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 Usage of en_cl_fix provided by Enclustra GmbH

For all fixed-point calculations, the package *en_cl_fix* provided by Enclustra GmbH (www.enclustra.com) is used. The Enclustra packabe is wrapped by a package called *psi_fix* for historical reasons. In first versions, the implementations were independent but with version 2.0.0 the decision was taken to utilize the existing package from Enclustra instead of maintaining a separate one.

Note that conversion functions between the *en_cl_fix* and the *psi_fix* are provided in all languages, so library elements of both worlds can easily be mixed.

1.2 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*. The same applies to the fixed-point package *en_cl_fix* provided by Enclustra GmbH (but forked to the PSI GitHub account).

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.

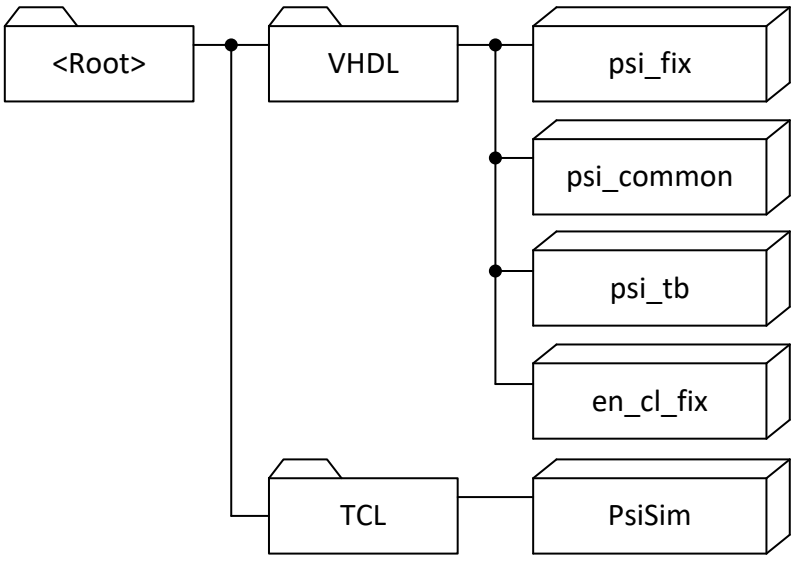


Figure 1: Working copy structure

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

1.3 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 command line 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

1.4 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*).

1.5 Running Simulations

1.5.1 Regression Test

1.5.1.1 Modelsim

To run the regression test, follow the steps below:

- Open Modelsim
- The TCL console, navigate to *<Root>/VHDL/psi_common/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.

1.5.1.2 GHDL

In order to run the regression tests using GHDL, GHDL must be installed and added to the path variable. Additionally a TCL interpreter must be installed.

To run the regression tests using GHDL, follow the steps below:

- Open the TCL interpreter (usually by running *tclsh*)
- The TCL console, navigate to *<Root>/VHDL/psi_common/sim*
- Execute the command "*source ./runGhdl.tcl*"

All test benches are executed automatically and at the end of the regression test, the result is reported

1.5.2 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_common/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>*

The steps for GHDL are the same, just in the TCL interpreter shall instead of the Modelsim TCL console.

1.6 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 the regression 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.

1.7 Handshaking Signals

1.7.1 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.

1.7.2 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.

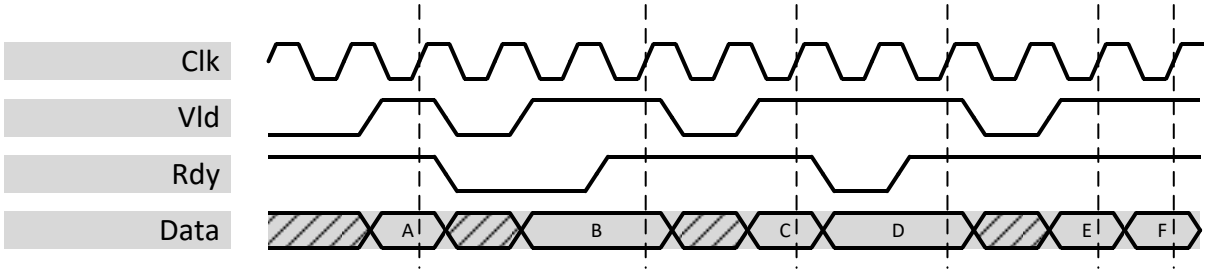


Figure 2: Handshaking signals

1.7.3 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 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 troubles:

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.

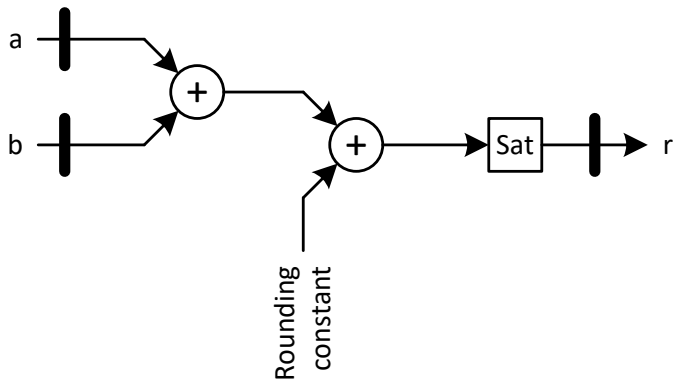


Figure 3: Heavy Pipelining, Problem Description

2.2.2 Solution 1: Register Retiming

Today's 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:

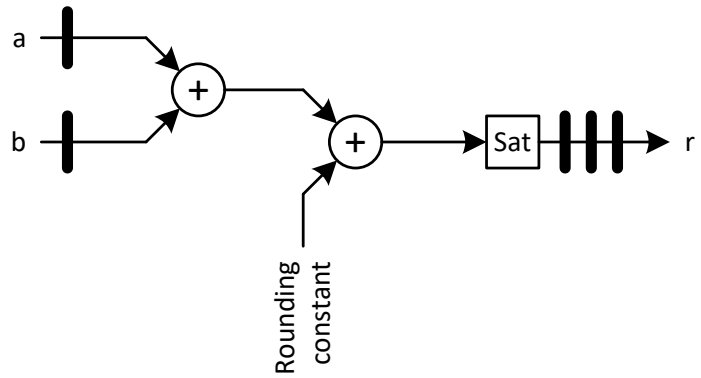


Figure 4: Heavy Pipelining, Retiming, Implementation without retiming

However, if 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.

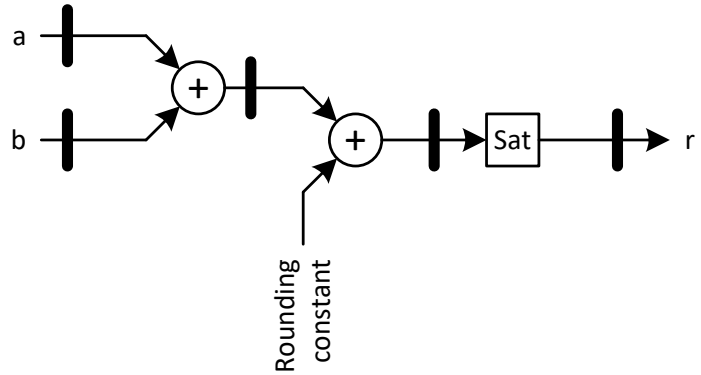


Figure 5: Heavy Pipelining, Retiming, Implementation with retiming

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 them 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.

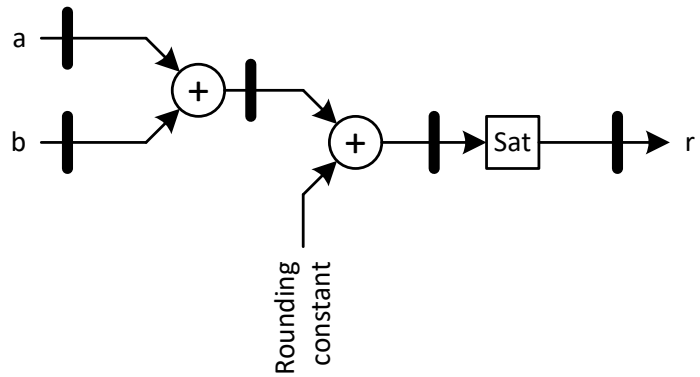


Figure 6: Heavy Pipelining, Manual Splitting

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

The disadvantage is that slightly more code is required.

Of 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.

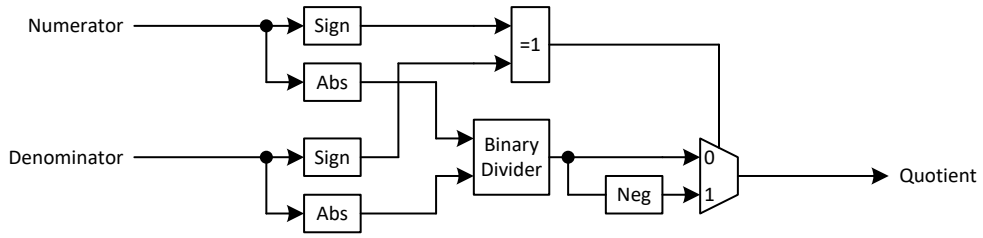


Figure 7: psi_fix_bin_div Architecture

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
Status interface			
CalcOngoing	Output	1	Asserted one clock cycle after *(InVld=1)*; deasserted one clock cycle after *(OutVld=1)* only when no further data is processed

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.

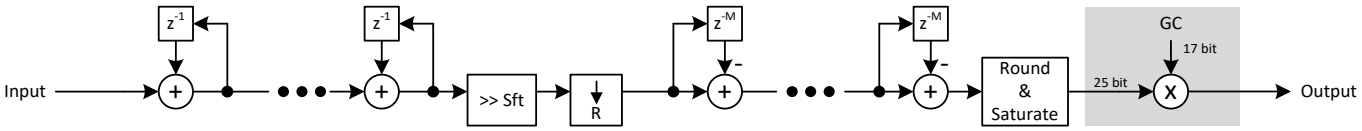


Figure 8: psi_fix_cic_dec_fix_1ch Architecture

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.

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

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

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

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_cfg_1ch

3.3.1 Description

This is the same CIC filter as *3.2 psi_fix_cic_dec_fix_1ch* but with the decimation ratio selectable at runtime. For general documentation, refer to the section given above. This section only describes the differences between the two filters.

3.3.2 Changed Generics

MaxRatio_g Maximum supported decimation ratio. Replaces the *Ratio_g* generic of the original filter.
AutoGainCorr_g True = Multiplier for exact gain compensation is implemented
False = compensation by shift only
The name of this generic is rather confusing but kept for reverse compatibility reasons.

3.3.3 Additional Interfaces

Signal	Direction	Width	Description
Configuration Interface (only modify while reset is asserted!)			
CfgRatio	Input	*ceil(log2(MaxRatio_g))*	Decimation ratio – 1 0 = no decimation, 1 = decimation by 2, etc.
CfgShift	Input	8	Number of bits to shift for gain compensation. Apply Sft from the formulas in 3.2.4
CfgGainCorr	Input	17	Gain correction factor in the format [0,1,16]. This port is only used if *AutoGainCorr_g = True.* Apply GC from the formulas in 3.2.4

3.3.4 Architecture

The static shift of the original filter is replaced by a pipelined dynamic shift with two stages. Otherwise, the architecture is unchanged.

3.4 psi_fix_cic_dec_fix_nch_par_tdm

3.4.1 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.

3.4.2 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*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, …
Status interface			
CalcOngoing	Output	1	Asserted one clock cycle after *(InVld=1)*; deasserted one clock cycle after *(OutVld=1)* only when no further data is processed

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.

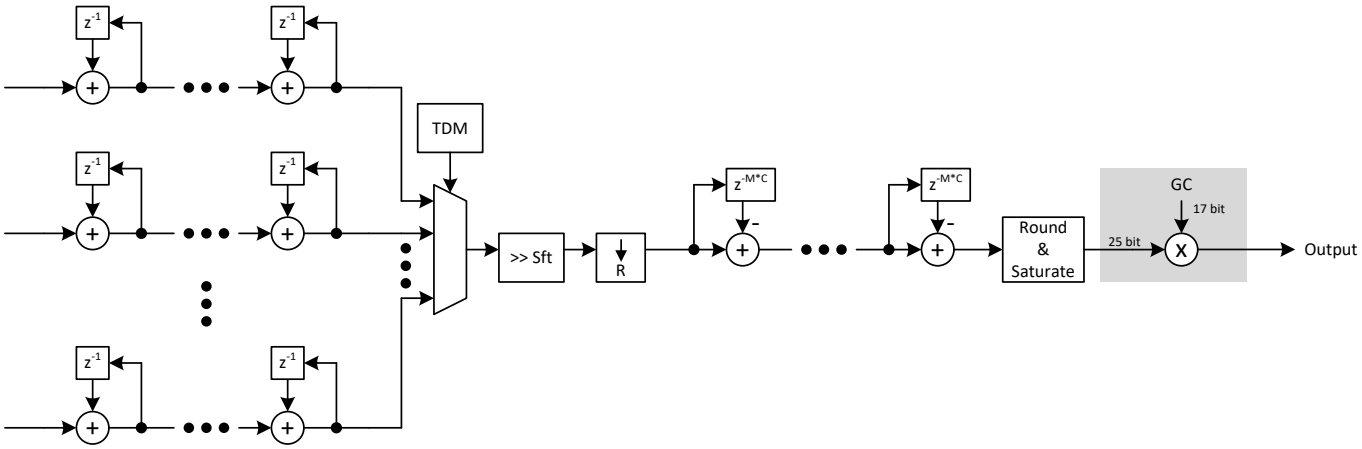


Figure 9: psi_fix_cic_dec_fix_nch_par_tdm Architecture

3.5 psi_fix_cic_dec_cfg_nch_par_tdm

3.5.1 Description

This is the same CIC filter as *3.4 psi_fix_cic_dec_fix_nch_par_tdm* but with the decimation ratio selectable at runtime. For general documentation, refer to the section given above. This section only describes the differences between the two filters.

3.5.2 Changed Generics

MaxRatio_g Maximum supported decimation ratio. Replaces the *Ratio_g* generic of the original filter.
AutoGainCorr_g True = Multiplier for exact gain compensation is implemented
False = compensation by shift only
The name of this generic is rather confusing but kept for reverse compatibility reasons.

3.5.3 Additional Interfaces

Signal	Direction	Width	Description
Configuration Interface (only modify while reset is asserted!)			
CfgRatio	Input	*ceil(log2(MaxRatio_g))*	Decimation ratio – 1 0 = no decimation, 1 = decimation by 2, etc.
CfgShift	Input	8	Number of bits to shift for gain compensation. Apply Sft from the formulas in 3.2.4
CfgGainCorr	Input	17	Gain correction factor in the format [0,1,16]. This port is only used if *AutoGainCorr_g = True.* Apply GC from the formulas in 3.2.4

3.5.4 Architecture

The static shift of the original filter is replaced by a pipelined dynamic shift with two stages. Otherwise, the architecture is unchanged.

3.6 psi_fix_cic_dec_fix_nch_tdm_tdm

3.6.1 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.

3.6.2 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

3.6.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Input			
InVld	Input	1	AXI-S handshaking signal
InData	Input	*InFmt_g*	Input data 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, …
Status interface			
CalcOngoing	Output	1	Asserted one clock cycle after *(InVld=1)*; deasserted one clock cycle after *(OutVld=1)* only when no further data is processed

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

3.7 psi_fix_cic_dec_cfg_nch_tdm_tdm

3.7.1 Description

This is the same CIC filter as *3.6 psi_fix_cic_dec_fix_nch_tdm_tdm* but with the decimation ratio selectable at runtime. For general documentation, refer to the section given above. This section only describes the differences between the two filters.

3.7.2 Changed Generics

MaxRatio_g Maximum supported decimation ratio. Replaces the *Ratio_g* generic of the original filter.
AutoGainCorr_g True = Multiplier for exact gain compensation is implemented
False = compensation by shift only
The name of this generic is rather confusing but kept for reverse compatibility reasons.

3.7.3 Additional Interfaces

Signal	Direction	Width	Description
Configuration Interface (only modify while reset is asserted!)			
CfgRatio	Input	*ceil(log2(MaxRatio_g))*	Decimation ratio – 1 0 = no decimation, 1 = decimation by 2, etc.
CfgShift	Input	8	Number of bits to shift for gain compensation. Apply Sft from the formulas in 3.2.4
CfgGainCorr	Input	17	Gain correction factor in the format [0,1,16]. This port is only used if *AutoGainCorr_g = True.* Apply GC from the formulas in 3.2.4

3.7.4 Architecture

The static shift of the original filter is replaced by a pipelined dynamic shift with two stages. Otherwise, the architecture is unchanged.

3.7.5 Architecture

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

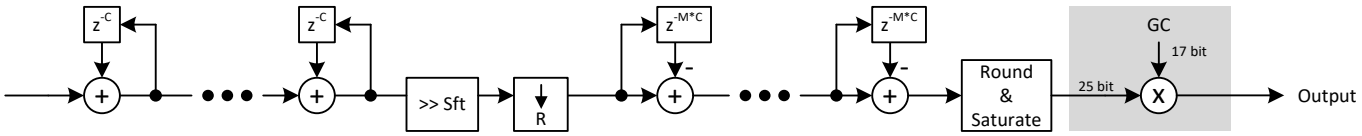


Figure 10: psi_fix_cic_dec_fix_nch_tdm_tdm Architecture

3.8 psi_fix_cic_int_fix_1ch

3.8.1 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.

3.8.2 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

3.8.3 Interfaces

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

3.8.4 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.

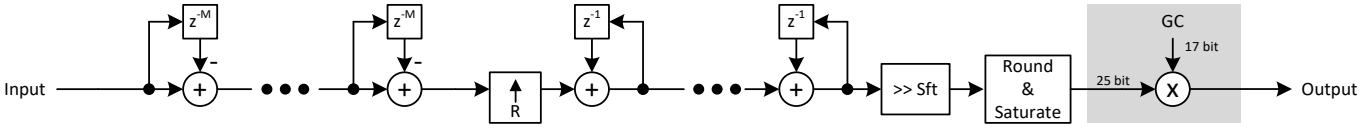


Figure 11: psi_fix_cic_int_fix_1ch Architecture

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.

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

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

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.

3.9 psi_fix_fir_3tap_hbw_dec2

3.9.1 Description

This component implements a decimating by 2 half-bandwidth filter. This particular implementation has 3 taps with fixed coefficients: 0.25, 0.5, 0.25. This enables efficient implementation based on bit shifting instead of multiplications. It can be used in a two modes of operation: Separate mode, in which all channels are independently processed (but still share AXI-S handshaking signals) and a second mode, in which input samples are processed as they come from one source. This enables decimate by N (where N is a power of two) by connecting more components of this type in a chained structure.

3.9.2 Generics

InFmt_g	Input format
OutFmt_g	Output format
Separate_g	Separate mode (when **true**, each pair of inputs is treated as from separate source)
Channels_g	Number of parallel channels (must be power of two when **Separate_g = false**)
Rnd_g	Rounding mode at the output (round or truncate)
Sat_g	Saturation mode at the output (saturate or wrap)

3.9.3 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	2*Channels_g*InFmt_g	Input data in parallel **Separete_g = true**, two samples for each channel: - Channel A Sample 0 [N-1:0] - Channel A Sample 1 [2*N-1:0] - Channel B Sample 0 [3*N-1:0] - Channel B Sample 1 [4*N-1:0] … **Separete_g = false**, one channel - Channel A Sample 0 [N-1:0] - Channel A Sample 1 [2*N-1:0] - Channel A Sample 2 [3*N-1:0] …
Output			
OutVld	Output	1	AXI-S handshaking signal
OutAbs	Output	Channels_g*OutFmt_g	Output data, one sample for each pair of inputs **Separete_g = true** - Channel A Sample 0 [N-1:0] - Channel B Sample 0 [2*N-1:0] … **Separete_g = false** - Channel A Sample 0 [N-1:0] - Channel A Sample 1 [2*N-1:0] …

3.9.4 Architecture

The figure below shows a base structure that is used in all filter configurations. The structure consists of three binary shifts (acting as a multiplication by constant coefficient) followed by two adders.

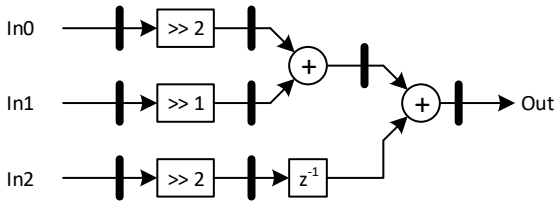


Figure 12: psi_fix_fir_3tap_hbw_dec base structure

Based on this structure, two different filter configurations can be build. When **Separate_g = true**, all channels run in parallel, although AXI-S handshaking signals are still shared. The figure below shows the example with **Channels_g = 2**. Here four samples for two channels A and B are processed in parallel that results in two decimated output samples, one for each channel.

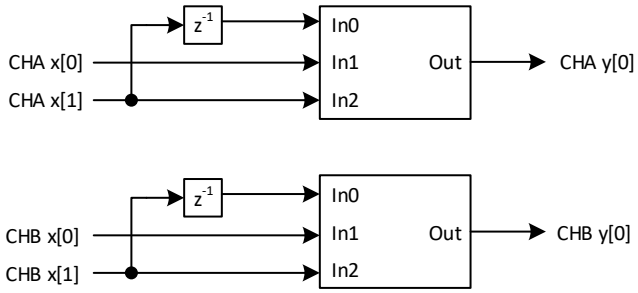


Figure 13: psi_fix_fir_3tap_hbw_dec
Separate_g = true, Channels_g = 2

When **Separate_g = false** the component processes samples from only one channel. The following figure displays example with **Channels_g = 2**. The filter takes 4 consecutive input samples from one source and decimates to two samples. It is now possible to use second filter that will further decimate from two to one sample.

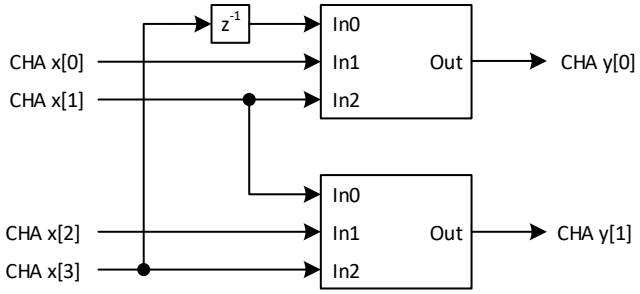


Figure 14: psi_fix_fir_3tap_hbw_dec
Separate_g = false, Channels_g = 2

3.10 psi_fix_mult_add_stage

3.10.1 Description

This entity implements a multiply-add stage that can be implemented efficiently for FPGAs supporting multiply-adder chains (as Xilinx 6- and 7-Series do for example). It is written in pure VHDL and also synthesizable for other FPGA families but it may lead to less optimal results.

This multiply-add stage is used for implementing parallel or semi-parallel filters efficiently.

3.10.2 Generics

InAFmt_g Input A format
InBFmt_g Input B format
AddFmt_g Adder chain format
InBIsCoef_g If True, *InBVld* is only used to write a constant coefficient into the input register of the DSP Slice. If False, *InBVld* leads to a multiply-add operation and is propagated to *AddChainOutVld*

3.10.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Input			
InAVld	Input	1	*InA* contains valid data. This always leads to the multiply-add being executed.
InA	Input	$InAFmt\_g$	Input data
InADel2	Output	$InAFmt\_g$	*InA* delayed by two clock cycles (this is useful for efficient pipelined FIR implementation)
InBVld	Input	1	*InB* contains valid data. If input B register is used as coefficient storage (see *InBIsCoef_g*), this does not lead to a multiply-add being executed. Otherwise this port shall be connected to the same signal as *InAVld*.
InB	Input	$InBFmt\_g$	Input data
InBDel2	output	$InBFmt\_g$	*InB* delayed by two clock cycles (this is useful for efficient pipelined FIR implementation)
Adder Chain			
AddChainIn	Input	$AddFmt\_g$	Adder chain input from the last slice. For the first slice in a chain, connect to zero.
AddChainOut	Output	$AddFmt\_g$	Adder chain output. Connect to next slice in chain or use as output.
AddChainOutVld	Output	1	Signals when AddChainOut is valid (based on *InAVld*).

3.10.4 Architecture

The figure below shows the architecture of the *psi_fix_mult_add_stage*.

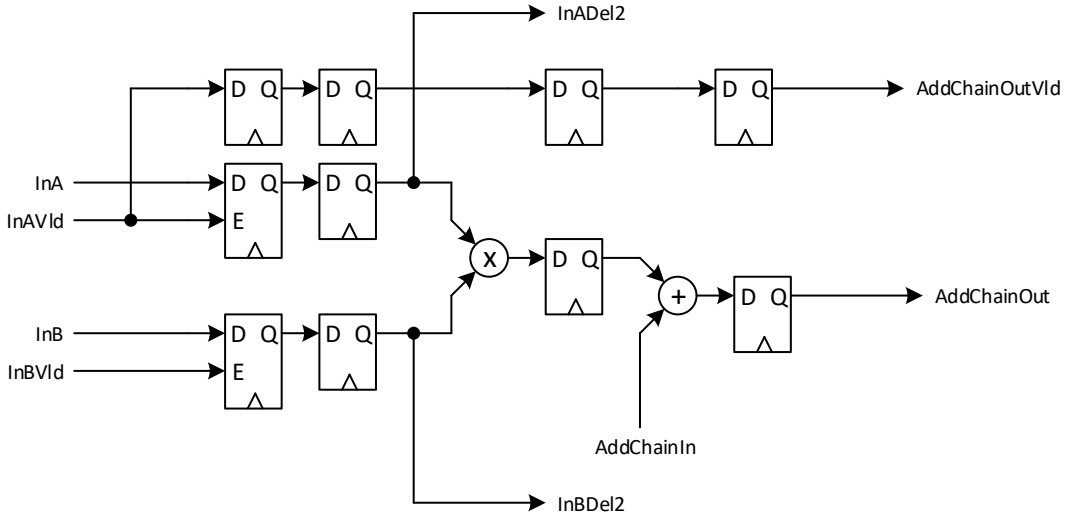


Figure 15: psi_fix_mult_add_stage Architecture

The example below shows how to use the multiply-add stage for an efficient fully-parallel single channel FIR implementation with three taps.

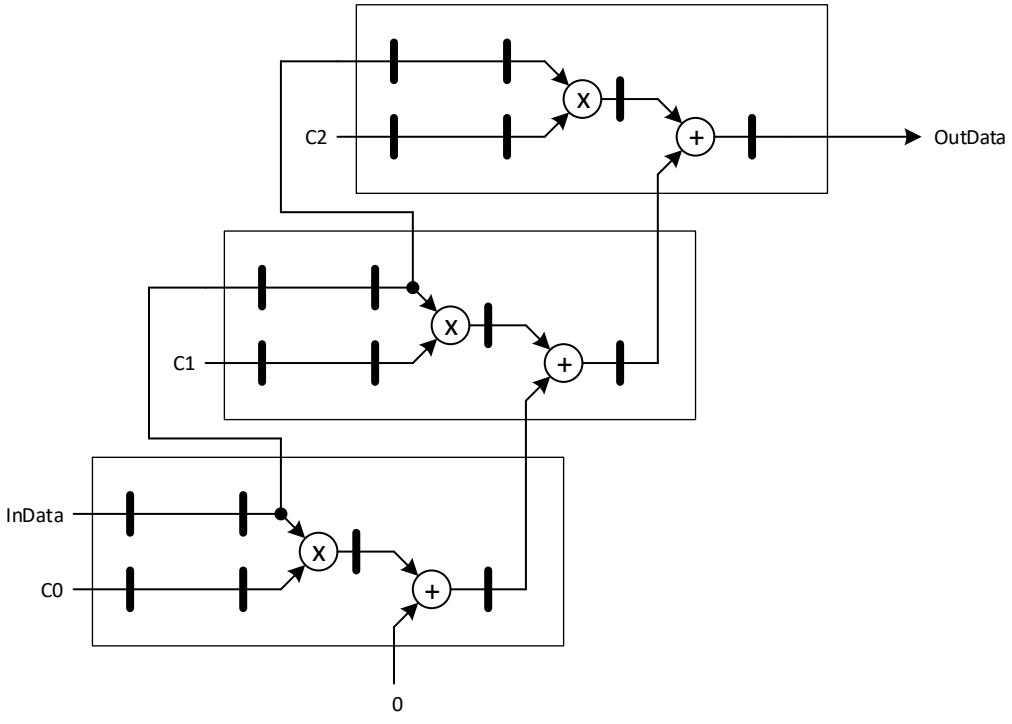


Figure 16: psi_fix_mult_add_stage Usage Example

3.11 psi_fix_fir_dec_ser_nch_chpar_conf

3.11.1 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.

3.11.2 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.
Coefs_g Initial value for coefficients

3.11.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Input			
InVld	Input	1	AXI-S handshaking signal
InData	Input	$InFmt\_g \cdot Channels\_g$	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	$ceil\left(\log_2(\mathrm{MaxRatio\_g})\right)$	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)
Status interface			
CalcOngoing	Output	1	Asserted one clock cycle after *(InVld=1)*; deasserted one clock cycle after *(OutVld=1)* only when no further data is processed

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 backpressure 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 handled is exceeded.

3.11.4 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.

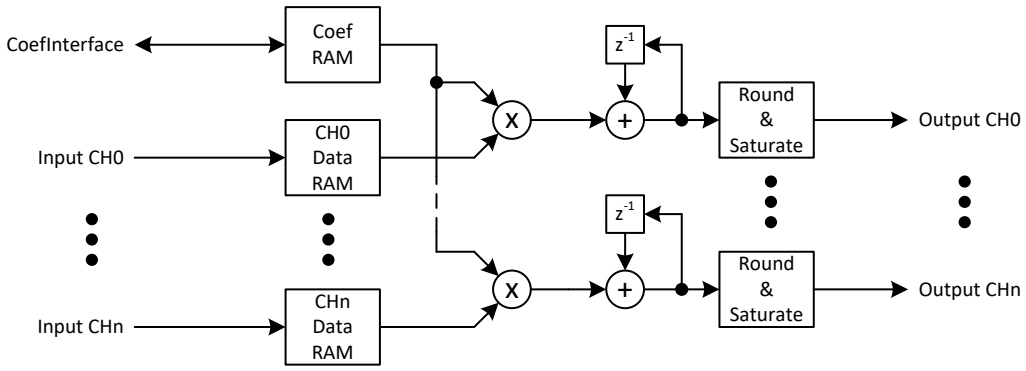


Figure 17: psi_fix_fix_dec_ser_nch_chpar_conf Architecture

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 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.

3.12 psi_fix_fir_dec_ser_nch_chtdm_conf

3.12.1 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.

3.12.2 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.
Coefs_g Initial value for coefficients

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	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	$ceil\left(\log_2(\mathrm{MaxRatio\_g})\right)$	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)
Status interface			
CalcOngoing	Output	1	Asserted one clock cycle after *(InVld=1)*; deasserted one clock cycle after *(OutVld=1)* only when no further data is processed

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 backpressure 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 handled is exceeded.

3.12.4 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).

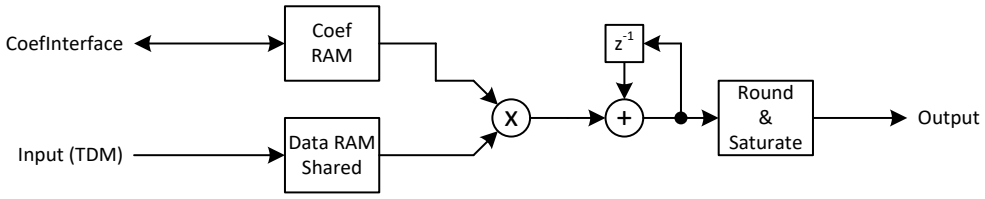


Figure 18: psi_fix_fix_dec_ser_nch_chtdm_conf Architecture

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.

3.13 psi_fix_fir_par_nch_chtdm_conf

3.13.1 Description

This entity implements a fully parallel FIR filter (without decimation). It can process one channel or more channels in TDM but for all channels the same coefficients are used. Coefficients can either be configured at runtime or statically via generics.

3.13.2 Generics

InFmt_g Input format
OutFmt_g Output format
CoefFmt_g Coefficient format
Channels_g Number of parallel channels
Taps_g Number of taps implemented
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.
Coefs_g Initial value for coefficients

3.13.3 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
Coefficient Interface			
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)

3.13.4 Architecture

The FIR filter is based on a chain of *psi_fix_mult_add_stage* stages. For the single channel implementation they are connected directly, for the multi-channel implementation, appropriate delays are placed in between them.

The filter is optimized for most efficient implementation at high clock speed since fully parallel filters are only used at the very input of a signal processing chain where clock-speed is highest.

The figure below shows an example with three taps.

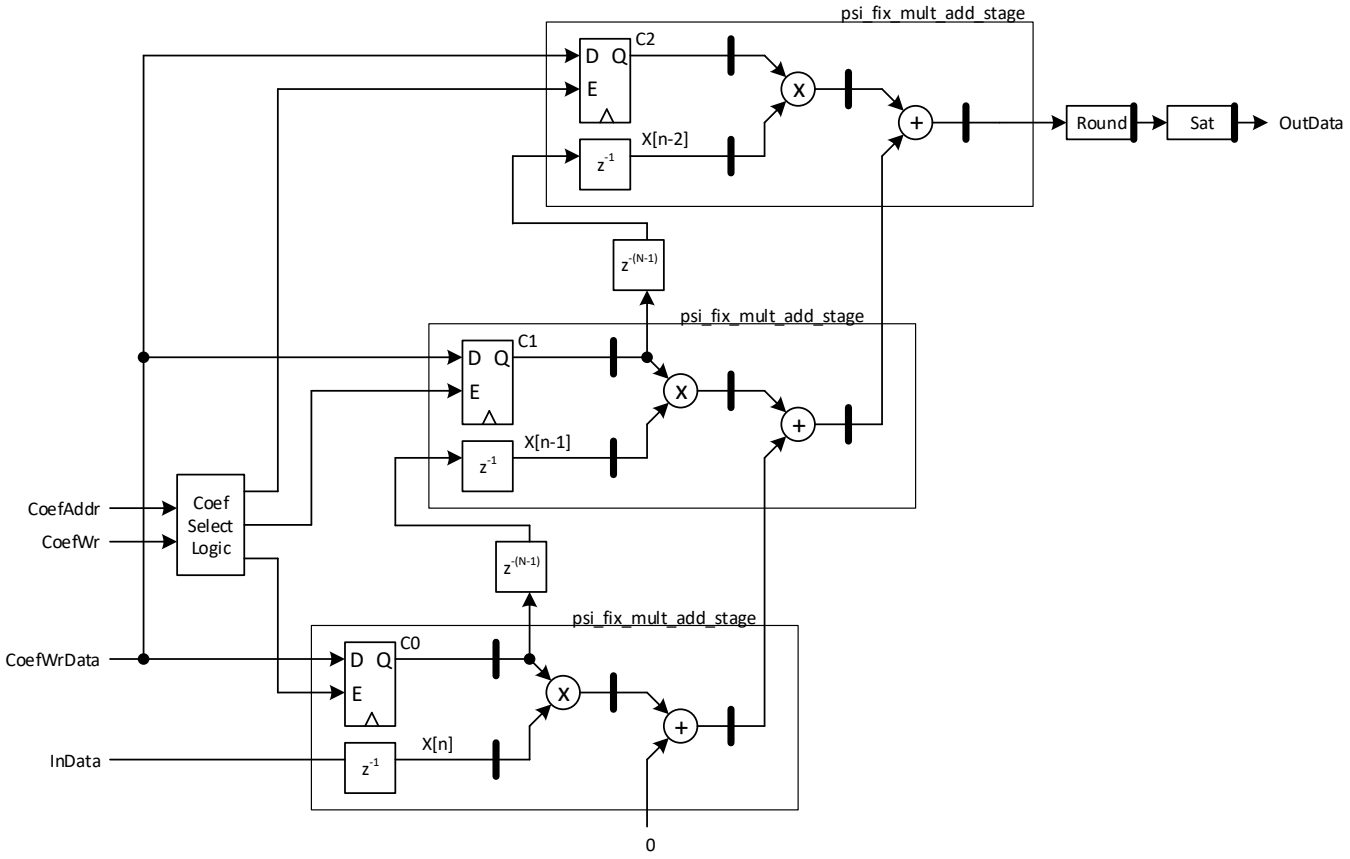


Figure 19: psi_fix_fir_par_nch_chtdm_conf Architecture

3.14 psi_fix_fir_dec_semi_nch_chtdm_conf

3.14.1 Description

This entity implements a semi-parallel FIR filter for multiple channels (TDM). A configurable number of multiplier is working in parallel. The number of cycles to calculate one output sample depends on the filter order and the number of multipliers chosen. As a result, this filter architecture allows a wide range of trade-offs between resource consumption and processing power.

This filter does only implement the *VLD* signal of the AXI-S handshaking. Due to the absence of back-pressure handling using the *RDY* signal, the user is responsible for making sure that the processing power of the filter is sufficient to process all samples. Violation of this rule may lead to undefined behavior. However, in Simulations an error (VHDL-assertion) is thrown if this situation occurs.

The processing power (in multiplications) required can be calculated by the formula below:

$$MultPerSec_{Required} = \frac{SR_{In} \cdot \text{Channels\_g} \cdot \text{Taps\_g}}{\text{Ratio\_g}}$$

Where:

SR_{In}: Input sample rate in SPS

The processing power available can be calculated by the formula below:

$$MultPerSec_{Available} = F_{clk} \cdot \text{Multipliers\_g}$$

To satisfy the rules, the following condition must be true:

$$MultPerSec_{Required} \geq MultPerSec_{Available}$$

$$\frac{SR_{In} \cdot \text{Channels\_g} \cdot \text{Taps\_g}}{\text{Ratio\_g}} \geq F_{clk} \cdot \text{Multipliers\_g}$$

From this, the required setting of *Multipliers_g* can be derived:

$$\text{Multipliers\_g} \geq \frac{SR_{In} \cdot \text{Channels\_g} \cdot \text{Taps\_g}}{\text{Ratio\_g} \cdot F_{clk}}$$

The filter can be implemented in two slightly different ways. One is more efficient in terms of memory required for the delay chain but it does not allow the *InVld* to be asserted in two consecutive cycles. As a result, the input sample rate is only half of the theoretical limit. The other architecture consumes more memory but allows *InVld* to be high all the time. The way the filter is implemented is controlled by the generic *FullInpRateSupport_g*.

Because the filter uses RAM blocks, old data may be left in the filter after a reset. Therefore, optionally a data flushing functionality can be implemented. This functionality allows overwriting all memories with zero after a reset.

3.14.2 Generics

InFmt_g	Input format
OutFmt_g	Output format
CoefFmt_g	Coefficient format
Channels_g	Number of parallel channels
Multipliers_g	Number of multipliers to use in parallel
Ratio_g	Decimation ratio
Taps_g	Number of taps implemented (Taps = Order+1)
Rnd_g	Rounding mode at the output (round or truncate)
Sat_g	Saturation mode at the output (saturate of wrap)
FullInpRateSupport_g	True = *InVld* can be high all the time

	Fals = *InVld* cannot be asserted in two consecutive clock cycles
UseFixCoefs_g	If true, fixed coefficients instead of configurable coefficients are implemented.
RamBehavior_g	"RBW" (read before write) or "WBR" (write before read), depending on the technology used
Coefs_g	Initial value for coefficients
ImplFlushIf_g	Implement memory flushing interface

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
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
Coefficient Interface			
CoefWr	Input	1	Coefficient write enable signal This port can be left unconnected for fixed coefficient implementation (*UseFixCoefs_g* = true)
CoefAddr	Input	$ceil\left(\log_2(\text{Taps\_g})\right)$	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)
Flushing Interface			
FlushMem	Input	1	Inject a pulse to flush all data memories (usually done after reset). This port can be left unconnected if the flushing interface is not implemented (*ImplFlushIf_g* = false).
FlushDone	Output	1	A pulse on this port indicates that a flush started by *FlushMem = '1'* was completed. Never use the filter while a flush is ongoing (i.e. after *FlushMem* = '1' but before *FlushDone* = '1').
Status interface			
CalcOngoing	Output	1	Asserted one clock cycle after *(InVld=1)*; deasserted one clock cycle after *(OutVld=1)* only when no further data is processed

3.14.4 Architecture

The filter general consists of a number of multiplier stages that all look the same. Each stage contains the multiply-add unit, a small RAM for the delay chain data and a small RAM or ROM for the coefficients (depending on *UseFixCoefs_g*).

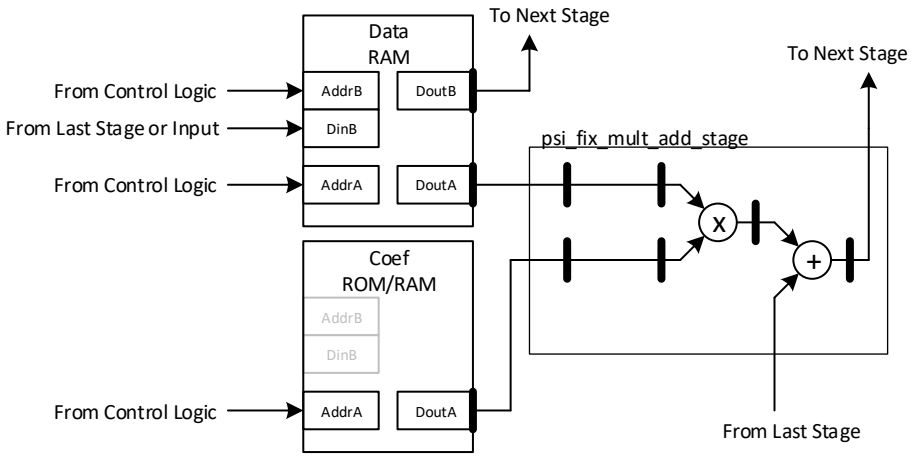


Figure 20: psi_fix_fir_dec_semi_nch_chtdm_conf Multiplier stage

These stages are then connected to a chain. Because multiple clock cycles are used for one calculation, all the partial results must be accumulated. At the very output the result is rounded.

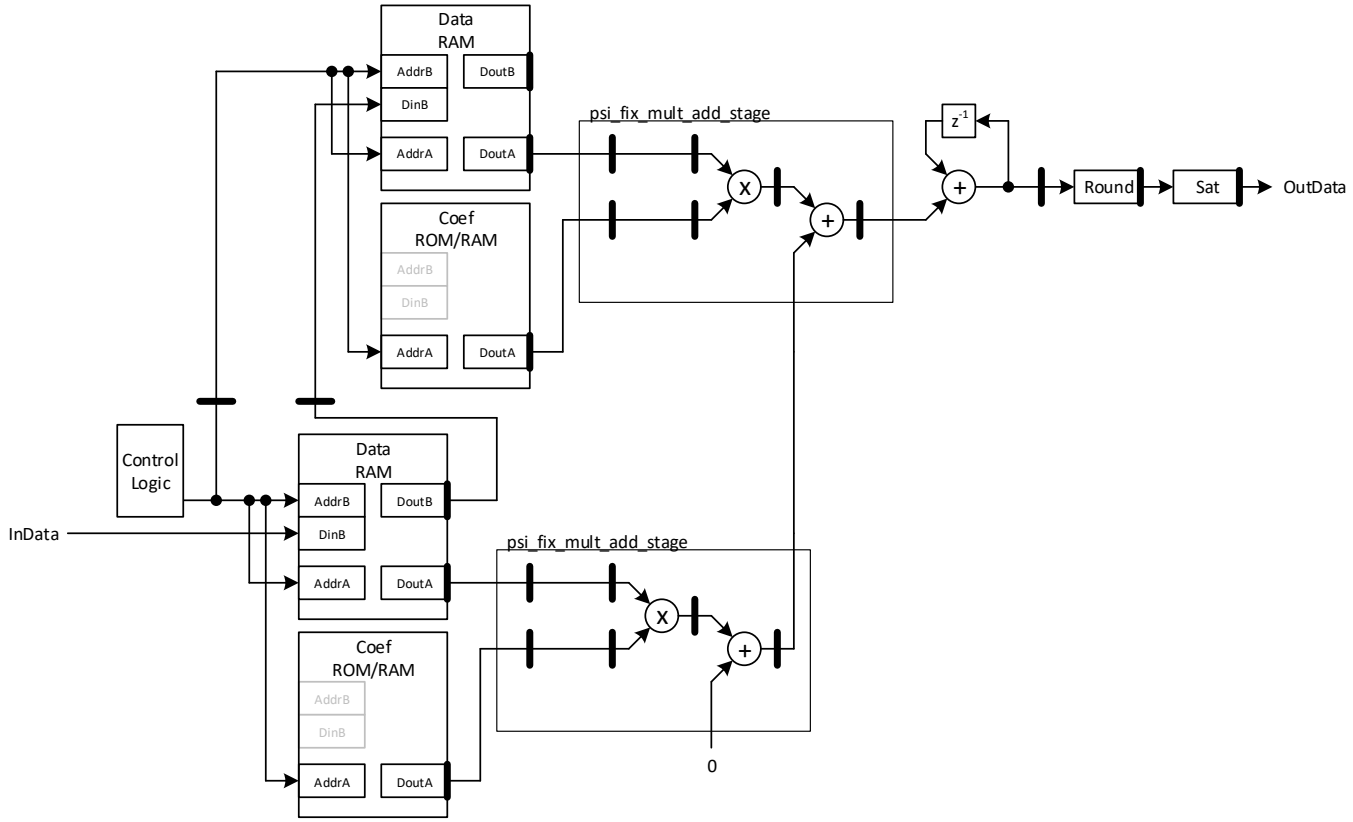


Figure 21: psi_fix_fir_dec_semi_nch_chtdm_conf Architecture

The figures above show the architecture for *FullInpRateSupport_g = False*. In this case, the access to the data RAM happens in two consecutive clock cycles. In the first cycle, the data to be sent to the next stage is read. In the next clock cycle the new data entry for this stage is written. This is required because the read and write address are not the same.

This may sound a bit awkward as for single channel FIRs usually the data in the cell that is written must be forwarded to the next stage and the access could be done in one clock cycle because write/read is done to the same address. However, for a multi-channel TDM filter, data must be kept in the RAM of the same stage even if new samples are arriving during the calculation of a new output sample being executed, which can take some clock cycles because not only one filter is calculated but one per channel. As a result, the RAM has some headroom.

As an example, let's assume an FIR filter that requires 8 samples per stage for the calculation of the output. Because more input samples must be stored in the RAM, it is larger than 8 entries. Let's say 10 entries. When the 9th sample is written (address 8) the sample at address 0 must be forwarded to the next stage. These addresses are different, so the accesses cannot be merged into one clock cycle.

As a result, a slightly different architecture is required for *FullInpRateSupport_g* = *True*. In this case, the delay for forwarding the data to the next stage is realized with a separate delay chain. The RAM is only used to keep the data available for calculations.

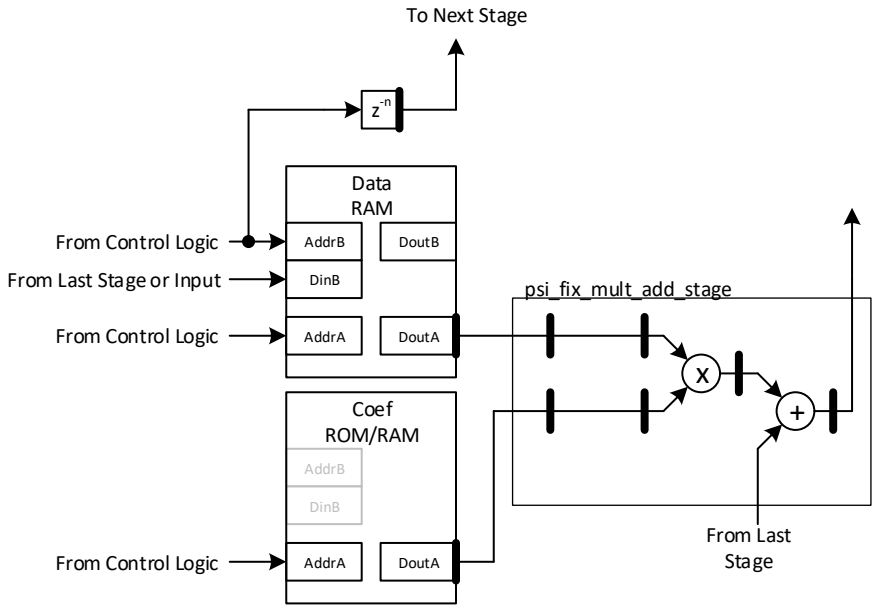


Figure 22: psi_fix_fir_dec_semi_nch_chtdm_conf Multiplier stage for full input rate

3.15 psi_fix_lin_approx_<function>

3.15.1 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()*.

3.15.2 Generics

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

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
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.

3.15.4 Architecture

The figure below shows the interpolation principle.

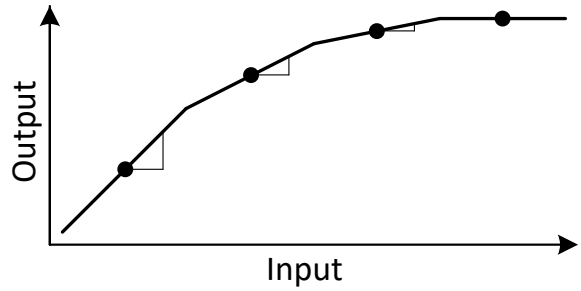


Figure 23: psi_fix_lin_approx Interpolation Principle

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.

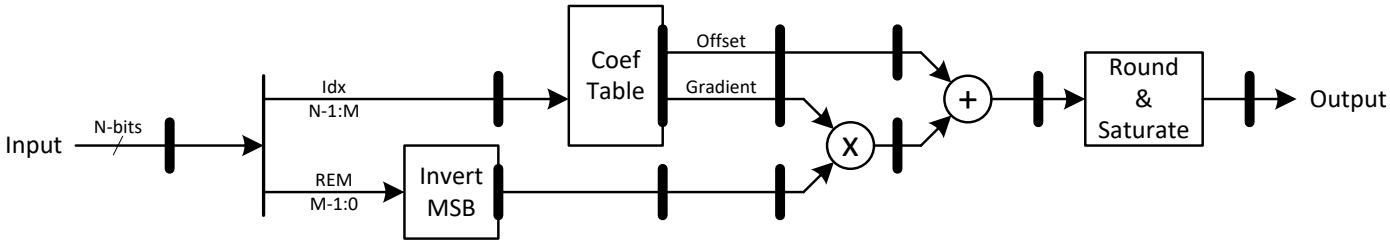


Figure 24: psi_fix_lin_approx Architecture

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.

3.15.5 Function Details

3.15.5.1 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.

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

3.15.5.3 Gaussify20b

Function: Inverse CDF of Gauss with σ=1/3
Input Range: -1 … 1
Remarks: This function can be used to map white noise to Gaussian distribution.

3.15.5.4 Inv18b

Function: Inversion (1/x)
Input Range: 1 ... 2
Remarks: If the input signal can be below 1, 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 left since $\frac{1}{x} = \frac{1}{2^N} \cdot \frac{1}{\frac{x}{2^N}} = \frac{1}{2^N} \cdot \frac{2^N}{x}$.

3.16 psi_fix_dds_18b

3.16.1 Description

This entity implements an 18-bit direct digital synthesizer (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. The DDS supports single-channel or multi-channel TDM implementation.

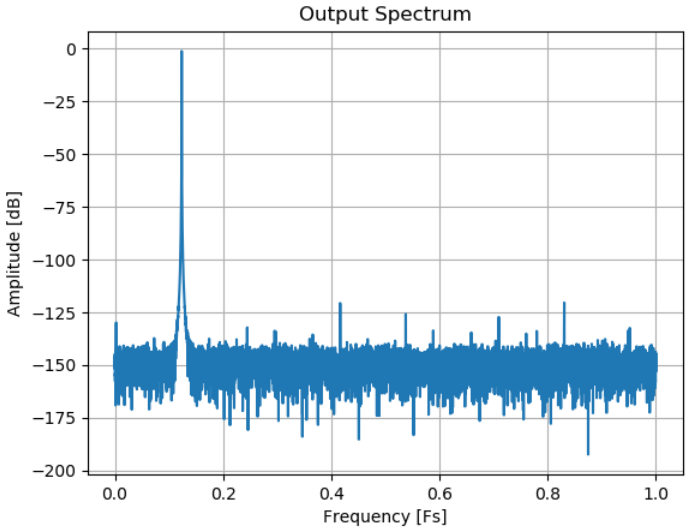


Figure 25: psi_fix_dds_18b Spectrum for PhaseStep=0.12345

3.16.2 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.

TdmChannels_g Number of channels in TDM mode (use 1 for single channel implementation)

RamBehavior_g "RBW" (read before write) or "WBR" (write before read), depending on the technology used

3.16.3 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. The inputs *Restart*, *PhaseStep* and *PhaseOffset* are latched on *InVld=1.*
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.

3.16.4 Architecture

The figure below shows the implementation of the DDS.

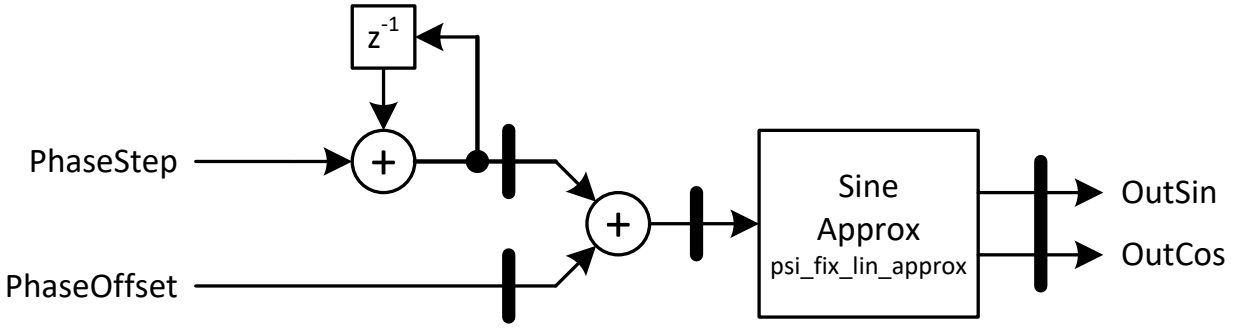


Figure 26: psi_fix_dds_18b Architecture

3.16.5 TDM Implementation Considerations

When using the DDS in multi-channel TDM configuration, the inputs (*PhaseStep, PhaseOffs, Restart*) of all channels must be applied in turns.

3.17 psi_fix_lowpass_iir_order1

3.17.1 Description

This entity implements a first order IIR low-pass 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.

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

$$\beta = 1 - alpha$$

3.17.2 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)

3.17.3 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

3.17.4 Architecture

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

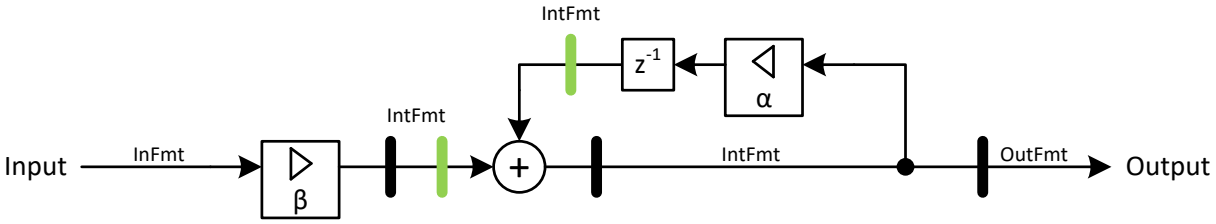


Figure 27: psi_fix_lowpass_iir_order1 Architecture

3.18 psi_fix_complex_addsub

3.18.1 Description

The block performs addition/subtraction on a complex number pair (*Inphase* & *Quadrature*, inputs of the block), let two complex numbers be:

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

The Addition result comes:

$$x + y = (a + c) + i(b + d)$$

The Addition result comes:

$$x - y = (a - c) + i(b - d)$$

The total pipeline delay of the block is 2 clock cycles if no pipeline activation is set through generics, otherwise the pipeline is doubled.

3.18.2 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
OutFmt_g Output format
Round_g Output rounding mode
Sat_g Output saturation mode
AddSub_g Select to use the block in adder or subtraction mode

3.18.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
InClk	Input	1	Clock
InRst	Input	1	Synchronous Reset
Input			
InInpADat	Input	InAFmt_g	Real part of input signal A
InQuaADat	Input	InAFmt_g	Imaginary part of input signal A
InInpBDat	Input	InBFmt_g	Real part of input signal B
InQuaBDat	Input	InBFmt_g	Imaginary part of input signal B
InVld	Input	1	AXI-S Handshaking signal
Output			
OutVld	Output	1	Data strobe output
OutInpo	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)

3.18.4 Architecture

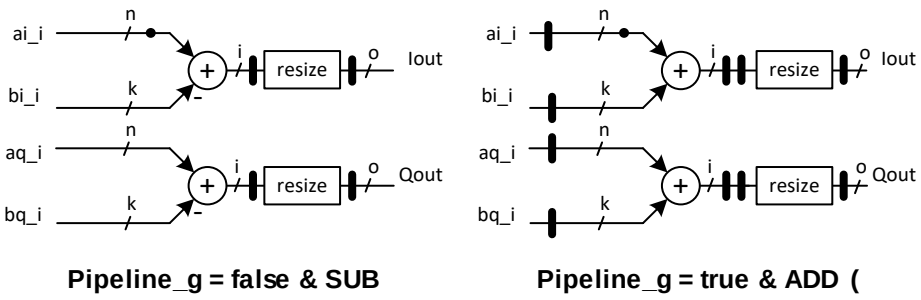


Figure 28: psi_fix_complex_addsub Architecture

3.19 psi_fix_complex_mult

3.19.1 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)

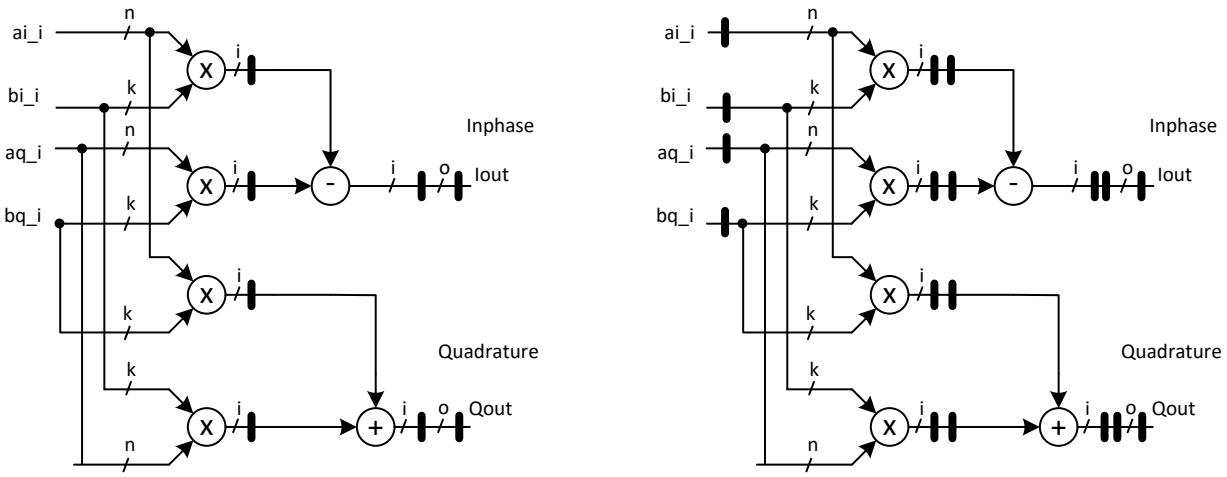
3.19.2 Generics

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
InAIsCplx_g Set to false if input A is real (not complex, *aq_i* = 0) to save multipliers
InBIsCplx_g Set to false if input B is real (not complex, *bq_i* = 0) to save multipliers

3.19.3 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)

3.19.4 Architecture



Pipeline_g = false **Pipeline_g = true**

Figure 29: psi_fix_complex_mult Architecture

3.20 psi_fix_mov_avg

3.20.1 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:

$$G_{None} = Taps$$

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

$$G_{Exact} = 1.0$$

3.20.2 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	

3.20.3 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

3.20.4 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.

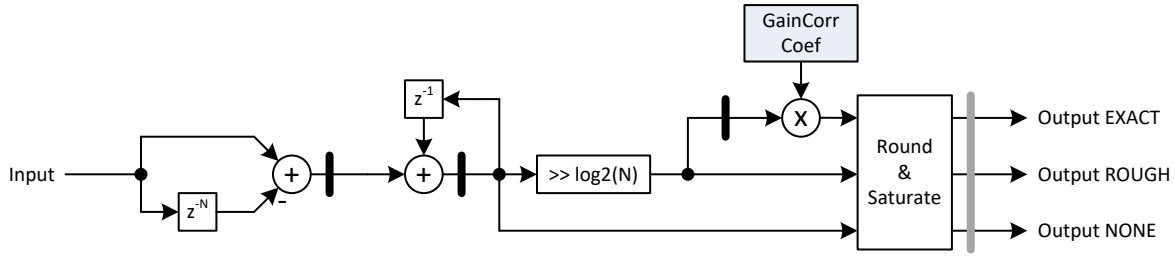


Figure 30: psi_fix_mov_avg Architecture

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.

3.21 psi_fix_demod_real2cplx

3.21.1 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.

The implementation supports multiple data channels with shared coefficients (i.e. the demodulation phase is the same for all channels). For multi-channel implementation, all channels must be synchronous (i.e. only one strobe/vld signal is provided that applies to all channels).

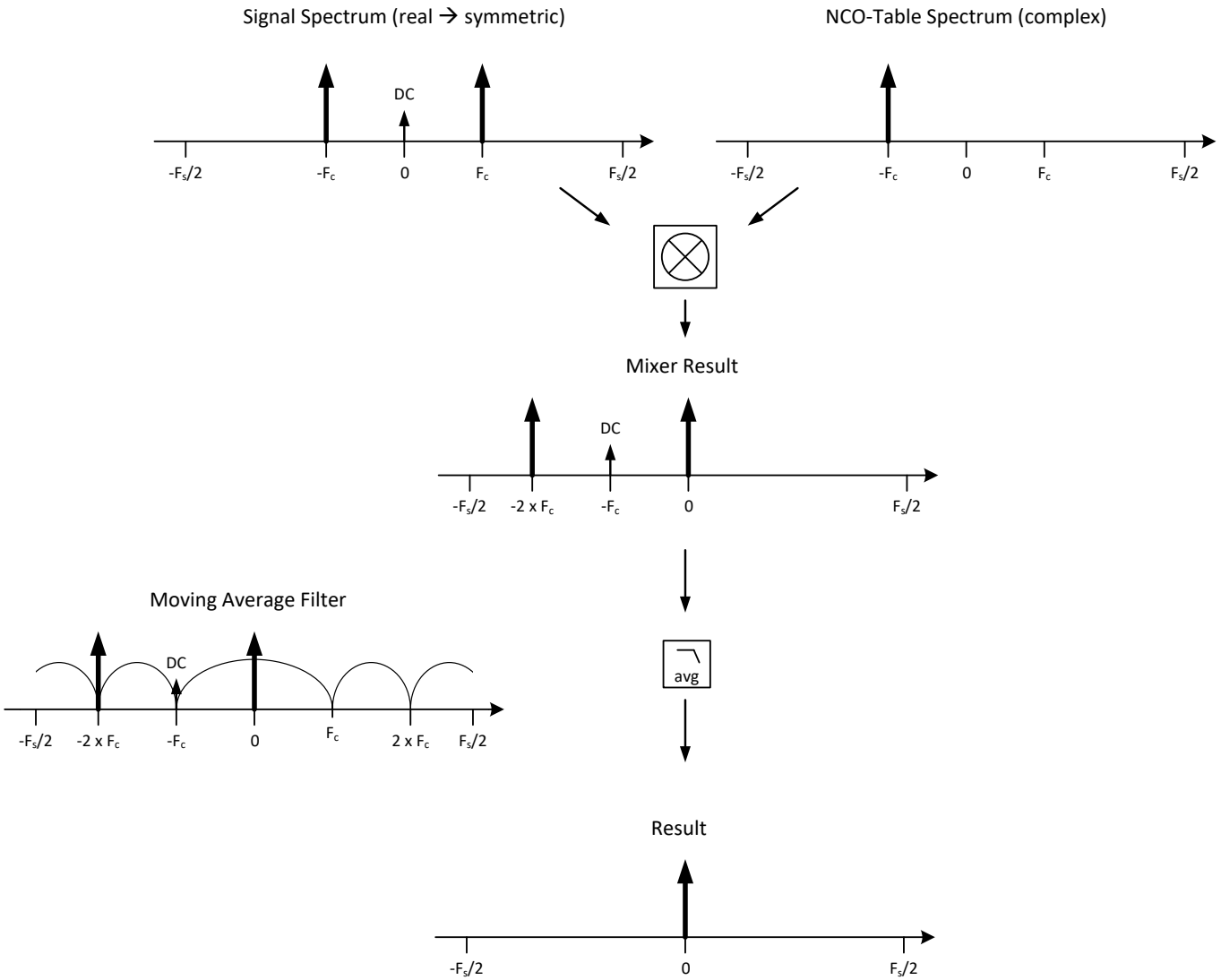


Figure 31: psi_fix_demod_real2cplx demodulation concept

3.21.2 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
Channels_g Number of channels to implement

3.21.3 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	InFmt_g*Channels_g	Data input For multiple channels, channels are concatenated*
phi_offset_i	Input	log2(Ratio_g)	Phase offset of the mixer frequency in $\frac{2\pi}{Ratio_g}$ Applied to all channels
Output			
data_I_o	Output	OutFmt_g*Channels_g	Real part of the output signal For multiple channels, channels are concatenated*
data_Q_o	Output	OutFmt_g*Channels_g	Imaginary part of the output signal For multiple channels, channels are concatenated*
str_o	Output	1	Output strobe (same as *Vld*)

* example: for 3 8-bit channels data is sorted as follows: ch0 -> bits 7..0, ch1 -> bits 15..8, ch2 -> bits 23..16

3.21.4 Architecture

The figure below shows the implementation of the demodulator. For simplicity only one channel is shown.

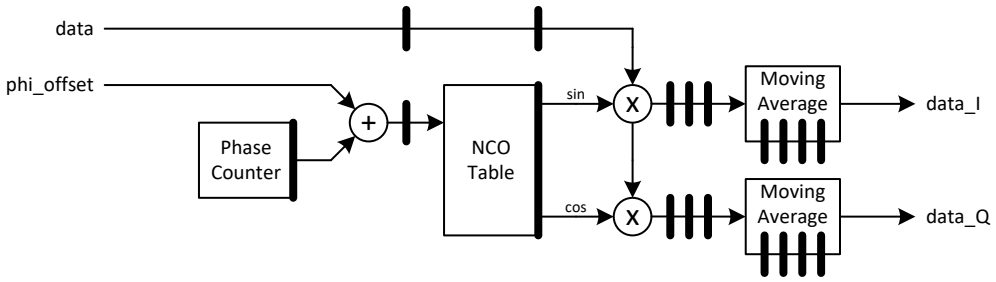


Figure 32: psi_fix_demod_real2cplx Architecture

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.

3.22 psi_fix_cordic_vect

3.22.1 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).

3.22.2 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.	

3.22.3 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°)

3.22.4 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.

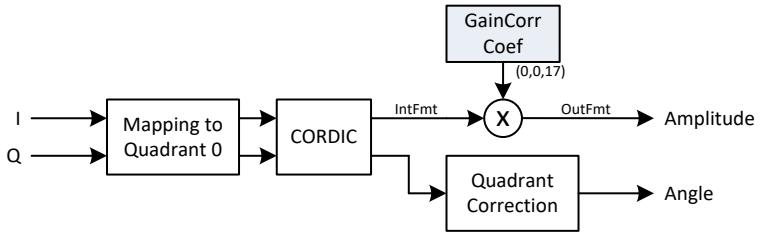


Figure 33: psi_fix_cordic_vect Architecture

3.23 psi_fix_cordic_rot

3.23.1 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 3.24) 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.

3.23.2 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

3.23.3 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)

3.23.4 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.

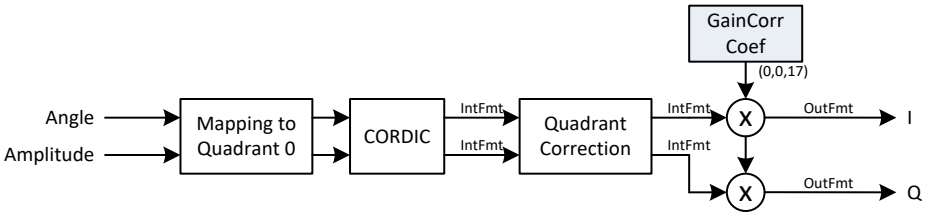


Figure 34: psi_fix_cordic_rot Architecture

3.24 psi_fix_pol2cart_approx

3.24.1 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.

3.24.2 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)

3.24.3 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)

3.24.4 Architecture

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

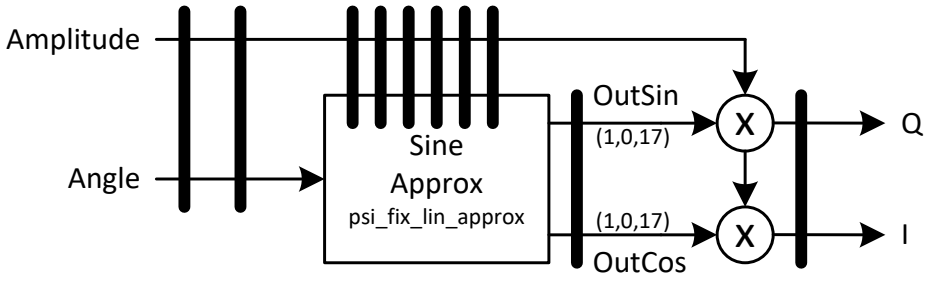


Figure 35: psi_fix_pol2cart_approx Architecture

3.25 psi_fix_mod_cplx2real

3.25.1 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 5 or 6 clock cycles depending on generics.

3.25.2 Generics

RstPol_g set the reset polarity
PlStages_g Number of pipeline stages (5 or 6), choose 6 for optimal timing.
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)*

3.25.3 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)

3.25.4 Architecture

The figure below shows the architecture of the demodulator. The pipeline stages in grey are optional (depending on generics).

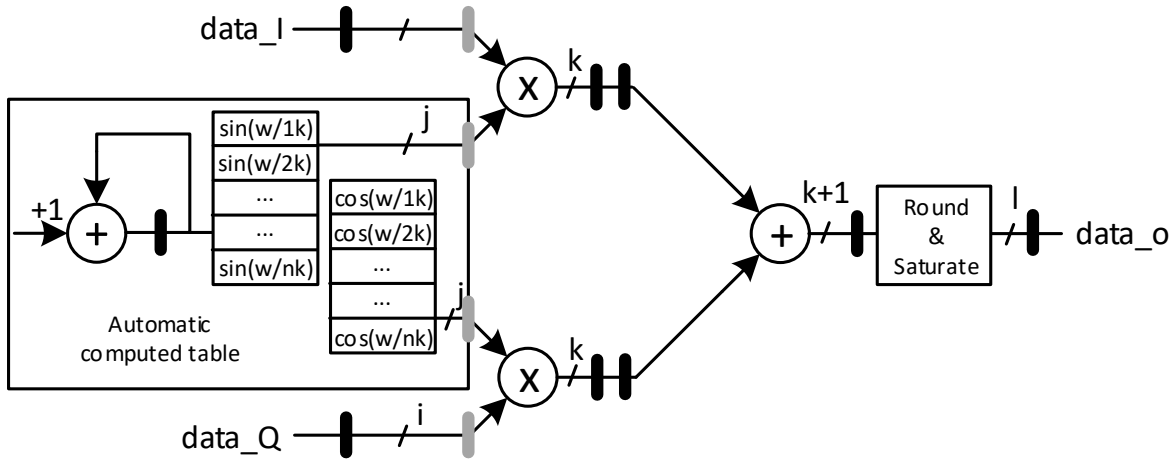


Figure 36: psi_fix_mod_cplx2real Archietcture

3.26 psi_fix_complex_abs

3.26.1 Description

This entity implements the absolute value calculation for a complex number based on the formula below:

$$|x| = \sqrt{real(x)^2 + imag(x)^2}$$

The square root function is approximated using *psi_fix_lin_approx_sqrt* function. The resulting implementation uses way less LUT than a CORDIC but multipliers and a bit or BRAM. Since the linear approximation of the square root function is limited to 18 bits, the result can have a relative error (relative to the absolute value of the output):

$$error = 2^{-18} = 3.8\,ppm$$

Note that the solution uses multipliers for the squaring operation, so for number formats that are wider than the multipliers available, this entity may not achieve optimal timing.

3.26.2 Generics

InFmt_g Format of the input
OutFmt_g Output format of the absolute value
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)
RamBehavior_g "RBW" (read before write) or "WBR" (write before read), depending on the technology used

3.26.3 Interfaces

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

3.26.4 Architecture

The figure below shows the architecture of the absolute value calculation.

For simple implementation, all formats are normalized to the range +/-1.0 and the normalized numbers are used for internal calculations. At the output, the normalization is reverted, so the normalization is completely invisible from outside.

Since the square root approximation is only valid in the range between 0.25 and 1.0, all numbers are first shifted into this range and the shift is compensated at the output of the calculation. This setup also allows for relatively precise results, eventhough the square-root approximation is limited to 18 bits.

This concept works because:

$$\frac{\sqrt{a \cdot 2^{2n}}}{2^n} = \sqrt{a \cdot 2^{2n}} \cdot 2^{-n} = \sqrt{a \cdot 2^{2n}} \cdot \sqrt{2^{-2n}} = \sqrt{a \cdot 2^{2n} \cdot 2^{-2n}} = \sqrt{a \cdot 2^{2n-2n}} = \sqrt{a \cdot 2^0} = \sqrt{a}$$

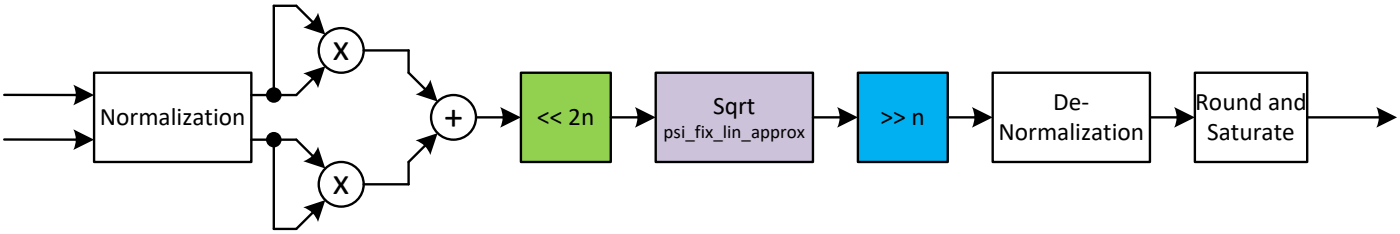


Figure 37: psi_fix_complex_abs

3.27 psi_fix_phase_unwrap

3.27.1 Description

This entity implements phase unwrapping by bringing all input angles into the range of +/- 1π (+/- 180°) of the previous sample and summing up the inputs.

Since phase unwrapping can accumulate infinitely, there is no theoretically sufficient output format. Therefore the user can choose the output format. If the unwrapping exceeds this format, the error is detected and signaled at the output. In this case the unwrapping is reset. This is shown by the figure below. In the figure, the output range is 4π (720°).

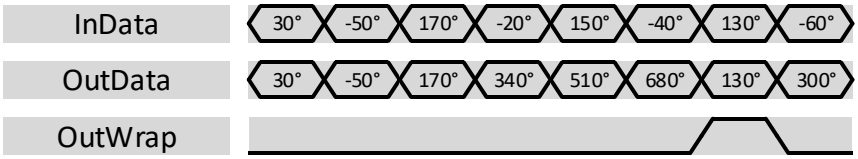


Figure 38: psi_fix_phase_unwrap overflow behavior

It is clearly visible, that when exceeding the maximum range, the output is reset to the input angle and unwrapping continues from there. So the phase itself is still correct (phase modulo 360° is correct) but there is a discontinuity in the output.

3.27.2 Generics

InFmt_g	Format of the input (usually [1,0,x] or [0,1,x])
OutFmt_g	Output format (must have at least one integer bit and be signed)
Round_g	Rounding mode at the output (use truncation for high clock speeds)

3.27.3 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 phase in π
Output			
OutVld	Output	1	AXI-S handshaking signal
OutData	Output	OutFmt_g	Output phase in π
OutWrap	Output	1	Indicates that the range was exceeded and that the output does therefore contain a discontinuity. See above.

3.28 psi_fix_white_noise

3.28.1 Description

This entity generates equally distributed white noise.

It does so by implementing an LFSR based pseudo-random binary sequence for each bit. Because the bits are fully uncorrelated (different seed), the signal generated is completely white.

The LFSRs are 32-bits wide, and the sequence repeats every 2'734'686'208 samples. For normal applications, this is long enough to be considered as uncorrelated.

The seed of the generator can be modified using generics. This allows generating different random sequences in the same design.

3.28.2 Generics

OutFmt_g Output format (max. 32 bits)
Seed_g Seed for the random number generator

3.28.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Input			
InVld	Input	1	AXI-S handshaking signal This signal is optional. If not connected, a new sample is generated each clock cycle.
Output			
OutVld	Output	1	AXI-S handshaking signal
OutData	Output	OutFmt_g	Output phase in π

The signal *InVld* is mainly used for bittrueness purposes. It allows adding noise to a signal exactly the same way as I the simulation by requesting the next sample *InVld='1'* whenever a new signal sample is available.

Of course the signal must be delayed for the processing time of the noise generator in this case in order to add the first noise sample to the first signal sample. This delay can easiest and most flexibly be achieved by a FIFO (this also still works if for any reasons the delay of the noise generator should slightly change).

3.29 psi_fix_noise_awgn

3.29.1 Description

This entity generates average-free, white, gaussian distributed noise in the range from -1 to + 1. It does so by using the *psi_fix_white_noise* generator and map the distribution to a gaussian one using *psi_fix_lin_approx*.

The LFSRs are 32-bits wide, and the sequence repeats every 2'734'686'208 samples. For normal applications, this is long enough to be considered as uncorrelated.

The seed of the generator can be modified using generics. This allows generating different random sequences in the same design.

3.29.2 Generics

OutFmt_g Output format, must be [1,0,x] where x must be <= 19.
Seed_g Seed for the random number generator

3.29.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset
Input			
InVld	Input	1	AXI-S handshaking signal This signal is optional. If not connected, a new sample is generated each clock cycle.
Output			
OutVld	Output	1	AXI-S handshaking signal
OutData	Output	OutFmt_g	Output phase in π

The signal *InVld* is mainly used for bittrueness purposes. It allows adding noise to a signal exactly the same way as I the simulation by requesting the next sample *InVld='1'* whenever a new signal sample is available.

Of course the signal must be delayed for the processing time of the noise generator in this case in order to add the first noise sample to the first signal sample. This delay can easiest and most flexibly be achieved by a FIFO (this also still works if for any reasons the delay of the noise generator should slightly change).

3.30 psi_fix_lut

3.30.1 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.

3.30.2 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

3.30.3 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

3.31 psi_fix_pkg_writer

3.31.1 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.

3.32 psi_fix_resize

3.32.1 Description

Pipleined psi_fix format change.

If rounding and saturation are done in the same clock cycle (or even in the same clock cycle as the actual operation), this often leads to timing issues. Therefore, a pipelined version of the resize that does rounding and saturation in separate clock cycles is provided.

The implementation is fully pipelined (i.e. can handle one conversion per clock cycle), has a delay of two clock cycles and supports full handshaking (including back-pressure).

3.32.2 Generics

InFmt_g Input format
OutFmt_g Output format
Round_g Rounding mode at the output (round or truncate)
Sat_g Saturation mode at the output (saturate of wrap)

3.32.3 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*	Input data
Output			
OutVld	Output	1	AXI-S handshaking signal
OutRdy	Input	1	AXI-S handshaking signal
OutData	Output	*OutFmt_g*	Output data

3.33 psi_fix_sqrt

3.33.1 Description

This entity implements a square root calculation.

The square root function is approximated in the range 0.25-1.0 using *psi_fix_lin_approx_sqrt* function and input/output are shifted to match the valid range of the approximation. The resulting implementation uses way less LUT than a CORDIC but multipliers and a bit or BRAM. Since the linear approximation of the square root function is limited to 18 bits, the result can have a relative error (relative to the absolute value of the output):

$$error = 2^{-18} = 3.8\,ppm$$

3.33.2 Generics

InFmt_g Format of the input
OutFmt_g Output format of the absolute value
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)
RamBehavior_g "RBW" (read before write) or "WBR" (write before read), depending on the technology used

3.33.3 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 signal
Output			
OutVld	Output	1	AXI-S handshaking signal
OutData	Output	OutFmt_g	Output data

3.33.4 Architecture

The figure below shows the architecture of the square roiot calculation.

For simple implementation, all formats are normalized to the range +/-1.0 and the normalized numbers are used for internal calculations. At the output, the normalization is reverted, so the normalization is completely invisible from outside.

Since the square root approximation is only valid in the range between 0.25 and 1.0, all numbers are first shifted into this range and the shift is compensated at the output of the calculation. This setup also allows for relatively precise results, even though the square-root approximation is limited to 18 bits.

This concept works because:

$$\frac{\sqrt{a \cdot 2^{2n}}}{2^n} = \sqrt{a \cdot 2^{2n}} \cdot 2^{-n} = \sqrt{a \cdot 2^{2n}} \cdot \sqrt{2^{-2n}} = \sqrt{a \cdot 2^{2n} \cdot 2^{-2n}} = \sqrt{a \cdot 2^{2n-2n}} = \sqrt{a \cdot 2^0} = \sqrt{a}$$

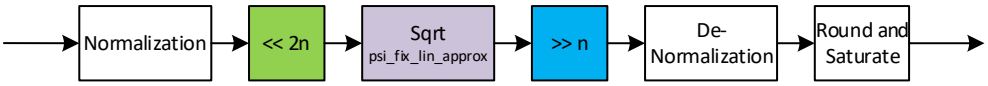


Figure 39: psi_fix_sqrt

3.34 psi_fix_inv

3.34.1 Description

This entity implements a inversion (1/x) calculation.

The inversion function is approximated in the range 1 ... 2 using *psi_fix_lin_approx_inv18b* function and input/output are shifted to match the valid range of the approximation. The resulting implementation uses way less LUT than a CORDIC but multipliers and a bit or BRAM. Since the linear approximation of the inversion function is limited to 18 bits, the result can have a relative error (relative to the absolute value of the output):

$$error = 2^{-18} = 3.8\,ppm$$

3.34.2 Generics

InFmt_g Format of the input
OutFmt_g Output format of the absolute value
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)
RamBehavior_g "RBW" (read before write) or "WBR" (write before read), depending on the technology used

3.34.3 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 signal
Output			
OutVld	Output	1	AXI-S handshaking signal
OutData	Output	OutFmt_g	Output data

3.34.4 Architecture

The figure below shows the architecture of the inversion calculation.

For simple implementation, all formats are normalized to the range +/- 2.0 and the normalized numbers are used for internal calculations. At the output, the normalization is reverted, so the normalization is completely invisible from outside.

Since the inversion approximation is only valid in the range between 1 and 2.0, all numbers are first shifted into this range and the shift is compensated at the output of the calculation. This setup also allows for relatively precise results, eventhough the inversion approximation is limited to 18 bits.

This concept works because:

$$\frac{\frac{1}{x} \cdot 2^n}{2^n} = \frac{1}{x}$$

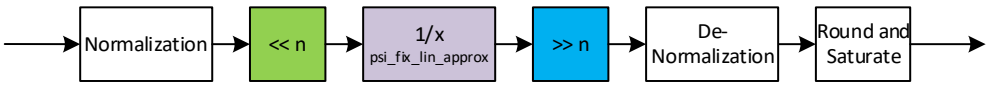


Figure 40: psi_fix_inv

3.35 psi_fix_comparator

3.35.1 Description

This entity implements two comparators using fixed point format in order to get flag within a window between minimum and maximum threshold. This entity is used in *psi_fix_nch_amalog_trigger_tdm*. The output latency is set by design to 3 clock cycles.

3.35.2 Generics

fmt_g Format of data
rst_pol_g reset polarity active high or low

3.35.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk_i	Input	1	Clock
Rst_i	Input	1	Reset
Input			
Set_min_i	Input	Fmt_g	Input parameter minimum threshold
Set_max_i	Input	Fmt_g	Input parameter maximum threshold
Data_i	Input	Fmt_g	Input data
Str_i	Input	1	AXI-S handshaking signal
Output			
Max_o	Output	1	Maximum flag output
Min_o	Output	1	Minimum flag output
Str_o	Output	1	AXI-S handshaking signal

3.36 psi_fix_nch_analog_trigger_tdm

3.36.1 Description

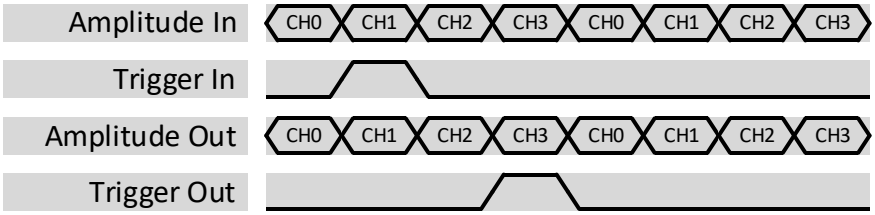
This unit implements an architecture prior to generate trigger that can be used for the ***MSDAQ*** available at the following link: https://github.com/paulscherrerinstitute/psi_multi_stream_daq

The specificity is that several channel can be checked with a set of dedicated thresholds, minimum and maximum. Data input must be fed in a TDM fashion whereas the parameters are set with parallel registers. Prior to avoid multiple use of comparator, setting registers are converted from parallel to TDM and aligned with data to perform comparison.

The results are connected to the digital trigger and the mechanism for arming and disarming is described in the ***psi_common*** library at the following link:

https://github.com/paulscherrerinstitute/psi_common/blob/master/hdl/psi_common_trigger_digital.vhd

The trigger output is aligned with the TDM data stream to ensure that the last value contains the trigger, see next datagram:



The previous datagram shows the case where an external trigger is coming in, however it is important that the trigger out is aligned to the last channel of the TDM data stream.

The VHDL file includes a specific package to define the data length with fixed-point format as well as the parameter in type. The next snippet shows the parameter record type definition.

```
type param_t is record
   mask_min_ena : std_logic_vector(CH_NUMBER_MAX_c-1 downto 0); --mask min results
   mask_max_ena : std_logic_vector(CH_NUMBER_MAX_c-1 downto 0); --mask max results
   thld         : param_array_t(0 to CH_NUMBER_MAX_c-1);        --thld to set Min/Max window
   trig         : trig_cfg_t;                                   --trigger configuration
   clr_ext_trig : std_logic;
end record;
```

3.36.2 Generics

ch_nb_g Number of channel
fix_fmt_g Data format

3.36.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk_i	Input	1	Clock
Rst_i	Input	1	Reset
Input			
Str_i	Input	1	AXI-S handshaking signal
Dat_i	Input	Fix_fmt_g	Input signal – TDM
Ext_i	Input	1	External trigger, better be aligned with strobe input
Param_i	Input	Param_t	Set of parameter shown in previous code snippet.
Output			
Str_pipe_o	Output	1	AXI-S handshaking signal
Data_pipe_o	Output	Fix_Fmt_g	Output signal - TDM
Trig_o	Output	1	External trigger aligned with last TDM channel
Is_arm_o	Ouutput	1	Trigger status

3.36.4 Architecture

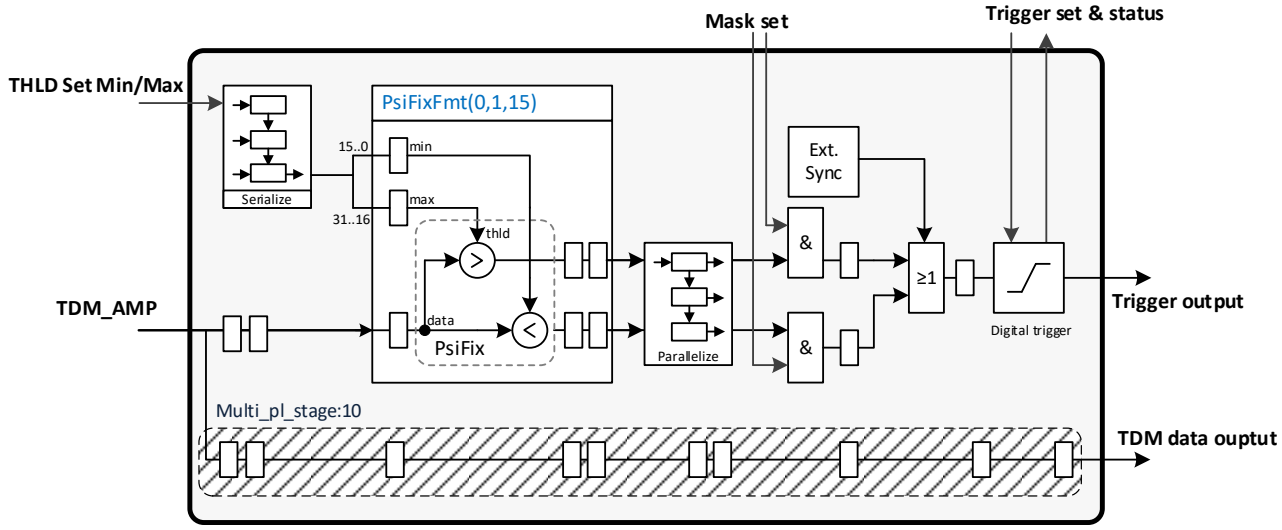


Figure 41 psi_fix_nch_analog_trigger_tdm

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 was deleted in 2.0.0.

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.