psi_fix

Documentation

Content

Table of Contents

Figures

1 Introduction

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

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

1.1 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.

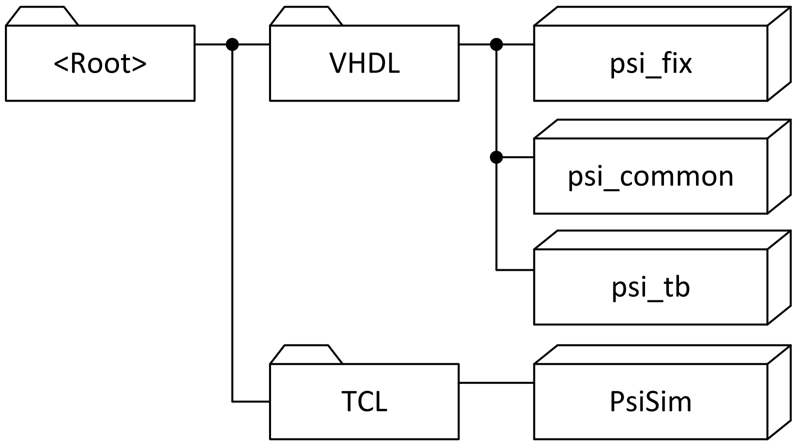


Figure 1: Working copy structure

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

1.2 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

1.3 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:

a) 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*).

b) 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*).

1.4 Running Simulations

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

1.4.1 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.

1.4.2 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>*

1.5 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.

1.6 Handshaking Signals

1.6.1 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.

1.6.2 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.

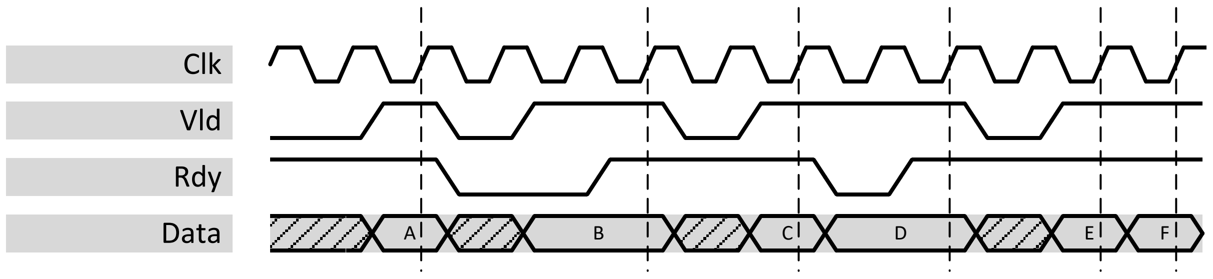


Figure 2: Handshaking signals

1.6.3 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.

2 Tipps & Tricks

2.1 Library Setup

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

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

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

2.2 Heavy Pipelining

2.2.1 Problem Description

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

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

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

…

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

This leads to the implementation shown below.

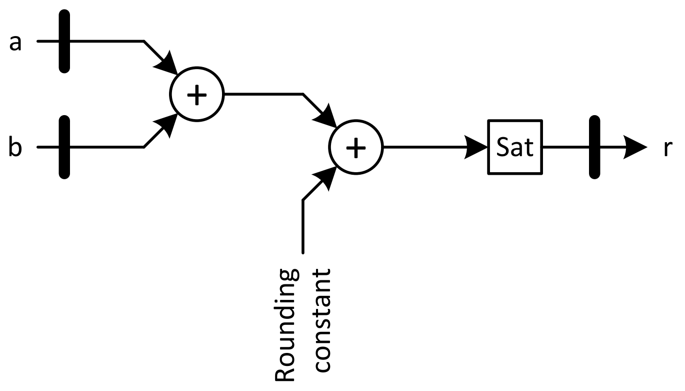


Figure 3: Heavy Pipelining, Problem Description

2.2.2 Solution 1: Register Retiming

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

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

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

…

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

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

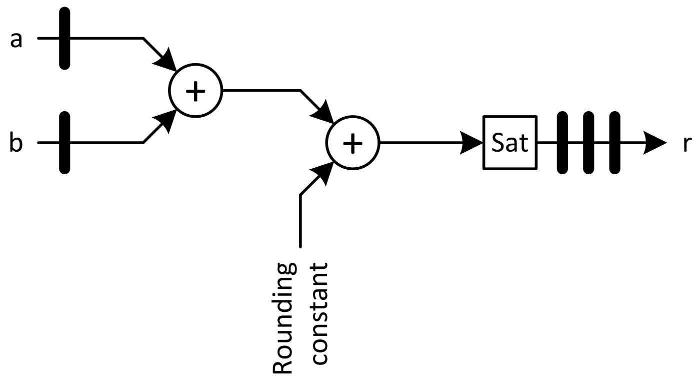


Figure 4: Heavy Pipelining, Retiming, Implementation without retiming

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.

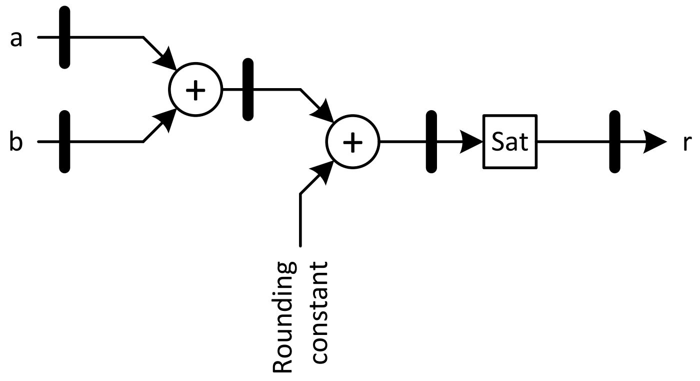


Figure 5: Heavy Pipelining, Retiming, Implementation with retiming

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.

2.2.3 Solution 2: Manual Splitting

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

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


…

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

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

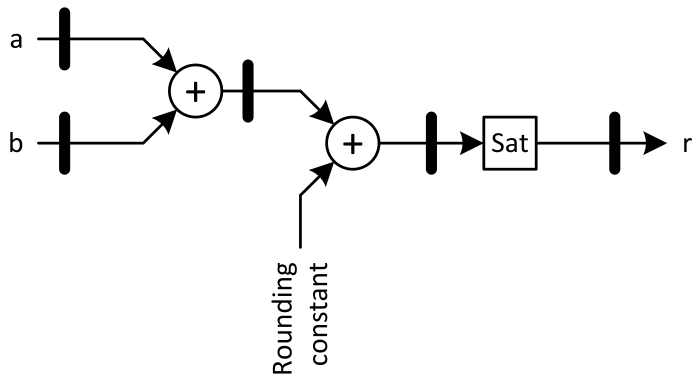


Figure 6: Heavy Pipelining, Manual Splitting

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.

3 RTL Descriptions

3.1 psi_fix_bin_div

3.1.1 Description

This component implements a fixed point binary divider.

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

3.1.2 Generics

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

3.1.3 Interfaces

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

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

3.1.4 Architecture

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

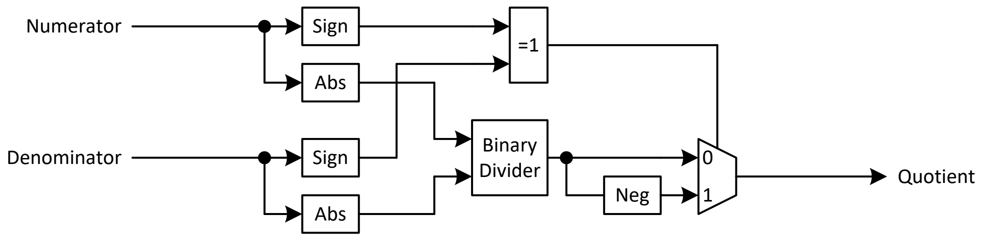


Figure 7: psi_fix_bin_div Architecture

3.2 psi_fix_cic_dec_fix_1ch

3.2.1 Description

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

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

3.2.2 Generics

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

3.2.3 Interfaces

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

3.2.4 Architecture

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

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

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

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

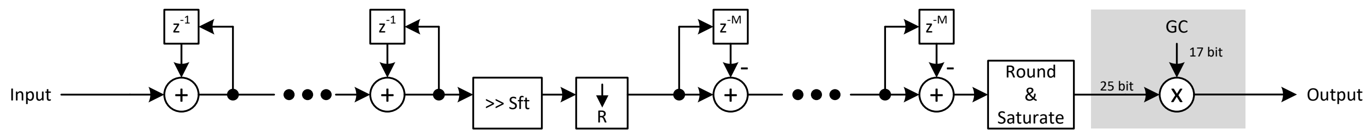


Figure 8: psi_fix_cic_dec_fix_1ch Architecture

The symbols are defined as follows:

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

Some of the most common formulas are given below.

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

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

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

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

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

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

3.3 psi_fix_cic_dec_fix_nch_par_tdm

3.3.1 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.

3.3.2 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

3.3.3 Interfaces

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

3.3.4 Architecture

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

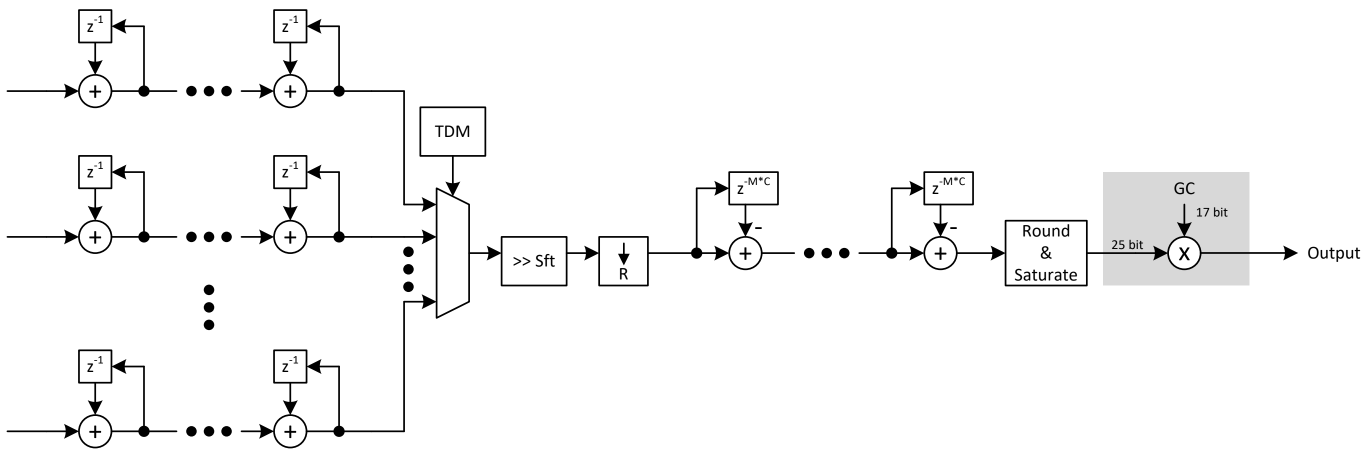


Figure 9: psi_fix_cic_dec_fix_nch_par_tdm Architecture

The symbols are defined as follows:

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

3.4 psi_fix_cic_dec_fix_nch_tdm_tdm

3.4.1 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.

3.4.2 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

3.4.3 Interfaces

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

3.4.4 Architecture

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

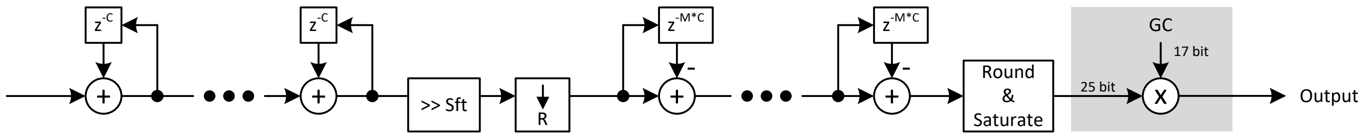


Figure 10: psi_fix_cic_dec_fix_nch_tdm_tdm Architecture

The symbols are defined as follows:

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

3.5 psi_fix_cic_int_fix_1ch

3.5.1 Description

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

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

3.5.2 Generics

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

3.5.3 Interfaces

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

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

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

3.5.4 Architecture

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

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

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

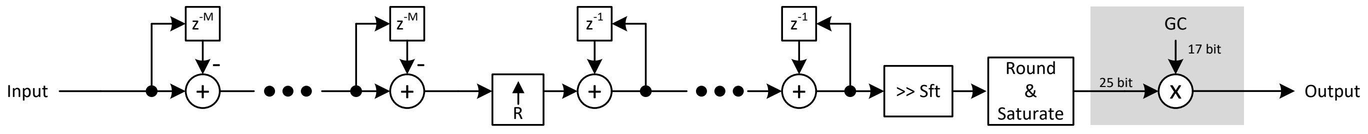


Figure 11: psi_fix_cic_int_fix_1ch Architecture

The symbols are defined as follows:

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

Some of the most common formulas are given below.

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

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

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

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

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

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

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

3.6 psi_fix_fir_dec_ser_nch_chpar_conf

3.6.1 Description

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

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

3.6.2 Generics

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

3.6.3 Interfaces

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

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

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

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

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

3.6.4 Architecture

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

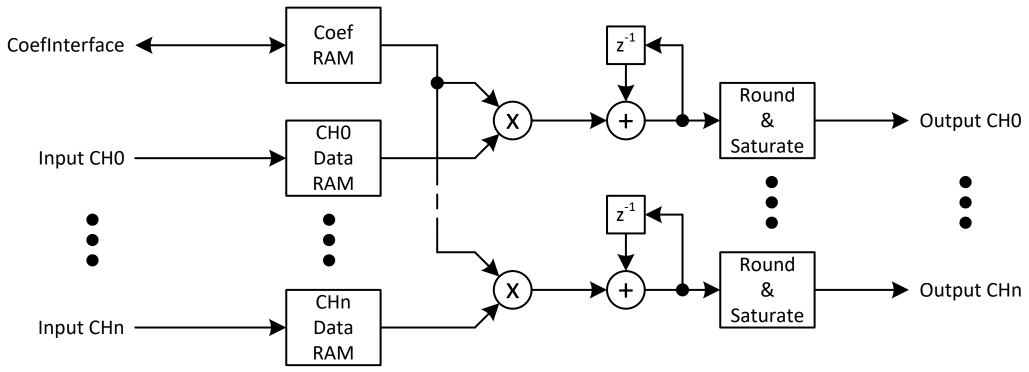


Figure 12: psi_fix_fix_dec_ser_nch_chpar_conf Architecture

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 ($Range_{Output} \geq 0.5 \cdot MaximumOutput$) and saturated externally.

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

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

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

3.7 psi_fix_fir_dec_ser_nch_chtdm_conf

3.7.1 Description

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

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

3.7.2 Generics

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

3.7.3 Interfaces

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

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

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

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

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

3.7.4 Architecture

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

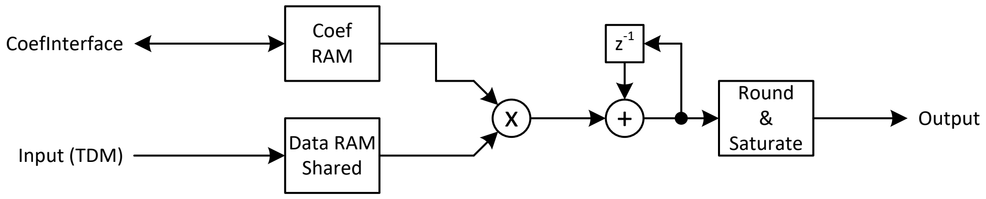


Figure 13: psi_fix_fix_dec_ser_nch_chtdm_conf Architecture

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 ($Range_{Output} \geq 0.5 \cdot MaximumOutput$) and saturated externally.

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

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

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

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

3.8 psi_fix_lin_approx_<function>

3.8.1 Description

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

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

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

3.8.2 Generics

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

3.8.3 Interfaces

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

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

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

3.8.4 Architecture

The figure below shows the interpolation principle.

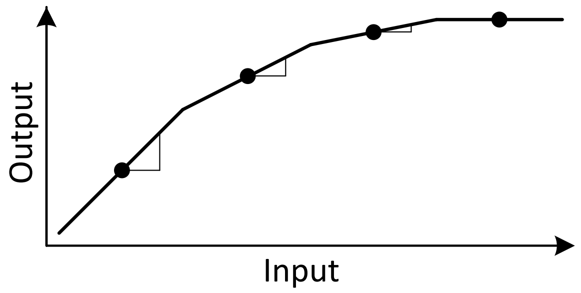


Figure 14: psi_fix_lin_approx Interpolation Principle

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.

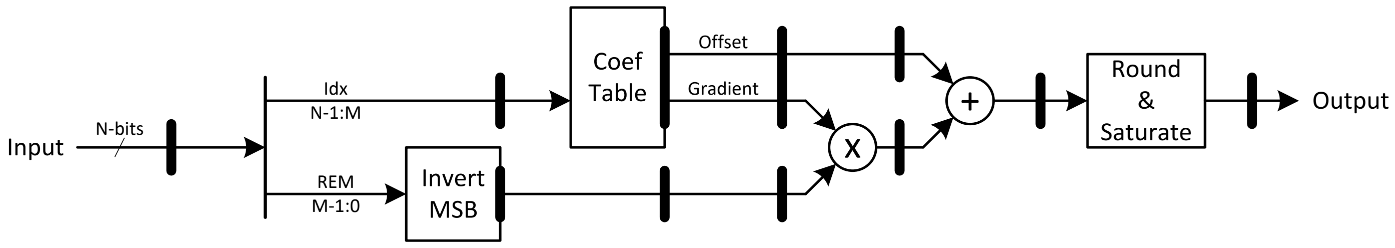


Figure 15: psi_fix_lin_approx Architecture

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.

3.8.5 Function Details

3.8.5.1 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.

3.8.5.2 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 $\sqrt{x} = \frac{1}{2^N}\sqrt{2^{2N}x}$.

3.9 psi_fix_dds_18b

3.9.1 Description

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

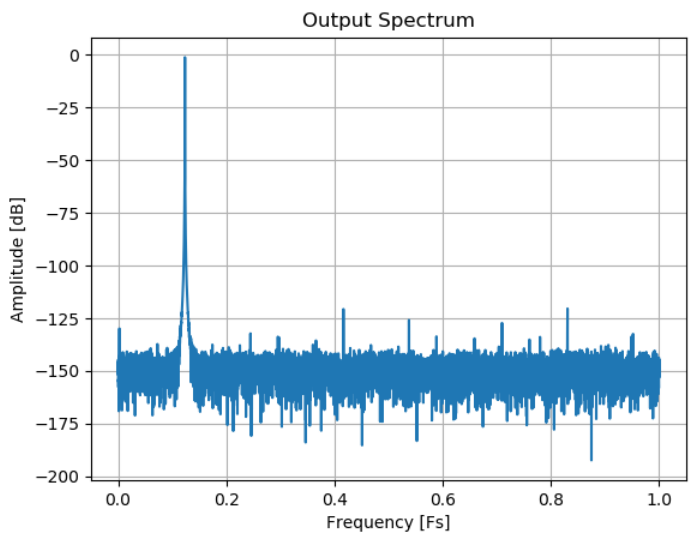


Figure 16: psi_fix_dds_18b Spectrum for PhaseStep=0.12345

3.9.2 Generics

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

3.9.3 Interfaces

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

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

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

3.9.4 Architecture

The figure below shows the implementation of the DDS.

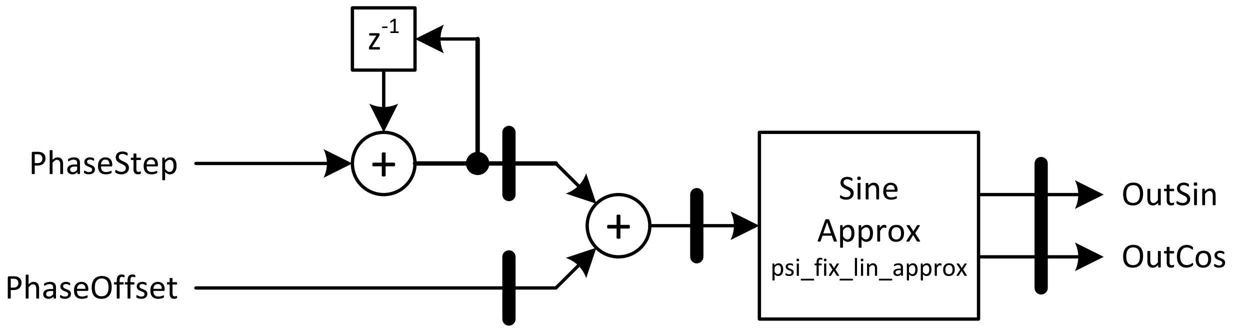


Figure 17: psi_fix_dds_18b Architecture

3.10 psi_fix_lowpass_iir_order1

3.10.1 Description

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

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

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

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

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

$$\beta = 1 - alpha$$

3.10.2 Generics

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

3.10.3 Interfaces

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

3.10.4 Architecture

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

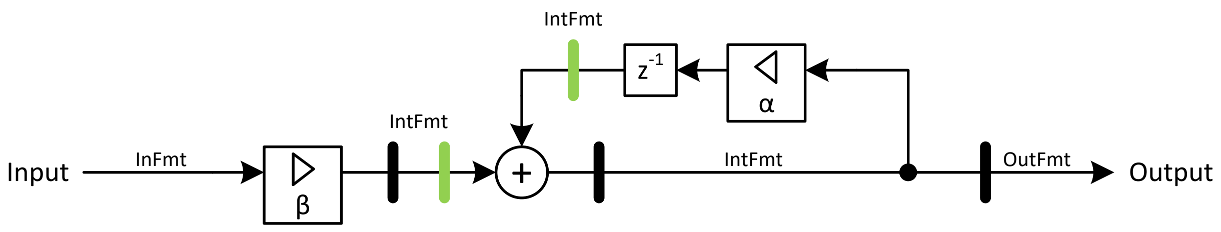


Figure 18: psi_fix_lowpass_iir_order1 Architecture

3.11 psi_fix_complex_mult

3.11.1 Description

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

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

The multiplication result comes:

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

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

3.11.2 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

3.11.3 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)

3.11.4 Architecture

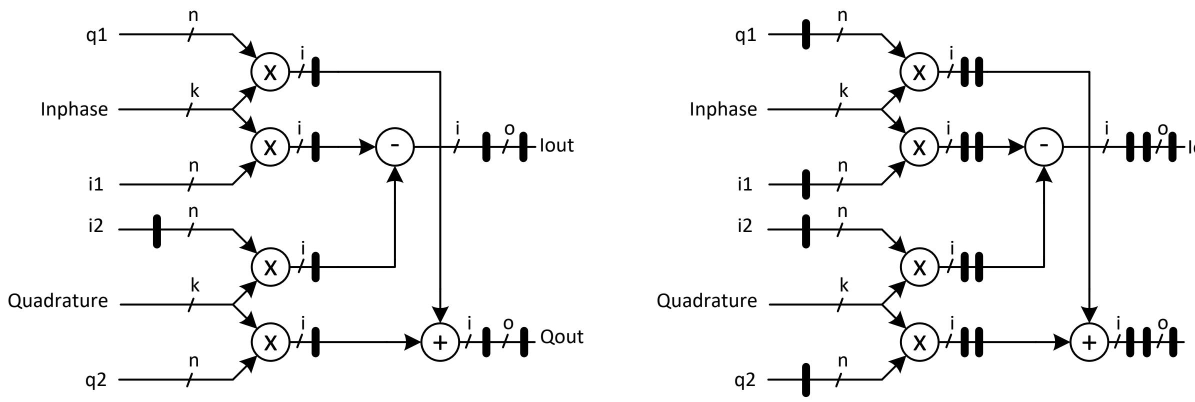


Figure 19: psi_fix_complex_mult Architecture – Pipeline_g = false (left) Pipeline_g = true

3.12 psi_fix_mov_avg

3.12.1 Description

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

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

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

$$G_{None} = Taps$$

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

$$G_{Exact} = 1.0$$

3.12.2 Generics

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

3.12.3 Interfaces

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

3.12.4 Architecture

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

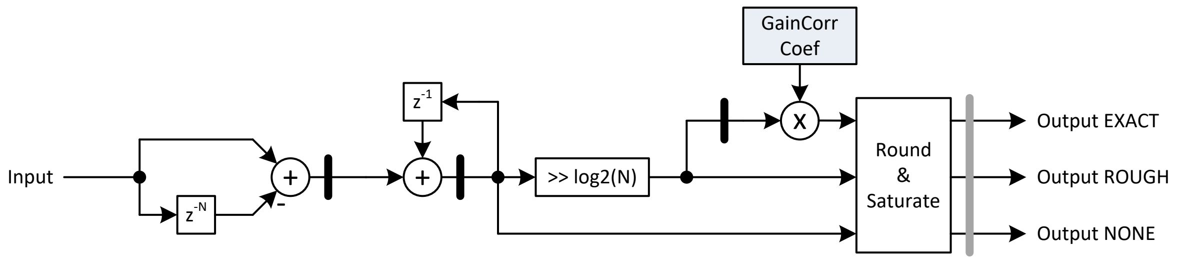


Figure 20: psi_fix_mov_avg Architecture

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.

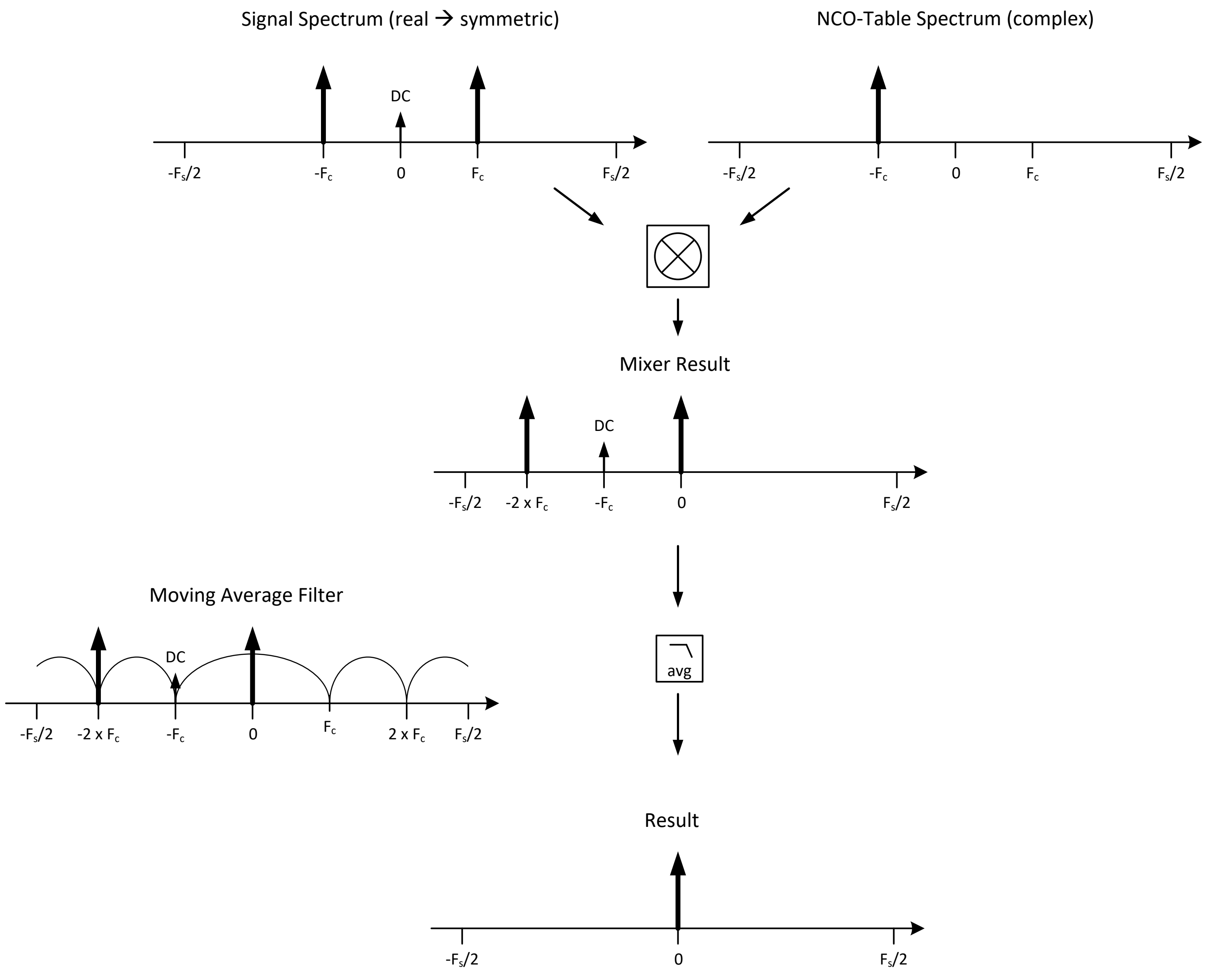
3.13 psi_fix_demod_real2cplx

3.13.1 Description

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

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

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



3.13.2 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

3.13.3 Interfaces

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

3.13.4 Architecture

The figure below shows the implementation of the demodulator.

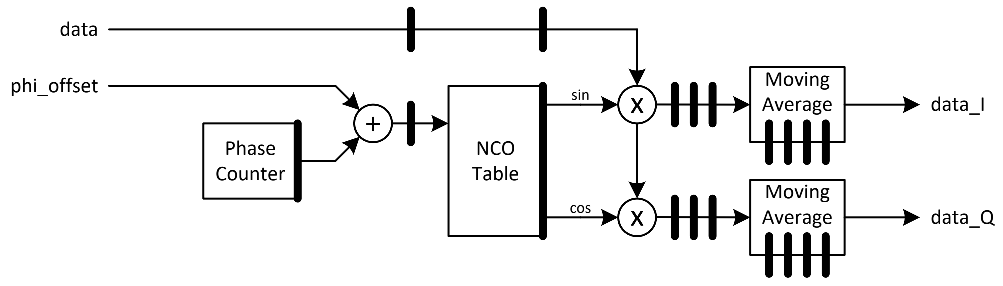


Figure 21: psi_fix_demod_real2cplx Architecture

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.

3.14 psi_fix_cordic_vect

3.14.1 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:

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

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).

3.14.2 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.

3.14.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Input			
InVld	Input	1	AXI-S handshaking signal
InRdy	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°)

3.14.4 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.

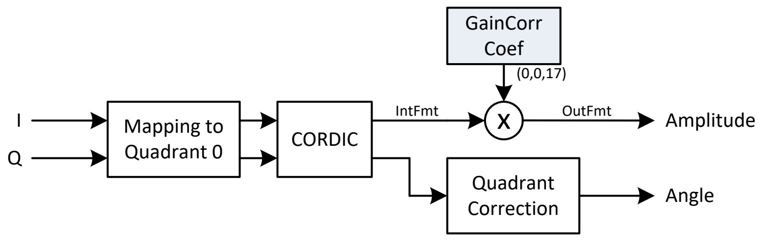


Figure 22: psi_fix_coric_vect Architecture

3.15 psi_fix_cordic_rot

3.15.1 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:

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

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 3.16) 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.

3.15.2 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

3.15.3 Interfaces

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

3.15.4 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.

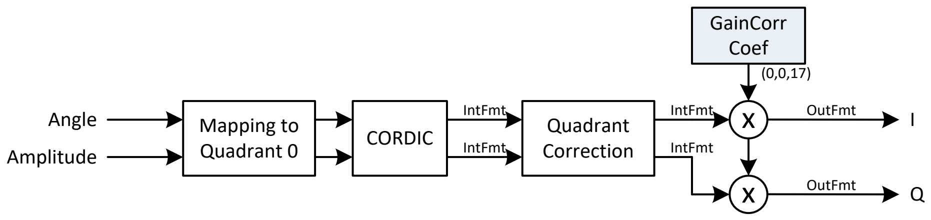


Figure 23: psi_fix_coric_rot Architecture

3.16 psi_fix_pol2cart_approx

3.16.1 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.

3.16.2 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)

3.16.3 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)

3.16.4 Architecture

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

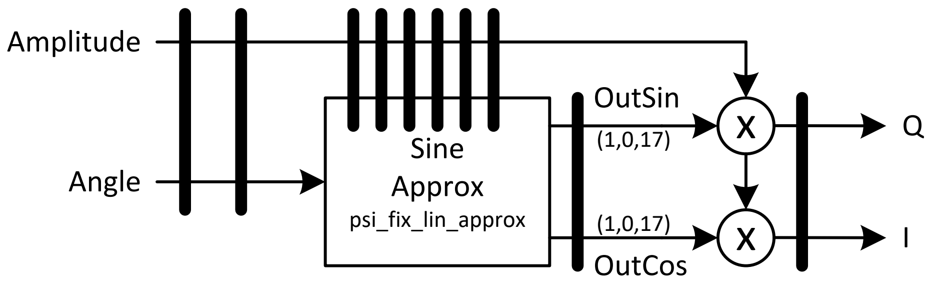


Figure 24: psi_fix_pol2cart_approx Architecture

3.17 psi_fix_mod_cplx2real

3.17.1 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:

$$x = I.sin(\omega t) + Q.\cos(\omega t)$$

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

$$\omega = \sum_{n=1}^{k} \frac{n}{k} \cdot 2\pi$$

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

RstPol_g	set the reset polarity
InpFmt_g	Fixed point Input format
CoefFmt_g	Fixed point Coefficient format
IntFmt_g	Fixed point Internal format (format the multiplication output is truncated to)
OutFmt_g	Fixed point Output format
Ratio_g	Frequency output ratio regarding clock frequency *(i.e. if Freq. clk 100MHz, ratio 5 => 20MHz)*

3.17.2 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)

3.17.3 Architecture

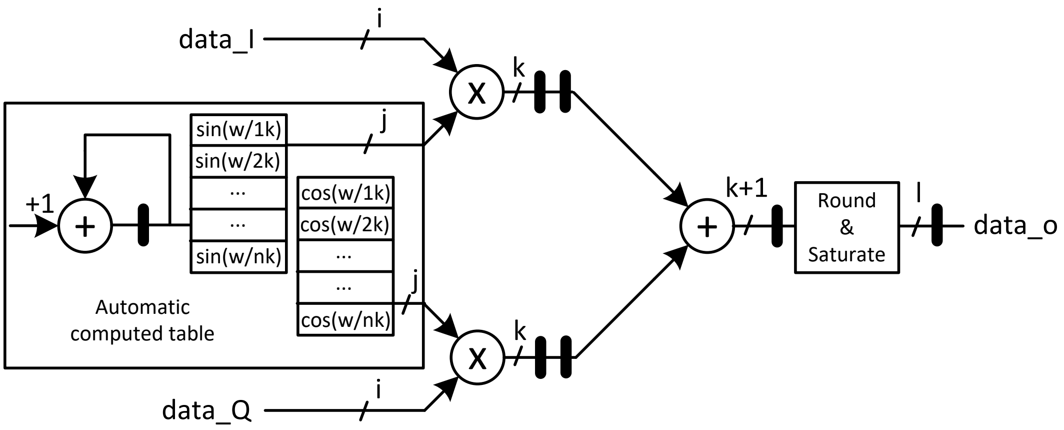


Figure 25: psi_fix_mod_cplx2real Archietcture

3.18 psi_fix_lut

3.18.1 Description

This is actually not an entity in the *hdl* directory but a python based code generator that generates lookup tables and corresponding simulation models. The name and location of the generated HDL file is given in python.

The LUT is implemented as synchronous ROM, so the read data is available one clock cycle after the read address is applied.

3.18.2 Generics

rst_pol_g	Reset polarity ('1' = reset is high-active)	
rom_style_g	Value passed to the attribute *rom_style* (Xilinx only)	
	"auto"	Tool choses resources
	"block"	Use block-RAM
	"distributed "	Use LUTs

3.18.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
clk_i	Input	1	Clock
rst_i	Input	1	Synchronous Reset
Data Interface			
radd_i	Input	*	LUT address input
rena_i	Input	1	Read enable
data_o	Input	*	LUT read data

3.19 psi_fix_pkg_writer

3.19.1 Description

This is actually not an entity in the *hdl* directory but a python based code generator that generates VHDL packages containing single value and array constants. This is very useful to pass values such as filter coefficients from python calculations into the VHDL implementation automatically.

There is no interface description for the package writer because it only generates a package containing constants and no entity.

4 Deprecated/Deleted Library Elements

This section contains a list of all library elements that were either removed or will be removed in near future. **Do not use them for new designs, even if they are not yet removed.**

For each deprecated library element a replacement strategy is described.

4.1 psi_fix_cordic_abs_pl

This library element is *DEPRECATED!* Do not use it for new designs, even if it is not yet deleted.

4.1.1 Replacement

The entity *psi_fix_cordic_abs_pl* can be replaced by *psi_fix_cordic_vect* with the angle output left unconnected. All logic related to the angle will get optimized away and the resource usage is roughly the same as for *psi_fix_cordic_abs_pl* according to test-routings.

The only thing that is not available in the new element is the generic *PipelineFactor_g* (implementing multiple iterations in one pipeline stage). However, this feature is regarded as useless.

4.1.2 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.

4.1.3 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)

4.1.4 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.

4.1.5 Architecture

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

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

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

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

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

The CORDIC gain can be calculated by the formula below:

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

Where:

G_{CORDIC}	Cordic Gain
N	Number of iterations

The formula converges towards 1.646760 with high numbers of iterations.

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