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.

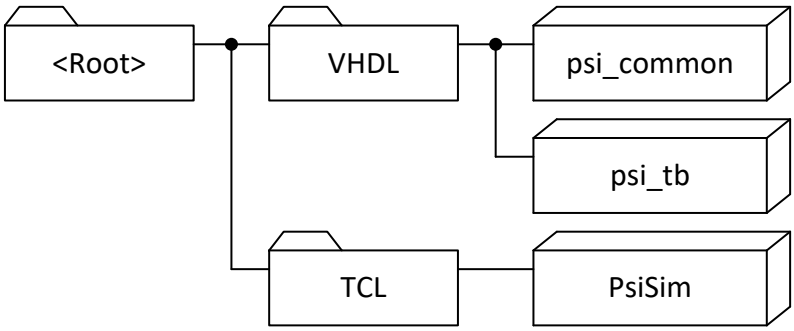


Figure 1: Working copy structure

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

1.3.1 Regression Test

1.3.1.1 Modelsim

To run the regression test, follow the steps below:

- Open Modelsim
- The TCL console, navigate to *<Root>/VHDL/psi_common/sim*
- Execute the command "*source ./run.tcl*"

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

1.3.1.2 GHDL

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

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

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

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

1.3.2 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>*
    - *launch_tb –contains <string>*

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

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.
  - Highlight all documentation changes in feature branches in yellow so they can be found easily when merging back to master.

- New test-benches must be added to the regression test-script
  - Change */sim/config.tcl* accordingly
  - Test if the regression test really runs the new test-bench and exits without errors before doing any merge requests.

- GIT commit annotations, please add a short description at first of your commit annotation, this ease the maintainer to merge, write the changelog.md file while doing new release and others to understand what has been committed, here below there are 6 inputs that are commonly use.
  - ***FEATURE***: Adding a new feature to library like a component or a package element (i.e. function, procedure, type and constant)
  - ***GIT***: when committing some GIT related issue, merging, etc…
  - ***BUGFIX***: when a fix has been made
  - ***DOCU***: documentation related comment
  - ***DEVEL***: when a commit is done but work is in development
  - ***TB***: test bench related commit

- Working with GIT
  - If a user wants to participate to the library he is free to do so, however some rules should be considered. All PSI libraries have at least two branches develop & master. The master branch is used for stable release version all changes are merge to master when required. The develop branch is the branch when a GIT user shall diverge from to add a new component.
  - A good practice is to call the branch the name of the new block, the user is free to push into this branch and once the work is over a pull request can be done to the base branch develop (Step 1). Members of the library may then have exchange and add comment during a code review (Step 2). The initial user takes into consideration the comment and once bug are fixed for instance a new pull request can be done. If members agree on the new feature then it is merged to develop (Step 3) and the branch will be safely deleted by the repository maintainer.

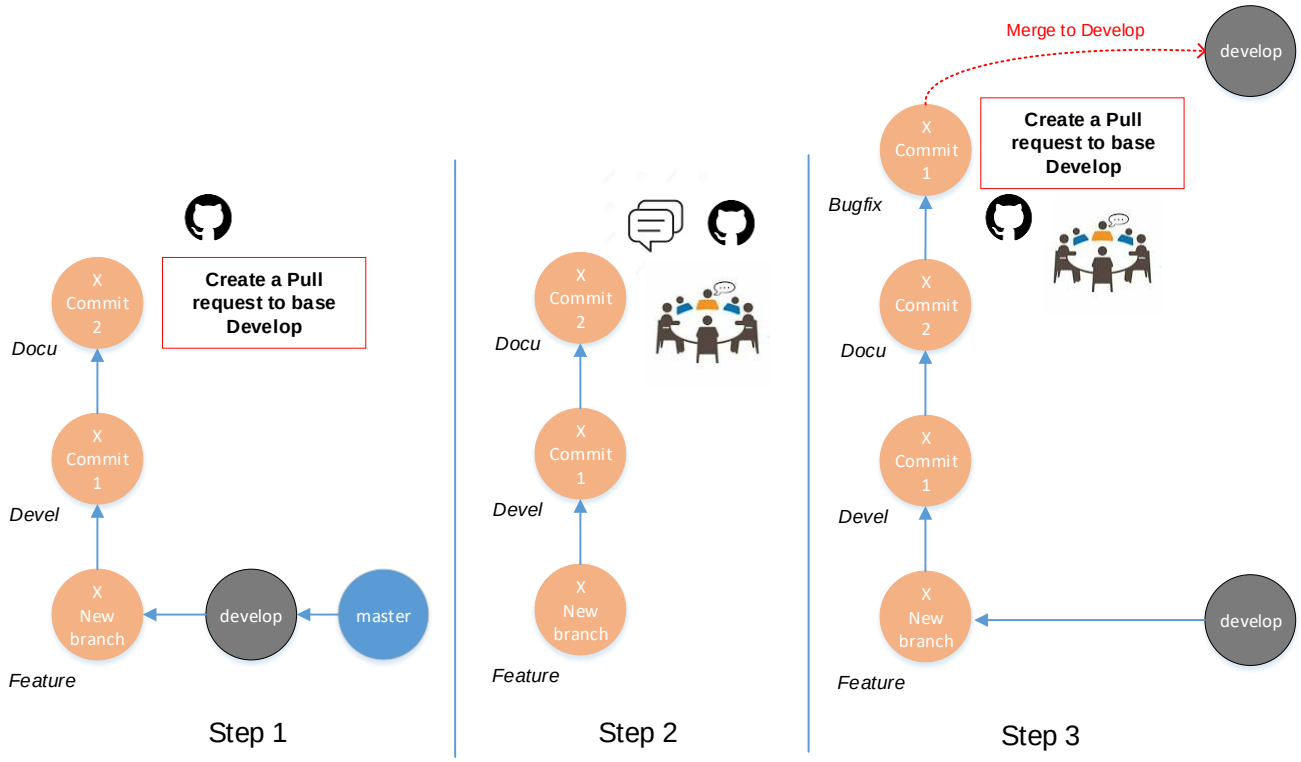


Figure 2: Adding a new component into the library with GIT - workflow

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.

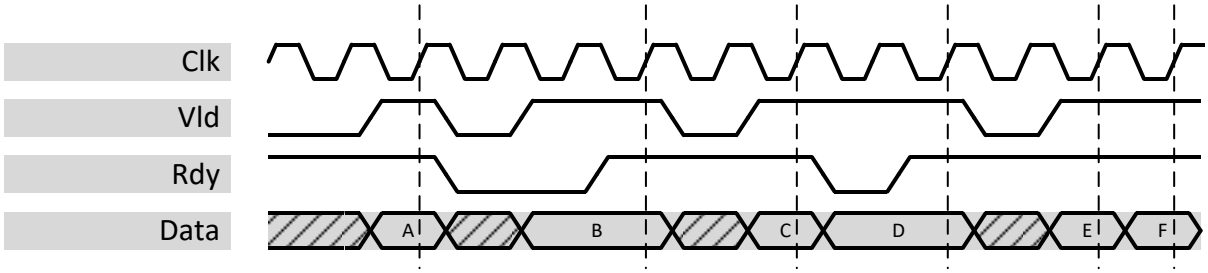


Figure 3: Handshaking signals

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.

1.6 TDM

Rules

- If multiple signals are transferred TDM (time-division-multiplexed) over the same interface and all signals have the same sample rate, no additional channel indicator is implemented and looping through the channels is implicit (e.g. for a three channel link, samples are transferred in the channel order 0-1-2-0-1-2-…).
- If multiple signals are transferred TDM over the same interface and the channels do not have the same sample rate, an additional channel indicator containing the channel number is present.

Reasoning

Not having an additional channel indicator for the most common TDM use-case of multiple signals at the same sample rate prevents any combinatorial blocks from having to know about being used for a TDM signal and maintaining the channel indicator. This implicitly allows using all combinatorial library elements (e.g. binary divider, function approximations, etc.) also for TDM signals

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
```

3.4 psi_common_tdp_ram_be

3.4.1 Description

Same as 3.3 psi_common_tdp_ram but with byte-enables. A byte is only written if *WrX* is set and the corresponding *BeX* bit is set too.

3.4.2 Generics

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

3.4.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Port A			
ClkA	Input	1	Port A clock
BeA	Input	Width_g/8	Port A byte enables
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
BeB	Input	Width_g/8	Port B byte enables
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

3.4.4 Constraints

Same as 3.3 psi_common_tdp_ram

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
RdyRstState_g	State of *InRdy* signal during reset. Usually this does not play a role and the default setting ('1') that leads to the least logic on the InRdy path is fine. Setting the value to '0' may lead to less optimal performance in terms of FMAX.	

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.

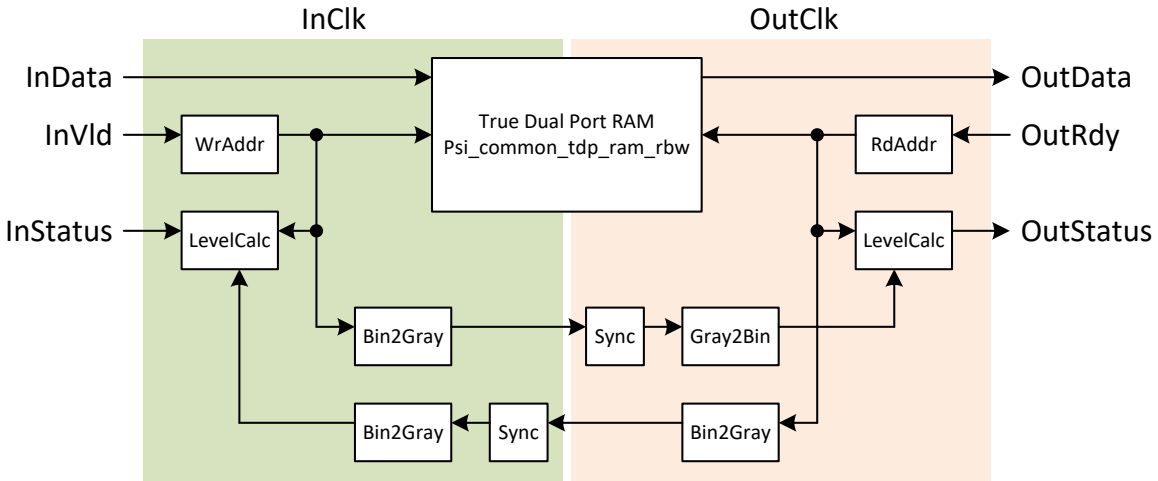


Figure 4: psi_common_async_fifo: Architecture

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
RdyRstState_g	State of *InRdy* signal during reset. Usually this does not play a role and the default setting ('1') that leads to the least logic on the InRdy path is fine. Setting the value to '0' may lead to less optimal performance in terms of FMAX.	

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.

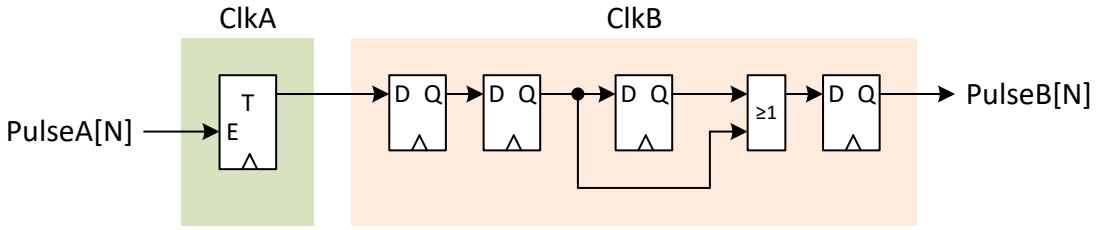


Figure 5: psi_common_pulse_cc: handling of pulses

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

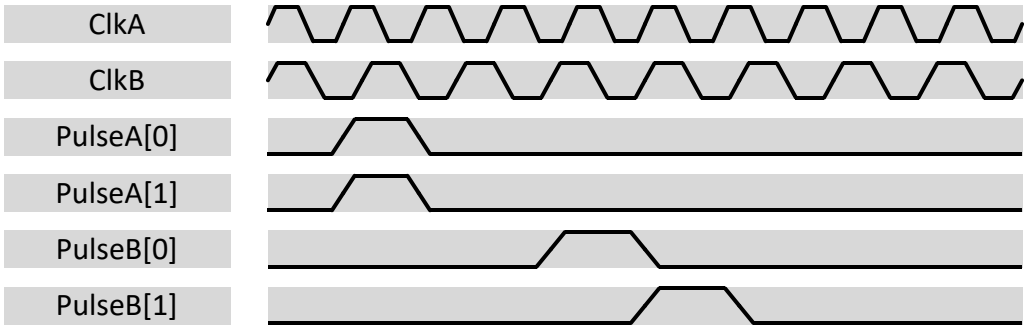


Figure 6: psi_common_pulse_cc: alignment of pulses can change

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.

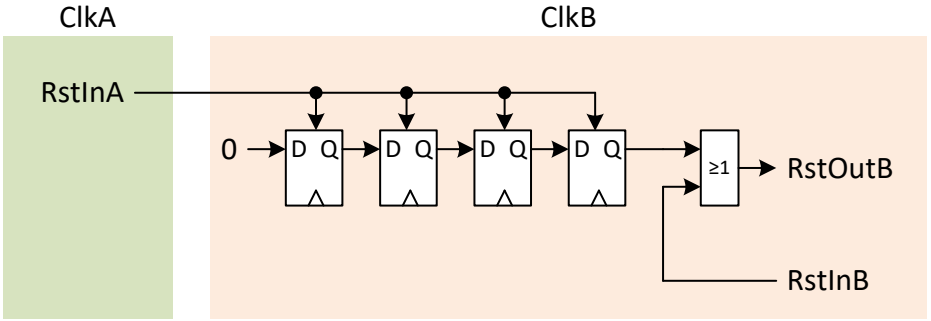


Figure 7: psi_common_pulse_cc: handling of resets

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.

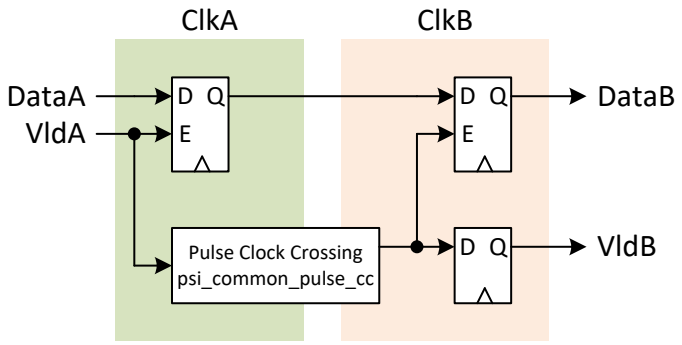


Figure 8: psi_common_simple_cc: Architecture

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.

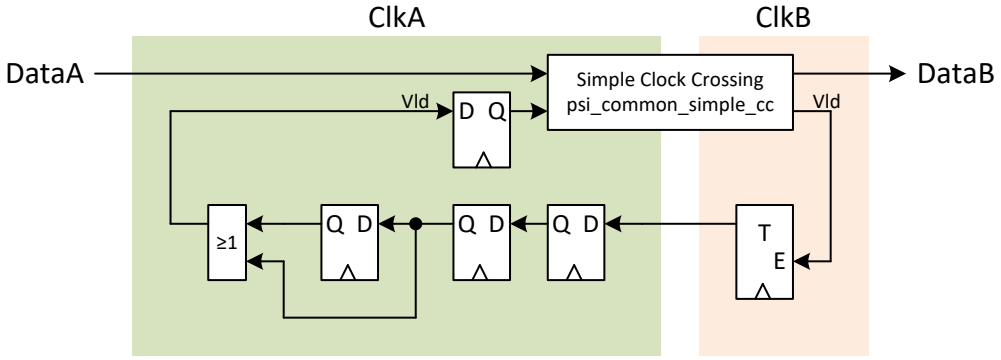


Figure 9: psi_common_status_cc: Architecture

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.

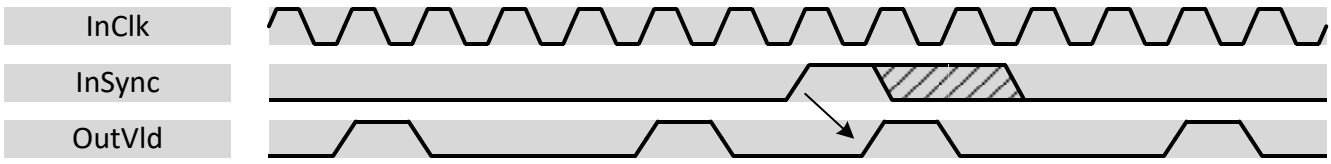


Figure 10: psi_common_strobe_generator: Strobe synchronization

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) 0 leads to the same behavior as 1.
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.

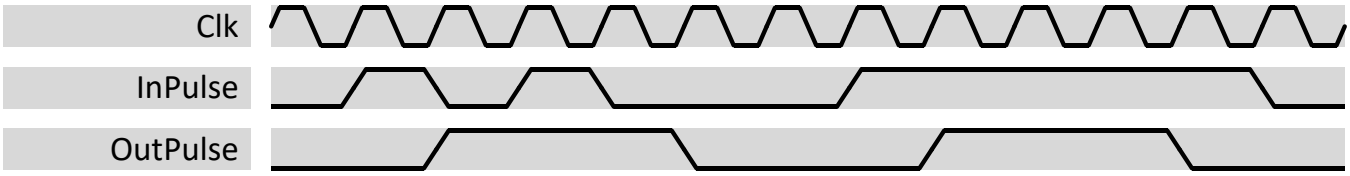


Figure 11: psi_common_pulse_shaper: Example waveform

6.4.2 Generics

Duration_g	Duration of the output pulse in clock cycles
HoldIn_g	If true it holds the input at the output, in case the pulse isn't pulse but a start signal
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_pulse_shaper_cfg

6.5.1 Description

This component is similar to *psi_common_pulse_shaper* (cd §6.4) but it is configurable in runtime. It allows giving the pulse width (duration) and the hold off time as registers in runtime.

6.5.2 Generics

MaxDuration_g Maximum duration allowed in clock cycles
HoldOffEna_g Enable hold-off mode – skip new pulses if arriving too fast
HoldIn_g If true it holds the input at the output, in case the pulse isn't pulse but a start signal
MaxHoldOff_g Maximum clock cycles for the minimum time between each new input pulse-rising-edges that are detected (in clock cycles) - Pulses arriving during the hold-off time are ignored
RstPol_g Defines reset polarity

6.5.3 Interfaces

Signal	Direction	Width	Description
clk_i	Input	1	Clock input
rst_i	Input	1	Reset input, polarity is set by generic
dat_i	Input	1	Input Pulse
width_i	Input	Log2ceil(MaxDuration_g)	Pulse width (duration) in clock cycles, if set to 0 no pulse will be generated.
hold_i	Input	Log2ceil(MaxHoldOff_g)	Minimum time between each new input pulse-rising edges that are detected (in clock cycles) – Pulse arriving during hold-off time are ignored
dat_o	Output	1	Output Pulse

6.6 psi_common_clk_meas

6.6.1 Description

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

6.6.2 Generics

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

6.6.3 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
FrequencyVld	Output	1	Handshaking signal (set on every update of *FrequencyHz*)

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). It also handles the last-flag correctly according to AXI specification. If *InLast* is asserted, all data is flushed out and the word enabled (*OutWe*) at the output are set only for words that contain data. *OutLast* is asserted accordingly.

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

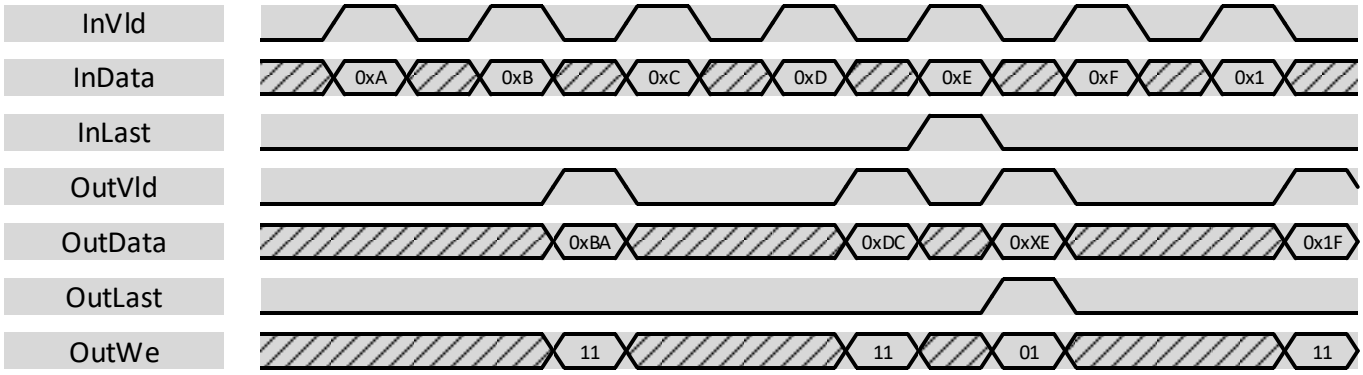


Figure 12: psi_common_wconv_n2xn: Example waveform

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
InLast	Input	1	AXI-S handshaking signal If InLast is asserted, the data stored inside the with-conversion is flushed out.
Output			
OutVld	Output	1	AXI-S handshaking signal
OutRdy	Input	1	AXI-S handshaking signal
OutData	Output	OutWidth_g	Data signal output
OutLast	Output	1	AXI-S handshaking signal
OutWe	Output	OutWidth_g/InWidth_g	Output word-enable. Works like byte-enable but with one bit per input-word. All bits in this signal are set, exept for with conversion results flushed out by *InLast='1'*. In this case, the *OutWe* bits indicate which *OutData* bits contain valid data.

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.

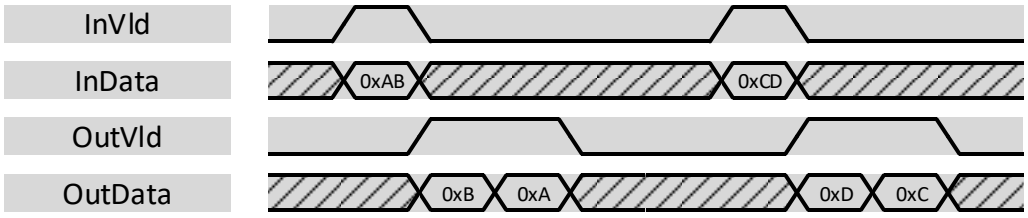


Figure 13: psi_common_wconv_xn2n: Data alignment

The with conversion does also handle the last-flag according to AXI specification and it can do alignment. To do so, an input word-enable signal *InWe* exists. Words that are not enabled are not sent to the output. If the input is marked with the *InLast* flag, the last enabled word is marked with *OutLast* at the output.

Note that with the assertion of *InLast* at least one byte of the data must be valid (*InWe* high). Otherwise it would be unclear when *OutLast* shall be assigned.

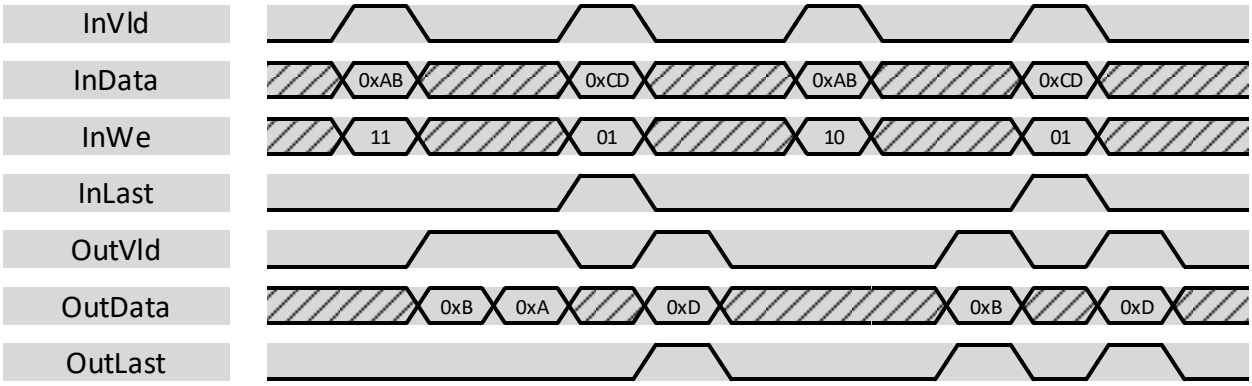


Figure 14: psi_common_wconv_xn2n: Last-Handling and alignment

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
InLast	Input	1	AXI-S handshaking signal
InWe	Input	InWidth_g/OutWidth_g	Output word-enable. Works like byte-enable but with one bit per input-word (not per byte). At least one word must be enabled together with the assertion of *InLast*
Output			
OutVld	Output	1	AXI-S handshaking signal
OutRdy	Input	1	AXI-S handshaking signal
OutData	Output	OutWidth_g	Data signal output
OutLast	Output	1	AXI-S handshaking signal

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.

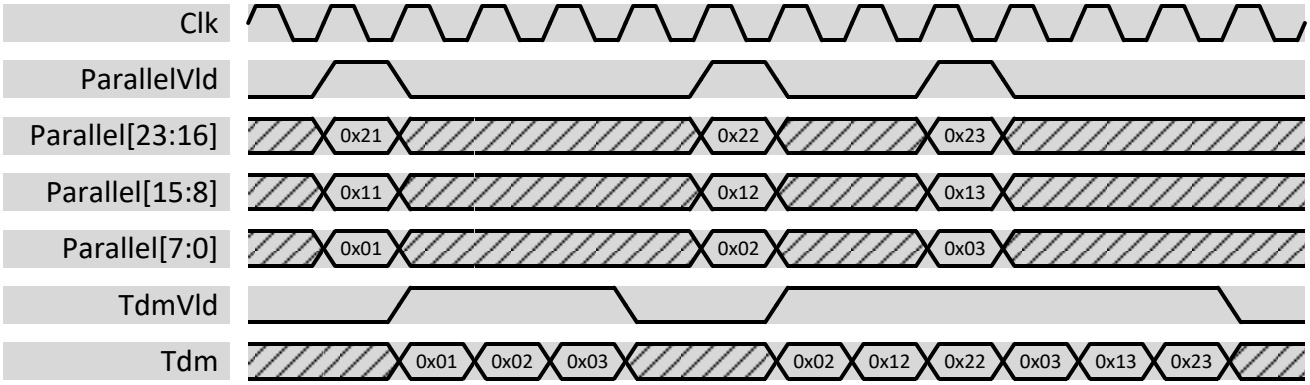


Figure 15: psi_common_par_tdm: Waveform

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.

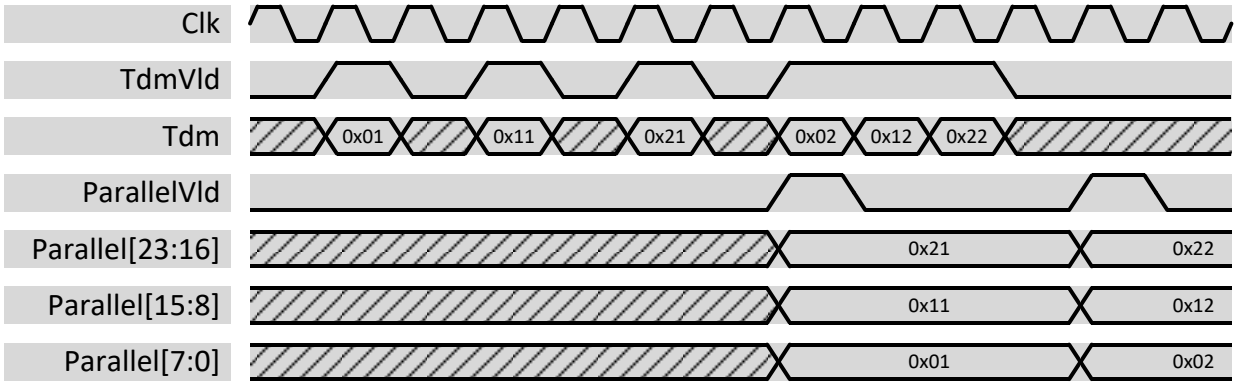


Figure 16: psi_common_tdm_par: Waveform

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_par_cfg

8.3.1 Description

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

The number of enabled channels is configurable. In addition, if used with an AXI stream, TdmLast can be used to ensure the correct correspondence between the channels and their indexes in Parallel when the number of enabled channels is changed.

The figures below show 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 enabled highest bits.

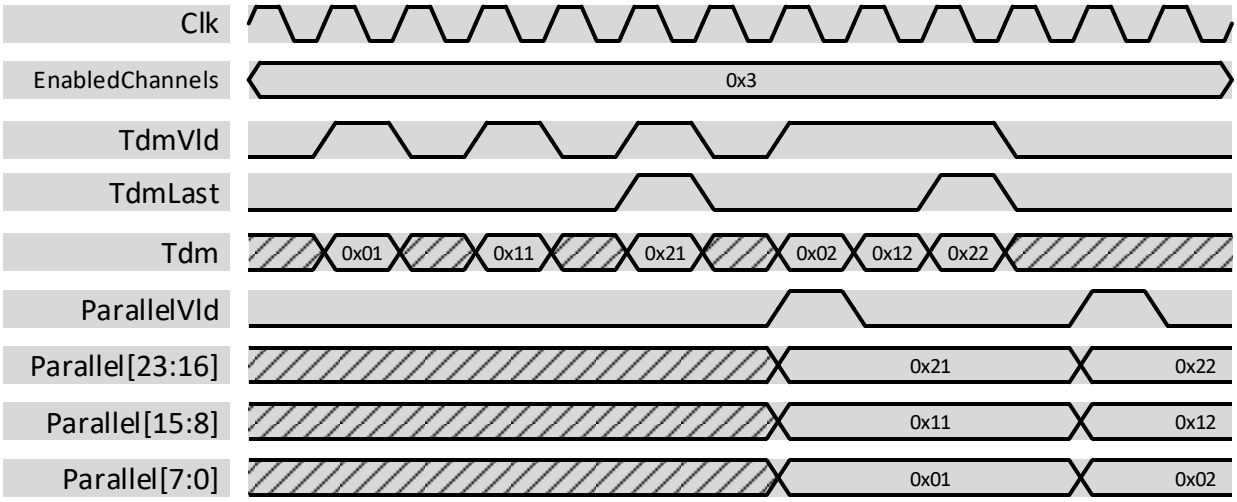


Figure 17: psi_common_tdm_par_cfg: 3 enabled channels waveform

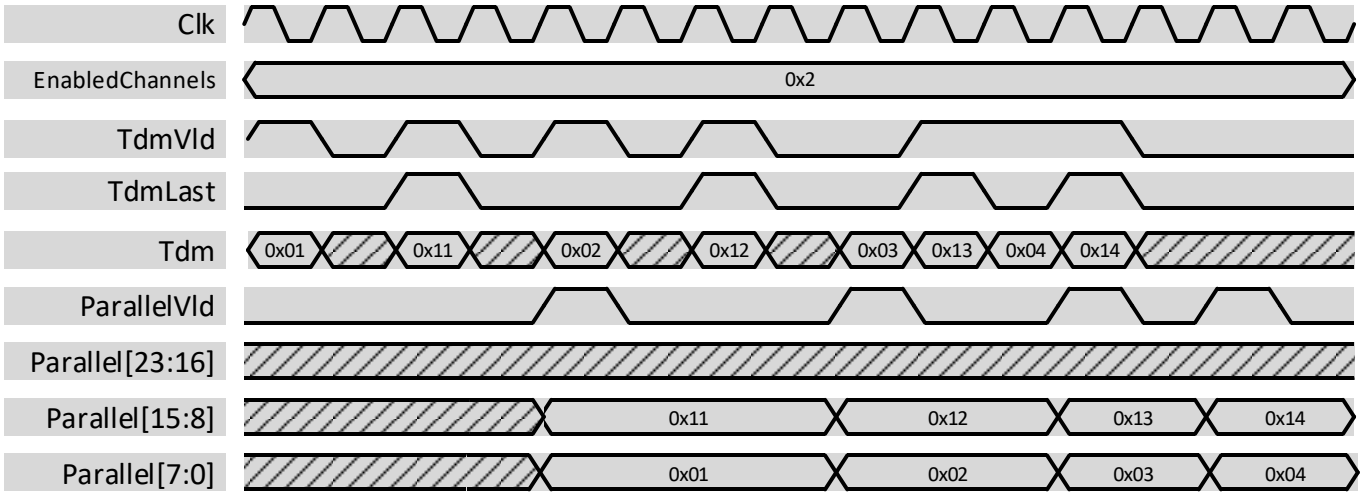


Figure 18: psi_common_tdm_par_cfg: 2 enabled channels waveform

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

8.3.2 Generics

ChannelCount_g	Number of channels
ChannelWidth_g	Number of bits per channel

8.3.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset (high active)
EnabledChannels	Input	ChannelCount_g	Number of enabled output channels
Input			
TdmVld	Input	1	AXI-S handshaking signal
TdmLast	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.

8.4 psi_common_tdm_mux

8.4.1 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.

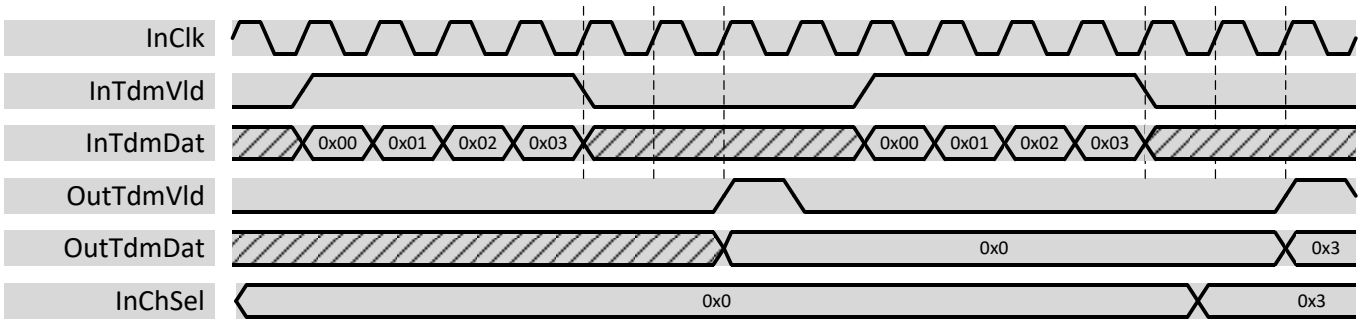


Figure 19: psi_common_tdm_mux: Waveform

8.4.2 Generics

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

8.4.3 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.

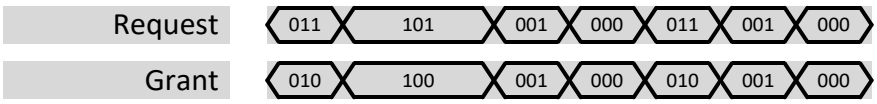


Figure 20: psi_common_arb_priority: Waveform

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.

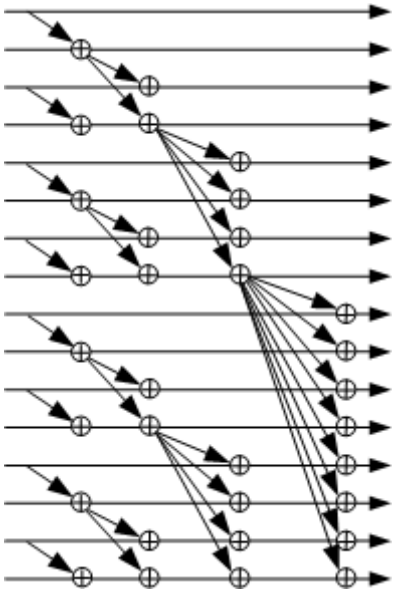


Figure 21: psi_common_arb_priority: Parallel prefix computation (PPC)

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.

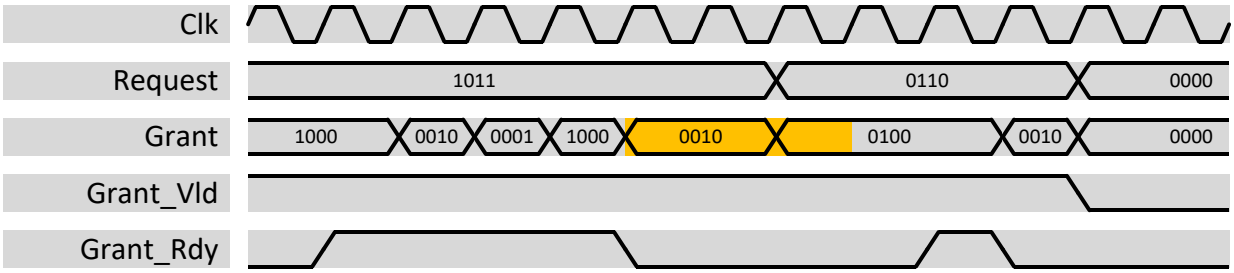


Figure 22: psi_common_arb_round_robin: Waveform

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 Interfaces

10.1 psi_common_spi_master

10.1.1 Description

This entity implements a simple SPI master. All common SPI settings are parametrizable to ensure the master can be configured for different applications.

The clock and data phase is configurable according to the SPI standard terminology described in the picture below:

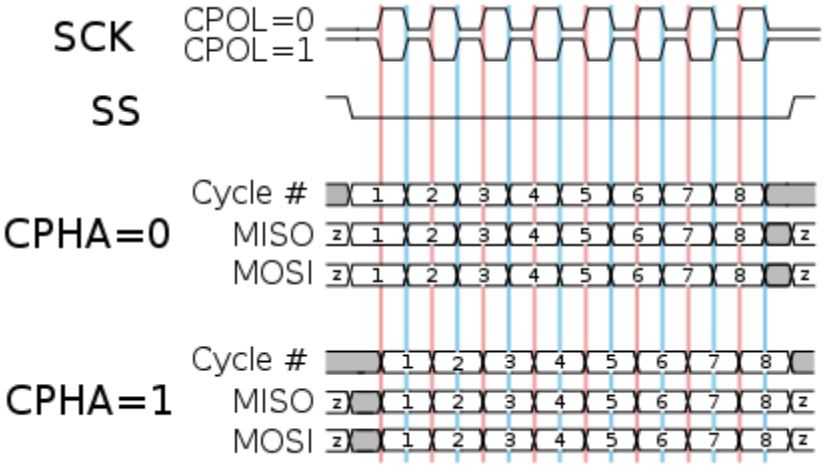


Figure 23: psi_common_spi_master: CPOL and CPHA meaning

For CPHA = 1, the sampling happens on the second edge (blue) and data is applied on the first edge (red). For CPHA = 0 it is the opposite way.

10.1.2 Generics

ClockDivider_g	Ratio between *Clk* and the *SpiSck* frequency
TransWidth_g	SPI Transfer width (bits per transfer)
CsHighCycles_g	Minimal number of *Cs_n* high cycles between two transfers
SpiCPOL_g	SPI clock polarity (see figure above)
SpiCPHA_g	SPI sampling edge configuration (see figure above)
SlaveCnt_g	Number of slaves to support (number of *Cs_n* lines)
LsbFirst_g	False = MSB first transmission, True = LSB first transmission

10.1.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset (high active)
Parallel Interface			
Start	Input	1	A high pulse on this line starts the transfer. Note that starting a transaction is only possible when *Busy* is low.
Slave	Input	$\log_2(SlaveCnt\_g)$	Slave number to access
Busy	Output	1	High during a transaction
Done	Output	1	Pulse that goes high for exactly one clock cycle after a transaction is done and *RdData* is valid
WrData	Input	*TransWidth_g*	Data to send to slave. Sampled during *Start = '1'*
RdData	Output	*TransWidth_g*	Data received from slave. Must be sampled during *Done = '1'* or *Busy = '0'*.
SPI Interface			
SpiSck	Output	1	SPI clock
SpiMosi	Output	1	SPI master to slave data signal
SpiMiso	Input	1	SPI slave to master data signal
SpiCs_n	Output	*SlaveCnt_g*	SPI slave select signal (low active)

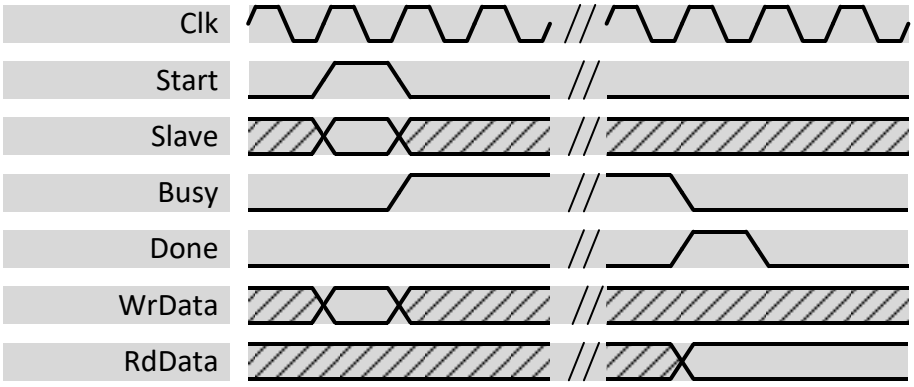


Figure 24: psi_common_spi_master: Parallel interface signal behavior

10.2 psi_common_i2c_master

10.2.1 Description

This entity implements an I2C master that is multi-master capable (i.e. supports arbitration). The *psi_common_i2c_master* allows generating start, stop and repeated start conditions as well as transferring bytes. This entity also supports slaves that do clock-stretching.

Addressing is not handled separately. To send the byte containing the address and the R/W bit, the normal byte transfer is used. The user is responsible for filling the address into bits 7:1 and the R/W flag into bit 0 of the data.

The same applies for 10-bit addressing. The user is responsible to send the 10-bit addressing pattern using two byte transfers.

The bus state (busy or free) is tracked based on start- and stop-conditions. However, if no transactions are ongoing (i.e. SCL = '1' and SDA = '1') for a configurable timeout, the bus is always regarded as free. This helps handling the case where a master starts a transaction and then (e.g. due to reset) aborts the transaction without sending a stop-condition.

The user has three main interfaces:

- The command interface allows the user to select the next bus action to do. Commands are:
  - *START* Send a start condition (only allowed if bus is idle)
  - *STOP* Send a stop condition (only allowed if bus is not idle)
  - *REPSTART* Send a repeated start condition (only allowed if the bus is not idle)
  - *SEND* Send a data byte (only allowed if the bus is not idle).
The data to send is provided along with the command.
  - *REC* Receive a data byte (only allowed if the bus is not idle)
The ACK to send is provided along with the command.
- On the response interface, the user receives one response per command as soon as the command is completed. More information like the received ACK bit, whether the command was aborted due to a lost arbitration, the received data or wether the command sequence was illegal is sent along with the response.
- On the status interface the user can see if the I2C bus is currently busy or free and if a command timed out (i.e. the user failed to provide the next command within time, so the bus was released).

For constants, a package *psi_common_i2c_master_pkg* is defined in the same VHDL file.

10.2.2 Generics

ClockFrequency_g	Frequency of the clock *Clk* in Hz
I2cFrequency_g	Frequency of the *I2cScl* signal in Hz
BusBusyTimeout_g	If *I2cScl* = '1' and *I2cSda* = '1' for this time in sec., the bus is regarded as free. If the user does not provide any command for this time, the *psi_common_i2c_master* automatically generates a stop-condition to release the bus.
CmdTimeout_g	When the *psi_common_i2c_master* is ready for the next command but the user does not provide a new command, after this timeout the bus is releases (and *TimeoutCmd* is pulsed to inform the user).
InternalTriState_g	True = Use internal tri-state buffer (*I2cScl* and *I2cSda*) are used. False = Use external tri-state buffer (*I2cScl_x* and *I2cSda_x*) are used.
DisableAsserts_g	If true, the *psi_common_i2c_master* does not print any messages during simulations.

10.2.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
Clk	Input	1	Clock
Rst	Input	1	Reset (high active)
Command Interface			
CmdRdy	Output	1	AXI-S handshaking signal for command interface
CmdVld	Input	1	AXI-S handshaking signal for command interface
CmdType	Input	3	Constant names “000” => Send start condition (*CMD_START*) “001” => Send stop condition (*CMD_STOP)* “010” => Send repeated start condition” (*CMD_REPSTART)* “011” => Send data byte (*CMD_SEND)* “100” => Receive data byte (*CMD_RECEIVE)*
CmdData	Input	8	Input data to send (only for *CMD_SEND* resp. *CmdType=”011”*).
CmdAck	Input	1	Acknowledge to send (only for *CMD_RECEIVE* resp. *CmdType=”100”).*
Response Interface			
RspVld	Output	1	AXI-S handshaking signal for response interface
RspType	Output	3	Type of the command that completed. See *CmdType* for details.
RspData	Output	8	Received data (only for *CMD_RECEIVE* resp. *CmdType=”100”*).
RspAck	Output	1	1 => ACK received 0 => NACK received
RspArbLost	Output	1	The command failed because arbitration was lost.
RspSeq	Output	1	The command failed because of wrong command sequence (e.g. attempt to do a *CMD_START* in the middle of an ongoing transfer)
Status Interface			
BusBusy	Output	1	I2C bus is busy (used by this master or another master)
TimeoutCmd	Output	1	Pulsed if the bus is released due to a timeout.
I2C Interface – Internal Tristate (InternalTriState_g=true)			
I2cScl	Bidir	1	SCL signal
I2cSda	Bidir	1	SDA signal
I2C Interface – External Tristate (InternalTriState_g=false)			
I2cScl_I	Input	1	SCL input signal
I2cScl_O	Output	1	SCL output signal
I2cScl_T	Output	1	SCL Tri-State signal (1 = tristated, 0 drive)
I2cSda_I	Input	1	SDA input signal
I2cSda_O	Output	1	SDA output signal

I2cSda_T	Output	1	SDA Tri-State signal (1 = tristated, 0 drive)

10.2.4 Typical Command Sequences

This section provides a few examples for command/response sequences for typical usecases.

Note that the response of the last command is always available before the next command must be asserted. So it is possible to apply commands based on responses received (e.g. not applying a new command if arbitration was lost).

10.2.4.1 Two Byte Read

- CMD_START – send start condition
- CMD_SEND – send address byte (slave responds with ACK)
- CMD_REC – receive data and send ACK
- CMD_REC – receive data and send NACK
- CMD_STOP – send stop condition

Order	CmdType	CmdData	CmdAck	RspType	RspData	RspAck	RspArbLost	RespSeq
1	CMD_START	N/A	N/A	CMD_START	N/A	N/A	0	0
2	CMD_SEND	Addr + R/W	N/A	CMD_SEND	N/A	1	0	0
3	CMD_REC	N/A	1	CMD_REC	Data	N/A	0	0
4	CMD_REC	N/A	0	CMD_REC	Data	N/A	0	0
5	CMD_STOP	N/A	N/A	CMD_STOP	N/A	N/A	0	0

10.2.4.2 Two Byte Write

- CMD_START – send start condition
- CMD_SEND – send address byte (slave responds with ACK)
- CMD_SEND – send data (slave responds with ACK)
- CMD_SEND – send data (slave responds with NACK)
- CMD_STOP – send stop condition

Order	CmdType	CmdData	CmdAck	RspType	RspData	RspAck	RspArbLost	RespSeq
1	CMD_START	N/A	N/A	CMD_START	N/A	N/A	0	0
2	CMD_SEND	Addr + R/W	N/A	CMD_SEND	N/A	1	0	0
3	CMD_SEND	Data	N/A	CMD_SEND	N/A	1	0	0
4	CMD_SEND	Data	N/A	CMD_SEND	N/A	0	0	0
5	CMD_STOP	N/A	N/A	CMD_STOP	N/A	N/A	0	0

10.2.4.3 One Byte Write followed by One Byte Read (with Repeated Start)

- CMD_START – send start condition
- CMD_SEND – send address byte (slave responds with ACK)
- CMD_SEND – send data (slave responds with ACK)
- CMD_REPSTART – Repeated start
- CMD_REC – receive data and send NACK
- CMD_STOP – send stop condition

Order	CmdType	CmdData	CmdAck	RspType	RspData	RspAck	RspArbLost	RespSeq
1	CMD_START	N/A	N/A	CMD_START	N/A	N/A	0	0
2	CMD_SEND	Addr + R/W	N/A	CMD_SEND	N/A	1	0	0
3	CMD_SEND	Data	N/A	CMD_REC	N/A	1	0	0
4	CMD_REPST.	N/A	N/A	CMD_REPST.	N/A	N/A	0	0
4	CMD_REC	N/A	0	CMD_REC	Data	N/A	0	0
5	CMD_STOP	N/A	N/A	CMD_STOP	N/A	N/A	0	0

10.2.4.4 Arbitration Lost

- CMD_START – send start condition
- CMD_SEND – send address byte (slave responds with ACK)
- CMD_SEND – send data (arbitration is lost during this byte)
- CMD_REPSTART – Repeated start
    - This command is ignored (RespSeq=’1’) because it is illegal when the bus is not owned

Order	CmdType	CmdData	CmdAck	RspType	RspData	RspAck	RspArbLost	RespSeq
1	CMD_START	N/A	N/A	CMD_START	N/A	N/A	0	0
2	CMD_SEND	Addr + R/W	N/A	CMD_SEND	N/A	1	0	0
3	CMD_SEND	Data	N/A	CMD_REC	N/A	N/A	1	0
4	CMD_REPST.	N/A	N/A	CMD_REPST.	N/A	N/A	0	1

10.2.5 Example Waveform

The waveform below shows the very simplest transaction possible: transmitting an address only to probe if the slave is available. This simple transaction was chosen to keep the waveform as short as possible. The main focus is on the sequence of events, not on the I2C transaction.

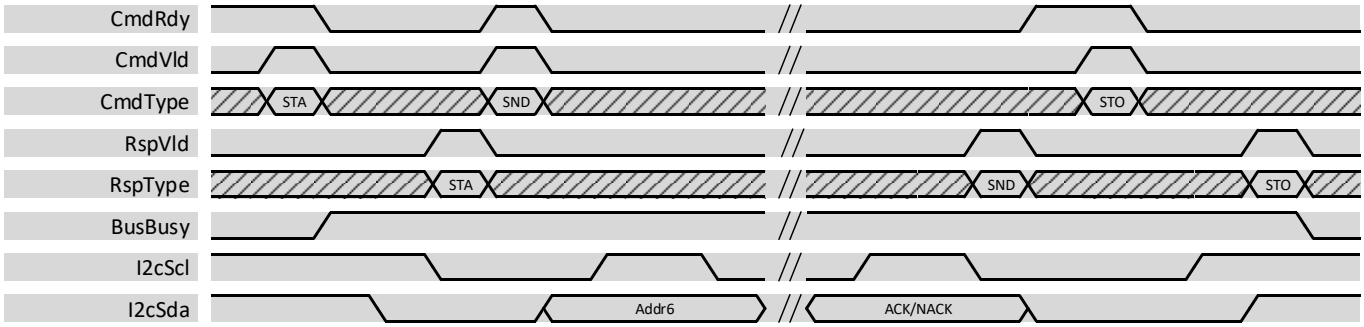


Figure 25: psi_common_i2c_master: Address only transaction

10.3 psi_common_axi_master_simple

10.3.1 Description

This entity executes transactions requested through a simple command interface on an AXI bus according to all specifications. This entity includes FIFOs to buffer read- and write-data but not for the commands.

The user can request transaction of any size and they will get split automatically in order to not exceed AXI bust size and other limitations. The response is sent to the user when his whole command is executed (which may involve multiple AXI transactions).

For each command there are two operation modes:

- High Latency
  - The AXI-master only starts the command after sufficient data (write-case) or space (read-case) is available in the corresponding data FIFO
  - This ensures that commands can be executed without blocking the AXI bus.
  - This approach leads to more latency, since he user has to handle data before the command is sent.
- Low Latency
  - The AXI-master starts the transaction immediately, with no regard on FIFO states.
  - If the user logic cannot provide the data in-time, the AXI bus may get blocked.
  - This approach leads to lowest latency since the user logic can prepare the data on the fly without the transaction being delayed.

This entity does not handle unaligned transactions and word-width conversions. So the data- and AXI-width are the same and all commands must be aligned to that word-width. This is the reason for the term *simple* in the name of the entity.

Read and write logic are fully independent. So reads and writes can happen at the same time.

There is no required timing relationship between command and data signals. So for writes the user can provide write data before, after or together with the command.

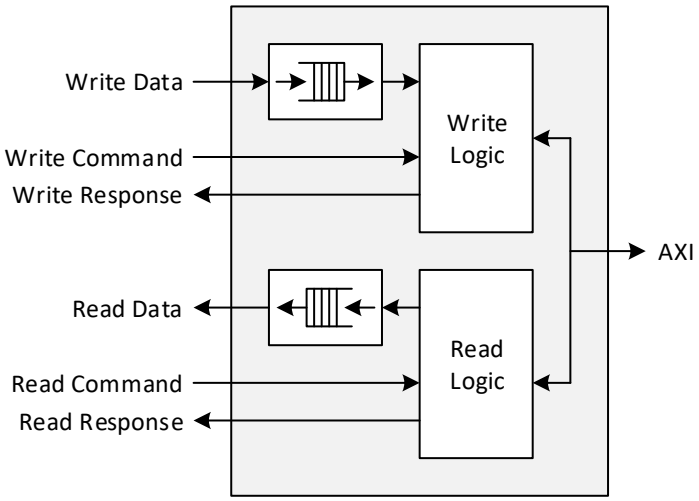


Figure 26: psi_common_axi_master_simple: Block diagram

10.3.2 Transaction Types

For simplicity, only burst transactions are shown. However, of course also single-word transactions are supported.

Note that for all examples, the maximum AXI burst length is assumed to be 4 (unusual low) for illustrative reasons.

Also not that latencies and delays may be drawn shorter than they actually are to keep the waveforms small. However, all relationship between signals are correct.

10.3.2.1 Write High-Latency

The example below shows a high latency burst read transaction.

Please read the description of all examples (10.3.2).

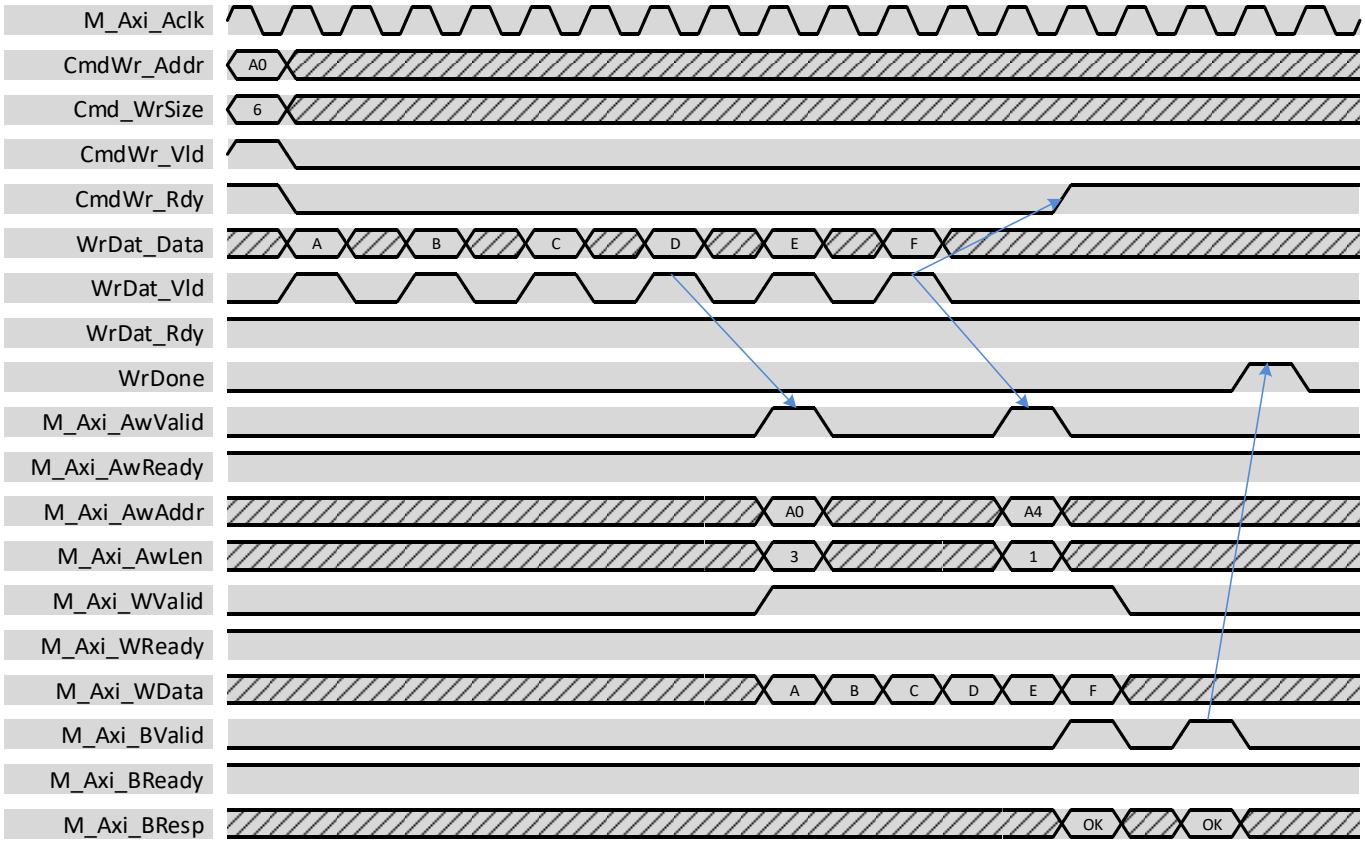


Figure 27: psi_common_axi_master_simple: High latency write

The waveform shows, that the write command (*M_Axi_Aw**) is held back until all data for a burst (4 words) is in the FIFO. The command is then executed. The next command is executed immediately, because the data is already available when the first transaction completed.

The waveform also clearly shows, that the *CmdWr_Rdy* signals goes high as soon as all AXI-commands related to the user command are sent. However, at the time *CmdWr_Rdy* goes high, not all data is transferred yet. This only indicates that the next command can be applied and does not have any meaning for the currently processed signal.

The *WrDone* signal is pulsed as soon as the response of the last AXI transaction is received.

The waveform also clearly shows that a user command is split into two AXI transactions automatically and that the *M_Axi_AwAddr* and *M_Axi_AwLen* signals are chosen appropriately.

Since the waveform above only shows that the first transaction is delayed according to high-latency operation, a second figure is shown below that shows this behavior also for the second transaction.

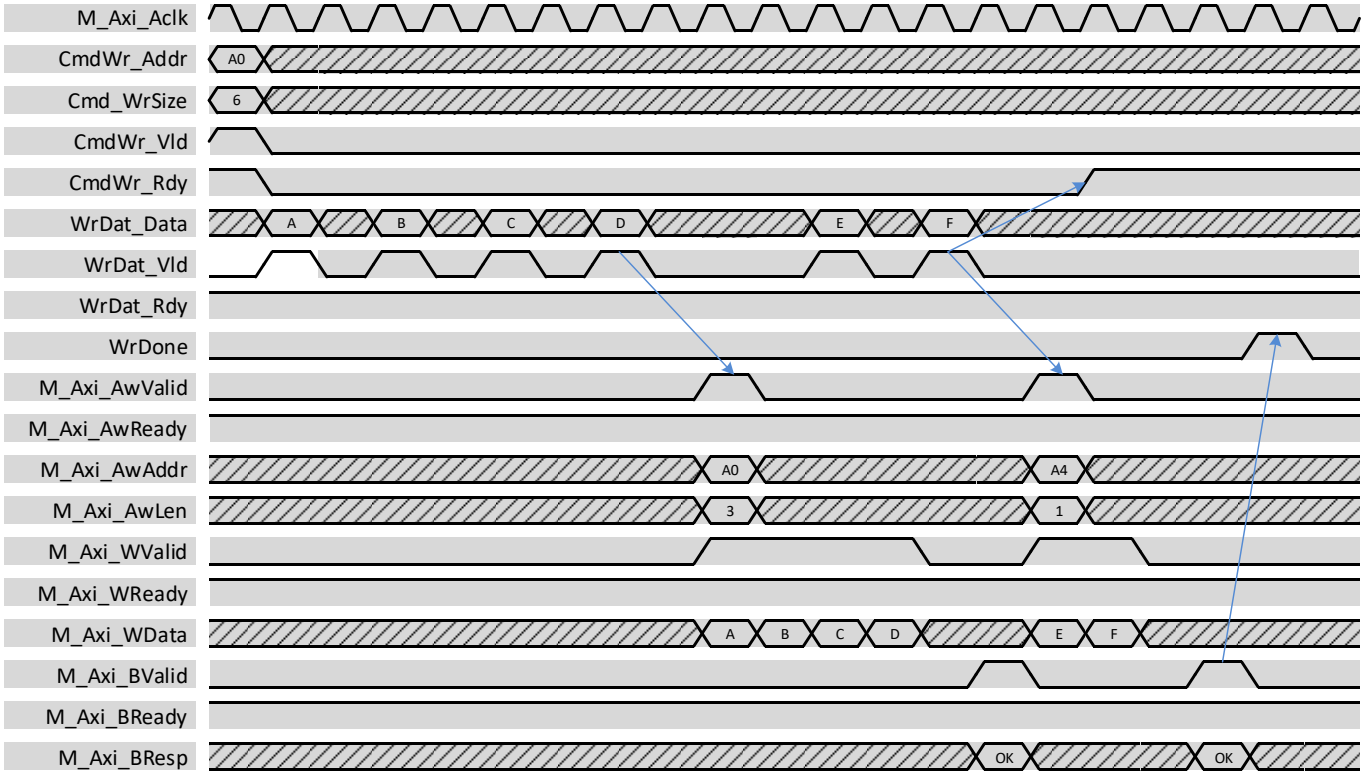


Figure 28: psi_common_axi_master_simple: High latency write with delay for second transaction

10.3.2.2 Write Low-Latency

The example below shows a low latency burst read transaction.

Please read the description of all examples (10.3.2).

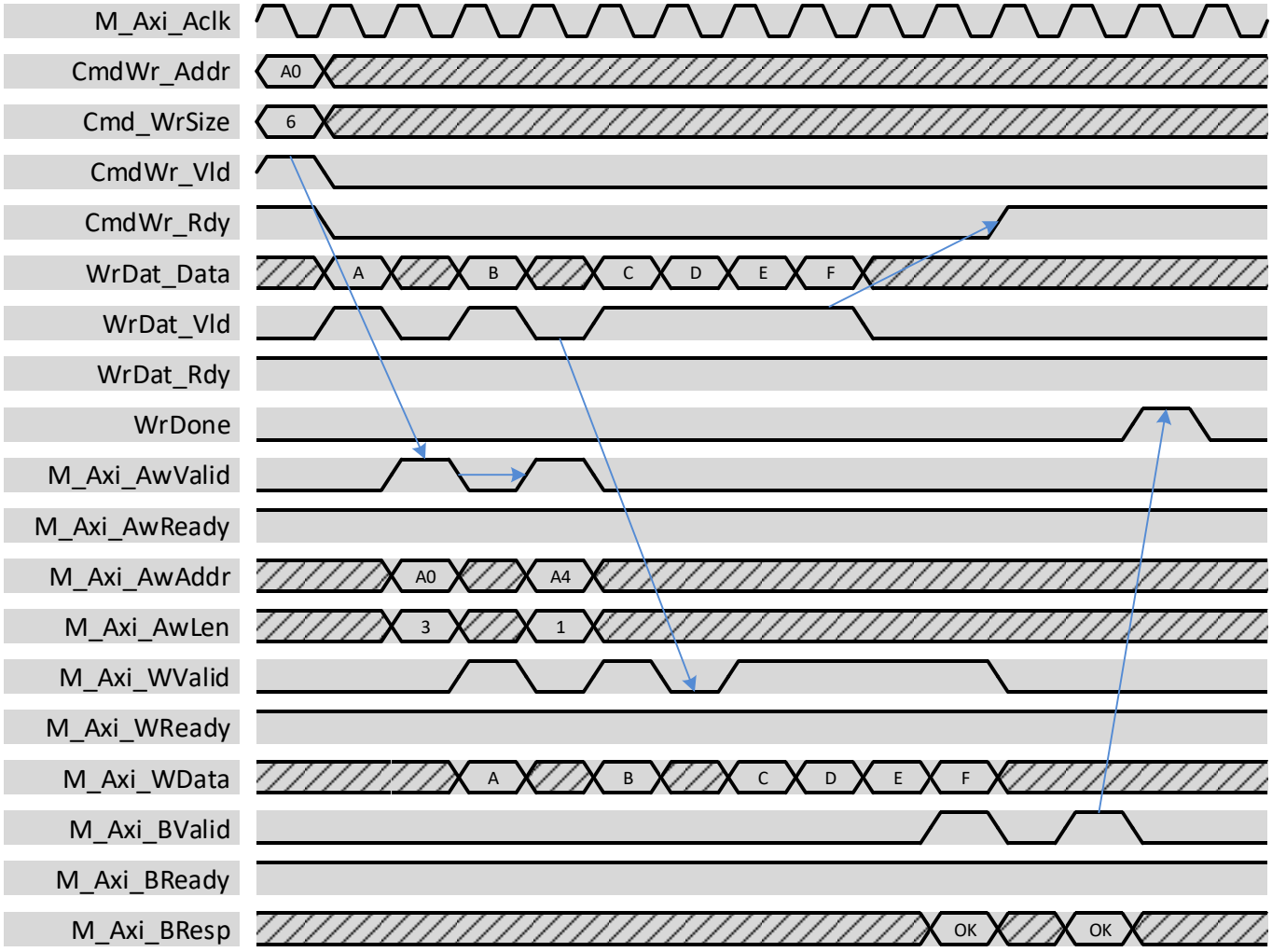


Figure 29: psi_common_axi_master_simple: Low latency write

The waveform shows, that in low latency operation, AXI commands are issued as soon as possible independently of the availability of data. Therefore both write commands are issued before even the data for the first one is in the FIFO.

The waveform also shows, that the *M_AXI_W** bus is blocked temporarily (*M_Axi_WValid* low) due to the data not being available. This situation has a negative impact on the AXI bandwidth, so it shall be avoided usually.

To avoid stalling the AXI bus, it is possible to prefill the write data FIFO. To do so, the write command is sent after the first few data samples are already written into the FIFO. This allows using the FIFO to prevent the AXI bus from stalling.

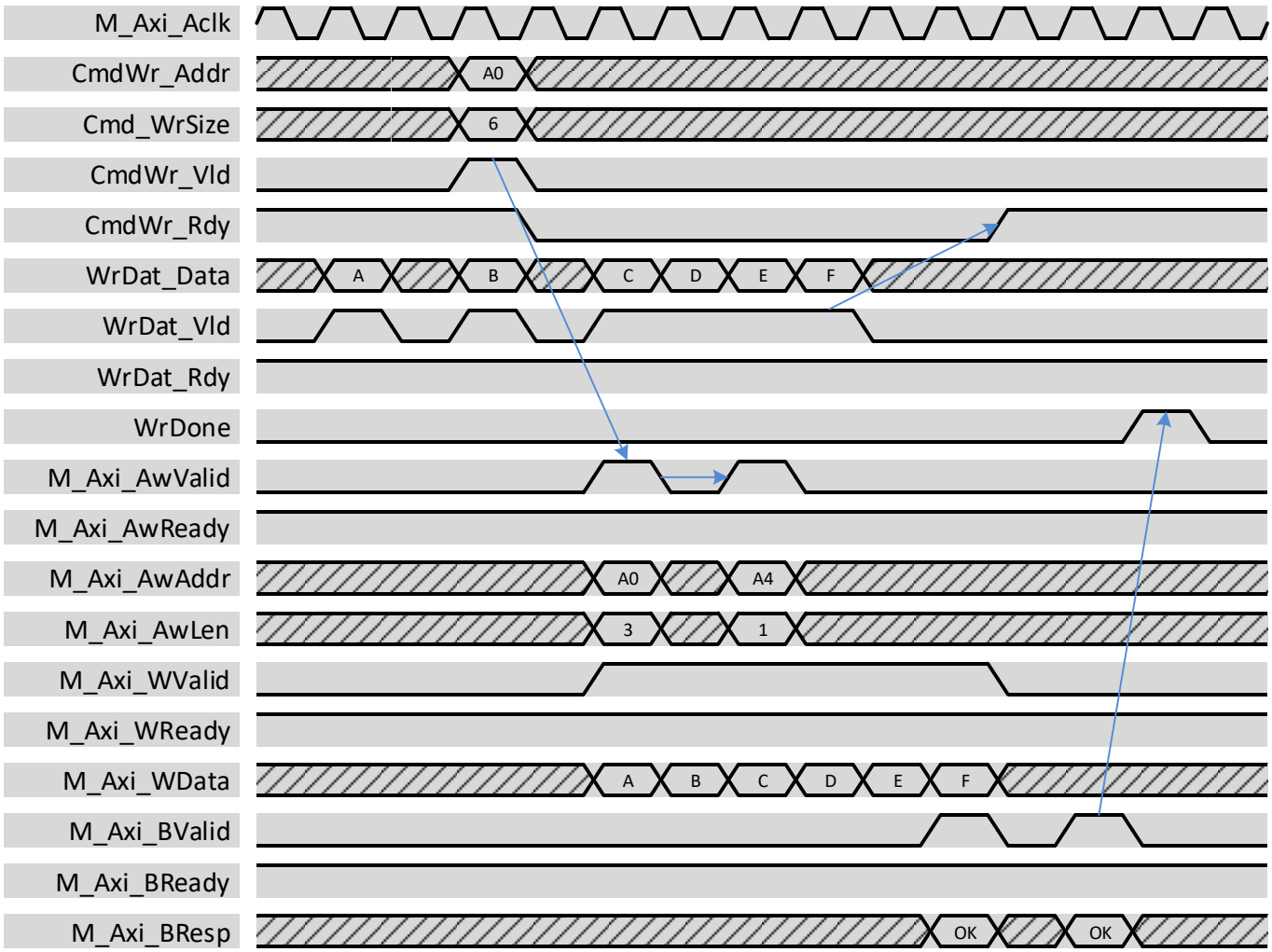


Figure 30: psi_common_axi_master_simple: Low latency write with FIFO prefill

10.3.2.3 Read

The example below shows a burst read transaction.

Please read the description of all examples (10.3.2).

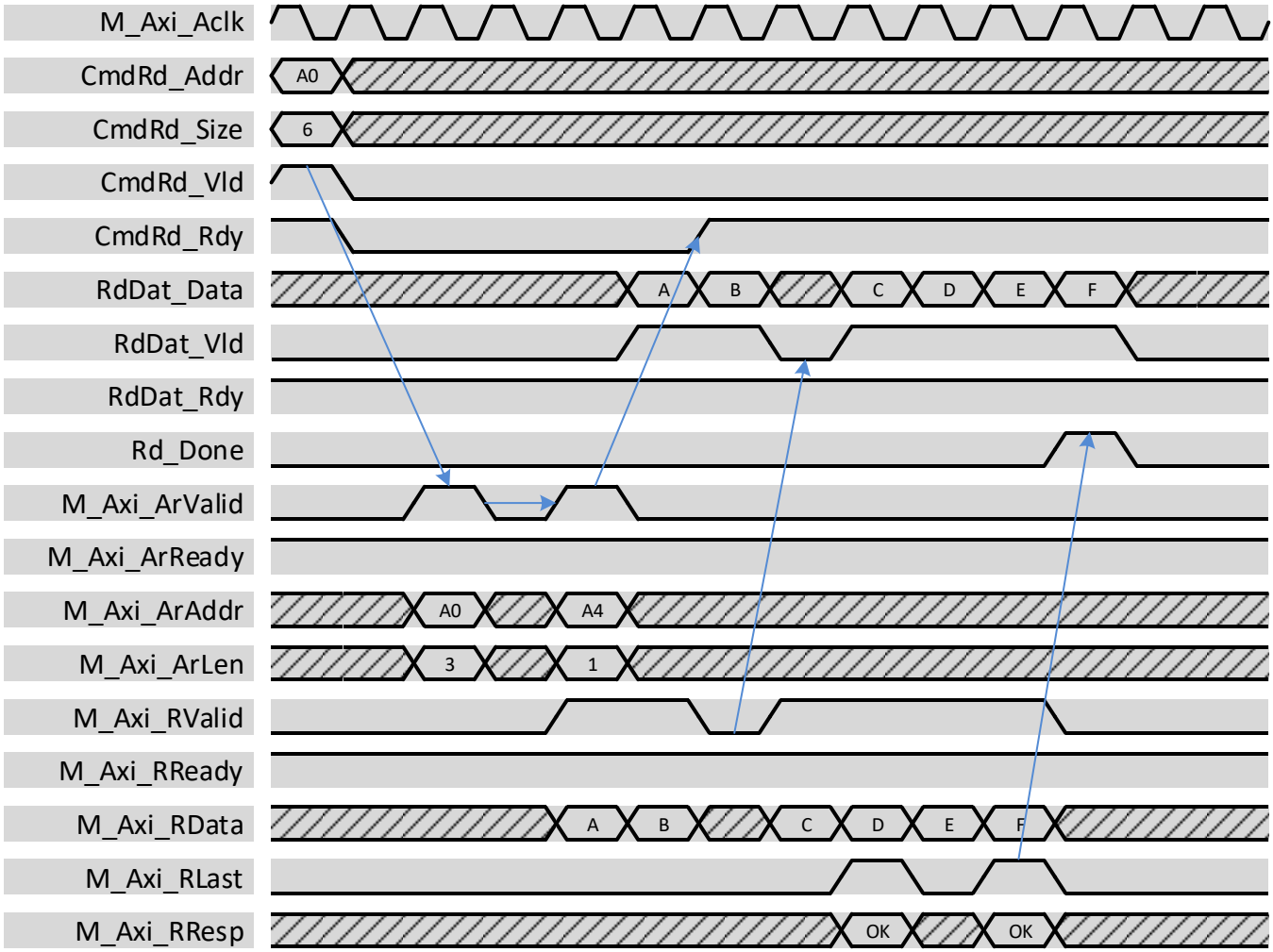


Figure 31: psi_common_axi_master_simple: Read transaction

Usually for reads there is enough space in the read FIFO. In this case, the user command directly triggers the transmission of the AXI-command on *M_Axi_Ar**. After all AXI commands are sent, the FSM is ready for the next command.

If the slave is not able to continuously burst data, this is reflected on the read data output. However, a FIFO is present and can compensate this effect if reading of the data is started a few beats after availability of first data.

10.3.2.4 Read FIFO full with Low Latency

The example below shows a burst read transaction in low latency mode. In contrast to the example above, the read FIFO is assumed to be full when the user command is issued.

Please read the description of all examples (10.3.2).

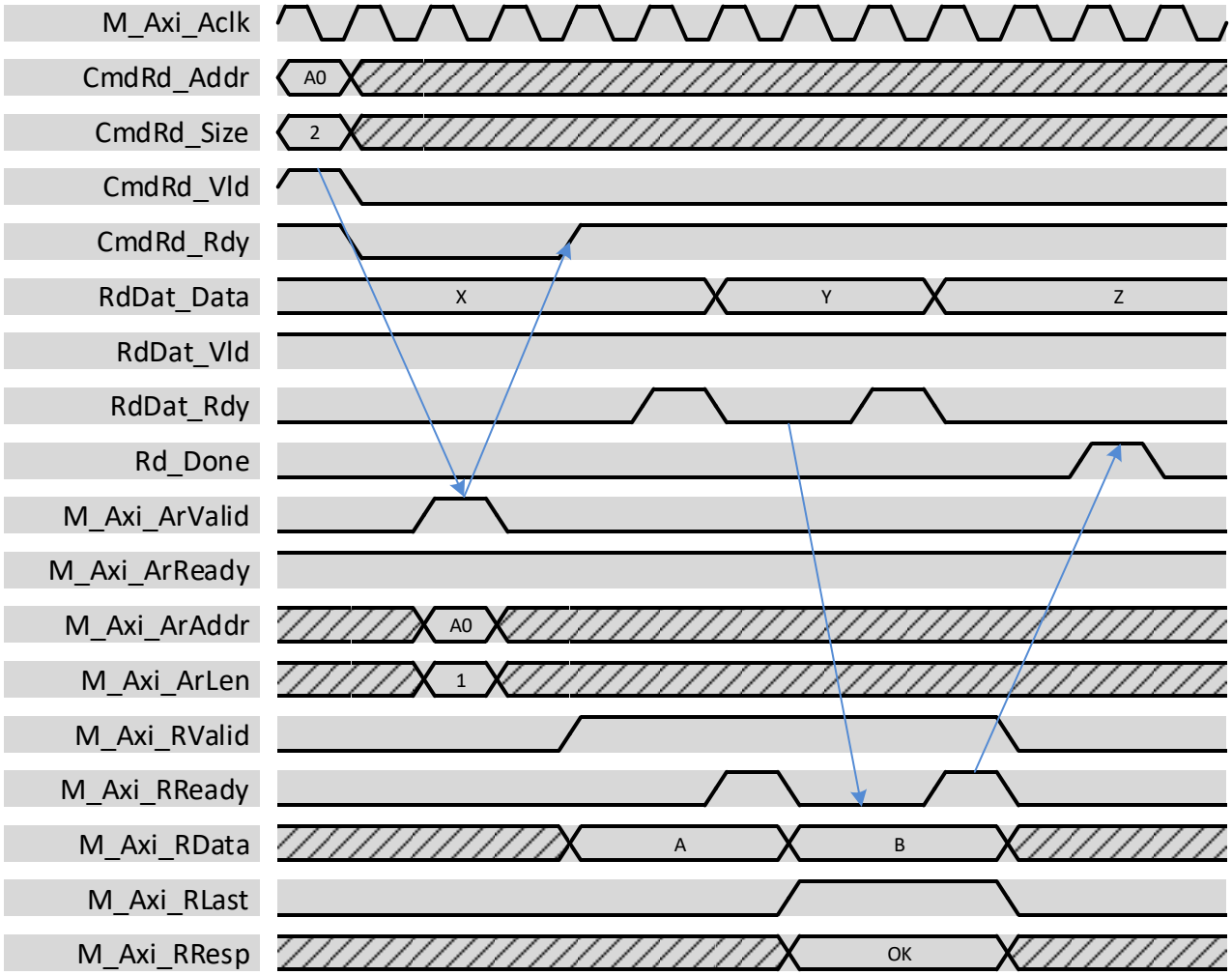


Figure 32: psi_common_axi_master_simple: Read transaction, low latency

Beause the command is issued in low-latency mode, the AXI read command is issued immediately. Because the FIFO is full, *M_Axi_RReady* is low and the AXI bus is stalled. The pattern of space in the FIFO becoming available is visible on the AXI bus directly.

10.3.2.5 Read FIFO full with High Latency

The example below shows a burst read transaction in high latency mode. In contrast to the example above, the read FIFO is assumed to be full when the user command is issued.

Please read the description of all examples (10.3.2).

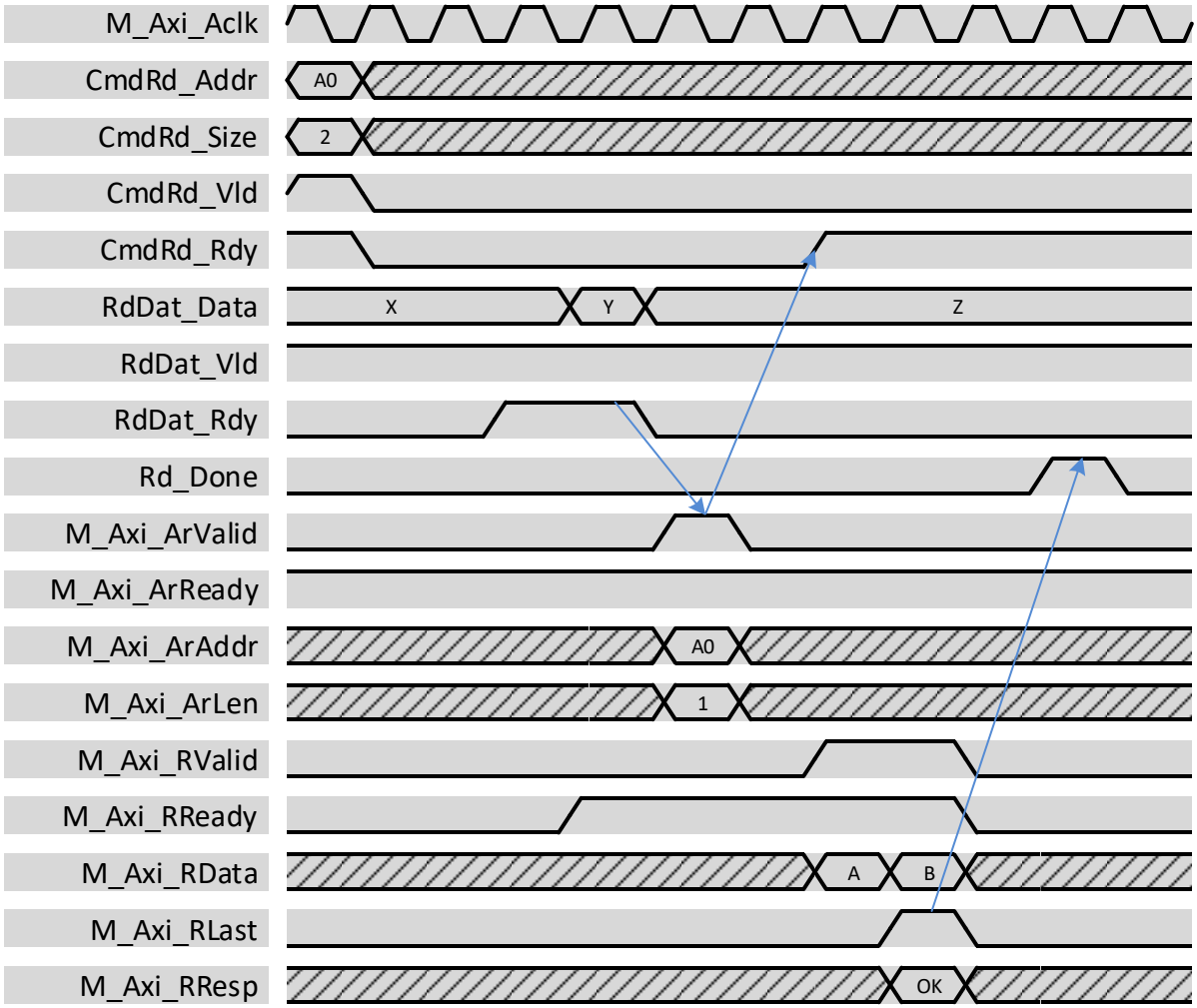


Figure 33: psi_common_axi_master_simple: Read transaction, high latency

Beause the command is issued in high-latency mode, the AXI read command is not issued until enough data is read from the FIFO in order for the command to complete in one burst. If this is the case, the AXI read command is issued and the transfer is completed in one burst with *M_Axi_RValid* high all the time.

This has the benefit of not blocking the AXI bus. In contrast to write commands, the high-latency mode does not lead to significantly more latency in the read-case because the user can still immediately read the first data after it was received. As a result, it is recommended to always execute read commands I the high-latency mode unless there is a very good reason for the low-latency mode.

10.3.3 Generics

AxiAddrWidth_g	Width of the AXI address bus
AxiDataWidth_g	Width of the AXI data bus
AxiMaxBeats_g	Maximum number of beats in one AXI transaction. Values given by the AXI specification are 16 for AXI-3 and 256 for AXI-4. However, the user may choose any other number for scheduling reasons.
AxiMaxOpenTransactions_g	Maximum number of AXI commands (AW/AR-channel) to send before the first command is completed (outstanding transactions).
UserTransactionSizeBits _g	Number of bits used to specify the number of beats to transfer on the user command interface. This is the only limiting factor for the transfer size requested.
DataFifoDepth_g	Number of entries in the read/write data FIFOs
ImplRead_g	Implement read functionality (can be disabled to save resources)
ImplWrite_g	Implement write functionality (can be disabled to save resources)
RamBehavior_g	Block-RAM style (must match FPGA architecture) “RBW” Read before write “WBR” Write before read

10.3.4 Interfaces

Signal	Direction	Width	Description
Control Signals			
M_Axi_Aclk	Input	1	Clock
M_Axi_Aresetn	Input	1	Reset (low active)
Write Command			
CmdWr_Addr	Input	*AxiAddrWidth_g*	Address to start writing at (must be aligned)
CmdWr_Size	Input	*UserTransactionSizeBits_g*	Number of beats in the transfer
CmdWr_LowLat	Input	1	'1' --> Low latency mode '0' --> High latency mode
CmdWr_Vld	Input	1	AXI-S handshaking signal
CmdWr_Rdy	Output	*1*	AXI-S handshaking signal
Read Command			
CmdRd_Addr	Input	*AxiAddrWidth_g*	Address to start reading at (must be aligned)
CmdRd_Size	Input	*UserTransactionSizeBits_g*	Number of beats in the transfer
CmdRd_LowLat	Input	1	'1' --> Low latency mode '0' --> High latency mode
CmdRd_Vld	Input	1	AXI-S handshaking signal
CmdRd_Rdy	Output	1	AXI-S handshaking signal
Write Data			
WrDat_Data	Input	*AxiDataWidth_g*	Write data
WrDat_Be	Input	*AxiDataWidth_g/8*	Byte enables for write data
WrDat_Vld	Input	1	AXI-S handshaking signal
WrDat_Rdy	Output	1	AXI-S handshaking signal
Read Data			
RdDat_Data	Output	*AxiDataWidth_g*	Read data
RdDat_Vld	Output	1	AXI-S handshaking signal
RdDat_Rdy	Input	1	AXI-S handshaking signal
Response			
Wr_Done	Output	1	Write command was completed successfully
Wr_Error	Output	1	Write command was completed but at least one transaction failed (AXI response from slave indicated an error)
Rd_Done	Output	1	Read command was completed successfully
Rd_Error	Output	1	Read command was completed but at least one transaction failed (AXI response from slave indicated an error)
AXI Master Interface			
M_Axi_*	*	*	AXI signals, see AXI specification

10.4 psi_common_axi_master_full

10.4.1 Description

This entity executes transactions requested through a simple command interface on an AXI bus according to all specifications. This entity includes FIFOs to buffer read- and write-data but not for the commands.

This this entity internally uses *psi_common_axi_master_simple* and works similarly Many generics are just forwarded to this component and the meaning of the low-latency function is the same. These topics are not explained in detail in this section, please refer to 10.3 for details.

For the case that the AXI data width is larger than the user data width, it is highly recommended to not use low-latency transfers. Due to the width conversion, the bandwidth is smaller than the maximum of AXI, so the bus would be blocked longer than required if low latency transfers are made.

The user can request transaction of any size and they will get split automatically in order to not exceed AXI bust size and other limitations. The response is sent to the user when his whole command is executed (which may involve multiple AXI transactions).

In contrast to the *psi_common_axi_master_simple*, this entity has the following additional features:

- Execution of unaligned and odd-sized transfers (alignment of the data according to AXI requirements)
- AXI data-width can be larger than user interface data width
- The transfer size is specified in bytes and not in beats (to allow uneven-length transfers)

Most of the logic in this entity is related to unaligned and odd-sized transfers. So if you only require an AXI data width that is larger than the user interface data width, you may consider using *psi_common_axi_master* along with *psi_common_wconv_n2xn* (with conversion) to achieve better performance with less resource utilization.

The *psi_common_axi_master_full* has some clock cycles of overhead for each command. As a result, large data transfers are efficient but the performance for very small transfers (only a few AXI bursts) is limited due to this overhead.

Read and write logic are fully independent. So reads and writes can happen at the same time.

There is no required timing relationship between command and data signals. So for writes the user can provide write data before, after or together with the command.

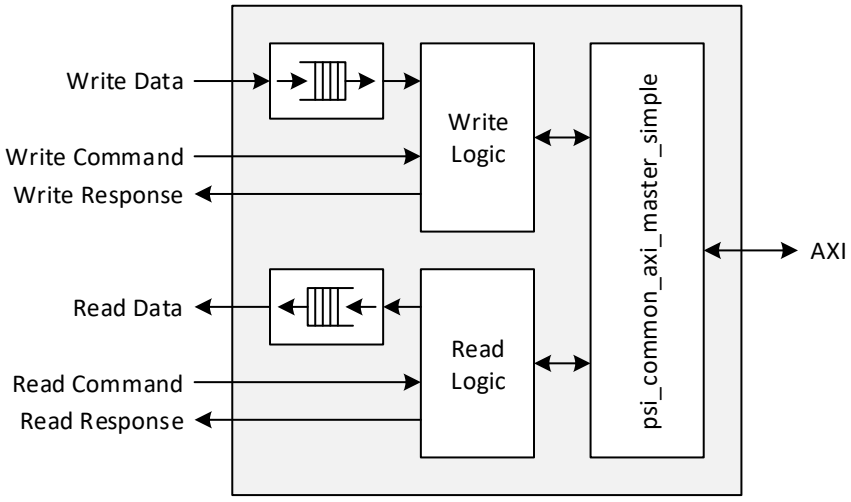


Figure 34: psi_common_axi_master_full: Block diagram

10.4.2 Transaction Types

For simplicity, only burst transactions are shown. However, of course also single-word transactions are supported.

Note that latencies and delays may be drawn shorter than they actually are to keep the waveforms small. However, all relationship between signals are correct.

For all figures, an AXI-width of 32 bits and a user data width of 16 bits is assumed to keep the figures simple.

For simplicity reasons, only transfers that consist of one AXI transaction are shown. For the user interface, larger transactions (consisting of multiple AXI bursts) do behave exactly the same.

For all transactions, user data is right-aligned. The byte at the start address specified in the command is the LSB of the user input/output data.

10.4.2.1 Read Transaction

The example below shows a read transaction.

Please read the description of all examples (10.4.2).

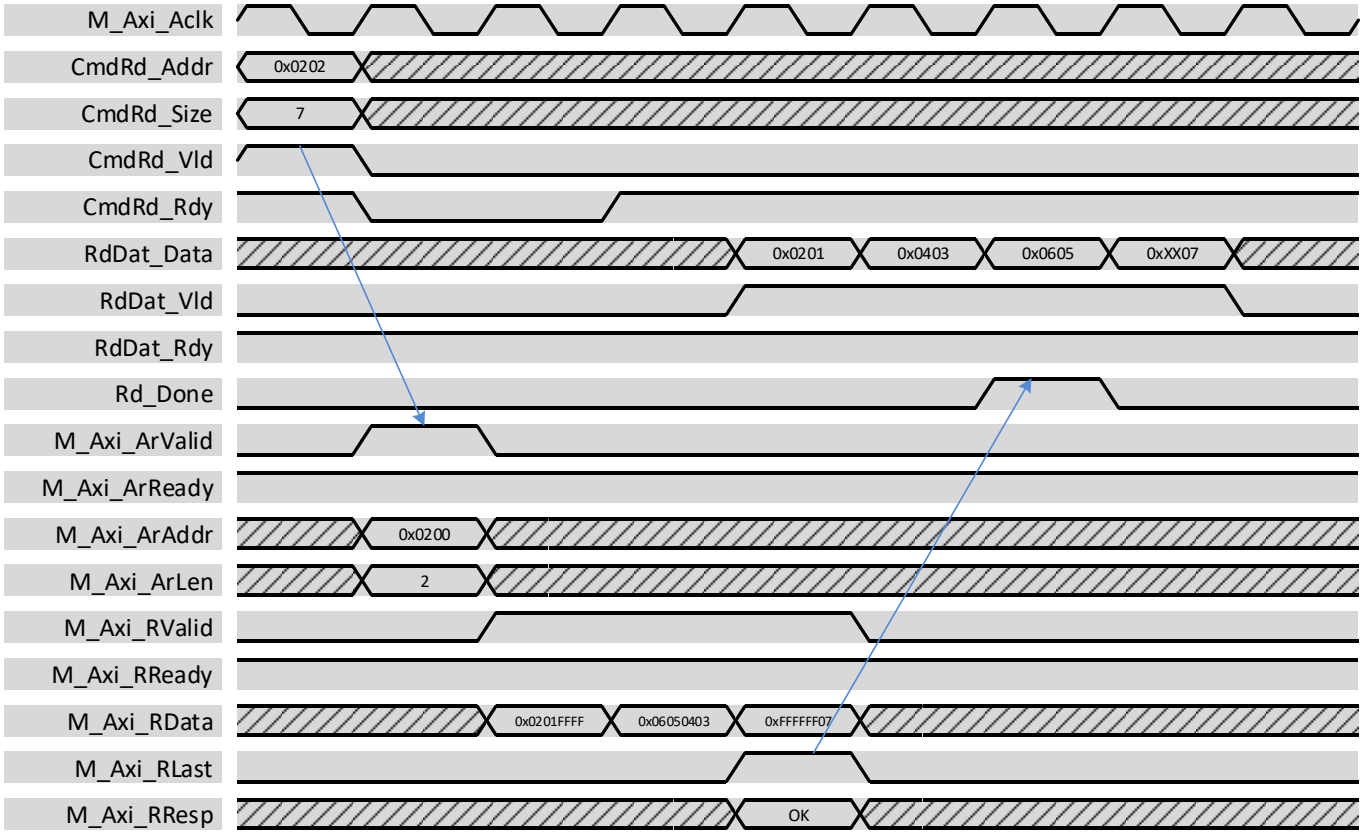


Figure 35: psi_common_axi_master_full: Read transaction

The read command together with the address of the first used byte and the size of the data required in bytes is asserted. The *psi_common_axi_master_full* then calculates the word-aligned AXI start address (0x0200) and the number of AXI-beats required (3 --> *M_Axi_ArLen*=2) and asserts the AXI AR-command.

The received data is aligned correctly in order to have the first byte the user requested as LSB of the first *RdDat* word. Unused trailing bytes may have any value, they shall never be interpreted.

Note that the *Rd_Done* signal is asserted as soon as all data is read from AXI, even if the data was not yet consumed on the *RdDat* interface.

10.4.2.2 Write Transaction

The example below shows a read transaction.

Please read the description of all examples (10.4.2).

Note that a high-latency write (AXI commands only sent after data is in buffer) is shown in this figure.

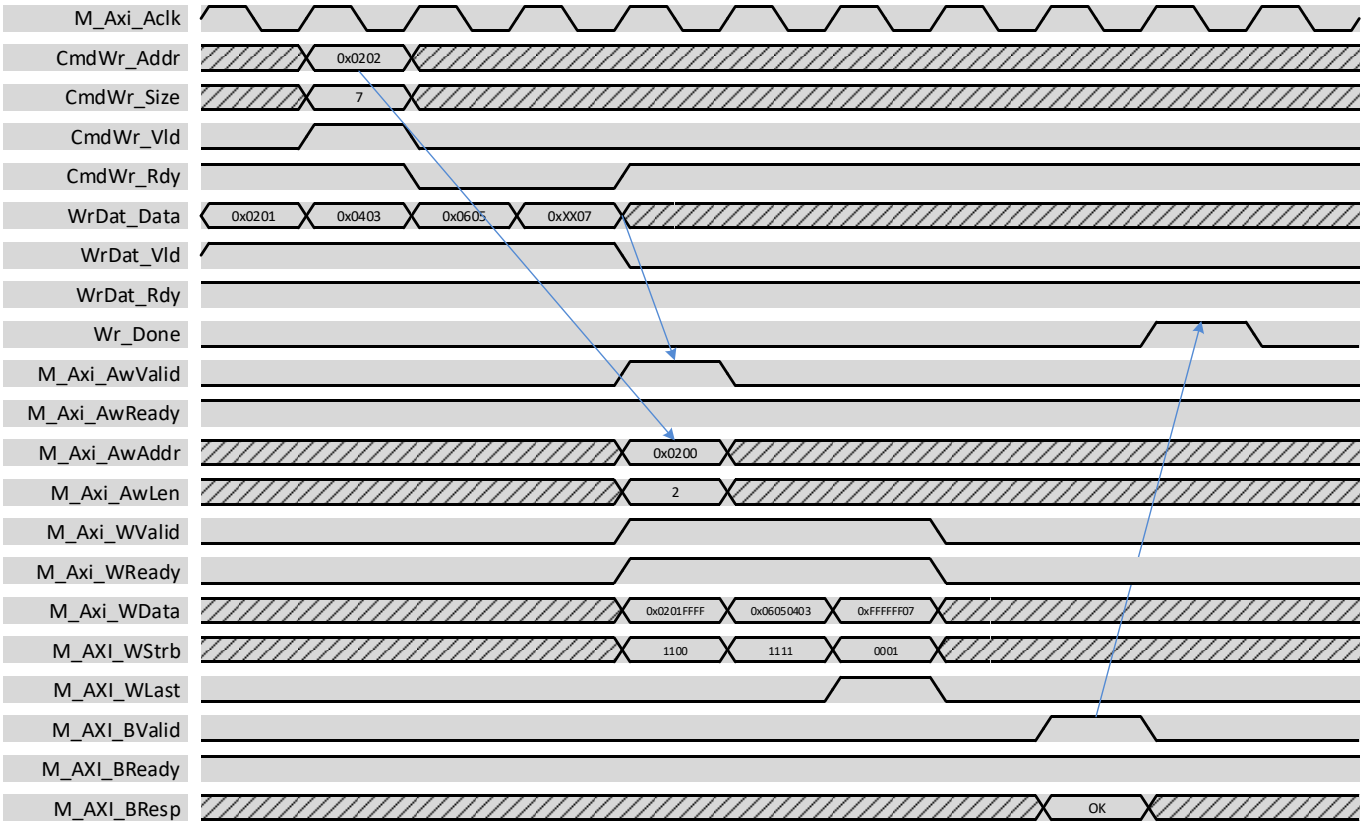


Figure 36: psi_common_axi_master_full: Write transaction

In this example, the user provides some data before the command. This is perfectly fine and allowed. Then the write command together with the address of the first byte to write and the size of the data that must be written in bytes is asserted. The *psi_common_axi_master_full* then calculates the word-aligned AXI start address (0x0200) and the number of AXI-beats required (3 --> *M_Axi_ArLen*=2) and asserts the AXI AW-command.

The user data is automatically aligned to the alignment of the AXI bus. Unused bytes may have any value, which is fine because the corresponding strobe signal is set to low. The same applies to the user interface: unused trailing bytes can have any value and do not influence the transaction.

As soon as the write response from AXI is received, the *Wr_Done* signal is asserted.

10.4.3 Generics

AxiAddrWidth_g	Width of the AXI address bus
AxiDataWidth_g	Width of the AXI data bus
AxiMaxBeats_g*	Maximum number of beats in one AXI transaction. Values given by the AXI specification are 16 for AXI-3 and 256 for AXI-4. However, the user may choose any other number for scheduling reasons.
AxiMaxOpenTransactions_g	Maximum number of AXI commands (AW/AR-channel) to send before the first command is completed (outstanding transactions).
UserTransactionSizeBits _g	Number of bits used to specify the number of bytes to transfer on the user command interface. This is the only limiting factor for the transfer size requested.
DataFifoDepth_g	Number of entries in the read/write data FIFOs (user side)
AxiFifoDepth_g	Number of entries in the FIFOs inside the AXI interface
DataWidth_g	Width of the user data interface
ImplRead_g	Implement read functionality (can be disabled to save resources)
ImplWrite_g	Implement write functionality (can be disabled to save resources)
RamBehavior_g	Block-RAM style (must match FPGA architecture) “RBW” Read before write “WBR” Write before read

10.4.3.1 FIFO Parametrization

This section explains the most important points about parametrizing buffer sizes on the user side (*DataFifoDepth_g*) and on the AXI side (*AxiFifoDepth_g)*.

The *DataFifoDepth_g* sets the depth of the FIFOs on the user interface as shown in the block diagram. For writes, this determines how much data can be accepted before a command is applied. The user side buffers have the width of the user-interface, so for writes, data in these buffers still must be converted to the AXI bus width (if *AxiDataWith_g* > *DataWidth_g*) and cannot be transmitted over AXI with ideal performance.

The *AxiFifoDepth_g* sets the depth of the FIFOs inside the AXI interface that have the same width as the AXI bus. These FIFOs can only be filled after a command is sent. On the other hand, the data buffered in these FIFOs can be sent over AXI with ideal performance, so this buffer shall be used to compensate effects due to the AXI bus being busy.

For reads, it does not matter much, in which FIFO the data is buffered since the bandwidth on the user-interface is always smaller or equal to the bandwidth of the AXI interface. As a result, the width-conversion does not lead to a bottleneck (like it is this case for writes).

10.4.4 Interfaces

Signal	Direction	Width	Description
Control Signals			
M_Axi_Aclk	Input	1	Clock
M_Axi_Aresetn	Input	1	Reset (low active)
Write Command			
CmdWr_Addr	Input	*AxiAddrWidth_g*	Address to start writing at (must be aligned)
CmdWr_Size	Input	*UserTransactionSizeBits_g*	Number of bytes in the transfer
CmdWr_LowLat	Input	1	'1' --> Low latency mode '0' --> High latency mode
CmdWr_Vld	Input	1	AXI-S handshaking signal
CmdWr_Rdy	Output	*1*	AXI-S handshaking signal
Read Command			
CmdRd_Addr	Input	*AxiAddrWidth_g*	Address to start reading at (must be aligned)
CmdRd_Size	Input	*UserTransactionSizeBits_g*	Number of bytes in the transfer
CmdRd_LowLat	Input	1	'1' --> Low latency mode '0' --> High latency mode
CmdRd_Vld	Input	1	AXI-S handshaking signal
CmdRd_Rdy	Output	1	AXI-S handshaking signal
Write Data			
WrDat_Data	Input	*DataWidth_g*	Write data (right-aligned)
WrDat_Vld	Input	1	AXI-S handshaking signal
WrDat_Rdy	Output	1	AXI-S handshaking signal
Read Data			
RdDat_Data	Output	*DataWidth_g*	Read data (right-aligned)
RdDat_Vld	Output	1	AXI-S handshaking signal
RdDat_Rdy	Input	1	AXI-S handshaking signal
Response			
Wr_Done	Output	1	Write command was completed successfully
Wr_Error	Output	1	Write command was completed but at least one transaction failed (AXI response from slave indicated an error)
Rd_Done	Output	1	Read command was completed successfully
Rd_Error	Output	1	Read command was completed but at least one transaction failed (AXI response from slave indicated an error)
AXI Master Interface			
M_Axi_*	*	*	AXI signals, see AXI specification

10.5 psi_common_axi_slave_ipif

10.5.1 Description

This entity implements a full AXI-4 slave interface that can be used to make custom IP-Cores accessible through AXI.

On the user interface (where the user code is attached), it supports using registers as well as access to synchronous memory (e.g. BRAMs). Burst are supported, also across the boundary between registers and memory range.

The limitations of this block are given below:

- It cannot be operated with memory only (at least 1 register must be used)
- The number of registers must be a power of two
- The latency of memory attached must be exactly one clock cycle
- AXI bus width is fixed to 32-bits

Especially the limitation of the memory latency to one clock cycle is suboptimal, since this prevents any additional pipelining in large IP-Cores.

For registers, this entity handles read/write registers completely independently. If readback of register values written via AXI should be possible, the user code must loop-back write values (*o_reg_wdata*) to read values (*i_reg_rdata*).

The memory range is placed in the memory map directly after the registers. Example: If 8 registers are implemented, the registers are at AXI addresses 0x00, 0x04, … 0x1C and memory starts at the AXI address 0x20.

The offset of the memory is removed internally. So in the example above, an access to the AXI address 0x24 (second memory cell) leads to the memory address (*o_mem_addr)* 0x04 because the offset of 0x20 is subtracted in the *psi_common_axi_slave_ipif* component.

10.5.2 IP Interface transactions

Only burst transactions of length 4 are shown in the waveforms for simplicity reasons. Single word transactions behave the same as length 1 bursts.

For all waveforms, an implementation with 4 registers (*NumReg_g* = 4) is assumed. Hence the memory range starts at address 0x10.

10.5.2.1 Register Write

When a register is written, a pulse on the corresponding *o_reg_wr* signal is asserted together with the new data value.

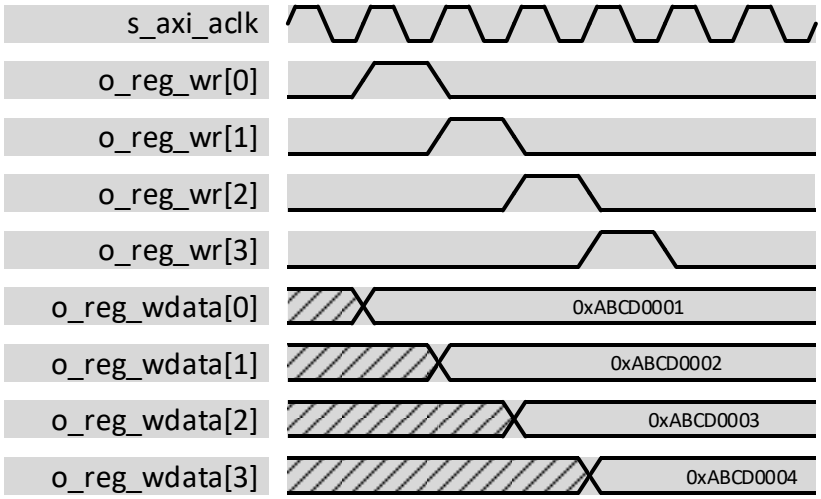


Figure 37: psi_common_axi_slave_ipif: Register Write

10.5.2.2 Register Read

When a register is read, its value is sampled together with the pulse being applied on the corresponding *o_reg_rd* signal. Hence he *o_reg_rd* signal can for example be used to acknowledge reading from a FIFO.

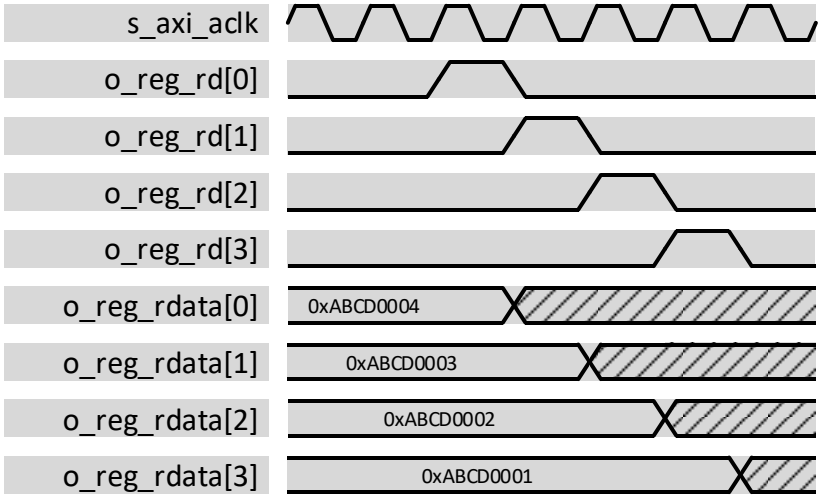


Figure 38: psi_common_axi_slave_ipif: Register Read

10.5.2.3 Memory Write

In this example, data in the AXI-addresses 0x12 … 0x1D is written. Since the example assumes four registers (addresses 0x00 … 0x0F), this translates to memory addresses 0x02 … 0x0D on the user interface, because the memory offset of 0x10 is subtracted internally.

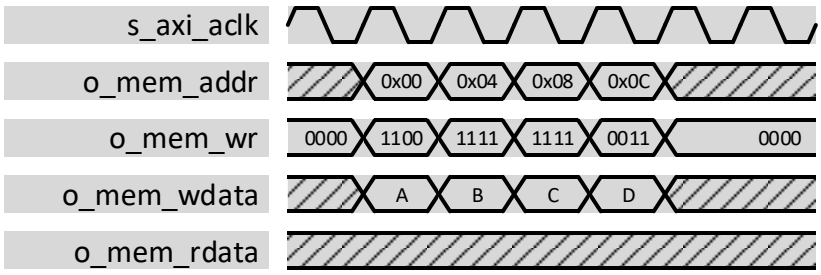


Figure 39: psi_common_axi_slave_ipif: Memory Write

10.5.2.4 Memory Read

In this example, data in the AXI-addresses 0x10 … 0x1F is read. Since the example assumes four registers (addresses 0x00 … 0x0F), this translates to memory addresses 0x01 … 0x0F on the user interface, because the memory offset of 0x10 is subtracted internally.

The example also nicely shows, that read data must be applied after exactly one clock cycle.

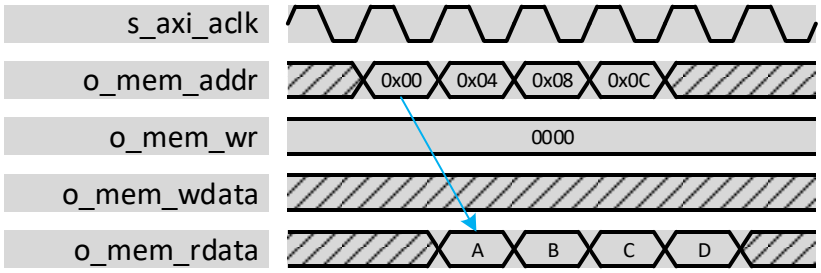


Figure 40: psi_common_axi_slave_ipif: Memory Read

10.5.2.5 Write over Register/Memory Boundary

In this example, four data words are written to the addresses 0x08 … 0x17. This includes two registers and two memory locations. Note that the register and memory interfaces are not delay compensated, therefore the first memory access happens at the same time as the last register access.

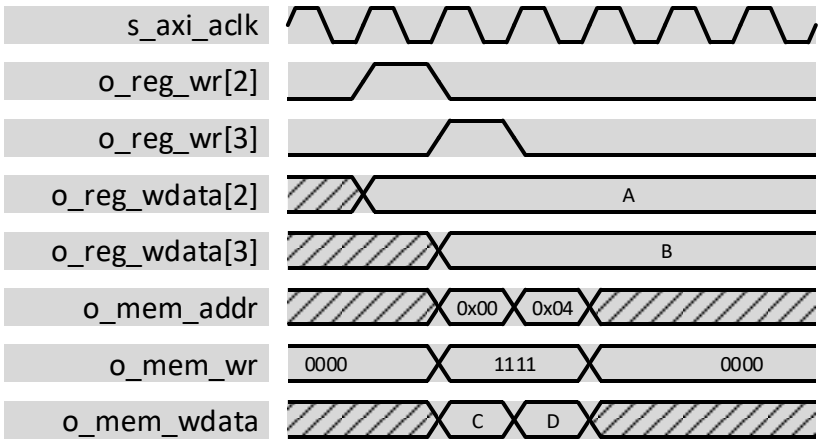


Figure 41: psi_common_axi_slave_ipif: Write over Register/Memory Boundary

10.5.3 Generics

NumReg_g	Number of registers to implement
ResetVal_g	Reset values for registers. The size of the array passed does not have to match *NumReg_g*, if it does not, the reset values are applied to the first N registers and the other registers are reset to zero.
UseMem_g	True = use memory interface, False = use registers only
AxiIdWidth_g	Number of bits used for the AXI ID signals
AxiAddrWidth _g	Number of AXI address bits supported

10.5.4 Interfaces

Signal	Direction	Width	Description
Control Signals			
s_axi_aclk	Input	1	Clock
s_axi_aresetn	Input	1	Reset (low active)
Register Interface			
o_reg_rd	Output	*NumReg_g*	Read-pulse for each register
i_reg_rdata	Input	*NumReg_g x 32*	Register read values
o_reg_wr	Input	*NumReg_g*	Write-pulse for each register
o_reg_wdata	Input	*NumReg_g x 32*	Register write values
Memory Interface			
o_mem_addr	Output	*AxiAddrWidth_g*	Memory address
o_mem_wr	Output	*4*	Memory byte write enables (one signal per byte)
o_mem_wdata	Output	32	Memory write data
i_mem_rdata	Input	32	Memory read data Must be valid one clock cycle after *o_mem_addr*
AXI Slave Interface			
s_axi_*	*	*	AXI signals, see AXI specification

11 Miscellaneous

11.1 psi_common_delay

11.1.1 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.

11.1.2 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.	

11.1.3 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

11.2 psi_common_pl_stage

11.2.1 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.

11.2.2 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)

11.2.3 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

11.3 psi_common_multi_pl_stage

11.3.1 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 11.2) entities.

11.3.2 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

11.3.3 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

11.4 psi_common_ping_pong

11.4.1 Description

This component implements a ping pong buffer mechanism around a ***single*** RAM block for multiple channels. It allows to stream data continuously in moving data back and forth at different memory space. While a buffer 'Ping' is written the other 'Pong' can be read. An interrupt is delivered when the buffer writing address swap, it gives the start indication to read.

The ***memory split between*** "buffers" which correspond to channels is defined as follow: the memory space necessary for one single channel will be extracted from the number of samples to be transferred and will be ceiled to its ***power of 2***. This means gaps may occur if one choose to forward 500 samples a memory fragmentation of 12 samples per canal is to be expected. This choice has been to ease the genericity of the component and its implementation. At the read port it is possible to select channel independent from sample. The ping pong component allows either receiving data in parallel either as time division multiplexed. Fig. 39 gives an overview of the TDM behavior whereas the fig. 40 shows the TDM mode.

If user wants to use a single channel then the parallel mode is required. **Beware**; data sample frequency ratio compared to the clock cannot be higher than the number of channel. It is not feasible to several channels simultaneously, in other words **only one sample per 'number of channels' clock cycles**. An associated test bench helps to determinate proper parameters.

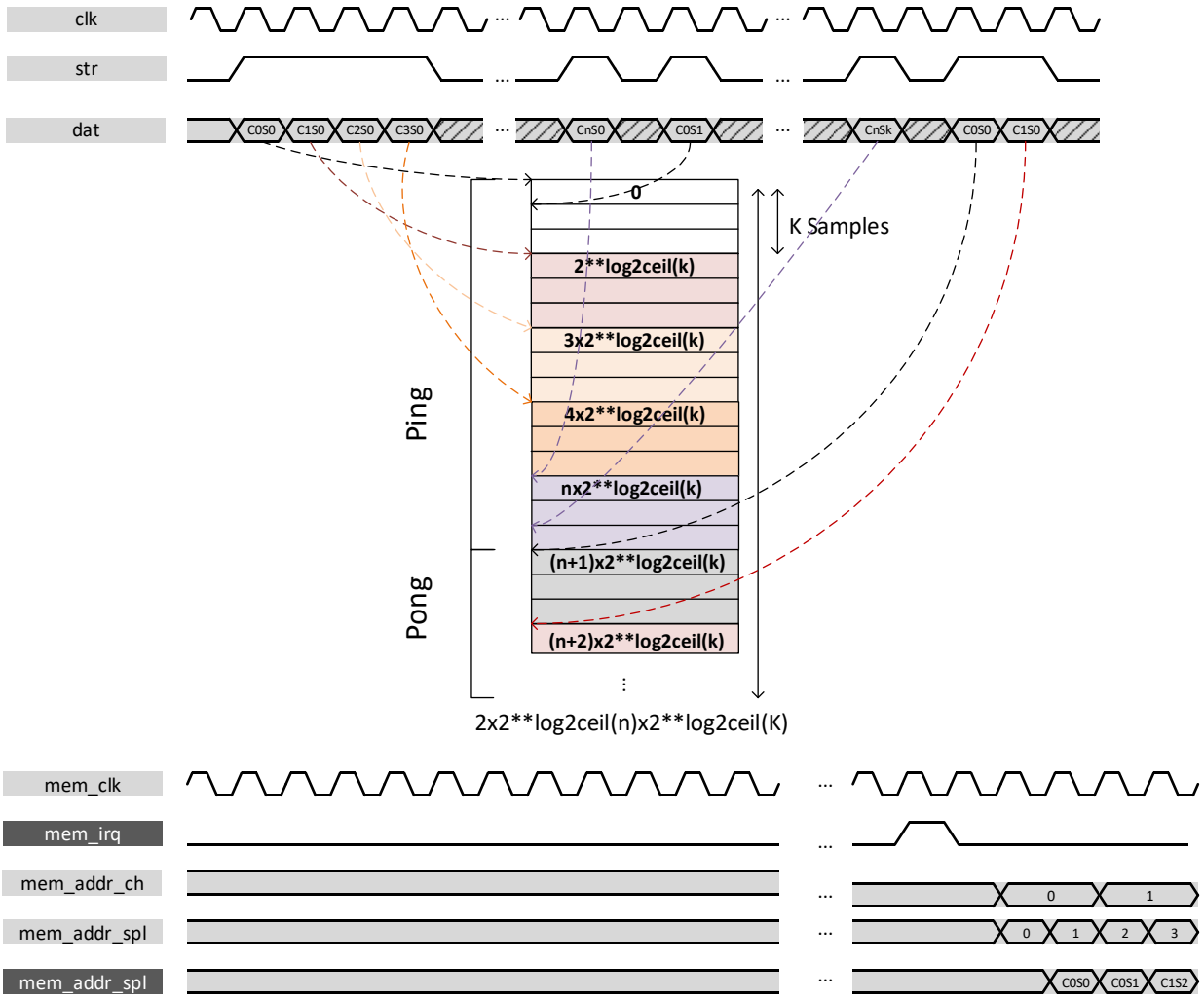


Figure 42: psi_common_ping_pong TDM mode

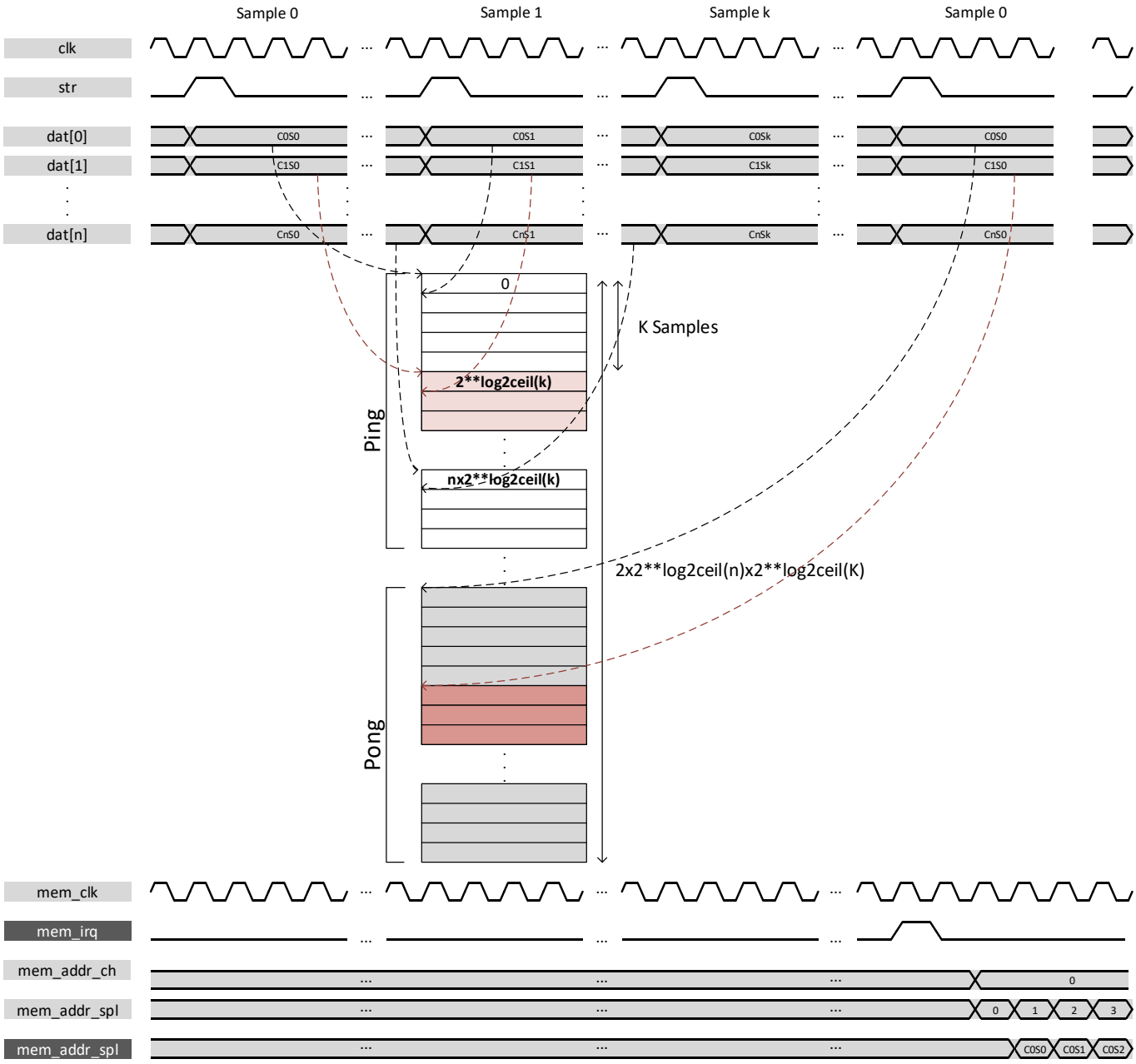


Figure 43: psi_common_ping_pong parallel

Both figures shows hypothetical behaviors for ping pong buffer and the following convention applies:

- *C: channel*
- *S: sample*
- *C0S1: Channel 0 & sample 1*

11.4.2 Generics

ch_nb_g	Number of Channel	
sample_nb_g	Number of sample to store	
dat_length_g	Vector width per channel	
tdm_g	True	TDM data flow mode
	False	Parallel mode
ram_behavior_g	“RBW”	Read-before-write implementation
	“WBR”	Write-before-read implementation
rst_pol_g	Reset polarity	

11.4.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
clk_i	Input	1	Clock processing
rst_i	Input	1	Reset processing
Input			
dat_i	Input	Generic	Width depends on the mode • parallel: *ch_nb_g* dat_length_g* • tdm: *dat_length_g*
str_i	Input	1	AXI-S handshaking signal
Memory interface			
mem_irq_o	Output	1	Interrupt signal high for one clock cycle that indicate that buffer swapped
mem_clk_i	Input	1	Clock memory
mem_addr_ch_i	Input	Log2ceil(ch_nb_g)	Address corresponding to channel (MSB) -
mem_addr_spl_i	Input	Log2ceil(sample_nb_g)	Address corresponding to sample (LSB) -
mem_dat_o	Output	dat_length_g	Data output

11.5 psi_common_delay_cfg

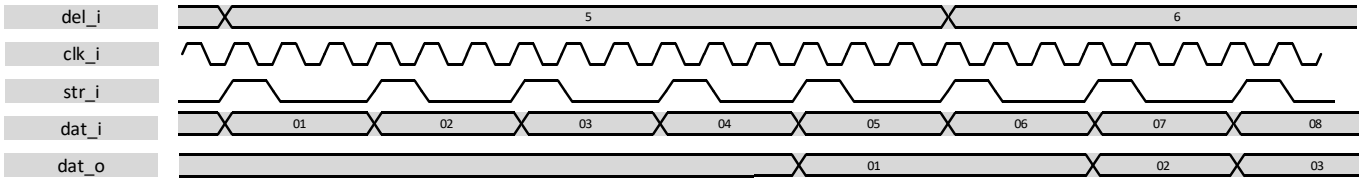
11.5.1 Description

This component is slightly the same as the ***psi_common_delay*** (cf. §11.1) but it is configurable during runtime. It allows setting the delay by a register. The architecture is based on block RAM and a SRL an overview is shown on next figure.

Since the latency is set by 3 registers to write, to read from block ram and to output the value a small shift register is implemented to output value when delay is inferior or equal to 3.

The delay value 0 isn't covered it will by default have a minimum of one clock cycle delay.

A generic allows to hold the value when a change is done to increase delay, this avoid having transient for example. A counter is launched to compute the delay difference and read address is hold during this time. This behavior can be skipped.



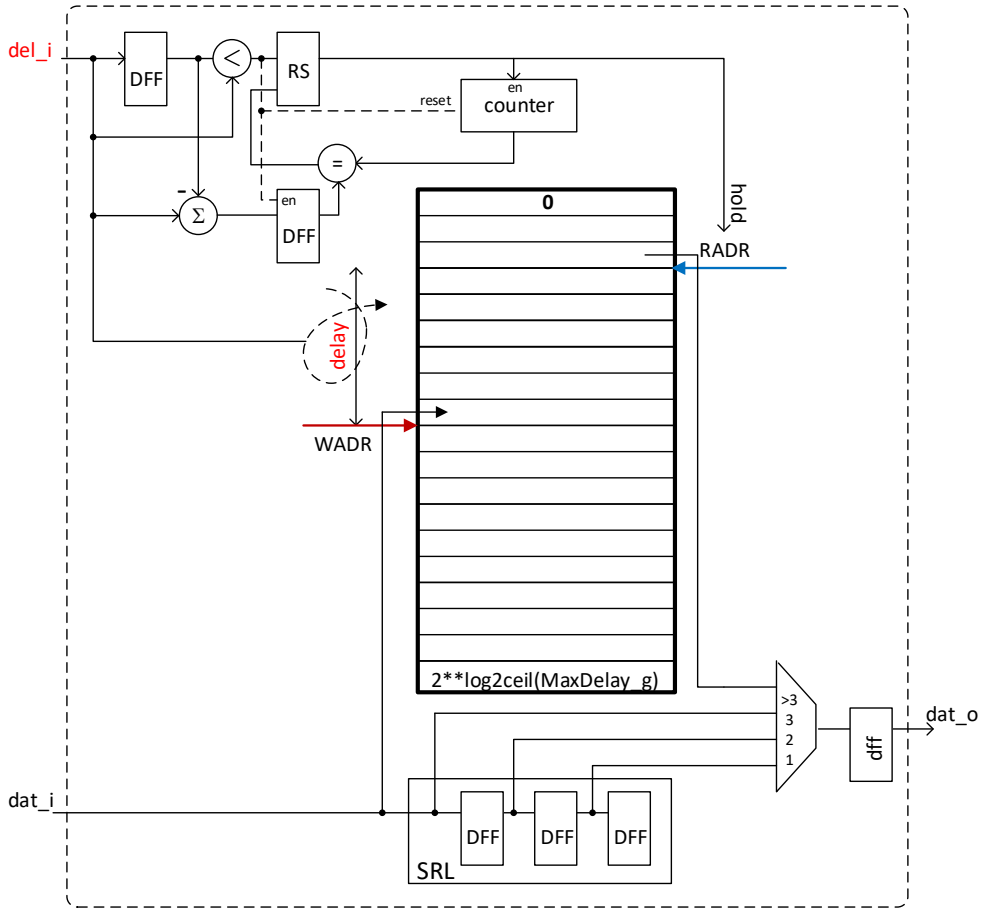


Figure 44: psi_common_delay_cfg: Hold behavior & pseudo-architecture

11.5.2 Generics

Width_g	Width of the data to delay
MaxDelay_g	Number of delay taps (*nb: The RAM size will be the next power of 2 of this valu*e)
RstPol_g	Polarity reset
RamBehavior_g	"RBW" Read-before-write implementation "WBR" Write-before-read implementation
Hold_g	**True** While a delay parameter value increase, it holds the last value that where generated and as soon as the delay has been compensated it output the expected value **False** output old value that where written previously into the RAM when the Delay excursion occurred

11.5.3 Interfaces

Signal	Direction	Width	Description
Control Signals			
clk_i	Input	1	Clock
rst_i	Input	1	Reset (polarity set by generic)
Data			
str_i	Input	1	InData valid (clock enable for shift register)
dat_i	Input	Width_g	Data input
del_i	Input	Log2ceil(MaxDelay_g)	Number of delay taps
dat_o	Output	Width_g	Data output

11.6 psi_common_watchdog

11.6.1 Description

This block is checking if an event signal is active within a predefined particular time period by user through generics. The typical usage of this block is to check pulse activity but vector can be fed in. It has three outputs, the missing counter value, warning flag and error flag, via the reset one can restart the missing value to zero and erase output flags. Two modes can be set via the generic "*thld_fault_succ_g*", setting a positive value will enable the mode which will count only successive missing events prior to rise a flag.

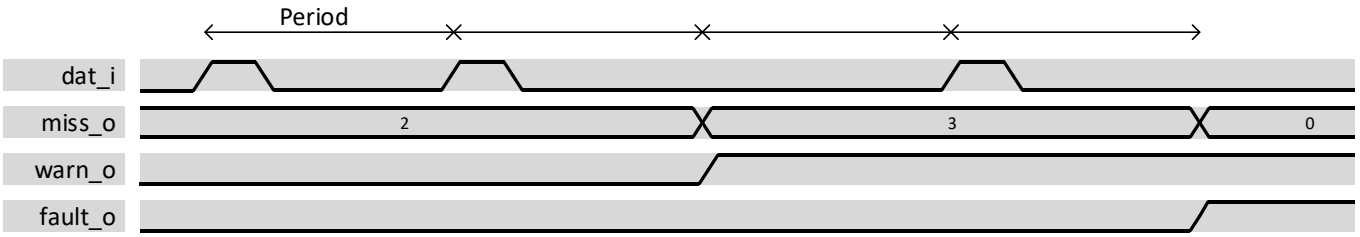


Figure 45: psi_common_watchdog, datagram total missing event mode

On the graph above the total amount of missing event is checked and the missing counter is not reset to 0 when an expected event occurs. Warning and Fault thresholds have been respectively set to 3 and 4. The successive fault threshold has been set 0 whereas on the graph below it has been set to 4, therefore as soon a new event occurs within period the missing counter is reset to 0 and flag are de-asserted.

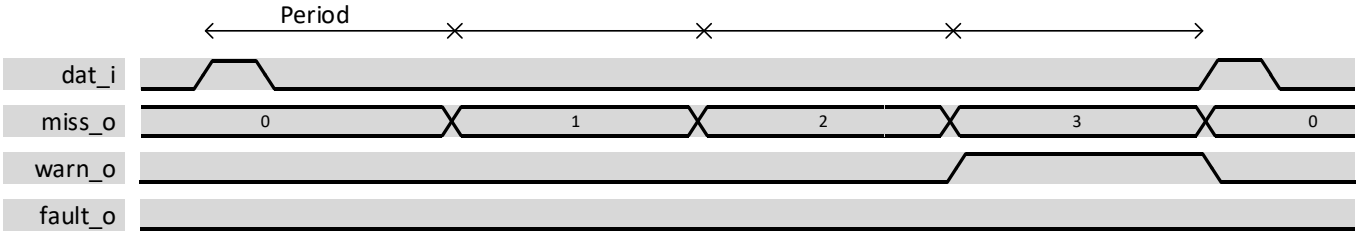


Figure 46: psi_common_watchdog, datagram successive missing event mode

11.6.2 Generics

freq_clk_g	Frequency clock
freq_act_g	Frequency at which the event should occur
thld_fault_total_g	Number of missing events that will rise the error flag
thld_warn_g	Number of missing events that will rise the warning flag
thld_fault_succ_g	Number of successive missing events that will rise the error flag **!n.b:** Setting this value to 0 will enable the total missing event behavior whereas setting positive value will enable successive missing events behavior
length_g	Data input vector length
rst_pol_g	reset polarity ('1' or '0')

11.6.3 Interfaces

Signal	Direction	Width	Description
clk_i	Input	1	Clock
rst_i	Input	1	Reset (polarity set by generic)
dat_i	Input	length_g	data input
warn_o	Output	1	When the number of missing event has reached the warning threshold value, the output is set to 1
miss_o	Output	Log2ceil(thdl_fault_g)	Number of missing event
fault_o	Output	1	Output is set to 1 when number of errors reached

11.7 psi_common_debouncer

11.7.1 Description

This component is a simple de-bouncer element where the filter time period is settable via generic as well as the polarity of both input and output. If the input is toggling during less time than the one set as generic in sec. the output won't be forwarded to the output. The counter start is triggered via the input change of state, once the counter reach the time predefined the output value is then forwarded depending on the desired polarity.

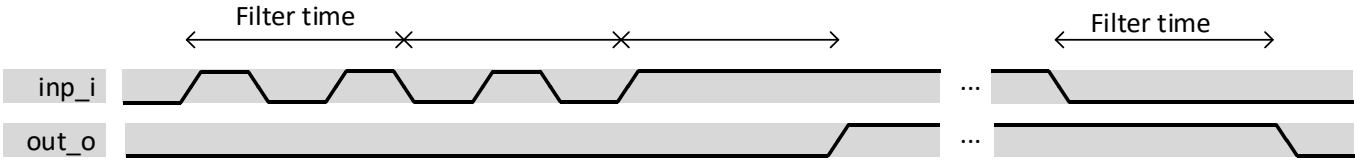


Figure 47: psi_common_debouncer datagram if len_g = 1

11.7.2 Generics

freq_clk_g	Frequency clock
dbnc_per_g	Filtering time in sec
sync_g	Add double stage synchronizer
out_pol_g	define the output polarity, active high or low
in_pol_g	define the input polarity, active high or low
rst_pol_g	reset polarity ('1' or '0')
len_g	vector input length

11.7.3 Interfaces

Signal	Direction	Width	Description
clk_i	Input	1	Clock
rst_i	Input	1	Reset (polarity set by generic)
inp_i	Input	len_g	data input
out_o	Output	len_g	data output