**psi\_common**

Documentation

**Content**

**Table of Contents**

[1 Introduction 5](#_Toc527457989)

[1.1 Working Copy Structure 5](#_Toc527457990)

[1.2 VHDL Libraries 5](#_Toc527457991)

[1.3 Running Simulations 6](#_Toc527457992)

[1.4 Contribute to PSI VHDL Libraries 7](#_Toc527457993)

[1.5 Handshaking Signals 8](#_Toc527457994)

[2 Packages 9](#_Toc527457995)

[2.1 psi\_common\_array\_pkg 9](#_Toc527457996)

[2.2 psi\_common\_logic\_pkg 9](#_Toc527457997)

[2.3 psi\_common\_axi\_pkg 9](#_Toc527457998)

[2.4 psi\_common\_math\_pkg 9](#_Toc527457999)

[3 Memories 10](#_Toc527458000)

[3.1 psi\_common\_sdp\_ram 10](#_Toc527458001)

[3.2 psi\_common\_sp\_ram\_be 11](#_Toc527458002)

[3.3 psi\_common\_tdp\_ram 12](#_Toc527458003)

[4 FIFOs 13](#_Toc527458004)

[4.1 psi\_common\_async\_fifo 13](#_Toc527458005)

[4.2 psi\_common\_sync\_fifo 16](#_Toc527458006)

[5 Clock Crossings 18](#_Toc527458007)

[5.1 psi\_common\_pulse\_cc 18](#_Toc527458008)

[5.2 psi\_common\_simple\_cc 20](#_Toc527458009)

[5.3 psi\_common\_status\_cc 22](#_Toc527458010)

[5.4 psi\_common\_sync\_cc\_n2xn 24](#_Toc527458011)

[5.5 psi\_common\_sync\_cc\_xn2n 25](#_Toc527458012)

[5.6 psi\_common\_bit\_cc 26](#_Toc527458013)

[5.7 Other Components that can be used as Clock Crossings 26](#_Toc527458014)

[6 Timing 27](#_Toc527458015)

[6.1 psi\_common\_strobe\_generator 27](#_Toc527458016)

[6.2 psi\_common\_strobe\_divider 28](#_Toc527458017)

[6.3 psi\_common\_tickgenerator 29](#_Toc527458018)

[6.4 psi\_common\_pulse\_shaper 30](#_Toc527458019)

[6.5 psi\_common\_clk\_meas 31](#_Toc527458020)

[7 Conversions 32](#_Toc527458021)

[7.1 psi\_common\_wconv\_n2xn 32](#_Toc527458022)

[7.2 psi\_common\_wconv\_xn2n 33](#_Toc527458023)

[8 TDM Handling 34](#_Toc527458024)

[8.1 psi\_common\_par\_tdm 34](#_Toc527458025)

[8.2 psi\_common\_tdm\_par 35](#_Toc527458026)

[8.3 psi\_common\_tdm\_mux 36](#_Toc527458027)

[9 Arbiters 37](#_Toc527458028)

[9.1 psi\_common\_arb\_priority 37](#_Toc527458029)

[9.2 psi\_common\_arb\_round\_robin 39](#_Toc527458030)

[10 Miscellaneous 40](#_Toc527458031)

[10.1 psi\_common\_delay 40](#_Toc527458032)

[10.2 psi\_common\_pl\_stage 41](#_Toc527458033)

[10.3 psi\_common\_multi\_pl\_stage 42](#_Toc527458034)

**Figures**

[Figure 1: Working copy structure 5](#_Toc527458035)

[Figure 2: Handshaking signals 8](#_Toc527458036)

[Figure 3: psi\_common\_async\_fifo: Architecture 14](#_Toc527458037)

[Figure 4: psi\_common\_pulse\_cc: handling of pulses 19](#_Toc527458038)

[Figure 5: psi\_common\_pulse\_cc: alignment of pulses can change 19](#_Toc527458039)

[Figure 6: psi\_common\_pulse\_cc: handling of resets 19](#_Toc527458040)

[Figure 7: psi\_common\_simple\_cc: Architecture 21](#_Toc527458041)

[Figure 8: psi\_common\_status\_cc: Architecture 23](#_Toc527458042)

[Figure 9: psi\_common\_strobe\_generator: Strobe synchronization 27](#_Toc527458043)

[Figure 10: psi\_common\_pulse\_shaper: Example waveform 30](#_Toc527458044)

[Figure 11: psi\_common\_wconv\_n2xn: Data alignment 32](#_Toc527458045)

[Figure 12: psi\_common\_wconv\_xn2n: Data alignment 33](#_Toc527458046)

[Figure 13: psi\_common\_par\_tdm: Waveform 34](#_Toc527458047)

[Figure 14: psi\_common\_tdm\_par: Waveform 35](#_Toc527458048)

[Figure 15 psi\_common\_tdm\_mux: Waveform 36](#_Toc527458049)

[Figure 16: psi\_common\_arb\_priority: Waveform 37](#_Toc527458050)

[Figure 17: psi\_common\_arb\_priority: Parallel prefix computation (PPC) 38](#_Toc527458051)

[Figure 18: psi\_common\_arb\_round\_robin: Waveform 39](#_Toc527458052)

# 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

## 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.

## 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:

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

## Running Simulations

### Regression Test

#### 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.

#### 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

### 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 vor GHDL are the same, just in the TCL interpreter shall instead of the Modelsim TCL console.

## 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.

## Handshaking Signals

### General Information

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

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

The most important points of the specification are outlined below.

### Excerpt of the AXI-S Standard

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

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

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



Figure 2: Handshaking signals

### Naming

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

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

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

TREADY Rdy, InRdy, OutRdy

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

# Packages

## psi\_common\_array\_pkg

### 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.

## psi\_common\_logic\_pkg

### Description

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

## psi\_common\_axi\_pkg

### 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.

## psi\_common\_math\_pkg

### 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.

# Memories

## psi\_common\_sdp\_ram

### 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).

### 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

### 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 |

## psi\_common\_sp\_ram\_be

### 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).

### 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

### 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 |

## psi\_common\_tdp\_ram

### 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).

### 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

### 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 |

### 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

# FIFOs

## psi\_common\_async\_fifo

### 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).

### 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

### 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* |

### 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

### 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

## psi\_common\_sync\_fifo

### 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).

### 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

### 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* |

# Clock Crossings

## psi\_common\_pulse\_cc

### 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.

### Generics

**NumPulses\_g** Width of the FIFO

### 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 |

### 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

### Constraints

This entity does not require any constraints.

## psi\_common\_simple\_cc

### 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.

### Generics

**Width\_g** Width of the data signal to implement the clock crossing for

### 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 |

### 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.

### 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

## psi\_common\_status\_cc

### 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.

### Generics

**Width\_g** Width of the data signal to implement the clock crossing for

### 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 |

### 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

### 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

## psi\_common\_sync\_cc\_n2xn

### 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).

### Generics

**Width\_g** Width of the data signal to implement the clock crossing for

### 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 |

### Constraints

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

## psi\_common\_sync\_cc\_xn2n

### 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).

### Generics

**Width\_g** Width of the data signal to implement the clock crossing for

### 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 |

### Constraints

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

## psi\_common\_bit\_cc

### 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.

### Generics

**NumBits\_g** Number of data bits to implement the clock crossing for

### Interfaces

|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Direction | Width | Description |
| ***Input*** | | | |
| BitsA | Input | NumBits\_g | Input signals |
| ***Output*** | | | |
| ClkB | Input | 1 | Destination clock |
| BitsB | Output | NumBits\_g | Output signals |

### Constraints

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

## Other Components that can be used as Clock Crossings

* psi\_common\_tdp\_ram (see 3.3)
* psi\_common\_async\_fifo (see 4.1)

# Timing

## psi\_common\_strobe\_generator

### 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.

### 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)

### 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 |

### 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

## psi\_common\_strobe\_divider

### Description

This component divides the rate of a strobe signal. Only every Nth 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.

### Generics

**length\_g** Width of the *ratio\_i* input in bits  
**rst\_pol\_g** Reset polarity (‘1’ = high active)

### 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 |

## psi\_common\_tickgenerator

### 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.

### 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.

### 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 |

## psi\_common\_pulse\_shaper

### 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

### 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.

### 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 |

## psi\_common\_clk\_meas

### Description

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

### Generics

**MasterFrequency\_g** Clock frequency of the master clock in Hz  
**MaxMeasFrequency\_g** Maximum supported frequency for *ClkTest*

### 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 |

# Conversions

## psi\_common\_wconv\_n2xn

### 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.

### Generics

**InWidth\_g** Input data width  
**OutWidth\_g** Output data width

The ratio must be an integer number and *OutWidth\_g* must be bigger or equal to *InWidth\_g*.

### 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 |

## psi\_common\_wconv\_xn2n

### 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.

### Generics

**InWidth\_g** Input data width  
**OutWidth\_g** Output data width

The ratio must be an integer number and *InWidth \_g* must be bigger or equal to *OutWidth\_g*.

### 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 |

# TDM Handling

## psi\_common\_par\_tdm

### 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

### Generics

**ChannelCount\_g** Number of channels  
**ChannelWidth\_g** Number of bits per channel

### 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 |

## psi\_common\_tdm\_par

### 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.

### Generics

**ChannelCount\_g** Number of channels  
**ChannelWidth\_g** Number of bits per channel

### 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. |

## psi\_common\_tdm\_mux

### 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

### Generics

**rst\_pol\_g** reset polarity selection  
**num\_channel\_g** Number of channels  
**data\_length\_g** Width of the data signals

### 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 |

# Arbiters

## psi\_common\_arb\_priority

### 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

### Generics

**Size\_g** Size of the arbiter (number of input/output bits)  
**OutputRegister\_g** True = Registered output  
 False = Combinatorial output

### 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 |

### 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)

## psi\_common\_arb\_round\_robin

### 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.

### Generics

**Size\_g** Size of the arbiter (number of input/output bits)

### 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’* |

# Miscellaneous

## psi\_common\_delay

### 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.

### 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.

### 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 |

## psi\_common\_pl\_stage

### 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.

### 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)

### 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 |

## psi\_common\_multi\_pl\_stage

### 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.

### 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

### 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 |