psi\_fix\_pol2cart\_approx* (see 3.17) offers a better trade-off between resource usage and performance than the *psi\_fix\_cordic\_rot*. So it may be worth considering switching to that component.

### 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. The value is always < 0.25   
 (corresponds to 0.5 \* π) since the calculation is always mapped into the same quadrant.  
**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 23: psi\_fix\_coric\_rot Architecture

## psi\_fix\_pol2cart\_approx

### Description

This entity implements a polar to cartesian conversion based on a linear approximation of the sine/cosine function. In most cases (especially for signals with less than 18 bits) this approach offers a better tradeoff between resource usage and performance.

Compared to the CORDIC implementation, 4 instead of 2 or 0 28x18 multipliers (depending on gain correction) are used and additional 72kBit of BRAM are used (= 4 RAMB18). On the other hand the LUT usage is lower than for the serial CORDIC implementation and the throughput is the same as for the pipelined CORDIC implementation.

### 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  
**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)

### Interfaces

|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Direction | Width | Description |
| ***Control Signals*** | | | |
| Clk | Input | 1 | Clock |
| Rst | Input | 1 | Reset |
| ***Input*** | | | |
| InVld | Input | 1 | AXI-S handshaking signal |
| 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

Note that some additional output registers outside the entity may be required if rounding and saturation are used.



Figure 24: psi\_fix\_pol2cart\_approx Architecture

## psi\_fix\_mod\_cplx2real

### Description

The block converts complex data to real output with weighted coefficient regarding the given clock ratio K as generic. Giving input data In-phase and Quadrature at the input gives the following result:

Where sin & cos angle are computed within a table as follow:

The total pipeline delay of the block is 4 clock cycles Generics

**RstPol\_g** set the reset polarity  
**InpFmt\_g** Fixed pointInput format

**CoefFmt\_g** Fixed pointCoefficient format

**IntFmt\_g** Fixed pointInternal format  
**OutFmt\_g** Fixed pointOutput format

**Ratio\_g** Frequency output ratio regarding clockfrequency*(i.e. if Freq. clk 100MHz, ratio 5 => 20MHz)*

### Interfaces

|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Direction | Width | Description |
| ***Control Signals*** | | | |
| clk\_i | Input | 1 | Clock |
| rst\_i | Input | 1 | Synchronous Reset |
| ***Input*** | | | |
| data\_I\_i | Input | InpFmt\_g | Real part of complex number input (in-phase data) |
| data\_Q\_i | Input | InpFmt\_g | Imaginary part of complex number input (quadrature data) |
| vld\_i | Input | 1 | Data strobe input |
| ***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 25: psi\_fix\_mod\_cplx2real Archietcture