

# **psi\_fix**

## Documentation

# Content

## Table of Contents

1	Introduction.....	3
2	Tipps & Tricks.....	3
2.1	Library Setup .....	3
2.2	Heavy Pipelining.....	3
3	RTL Descriptions.....	7
3.1	psi_fix_bin_div .....	7
3.2	psi_fix_cic_dec_fix_1ch.....	9
3.3	psi_fix_cic_int_fix_1ch.....	11
3.4	psi_fix_cordic_abs_pl.....	13
3.5	psi_fix_fir_dec_ser_nch_chpar_conf.....	15
3.6	psi_fix_fir_dec_ser_nch_chtdm_conf.....	18
3.7	psi_fix_lin_approx_<function>.....	21
3.8	psi_fix_dds_18b.....	23

## Figures

Figure 1: Heavy Pipelining, Problem Description .....	4
Figure 2: Heavy Pipelining, Retiming, Implementation without retiming .....	4
Figure 3: Heavy Pipelining, Retiming, Implementation with retiming .....	5
Figure 4: Heavy Pipelining, Manual Splitting .....	6
Figure 5: psi_fix_bin_div Architecture .....	8
Figure 6: psi_fix_cic_dec_fix_1ch Architecture.....	10
Figure 7: psi_fix_cic_int_fix_1ch Architecture.....	12
Figure 8: psi_fix_fir_dec_ser_nch_chpar_conf Architecture .....	17
Figure 9: psi_fix_fir_dec_ser_nch_chtdm_conf Architecture .....	20
Figure 10: psi_fix_lin_approx Interpolation Principle .....	22
Figure 11: psi_fix_lin_approx Architecture.....	22
Figure 12: psi_fix_dds_18b Spectrum for PhaseStep=0.12345.....	23
Figure 13: psi_fix_dds_18b Architecture .....	24

# 1 Introduction

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

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

## 2 Tipps & Tricks

### 2.1 Library Setup

The *psi\_fix* library refers to *psi\_common* and *psi\_tb* relatively and assumes the contents of these repositories are compiled into the same VHDL library.

There are two common ways of setting up projects without 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)
- Limiting

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

...

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

This leads to the implementation shown below.

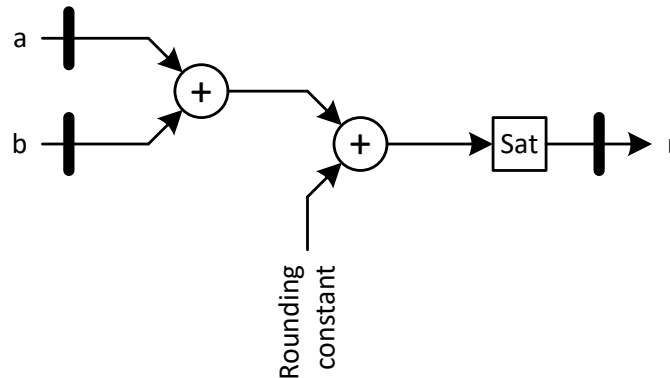


Figure 1: Heavy Pipelining, Problem Description

### 2.2.2 Solution 1: Register Retiming

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

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

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

...

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

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

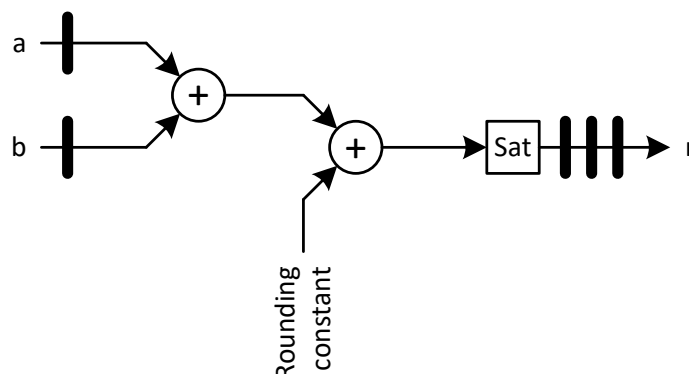
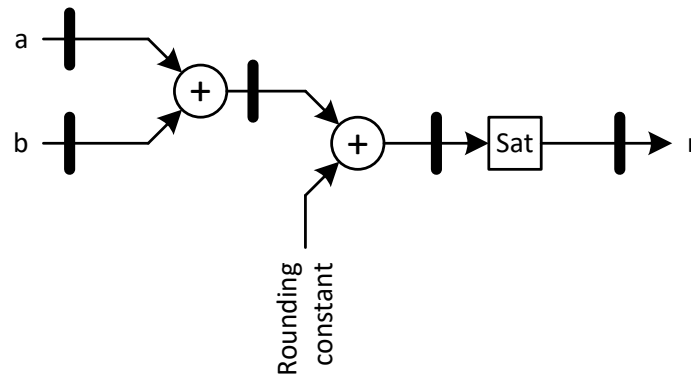


Figure 2: Heavy Pipelining, Retiming, Implementation without retiming

However, 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 3: Heavy Pipelining, Retiming, Implementation with retiming**

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

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

### 2.2.3 Solution 2: Manual Splitting

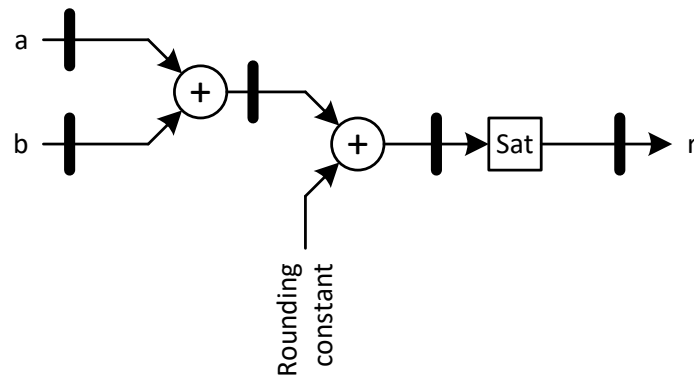
The operation can be split into multiple stages manually on VHDL level. This can be done by not doing all steps in one VHDL line but one after the other in multiple lines. Of course intermediate number formats must be chosen accordingly to ensure correct operation. An example is given below.

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

...

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

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



**Figure 4: Heavy Pipelining, Manual Splitting**

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

The disadvantage is that slightly more code is required.

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.

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

#### 3.1.2 Generics

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

#### 3.1.3 Interfaces

Signal	Direction	Width	Description
<b>Control Signals</b>			
Clk	Input	1	Clock
Rst	Input	1	Reset
<b>Input</b>			
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
<b>Output</b>			
OutVld	Output	1	AXI-S handshaking signal
OutQuot	Output	QuotFmt_g	Quotient output

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

### 3.1.4 Architecture

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

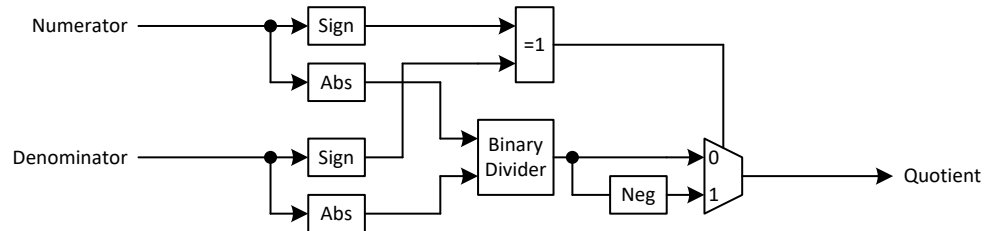


Figure 5: `psi_fix_bin_div` Architecture



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

### 3.2.1 Description

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

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

### 3.2.2 Generics

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

### 3.2.3 Interfaces

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

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

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

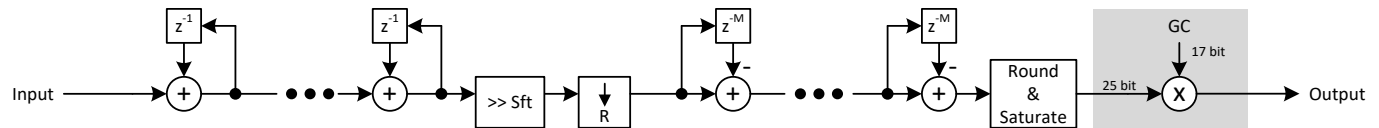
### 3.2.4 Architecture

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

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

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

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



**Figure 6: psi\_fix\_cic\_dec\_fix\_1ch Architecture**

The symbols are defined as follows:

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

Some of the most common formulas are given below.

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

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

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

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

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

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

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

#### 3.3.1 Description

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

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

#### 3.3.2 Generics

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

#### 3.3.3 Interfaces

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

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

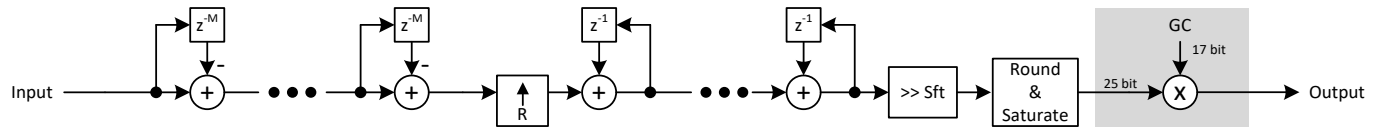
Full handshaking at the output side was implemented mainly to allow equally spaced output samples (in time). By nature the filter calculates multiple output samples back-to-back after an input sample arrived. For output rates lower than the clock-speed, this leads to a bursting behavior which is often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

### 3.3.4 Architecture

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

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

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



**Figure 7: psi\_fix\_cic\_int\_fix\_1ch Architecture**

The symbols are defined as follows:

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

Some of the most common formulas are given below.

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

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

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

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

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  $\frac{f_s}{2}$ ) and 2 is used otherwise.

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

### 3.4 psi\_fix\_cordic\_abs\_pl

#### 3.4.1 Description

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

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

#### 3.4.2 Generics

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

#### 3.4.3 Interfaces

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

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

### 3.4.4 Architecture

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

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

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

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

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

The CORDIC gain can be calculated by the formula below:

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

Where:

$G_{CORDIC}$	Cordic Gain
$N$	Number of iterations

The formula converges towards 1.646760 with high numbers of iterations.

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

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

### 3.5.1 Description

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

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

### 3.5.2 Generics

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

### 3.5.3 Interfaces

Signal	Direction	Width	Description
<b>Control Signals</b>			
Clk	Input	1	Clock
Rst	Input	1	Reset
<b>Input</b>			
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] - ...
<b>Output</b>			
OutVld	Output	1	AXI-S handshaking signal
OutAbs	Output	$OutFmt\_g \cdot Channels\_g$	Output data in parallel (see <i>InData</i> )
<b>Configuration</b>			
Ratio	Input	$ceil(\log_2(MaxRatio\_g))$	Decimation ratio -1 0 → no decimation 1 → decimation by 2)  This port is optional. If it is not connected, <i>MaxRatio_g</i> is used as fixed ratio.

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

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

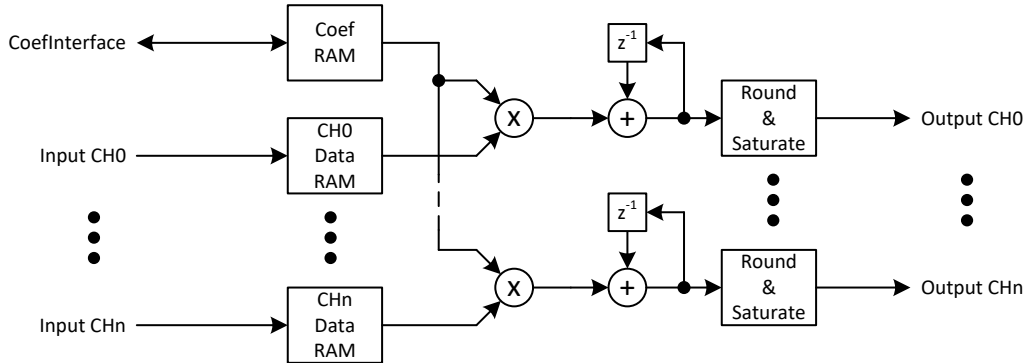
The filter can continue taking new input data even if a calculation is ongoing. As a result, the handling of backpressure is not required as long as the processing power of the filter is sufficient to handle all input data. For the calculation, see below.

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



### 3.5.4 Architecture

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



**Figure 8: psi\_fix\_fix\_dec\_ser\_nch\_chpar\_conf Architecture**

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

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

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

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

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

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

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

### 3.6.1 Description

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

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

### 3.6.2 Generics

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

### 3.6.3 Interfaces

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

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

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

The filter can continue taking new input data even if a calculation is ongoing. As a result, the handling of 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.

### 3.6.4 Architecture

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

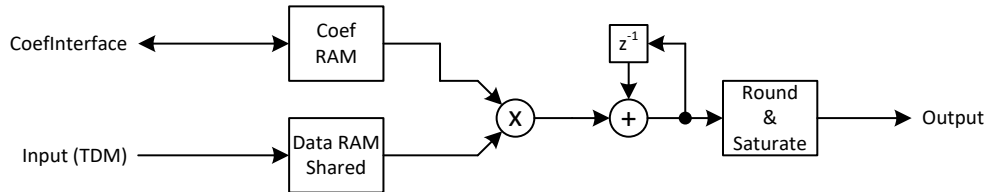


Figure 9: `psi_fix_fix_dec_ser_nch_chtdm_conf` Architecture

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

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

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

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

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

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

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

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

### 3.7.1 Description

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

All components are based on the same implementation of the approximation (*psi\_fix\_lin\_approx\_calc.vhd*) and they only vary in number formats and coefficient tables.

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

### 3.7.2 Generics

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

### 3.7.3 Interfaces

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

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

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

### 3.7.4 Architecture

The figure below shows the interpolation principle.

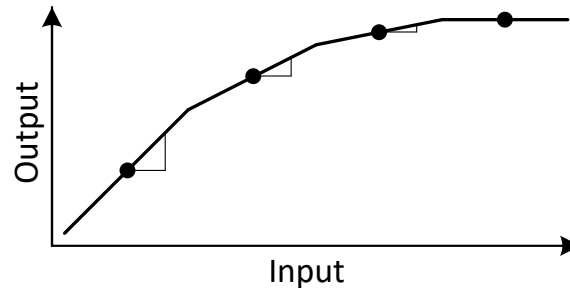


Figure 10: psi\_fix\_lin\_approx Interpolation Principle

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

The figure below shows the implementation of the approximation.

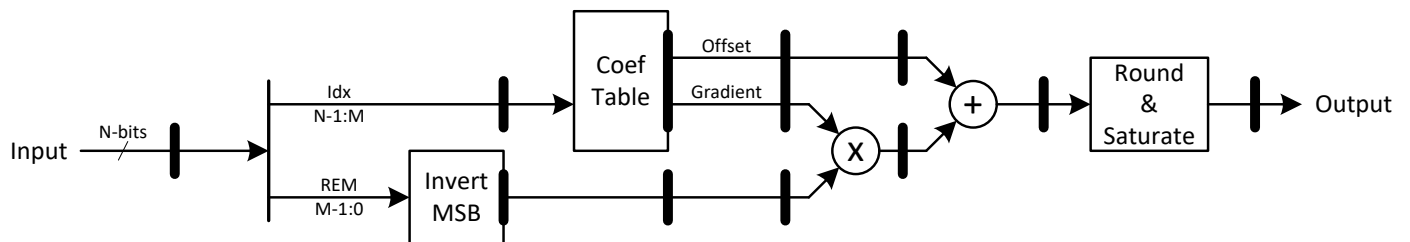


Figure 11: psi\_fix\_lin\_approx Architecture

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

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

## 3.8 psi\_fix\_dds\_18b

### 3.8.1 Description

This entity implements an 18-bit DDS. The sine-wave is generated using the entity *psi\_fix\_lin\_approx\_sin\_18b* and it has an error of less than one LSB for all values. As a result, there are no significant spurs in the generated spectrum (significant in terms of above the quantization noise floor) as shown in the figure below.

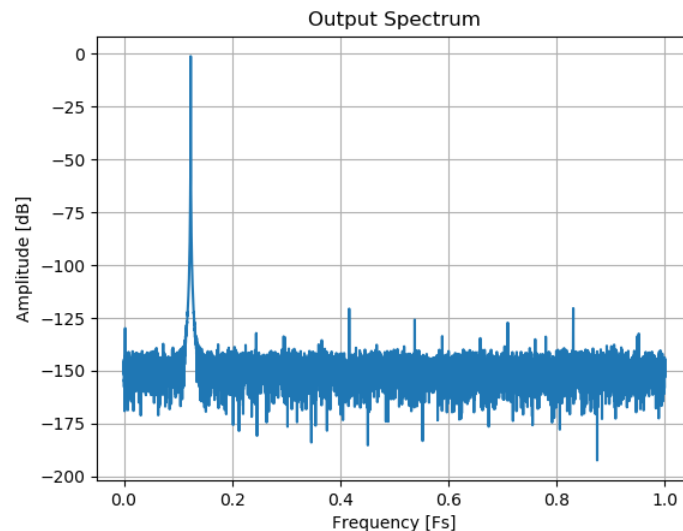


Figure 12: psi\_fix\_dds\_18b Spectrum for PhaseStep=0.12345

### 3.8.2 Generics

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

### 3.8.3 Interfaces

Signal	Direction	Width	Description
<b>Control Signals</b>			
Clk	Input	1	Clock
Rst	Input	1	Reset
<b>Configuration</b>			
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 $2\pi$ (0.5 corresponds to $\pi$ ). 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 $2\pi$ (0.5 corresponds to $\pi$ ). The phase offset can be changed at runtime safely.

<b>Input</b>			
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.
<b>Output</b>			
OutVld	Output	1	AXI-S handshaking signal
OutSin	Output	18	Sine wave output in the format [1,0,17]
OutCos	Output	18	Cosine wave output in the format [1,0,17]

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

### 3.8.4 Architecture

The figure below shows the implementation of the DDS.

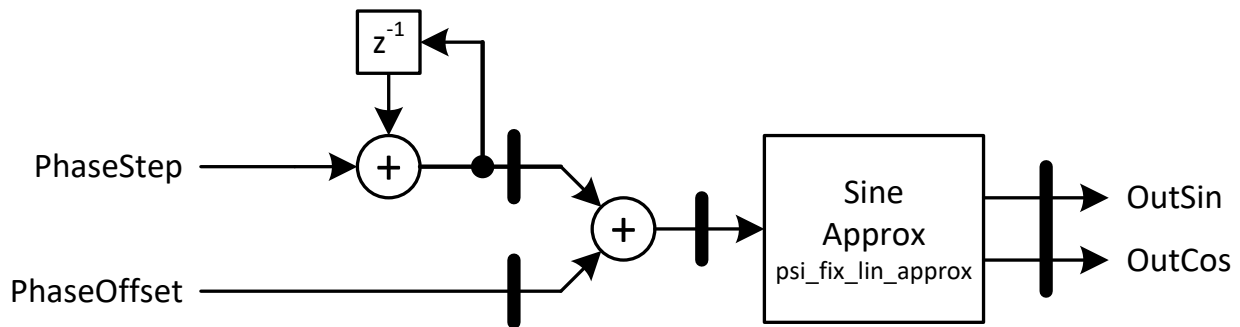


Figure 13: psi\_fix\_dds\_18b Architecture