psi_common

Documentation

Content

Table of Contents

Figures

1 Introduction

The purpose of this library is to provide HDL implementations for commonly used VHDL functionality such as memories, FIFOs and clock crossings.

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

1.1 Working Copy Structure

If you just want to use some components out of the *psi_common* library, no special structure is required and the repository can be used standalone.

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

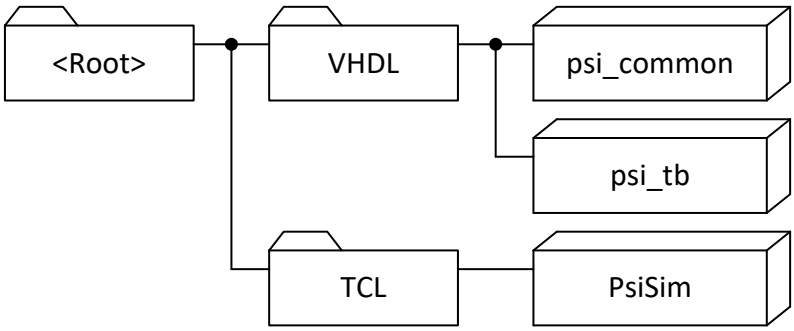

Figure 1: Working copy structure

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

1.2 VHDL Libraries

The PSI VHDL libraries (including *psi_common*) 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_common_pl_stage* 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_common_pl_stage* or *psi_lib.psi_common_array_pkg.all*).

1.3 Running Simulations

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

1.3.1 Regression Test

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.

1.3.2 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>*

1.4 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.
- 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.
- Documentation
  - Extend this document with proper documentation of your code.
- New test-benches must be added to theregression test-script
  - Change */sim/config.tcl* accordingly
  - Test if the regression test really runs the new test-bench and exits without errors before doing any merge requests.

1.5 Handshaking Signals

1.5.1 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.

1.5.2 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.

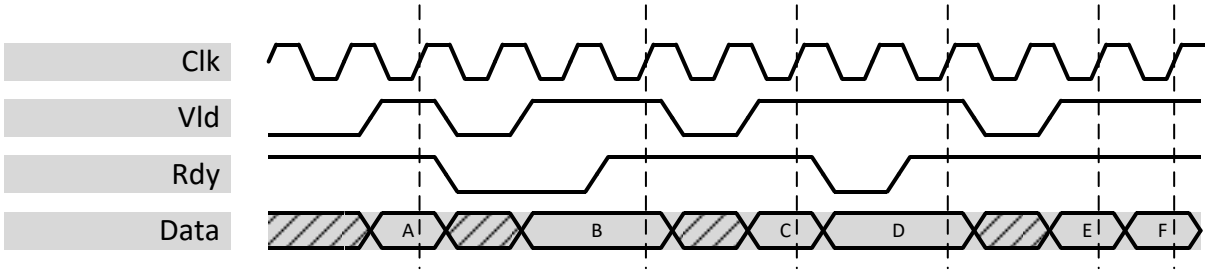

Figure 2: Handshaking signals

1.5.3 Naming

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

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

TVALID Vld, InVld, OutVld, Valid, str, str_i

TREADY Rdy, InRdy, OutRdy

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

2 Packages

2.1 psi_common_array_pkg

2.1.1 Description

This package defines various array types that are not defined by VHDL natively. Some of these definitions are no more required in VHDL 2008 but since VHDL 2008 is not yet fully synthesizable, the package is kept.

2.2 psi_common_logic_pkg

2.2.1 Description

This package contains various logic functions (e.g. combinatorial conversions) that can be synthesized.

2.3 psi_common_axi_pkg

2.3.1 Description

This package contains record definitions to allow representing a complete AXI interface including all ports by only two records (one in each direction). This helps improving the readability of entities with AXI interfaces.

2.4 psi_common_math_pkg

2.4.1 Description

This package contains various mathematical functions (e.g. log2). The functions are meant for calculating compile-time constants (i.e. constans, port-widths, etc.). They can potentially be synthesized as combinatorial functions but this is neither guaranteed nor will it lead to optimal results.

3 Memories

3.1 psi_common_sdp_ram

3.1.1 Description

This component implements a simple dual port RAM. It has one write port and one read port and both ports are running at the same clock. The RAM is described in a way that it utilizes RAM resources (Block-RAM and Distributed-RAM) available in FPGAs with commonly used tools.

The RAM is a synchronous RAM, so data is available at the read port one clock cycle after applying the address.

The RAM behavior (read-before-write or write-before-read) can be selected. This allows efficiently implementing RAMs for different technologies (some technologies implement one, some the other behavior).

3.1.2 Generics

Depth_g	Depth of the memory	
Width_g	Width of the memory	
IsAsync_g	true = Memory is asynchronous, *Clk* is used for write, *RdClk* for read	
	false = Memory is synchronous, *Clk* is used for read and write	
RamStyle_g	"auto" (default)	Automatic choice of block- or distributed-RAM
	"distributed"	Use distributed RAM (LUT-RAM)
	"block"	Use block RAM
Behavior_g	"RBW"	Read-before-write implementation
	"WBR"	Write-before-read implementation

3.1.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
RdClk	Input	1	Read clock (only used if *IsAsync_g* = true)
Write Port			
WrAddr	Input	ceil(log2(Depth_g))	Write address
Wr	Input	1	Write enable (active high)
WrData	Input	Width_g	Write data
Read Port			
RdAddr	Input	ceil(log2(Depth_g))	Read address
Rd	Input	1	Read enable (active high)
RdData	Output	Width_g	Read data

3.2 psi_common_sp_ram_be

3.2.1 Description

This component implements a single port RAM with byte enables. The RAM is described in a way that it utilizes RAM resources (Block-RAM and Distributed-RAM) available in FPGAs with commonly used tools.

The RAM is a synchronous RAM, so data is available at the read port one clock cycle after applying the address.

The RAM behavior (read-before-write or write-before-read) can be selected. This allows efficiently implementing RAMs for different technologies (some technologies implement one, some the other behavior).

3.2.2 Generics

Depth_g Depth of the memory
Width_g Width of the memory in bits (must be a multiple of 8)
Behavior_g "RBW" Read-before-write implementation
"WBR" Write-before-read implementation

3.2.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Access Port			
Addr	Input	ceil(log2(Depth_g))	Access address
Wr	Input	1	Write enable (active high)
Be	Input	Width_g/8	Byte enables (Be[0] corresponds do Din[7:0])
Din	Input	Width_g	Write data
Dout	Output	Width_g	Read data

3.3 psi_common_tdp_ram

3.3.1 Description

This component implements a true dual port RAM. It has one write port and one read port and both ports can be running at different clocks (completely asynchronous clocks are allowed). The RAM is described in a way that it utilizes RAM resources (Block-RAM) available in FPGAs with commonly used tools.

The RAM is a synchronous RAM, so data is available at the read port one clock cycle after applying the address.

The RAM behavior (read-before-write or write-before-read) can be selected. This allows efficiently implementing RAMs for different technologies (some technologies implement one, some the other behavior).

3.3.2 Generics

Depth_g Depth of the memory
Width_g Width of the memory
Behavior_g "RBW" Read-before-write implementation
"WBR" Write-before-read implementation

3.3.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Port A			
ClkA	Input	1	Port A clock
AddrA	Input	ceil(log2(Depth_g))	Port A address
WrA	Input	1	Port A write enable (active high)
DinA	Input	Width_g	Port A write data
DoutA	Output	Width_g	Port A read data
Port B			
ClkB	Input	1	Port B clock
AddrB	Input	ceil(log2(Depth_g))	Port B address
WrB	Input	1	Port B write enable (active high)
DinB	Input	Width_g	Port B write data
DoutB	Output	Width_g	Port B read data

3.3.4 Constraints

For the RAM to work correctly, signals from one clock domain to the other must be constrained to have not more delay that one clock cycle of the faster clock.

Example with a 100 MHz clock (10.0 ns period) and a 33.33 MHz clock (30 ns period) for Vivado:

```
set_max_delay -datapath_only -from <ClkA> -to <ClkB> 10.0
set_max_delay -datapath_only -from <ClkB> -to <ClkA> 10.0
```

4 FIFOs

4.1 psi_common_async_fifo

4.1.1 Description

This component implements an asynchronous FIFO (different clocks for write and read port). The memory is described in a way that it utilizes RAM resources (Block-RAM) available in FPGAs with commonly used tools.

The FIFO is a fall-through FIFO and has AXI-S interfaces on read and write side.

The RAM behavior (read-before-write or write-before-read) can be selected. This allows efficiently implementing FIFOs for different technologies (some technologies implement one, some the other behavior).

4.1.2 Generics

Width_g	Width of the FIFO	
Depth_g	Depth of the FIFO	
AlmFullOn_g	True = Almost full output is provided, False = Almost full output is omitted	
AlmFullLevel_g	Almost full output is high if the level is >= AlmFullLevel_g	
AlmEmptyOn_g	True = Almost empty output is provided, False = Almost empty output is omitted	
AlmEmptyLevel_g	Almost empty output is high if the level is <= AlmFullLevel_g	
RamStyle_g	"auto" (default)	Automatic choice of block- or distributed-RAM
	"distributed"	Use distributed RAM (LUT-RAM)
	"block"	Use block RAM
RamBehavior_g	"RBW"	Read-before-write implementation
	"WBR"	Write-before-read implementation

4.1.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
InClk	Input	1	Write side clock
InRst	Input	1	Write side reset input (active high)
OutClk	Input	1	Read side clock
OutRst	Input	1	Read side reset input (active high)
Input Data (InClk domain)			
InData	Input	Width_g	Write data
InVld	Input	1	AXI-S handshaking signal
InRdy	Output	1	AXI-S handshaking signal
Output Data (OutClk domain)			
OutData	Output	Width_g	Read data
OutVld	Output	1	AXI-S handshaking signal
OutRdy	Input	1	AXI-S handshaking signal

Input Status (InClk domain)			
InFull	Output	1	FIFO full signal synchronous to *InClk*
InEmpty	Output	1	FIFO empty signal synchronous to *InClk*
InAlmFull	Output	1	FIFO almost full signal synchronous to *InClk* Only exists if *AlmFullOn_g* = true
InAlmEmpty	Output	1	FIFO almost empty signal synchronous to *InClk* Only exists if *AlmEmptyOn_g* = true
InLevel	Output	ceil(log2(Depth_g))+1	FIFO level synchronous to *InClk*
Output Status (OutClk domain)			
OutFull	Output	1	FIFO full signal synchronous to *OutClk*
OutEmpty	Output	1	FIFO empty signal synchronous to *OutClk*
OutAlmFull	Output	1	FIFO almost full signal synchronous to *OutClk* Only exists if *AlmFullOn_g* = true
OutAlmEmpty	Output	1	FIFO almost empty signal synchronous to *OutClk* Only exists if *AlmEmptyOn_g* = true
OutLevel	Output	ceil(log2(Depth_g))+1	FIFO level synchronous to *OutClk*

4.1.4 Architecture

The rough architecture of the FIFO is shown in the figure below. Note that the figure does only depict the general architecture and not each and every detail.

Read and write address counters are handled in their corresponding clock domain. The current address counter value is then transferred to the other clock-domain by converting it to gray code, synchronizing it using a double-stage synchronizer and convert it back to a two's complement number. This approach ensures that a correct value is received, even if the clock edges are aligned in a way that causes metastability on the first flip-flop. Because the data is transferred in gray code, in this case either the correct value before an increment of the counter or the correct value after the increment is received, so the result is always correct.

All status information is calculated separately in both clock domains to make it available synchronously to both clocks.

This architecture is independent of the FPGA technology used and can also be used to combine more than just one Block-RAM into one big FIFO.

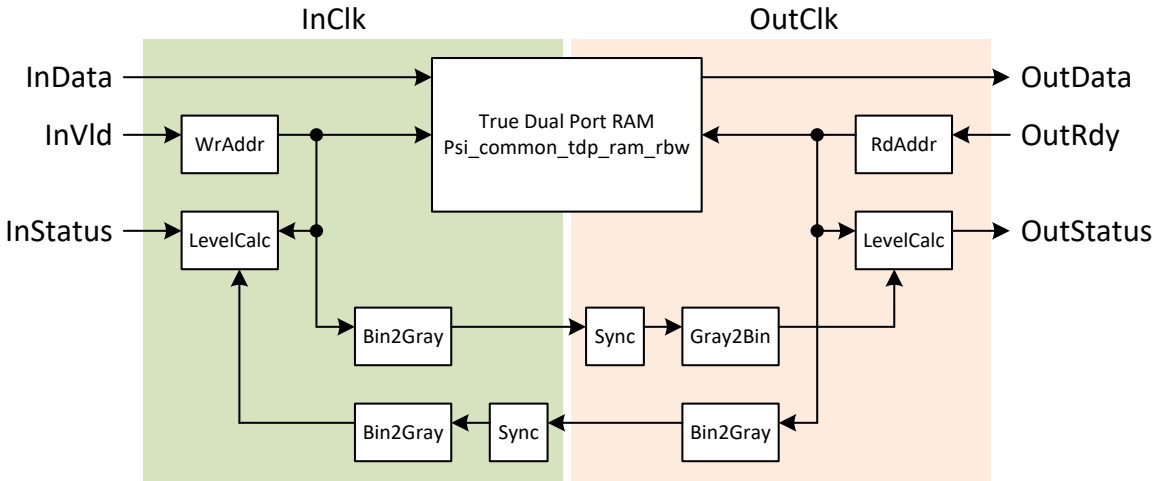

Figure 3: psi_common_async_fifo: Architecture

4.1.5 Constraints

For the FIFO to work correctly, signals from one clock domain to the other must be constrained to have not more delay that one clock cycle of the faster clock.

Example with a 100 MHz clock (10.0 ns period) and a 33.33 MHz clock (30 ns period) for Vivado:

```
set_max_delay -datapath_only -from <ClkA> -to <ClkB> 10.0
set_max_delay -datapath_only -from <ClkB> -to <ClkA> 10.0
```

4.2 psi_common_sync_fifo

4.2.1 Description

This component implements a synchronous FIFO (same clock for write and read port). The memory is described in a way that it utilizes RAM resources (Block-RAM or distributed RAM) available in FPGAs with commonly used tools.

The FIFO is a fall-through FIFO and has AXI-S interfaces on read and write side.

The RAM behavior (read-before-write or write-before-read) can be selected. This allows efficiently implementing FIFOs for different technologies (some technologies implement one, some the other behavior).

4.2.2 Generics

Width_g	Width of the FIFO	
Depth_g	Depth of the FIFO	
AlmFullOn_g	True = Almost full output is provided, False = Almost full output is omitted	
AlmFullLevel_g	Almost full output is high if the level is >= AlmFullLevel_g	
AlmEmptyOn_g	True = Almost empty output is provided, False = Almost empty output is omitted	
AlmEmptyLevel_g	Almost empty output is high if the level is <= AlmFullLevel_g	
RamStyle_g	"auto" (default)	Automatic choice of block- or distributed-RAM
	"distributed"	Use distributed RAM (LUT-RAM)
	"block"	Use block RAM
RamBehavior_g	"RBW"	Read-before-write implementation
	"WBR"	Write-before-read implementation

4.2.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset input (active high)
Input Data (InClk domain)			
InData	Input	Width_g	Write data
InVld	Input	1	AXI-S handshaking signal
InRdy	Output	1	AXI-S handshaking signal
Output Data (OutClk domain)			
OutData	Output	Width_g	Read data
OutVld	Output	1	AXI-S handshaking signal
OutRdy	Input	1	AXI-S handshaking signal

Input Status (InClk domain)			
InFull	Output	1	FIFO full signal synchronous to *InClk*
InEmpty	Output	1	FIFO empty signal synchronous to *InClk*
InAlmFull	Output	1	FIFO almost full signal synchronous to *InClk* Only exists if *AlmFullOn_g* = true
InAlmEmpty	Output	1	FIFO almost empty signal synchronous to *InClk* Only exists if *AlmEmptyOn_g* = true
InLevel	Output	ceil(log2(Depth_g))+1	FIFO level synchronous to *InClk*
Output Status (OutClk domain)			
OutFull	Output	1	FIFO full signal synchronous to *OutClk*
OutEmpty	Output	1	FIFO empty signal synchronous to *OutClk*
OutAlmFull	Output	1	FIFO almost full signal synchronous to *OutClk* Only exists if *AlmFullOn_g* = true
OutAlmEmpty	Output	1	FIFO almost empty signal synchronous to *OutClk* Only exists if *AlmEmptyOn_g* = true
OutLevel	Output	ceil(log2(Depth_g))+1	FIFO level synchronous to *OutClk*

5 Clock Crossings

5.1 psi_common_pulse_cc

5.1.1 Description

This component implements a clock crossing for transferring single pulses from one clock domain to another (completely asynchronous clocks).

The entity shall only be used for single-cycle pulses and the pulse frequency must be lower than the frequency of the slower clock for it to work correctly.

The entity does only guarantee that all pulses arrive at the destination clock domain. It does not guarantee that pulses that occur in the same clock cycle on the source clock domain, occur on the target clock domain in the same clock cycle. As a result it should only be used to do clock-crossings for individual pulses.

This entity does also do the clock-crossing for the reset by using “asynchronously assert, synchronously de-assert” synchronizer chains and applying all attributes to synthesize them correctly.

5.1.2 Generics

NumPulses_g Width of the FIFO

5.1.3 Interfaces

Signal	Direction	Width	Description
Clock Domain A			
ClkA	Input	1	Clock A
RstInA	Input	1	Clock domain A reset input (active high)
RstOutA	Output	1	Clock domain A reset output (active high) - active if *RstInA* or *RstInB* is asserted - de-asserted synchronously to *ClkA*
PulseA	Input	NumPulses_g	Input of the pulse signals
Clock Domain B			
ClkB	Input	1	Clock B
RstInB	Input	1	Clock domain A reset input (active high)
RstOutB	Output	1	Clock domain B reset output (active high) - active if *RstInA* or *RstInB* is asserted - de-asserted synchronously to *ClkA*
PulseB	Output	NumPulses_g	Output of the pulse signals

5.1.4 Architecture

The figure below shows how the pulses are transferred from one clock domain to the other.

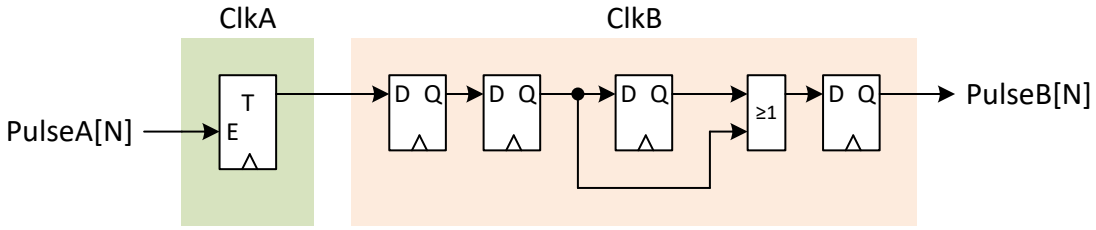

Figure 4: psi_common_pulse_cc: handling of pulses

Since each pulse is handles separately, the pulse alignment may change because of the clock crossing. This is shown in the figure below.

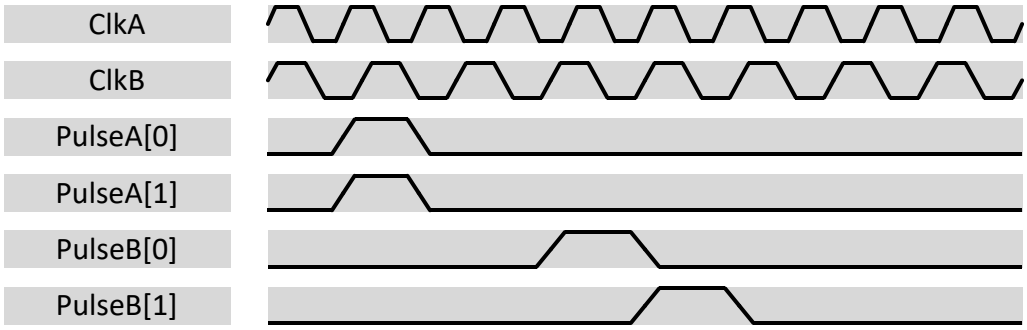

Figure 5: psi_common_pulse_cc: alignment of pulses can change

The figure below shows how the reset signal is transferred from one clock domain to the other. This concept is used to transfer resets in both directions between the clock domains but for simplicity only one direction is shown in the figure.

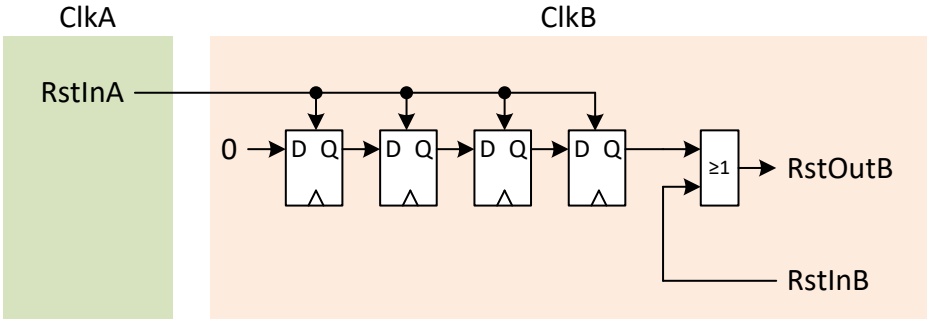

Figure 6: psi_common_pulse_cc: handling of resets

5.1.5 Constraints

This entity does not require any constraints.

5.2 psi_common_simple_cc

5.2.1 Description

This component implements a clock crossing for transferring single values from one clock domain to another (completely asynchronous clocks). In both clock domains the valid samples are marked with a *Vld* signal according to the AXI-S specification but back-pressure (*Rdy*) is not handled.

For the entity to work correctly, the data-rate must be significantly lower (4x lower) than the destination clock frequency.

This entity does also do the clock-crossing for the reset by using “asynchronously assert, synchronously de-assert” synchronizer chains and applying all attributes to synthesize them correctly.

5.2.2 Generics

Width_g Width of the data signal to implement the clock crossing for

5.2.3 Interfaces

Signal	Direction	Width	Description
Clock Domain A			
ClkA	Input	1	Clock A
RstInA	Input	1	Clock domain A reset input (active high)
RstOutA	Output	1	Clock domain A reset output (active high) - active if *RstInA* or *RstInB* is asserted - de-asserted synchronously to *ClkA*
DataA	Input	Width_g	Data signal input
VldA	Input	1	AXI-S handshaking signal
Clock Domain B			
ClkB	Input	1	Clock B
RstInB	Input	1	Clock domain A reset input (active high)
RstOutB	Output	1	Clock domain B reset output (active high) - active if *RstInA* or *RstInB* is asserted - de-asserted synchronously to *ClkA*
DataB	Output	Width_g	Data signal output
VldB	Output	1	AXI-S handshaking signal

5.2.4 Architecture

The concept of this clock crossing is to use *psi_common_pulse_cc* for the clock crossing of the valid signal and latch the data signal on in both clock domains when it is valid. Since the data signal stays stored on the source clock domain, it if for sure valid when the *Vld* signal arrives at the destination clock domain.

To ensure the clock crossing works, the next *Vld* signal is only allowed to arrive after the last one was processed. This is the reason for the maximum data rate allowed being limited to one quarter of the destination clock frequency.

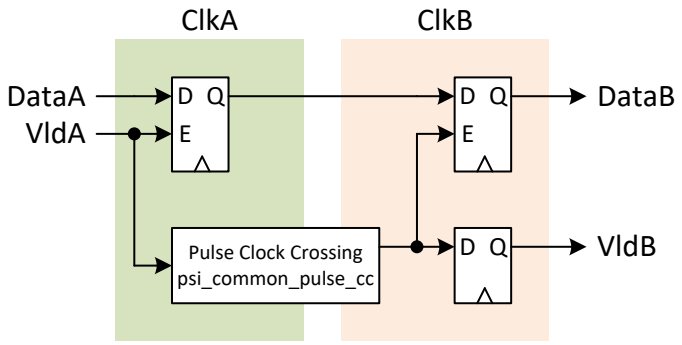

Figure 7: psi_common_simple_cc: Architecture

For details about the reset clock crossing, refer to 5.1.4.

5.2.5 Constraints

For the entity to work correctly, signals from the source clock domain to the destination clock domain must be constrained to have not more delay that one clock cycle of the destination clock.

Example with a 100 MHz source clock (10.0 ns period) and a 33.33 MHz destination clock (30 ns period) for Vivado:

```
set_max_delay -datapath_only -from <ClkA> -to <ClkB> 30.0
```

5.3 psi_common_status_cc

5.3.1 Description

This component implements a clock crossing for slowly changing status information that does not have exact sample rates. It can for example be used to transfer a buffer fill level from one clock domain to another with minimal effort.

The entity ensures that data from the source clock domain is correctly transferred to the destination clock domain. The value at the destination clock domain is always correct in terms of “either the last transferred value or the next one”. The exact timing of the sampling points at which the data is transferred is generated by the entity itself, so it is unknown to the user. As a result, the entity does not guarantee to show transfer state of the data signal in the source clock domain to the destination clock domain in cases of fast changing signals.

For the entity to work correctly, the data-rate must be significantly lower (10 x lower) than the slower clock frequency. Of course the signal can change more quickly but the clock crossing will skip some values in this case.

This entity does also do the clock-crossing for the reset by using “asynchronously assert, synchronously de-assert” synchronizer chains and applying all attributes to synthesize them correctly.

5.3.2 Generics

Width_g Width of the data signal to implement the clock crossing for

5.3.3 Interfaces

Signal	Direction	Width	Description
Clock Domain A			
ClkA	Input	1	Clock A
RstInA	Input	1	Clock domain A reset input (active high)
RstOutA	Output	1	Clock domain A reset output (active high) - active if *RstInA* or *RstInB* is asserted - de-asserted synchronously to *ClkA*
DataA	Input	Width_g	Data signal input
Clock Domain B			
ClkB	Input	1	Clock B
RstInB	Input	1	Clock domain A reset input (active high)
RstOutB	Output	1	Clock domain B reset output (active high) - active if *RstInA* or *RstInB* is asserted - de-asserted synchronously to *ClkA*
DataB	Output	Width_g	Data signal output

5.3.4 Architecture

The concept of this clock crossing is to use *psi_common_simple_cc* for the data signal, so the main functionality of this entity is to automatically generate valid pulses.

The first *Vld* pulse is generated in clock domain *ClkA* after the reset. At this point, the data is sampled and transferred to *ClkB*. When the *Vld* pulse arrives at *ClkB*, it is transferred back to *ClkA* and the next data word is transferred. This setup ensures that *Vld* pulses are generated at a rate that allows the *psi_common_simiple_cc* to transfer the data correctly without any knowledge about the frequencies of both clock domains.

The concept is shown in the figure below.

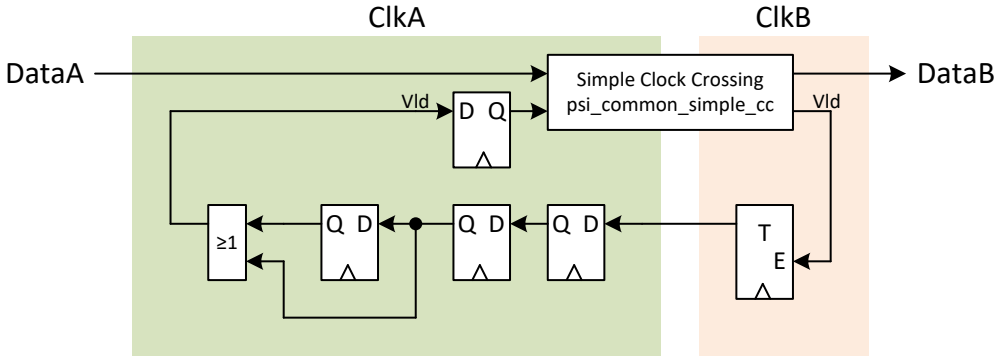

Figure 8: psi_common_status_cc: Architecture

5.3.5 Constraints

For the entity to work correctly, signals from the source clock domain to the destination clock domain must be constrained to have not more delay that one clock cycle of the destination clock.

Example with a 100 MHz source clock (10.0 ns period) and a 33.33 MHz destination clock (30 ns period) for Vivado:

```
set_max_delay -datapath_only -from <ClkA> -to <ClkB> 30.0
```

5.4 psi_common_sync_cc_n2xn

5.4.1 Description

This component implements a clock crossing with AXI-S handshaking for transferring data from one clock domain to another one that runs at an integer multiple of the frequency of the input clock frequency. It can for example be used to transfer data from a 50 MHz clock domain to a 100 MHz clock domain (both generated by the same PLL).

5.4.2 Generics

Width_g Width of the data signal to implement the clock crossing for

5.4.3 Interfaces

Signal	Direction	Width	Description
Input			
InClk	Input	1	Input side clock
InRst	Input	1	Input side reset
InVld	Input	1	AXI-S handshaking signal
InRdy	Output	1	AXI-S handshaking signal
InData	Input	Width_g	Data signal input
Output			
OutClk	Input	1	Output side clock
OutRst	Input	1	Output side reset
OutVld	Output	1	AXI-S handshaking signal
OutRdy	Input	1	AXI-S handshaking signal
OutData	Output	Width_g	Data signal output

5.4.4 Constraints

Constraints are derived by the tools automatically since the clocks are synchronous. Therefore no user constraints are required.

5.5 psi_common_sync_cc_xn2n

5.5.1 Description

This component implements a clock crossing with AXI-S handshaking for transferring data from one clock domain to another one that runs at an integer fractional of the frequency of the input clock frequency. It can for example be used to transfer data from a 100 MHz clock domain to a 50 MHz clock domain (both generated by the same PLL).

5.5.2 Generics

Width_g Width of the data signal to implement the clock crossing for

5.5.3 Interfaces

Signal	Direction	Width	Description
Input			
InClk	Input	1	Input side clock
InRst	Input	1	Input side reset
InVld	Input	1	AXI-S handshaking signal
InRdy	Output	1	AXI-S handshaking signal
InData	Input	Width_g	Data signal input
Output			
OutClk	Input	1	Output side clock
OutRst	Input	1	Output side reset
OutVld	Output	1	AXI-S handshaking signal
OutRdy	Input	1	AXI-S handshaking signal
OutData	Output	Width_g	Data signal output

5.5.4 Constraints

Constraints are derived by the tools automatically since the clocks are synchronous. Therefore no user constraints are required.

5.6 psi_common_bit_cc

5.6.1 Description

This component implements a clock crossing for multiple independent single-bit signals. It contains double-stage synchronizers and sets all the attributes required for proper synthesis.

Note that this clock crossing does not guarantee that all bits arrive in the same clock cycle at the destination clock domain, therefore it can only be used for independent single-bit signals.

5.6.2 Generics

NumBits_g Number of data bits to implement the clock crossing for

5.6.3 Interfaces

Signal	Direction	Width	Description
Input			
BitsA	Input	NumBits_g	Input signals
Output			
ClkB	Input	1	Destination clock
BitsB	Output	NumBits_g	Output signals

5.6.4 Constraints

No special constraints are required (only the period of the output clock.

5.7 Other Components that can be used as Clock Crossings

- psi_common_tdp_ram (see 3.3)
- psi_common_async_fifo (see 4.1)

6 Timing

6.1 psi_common_strobe_generator

6.1.1 Description

This component generates a strobe (pulse signal with 1 clock cycle pulse-width) at a compile-time configurable frequency. Clock frequency and strobe frequency can be passed as generics.

Optionally the strobe generation can be synchronized to an external signal.

6.1.2 Generics

freq_clock_g Frequency of the clock in Hz
freq_strobe_g Frequency of the strobe output in Hz
rst_pol_g Reset polarity (‘1’ = high active)

6.1.3 Interfaces

Signal	Direction	Width	Description
InClk	Input	1	Clock
InRst	Input	1	Reset input
InSync	Input	1	Synchronization signal (optional)
OutVld	Output	1	Strobe output

6.1.4 Synchronization

The strobe synchronization is optional. If no synchronization is required, *sync_i* can be left unconnected or tied to ‘0’.

When strobe synchronization is used, the strobe signal is synchronized to rising edges detected on the *sync_i* input. If a rising edge is detected on the *sync_i* input, a strobe is generated in the next cycle. From there, the strobe is running at the normal frequency.

The figure below shows the behavior of strobe synchronization for a strobe output at ¼ of the clock frequency.

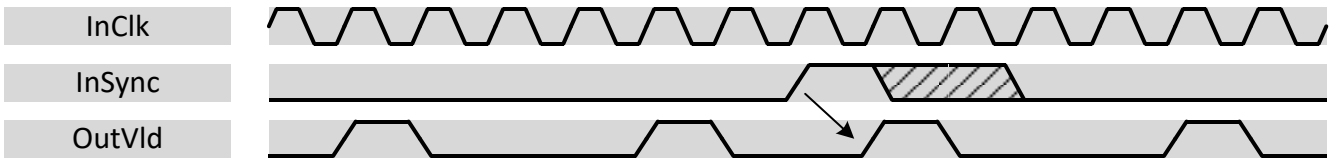

Figure 9: psi_common_strobe_generator: Strobe synchronization

6.2 psi_common_strobe_divider

6.2.1 Description

This component divides the rate of a strobe signal. Only every N^{th} strobe signal is forwarded to the output. If the input is not a single cycle strobe signal, a rising edge detection is done (strobe is detected on the first cycle the input is high).

The division ratio is selectable at runtime.

6.2.2 Generics

length_g Width of the *ratio_i* input in bits
rst_pol_g Reset polarity ('1' = high active)

6.2.3 Interfaces

Signal	Direction	Width	Description
InClk	Input	1	Clock
InRst	Input	1	Reset input
InVld	Input	1	Strobe input
InRatio	Input	length_g	Division ratio (1 = no division, 2 = division by 2)
OutVld	Output	1	Strobe output

6.3 psi_common_tickgenerator

6.3.1 Description

This component generated pulses at the commonly used time bases in a system (second, millisecond, microsecond) based on the clock frequency. The width of the tick-pulses is configurable.

6.3.2 Generics

g_CLK_IN_MHZ	Clock frequency in MHz
g_TICK_WIDTH	Pulse-width of the tick outputs
g_SIM_SEC_SPEEDUP_FACTOR	Factor to speedup the second tick in simulations (reduction of simulation runtimes). For implementation this generic must be set to 1.

6.3.3 Interfaces

Signal	Direction	Width	Description
clock_i	Input	1	Clock input
tick1us_o	Output	1	Microsecond tick output
tick1ms_o	Output	1	Millisecond tick output
tick1sec_o	Output	1	Second tick output

6.4 psi_common_pulse_shaper

6.4.1 Description

This component creates pulses of an exactly known length from pulses with unknown length. Additionally it can limit the maximum pulse rate by applying a hold-off time.

Input pulses are detected based on their rising edge.

The figure below shows an example behavior for *Duration_g=3* and *HoldOff_g=4*. The first pulse is stretched to three cycles, the second pulse is ignored because it is within the hold-off time and the third pulse is shortened to three cycles.

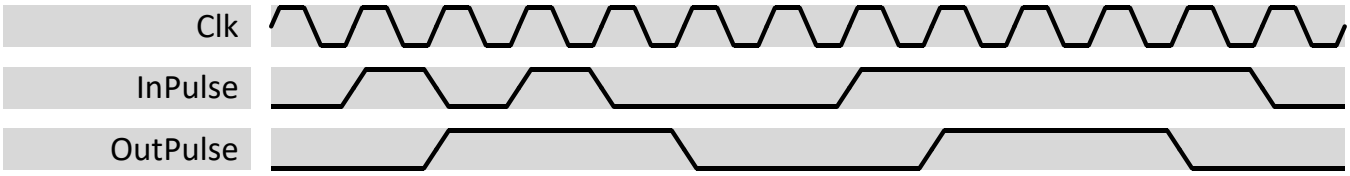

Figure 10: psi_common_pulse_shaper: Example waveform

6.4.2 Generics

Duration_g Duration of the output pulse in clock cycles
HoldOff_g Minimum time between input pulse-rising-edges that are detected (in clock cycles)
Pulses arriving during the hold-off time are ignored.

6.4.3 Interfaces

Signal	Direction	Width	Description
Clk	Input	1	Clock input
Rst	Input	1	Reset input
InPulse	Input	1	Input Pulse
OutPulse	Output	1	Output Pulse

6.5 psi_common_clk_meas

6.5.1 Description

This component measures the clock (*ClkTest*) under the assumption that a second clock (*ClkMaster*) runs at a known frequency.

6.5.2 Generics

MasterFrequency_g	Clock frequency of the master clock in Hz
MaxMeasFrequency_g	Maximum supported frequency for *ClkTest*

6.5.3 Interfaces

Signal	Direction	Width	Description
ClkMaster	Input	1	Master input clock
Rst	Input	1	Reset (synchronous to *ClkMaster*)
ClkTest	Input	1	Test input clock
FrequencyHz	Output	32	Clock frequency for *ClkTest* in Hz

7 Conversions

7.1 psi_common_wconv_n2xn

7.1.1 Description

This component implements a data width conversion from N-bits to a multiple of N-bits. The sample rate is reduced accordingly. The width conversion implements AXI-S handshaking signals to handle back-pressure.

The with conversion supports back-to-back conversions (*InVld* can stay high all the time)

The entity does little-endian data alignment as shown in the figure below.

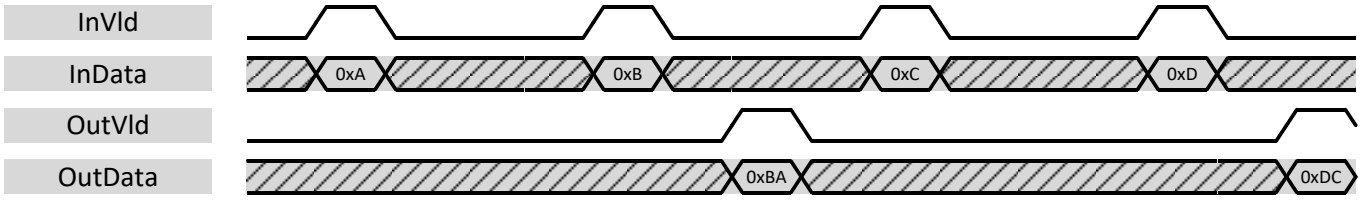

Figure 11: psi_common_wconv_n2xn: Data alignment

This entity does only do a width conversion but not clock crossing. If a half-clock-double-width conversion is used, *psi_common_sync_cc_xn2n* component can be used after the width conversion.

7.1.2 Generics

InWidth_g Input data width
OutWidth_g Output data width

The ratio $\frac{\text{OutWidth\_g}}{\text{InWidth\_g}}$ must be an integer number and *OutWidth_g* must be bigger or equal to *InWidth_g*.

7.1.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset (high active)
Input			
InVld	Input	1	AXI-S handshaking signal
InRdy	Output	1	AXI-S handshaking signal
InData	Input	InWidth_g	Data signal input
Output			
OutVld	Output	1	AXI-S handshaking signal
OutRdy	Input	1	AXI-S handshaking signal
OutData	Output	OutWidth_g	Data signal output

7.2 psi_common_wconv_xn2n

7.2.1 Description

This component implements a data width conversion from a multiple N-bits to a N-bits. The sample rate is increased accordingly. The width conversion implements AXI-S handshaking signals to handle back-pressure.

The width conversion does support back-to-back conversions (*OutVld/OutRdy* can stay high all the time).

The entity does little-endian data alignment as shown in the figure below.

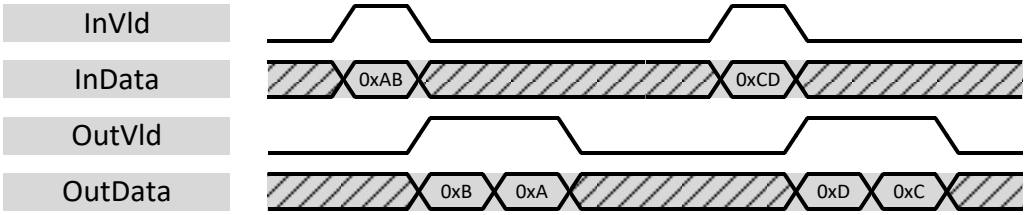

Figure 12: psi_common_wconv_xn2n: Data alignment

This entity does only do a width conversion but not clock crossing. If a double-clock-half-width conversion is used, *psi_common_sync_cc_n2xn* component can be used in front of the width conversion.

7.2.2 Generics

InWidth_g Input data width
OutWidth_g Output data width

The ratio $\frac{\text{InWidth\_g}}{\text{OutWidth\_g}}$ must be an integer number and *InWidth _g* must be bigger or equal to *OutWidth_g*.

7.2.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset (high active)
Input			
InVld	Input	1	AXI-S handshaking signal
InRdy	Output	1	AXI-S handshaking signal
InData	Input	InWidth_g	Data signal input
Output			
OutVld	Output	1	AXI-S handshaking signal
OutRdy	Input	1	AXI-S handshaking signal
OutData	Output	OutWidth_g	Data signal output

8 TDM Handling

8.1 psi_common_par_tdm

8.1.1 Description

This component changes the representation of multiple channels from parallel to time-division-multiplexed. It does not implement any flow-control, so the user is responsible to not apply input data faster than it can be represented at the output (time-division-multiplexed).

The figure below shows some waveforms of the conversion. The lowest bits of the input vector are interpreted as channel 0 and played out first, the highest bits of the input vector are played out last.

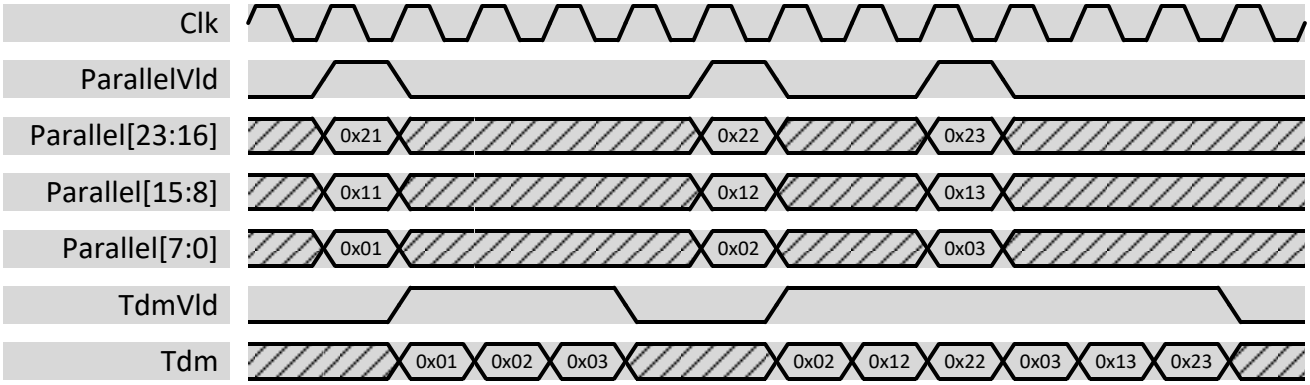

Figure 13: psi_common_par_tdm: Waveform

8.1.2 Generics

ChannelCount_g	Number of channels
ChannelWidth_g	Number of bits per channel

8.1.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset (high active)
Input			
ParallelVld	Input	1	AXI-S handshaking signal
Parallel	Input	ChannelCount_g*ChannelWidth_g	Data of all channels in parallel. Channel 0 is in the lowest bit and played out first.
Output			
TdmVld	Output	1	AXI-S handshaking signal
Tdm	Output	ChannelWidth	Data signal output

8.2 psi_common_tdm_par

8.2.1 Description

This component changes the representation of multiple channels from time-division-multiplexed to parallel. It does not implement any flow-control.).

The figure below shows some waveforms of the conversion. The first input sample is interpreted as channel 0 and played out in the lowest bits of the output, the last input sample is played out in the highest bits.

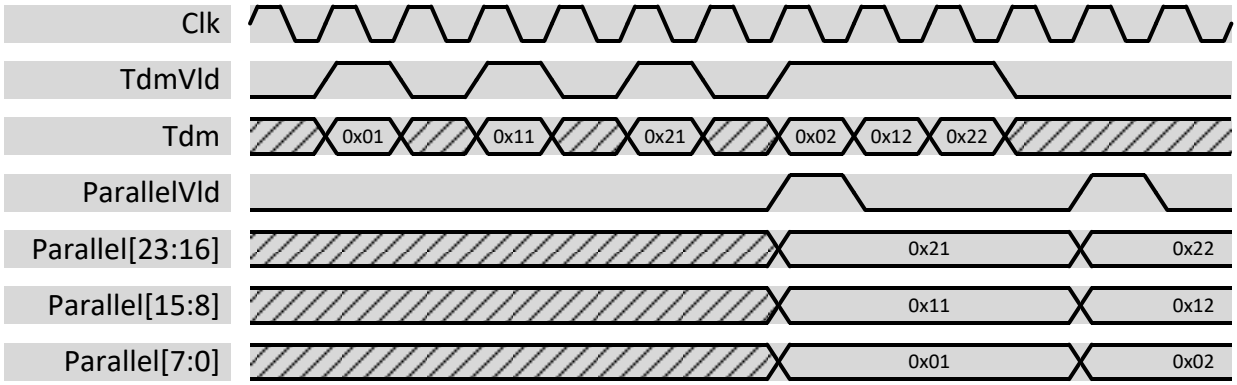

Figure 14: psi_common_tdm_par: Waveform

Note that the output stays stable also after the *Vld* pulse.

8.2.2 Generics

ChannelCount_g Number of channels
ChannelWidth_g Number of bits per channel

8.2.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset (high active)
Input			
TdmVld	Input	1	AXI-S handshaking signal
Tdm	Input	ChannelWidth	TDM input signal, first sample is channel 0.
Output			
ParallelVld	Output	1	AXI-S handshaking signal
Parallel	Output	ChannelCount_g*ChannelWidth_g	Data of all channels in parallel. Channel 0 is in the lowest bits.

8.3 psi_common_tdm_mux

8.3.1 Description

This component allows selecting one unique channel over a bunch of “N” time division multiplexed (tdm) data. The output comes with a strobe/valid signal at the falling edge of the “tdm” strobe/valid input with a two clock cycles latency.

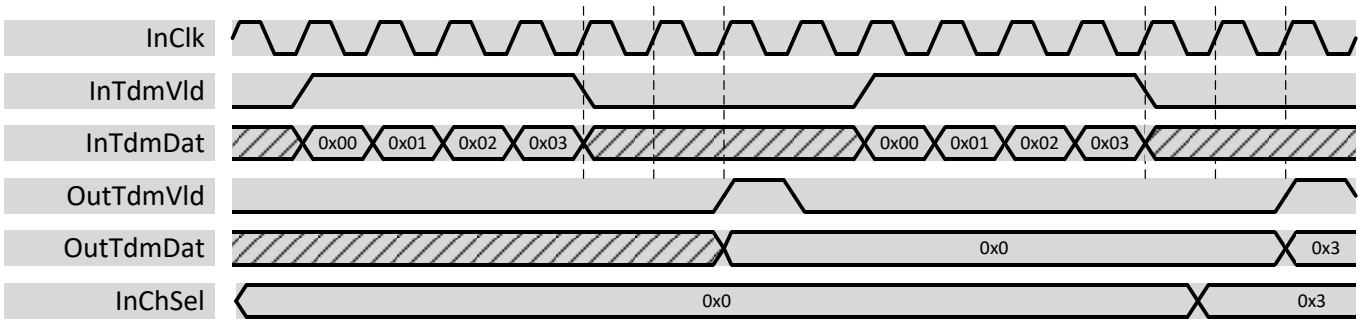

Figure 15 psi_common_tdm_mux: Waveform

8.3.2 Generics

rst_pol_g	reset polarity selection
num_channel_g	Number of channels
data_length_g	Width of the data signals

8.3.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
InClk	Input	1	Clock
InRst	Input	1	Reset
Inputs			
InChSel	Input	Log2ceil(num_channel_g)	Mux select
InTdmVld	Input	1	Strobe/valid input signal *(num_channel_g *clock cycle)*
InTdmDat	Input	data_length_g	Data input
Ouputs			
OutTdmVld	Output	1	AXI-S handshaking signal
OutTdmDat	Output	data_length_g	Data output

9 Arbiters

9.1 psi_common_arb_priority

9.1.1 Description

This entity implements a priority arbiter. The left-most bit (highest bit) of the request vector that was asserted is granted (i.e. asserted in the grant vector). The arbiter is implemented using the very logic- and timing-efficient parallel prefix computation approach.

The arbiter can be implemented with or without an output register. The waveform below shows its implementation without output register (*OutputRegister_g* = *false*), since the delay would make the waveform less easy to read.

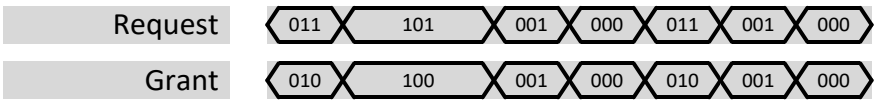

Figure 16: psi_common_arb_priority: Waveform

9.1.2 Generics

Size_g	Size of the arbiter (number of input/output bits)
OutputRegister_g	True = Registered output False = Combinatorial output

9.1.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset (high active)
Data			
Request	Input	*Size_g*	Request input signals The highest (left-most) bit has highest priority
Grant	Output	*Size_g*	Grant output signal

9.1.4 Architecture

Parallel prefix computation is used to calculate a vector that contains a ‘1’ on the highest-priority bit that was asserted and on all bits with lower priority. The vector then looks for example like this “0001111”. The bit to assert in the *Grant* output can then be determined by finding the 0-1 edge inside that vector.

The figure below shows the parallel prefix computation graphically.

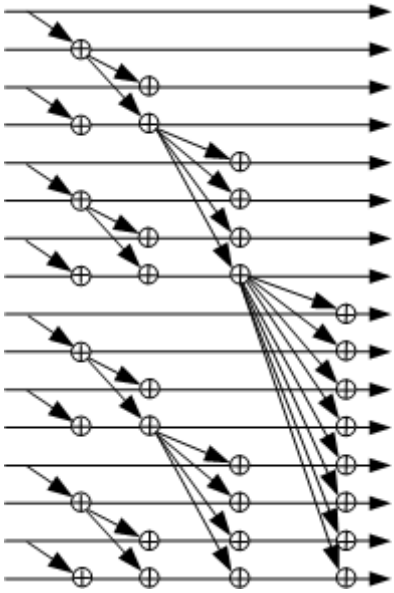

Figure 17: psi_common_arb_priority: Parallel prefix computation (PPC)

9.2 psi_common_arb_round_robin

9.2.1 Description

This entity implements a round-robin arbiter. If multiple bits are asserted in the request vector, the left-most bit is forwarded to the grant vector first. Next, the second left-most bit that is set is forwarded etc. Whenever at least one bit in the *Grant* vector is asserted, the *Grant_Vld* handshaking signal is asserted to signal that a request was granted. The consumer of the *Grant* vector signalizes that the granted access was executed by pulling *Grant_Rdy* high.

Note that the round-robin arbiter is implemented without an output register. Therefore combinatorial paths between input and output exist and it is recommended to add a register-stage to the output as early as possible.

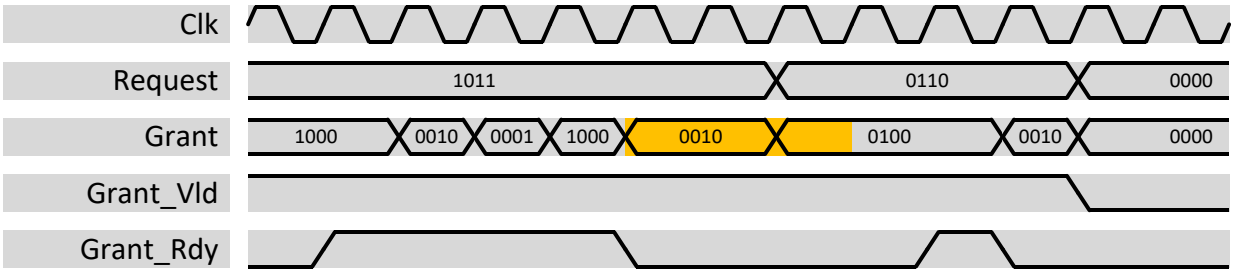

Figure 18: psi_common_arb_round_robin: Waveform

Especially interesting is the part in orange. At this point the arbiter does not grant access to bit 3 because it already granted this request in the clock cycle before. However, it continues to grant access to the highest-priority (i.e. left-most) bit of the request vector that is still left of the bit set in the last *Grant* output. If the request vector asserts a higher priority this change is directly forwarded to the output. This is shown in the orange section of the waveform.

9.2.2 Generics

Size_g Size of the arbiter (number of input/output bits)

9.2.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset (high active)
Data			
Request	Input	*Size_g*	Request input signals The highest (left-most) bit has highest priority
Grant	Output	*Size_g*	Grant output signal
Grant_Vld	Outpu	1	AXI-S handshaking signal Asserted whenever Grant != 0
Grant_Rdy	Input	1	AXI-S handshaking signal The state of the arbiter is updated upon *Grant_Rdy = '1'*

10 Miscellaneous

10.1 psi_common_delay

10.1.1 Description

This component is an efficient implementation for delay chains. It uses FPGA memory resources (Block-RAM and distributed RAM resp. SRLs) for implementing the delays (instead of many FFs). The last delay stage is always implemented in FFs to ensure good timing (RAM outputs are usually slow).

One Problem with using RAM resources to implement delays is that they don't have a reset, so the content of the RAM persists after resetting the logic. The *psi_common_delay* entity works around this issue by some logic that ensures that any persisting data is replaced by zeros after a reset. The replacement is done at the output of the *psi_common_delay*, so no time to overwrite memory cells after a reset is required and the entity is ready to operate on the first clock cycle after the reset.

If the delay is implemented using a RAM, the behavior of the RAM (read-before-write or write-before-read) can be selected to allow efficient implementation independently of the target technology.

10.1.2 Generics

Width_g	Width of the data to delay	
Delay_g	Number of delay taps	
Resource_g	**"**AUTO" (default)	automatically use SRL or BRAM according to *BramThreshold_g*
	*"*BRAM"	use Block RAM to implement the delay taps
	"SRL"	use SRLs (LUTs used as shift registers) to implement the delay taps
BramThreshold_g	This generic controls the resources to use for the delay taps in case *Resource_g = "AUTO".* SRLs are used if *Delay_g* < *BramThreshold_g*. Otherwise BRAMs are used.	
RstState_g	True	Persisting memory content is replaced by zeros after reset
	False	Persisting memory content is visible at output after reset (less resource usage)
RamBehavior_g	"RBW"	Read-before-write implementation
	"WBR"	Write-before-read implementation
	This generic is only used if a BRAM is used for the delay.	

10.1.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset (high active)
Data			
Vld	Input	1	InData valid (clock enable for shift register)
InData	Input	Width_g	Data input
OutData	Output	Width_g	Data output

10.2 psi_common_pl_stage

10.2.1 Description

This component implements a pipeline stage that supports full AXI-S handshaking (including the handling of back-pressure). The pipeline breaks any combinatorial paths on all lines (*Rdy, Vld* and *Data*). So not only the forward signals *Vld* and *Data* are registered but also *Rdy*. This is important since long combinatorial paths are common to occur on the *Rdy* signal (it is often handled combinatorial).

Correct handling of the *Rdy* signal requires some additional resources. Therefore the handling of *Rdy* can be disabled using a generic to reduce resource usage if back-pressure must not be handled.

10.2.2 Generics

Width_g	Width of the data signal	
UseRdy_g	True	Backpressure is handled (*Rdy* is used and pipelined)
	False	Backpressure is not handled (*Rdy* is not connected at all in this case)

10.2.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset (high active)
Input			
InVld	Input	1	AXI-S handshaking signal
InRdy	Output	1	AXI-S handshaking signal
InData	Input	Width_g	Data input
OutVld	Output	1	AXI-S handshaking signal
OutRdy	Input	1	AXI-S handshaking signal
OutData	Output	Width_g	Data output

10.3 psi_common_multi_pl_stage

10.3.1 Description

This component implements allows easily adding multiple pipeline stages to a signal path and maintain full AXI-S handshaking including back-pressure. It does so by chaining multiple *psi_common_pl_stage* (see 10.2) entities.

10.3.2 Generics

Width_g Width of the data signal
UseRdy_g True Backpressure is handled (*Rdy* is used and pipelined)
False Backpressure is not handled (*Rdy* is not connected at all in this case)
Stages_g Number of pipeline stages

10.3.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset (high active)
Input			
InVld	Input	1	AXI-S handshaking signal
InRdy	Output	1	AXI-S handshaking signal
InData	Input	Width_g	Data input
OutVld	Output	1	AXI-S handshaking signal
OutRdy	Input	1	AXI-S handshaking signal
OutData	Output	Width_g	Data output