PCI IP Core
Design document

Authors: Miha Dolenc & Tadej Markovic
mihad@opencores.org
tadej@opencores.org

Rev. 0.1
January 28, 2002

This Page is Intentionally Blank

Revision History

Rev.	Date	Author	Description
0.0	13/11/01	Miha Dolenc Tadej Markovic	First Draft
0.1	24/01/02	Miha Dolenc Tadej Markovic	Testbench description added

Contents

1 Introduction

1.1 PCI IP Core Introduction

The PCI IP core (PCI bridge) provides an interface between the WISHBONE SoC bus and the PCI local bus. It consists of two independent units, one handling transactions originating on the PCI bus, the other one handling transactions originating on the WISHBONE bus.

The core has been designed to offer as much flexibility as possible to all kinds of applications.

The chapter 2 describes file hierarchy, description of modules, core design considerations and constants regarding the PCI IP Core.

The chapter 3 describes file hierarchy, description of modules, testbench design considerations, description of testcases and constants regarding the Testbench.

1.2 PCI IP Core Features

The following lists the main features of the PCI IP core:

- 32-bit PCI interface
- Fully PCI 2.2 compliant (with 66 MHz PCI specification)
- Separated initiator and target functional blocks
- Supported initiator commands and functions:
  - Memory Read, Memory Write

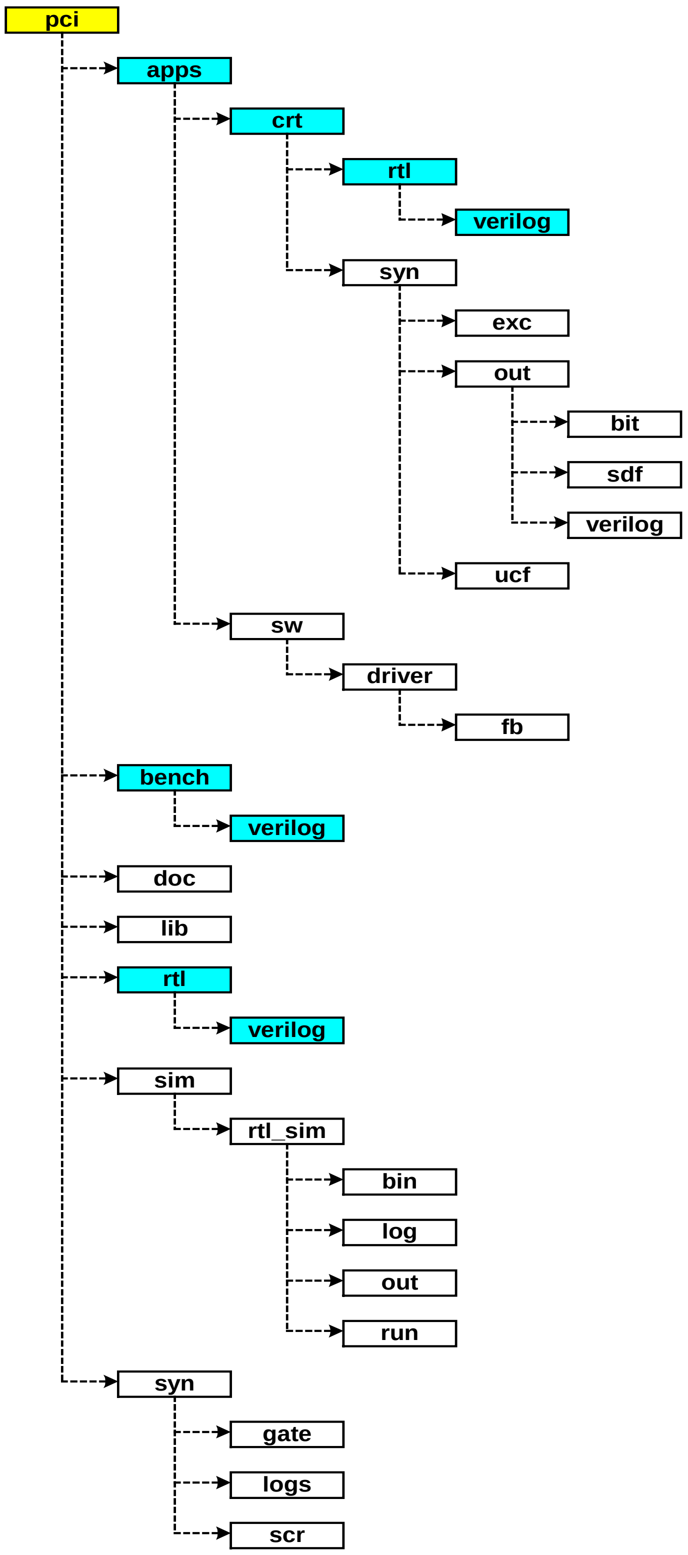
    - Memory Read Multiple (MRM)
    - Memory Read Line (MRL)
    - I/O Read, I/O Write
    - Configuration Read, Configuration Write
    - Bus Parking
    - Interrupt Acknowledge
    - Host Bridging
- Supported target commands and functions:
    - Type 0 Configuration Space Header

      (Type 0 is used to configure agents on the same bus segment)

      (Type 1 is used to configure across PCI-to-PCI bridges) Parity Generation (PAR), Parity Error Detection (PERR# and SERR#)
    - Memory Read, Memory Write
    - Memory Read Multiple (MRM)
    - Memory Read Line (MRL)
    - Memory Write and Invalidate (MWI)
    - I/O Read, I/O Write
    - Configuration Read, Configuration Write
    - Target Abort, Target Retry, Target Disconnect
    - Fast Back-to-Back Capable response
- Full Command/Status registers
- WISHBONE SoC Interconnection Rev. B compliant interface on processor side (master with Target PCI and slave with Initiator PCI interface)
- Configurable on-chip FIFOs

1.3 PCI IP Core Directory Structure

Following picture shows the structure of directories of the PCI IP Core.


pci
apps
crt
rtl
verilog
syn
exc
out
bit
sdf
verilog
ucf
sw
driver
fb
bench
verilog
doc
lib
rtl
verilog
sim
rtl_sim
bin
log
out
run
syn
gate
logs
scr

There are three major parts of the Verilog code in the **pci** directory. First one is the code for the PCI Bridge. The Verilog files are in the **pci\rtl\verilog** subdirectory. The second one is the code for the PCI Testbench. This files are used together with files for the PCI Bridge. There are also some exceptions, but those will be mentioned later. The Verilog files are in the **pci\bench\verilog** subdirectory. The third one is the code for the application, which is PCI Crt Controller. This files are used together with files for the PCI Bridge. There are also some exceptions, but those will be mentioned later. The Verilog files are in the **pci\apps\crt\rtl\verilog** subdirectory.

The documentation is in the subdirectory **pci\doc**. Documentation consists of PCI Bridge White Paper, PCI Bridge Product Preview, PCI IP Core Specification and PCI IP Core Design document.

Library files of the device specific subelements are in the subdirectory **pci\lib**. This device specific elements are Memory Blocks, Global Buffers and I/O Buffers. I/O Buffers used in FPGA are written in generic Verilog code and are put into FPGA IOB cells with constraint file. This files must be changed with device specific IOB cells when the design is used in ASIC. Memory Blocks and Global Buffers are device specific in both, FPGA and ASIC.

pci\sim subdirectory is used for running simulation – testbench. The **rtl_sim** subdirectory is used for RTL (functional) simulation of the core. There are a few directories included here:

- **bin** – includes various scripts needed for running Ncsim simulator
- **run** – the directory from which the simulation is ran. It provides a script for starting the simulation and a script for cleaning all the results produced by previous simulation runs
- **log** – Ncvlog, Ncelab and Ncsim log files are stored here for review. There is also a script provided for extracting errors from this logs.
- **out** – simulation output directory – simulation stores all the results into this directory (dump files for viewing with Signalscan, testbench text output etc.)

The **gate_sim** directory is used the same way as **rtl_sim**, except simulation is run with the netlist provided from synthesis tool.

Generated files from synthesis tools, like gate level Verilog and log files, are stored in the **pci\syn** subdirectory and its subdirectories.

Regarding the application, there are other subdirectories under **pci\apps** directory. User constraint file for pre-synthesis is in the **pci\apps\crt\syn\exc** subdirectory. This is used for preserving some of the hierarchy to meet PCI timing constraints in FPGA. User constraint file after synthesis and before implementation is in the **pci\apps\crt\syn\ucf** subdirectory. This is used for pin locations, pin types and timing constraints. Generated bit file from FPGA synthesis tool is stored in the **pci\apps\crt\syn\out\bit** subdirectory.

This is used for FPGA programming. Generated sdf file from FPGA synthesis tool is stored in the **pci\apps\crt\syn\out\sdf** subdirectory. This file contains all timing delays. Sdf file back annotated to Verilog is stored in the **pci\apps\crt\syn\out\verilog** subdirectory. This file is used for timing simulation.

2

PCI Bridge Core

2.1 Overview

The PCI bridge consists of two units: the PCI target unit and the WISHBONE slave unit. Each holds its own set of functions to support bridging operations from WISHBONE to PCI and from PCI to WISHBONE. The WISHBONE slave unit acts as a slave on the WISHBONE side of the bridge and initiates transactions as a master on the PCI bus. The PCI target unit acts as a target on the PCI side of the bridge and as a master on its WISHBONE side. Both units operate independently of each other.

The PCI interface is *PCI Specification 2.2* compliant, whereas the WISHBONE is *SoC Interconnection Specification Rev. B* compliant. The WISHBONE implementation carries out a 32-bit bus operation and does not support other bus widths.

WISHBONE COMPATIBLE

2.2 Core File Hierarchy

The hierarchy of modules in the PCI Bridge core is shown here with file tree. Each file here implements one module in a hierarchy RTL source files of the PCI Bridge core are in the **pci\rtl\verilog** subdirectory.

```
top.v
        pci_bridge32.v
        .       wb_slave_unit.v
        .       .       wb_slave.v
        .       .       wbw_wbr_fifos.v
        .       .       .       pci_tpram.v *
        .       .       .       wbw_fifo_control.v
        .       .       .       wbr_fifo_control.v
        .       .       wb_addr_mux.v
        .       .       .       decoder.v **
        .       .       delayed_sync.v *
        .       .       .       synchronizer_flop.v **
        .       .       delayed_write_reg.v
        .       .       conf_cyc_addr_dec.v
        .       .       pci_master32_sm_if.v
        .       .       pci_master32_sm.v
        .       .       .       frame_crit.v
        .       .       .       frame_load_crit.v
        .       .       .       irdy_out_crit.v
        .       .       .       mas_ad_load_crit.v
        .       .       .       mas_ch_state_crit.v
        .       .       .       mas_ad_en_crit.v
        .       .       .       cbe_en_crit
        .       .       .       frame_en_crit.v
        .       pci_target_unit.v
        .       .       wb_master.v
        .       .       pciw_pcir_fifos.v
        .       .       .       pci_tpram.v *
        .       .       .       pciw_fifo_control.v
        .       .       .       fifo_control.v
        .       .       delayed_sync.v *
```

. . . **synchronizer_flop.v** **
. . **pci_target32_interface.v**
. . . **pci_decoder.v** **
. . **pci_target32_sm.v**
. . . **pci_target32_clk_en.v**
. . . **pci_target32_trdy_crit.v**
. . . **pci_target32_stop_crit.v**
. . . **pci_target32_devs_crit.v**
. **conf_space.v**
. . **synchronizer_flop.v** **
. . **sync_module.v****
. . . **synchronizer_flop.v** **
. **pci_io_mux.v**
. . **pci_io_mux_ad_en_crit.v** **
. . **pci_io_mux_ad_load_crit.v** **
. . **out_reg.v** **
. **cur_out_reg.v**
. **pci_in_reg.v**
. **pci_parity_check.v**
. . **par_crit.v**
. . **perr_crit.v**
. . **perr_en_crit.v**
. . **serr_crti.v**
. . **serr_en_crit.v**
. **pci_rst_int.v**
bufif0 ***

* Files are used within more than one module: **pci_tpram.v** is used within wbw_wbr_fifos.v and pciw_pcir_fifos.v files, **delayed_sync.v** is used within wb_slave_unit.v and pci_target_unit.v files.

** Files are used within one module more than once: **synchronizer_flop.v** is used within delayed_sync.v for 5 times and within conf_space.v for several times as **sync_module.v** etc., **decoder.v** is used within wb_addr_mux.v for as much times as there is the number of WB images, **pci_decoder.v** is used within pci_target32_interface.v for as much times as there is the number of PCI images, **pci_io_mux_ad_en_crit.v** and **pci_io_mux_ad_load_crit.v** are used within pci_io_mux.v for 4 times each, **out_reg.v** is used within pci_io_mux.v for 47 times (for AD, CBE and Control signals).

*** Instantiation **bufif0** is a generic tri-state buffer with active low output enable' and is used within top.v for 47 times (for AD, CBE and Control signals). This is Verilog generic primitive and it works fine in FPGA implementation (I/O pads are defined in constraint files). For an ASIC implementation, user must edit **top.v** file to instantiate PCI I/O pads before synthesys.

2.2.1 Core Module Hierarchy

Module hierarchy is shown in detail in the following four pictures. First picture explains how the following three pictures should be viewed as a whole. Dashed line means a boundary between two pictures. Fourth picture shows connections between modules in the PCI target Unit (PCIU). Third picture shows connections between modules in the WB slave Unit (WBU) and the Configuration space connections. Second picture shows connections between modules in the PCI Input/Output interface (I/O).

The name of each module is upper-case name of the respective file. Description of modules and their connections is in the chapter Description of Core Modules.

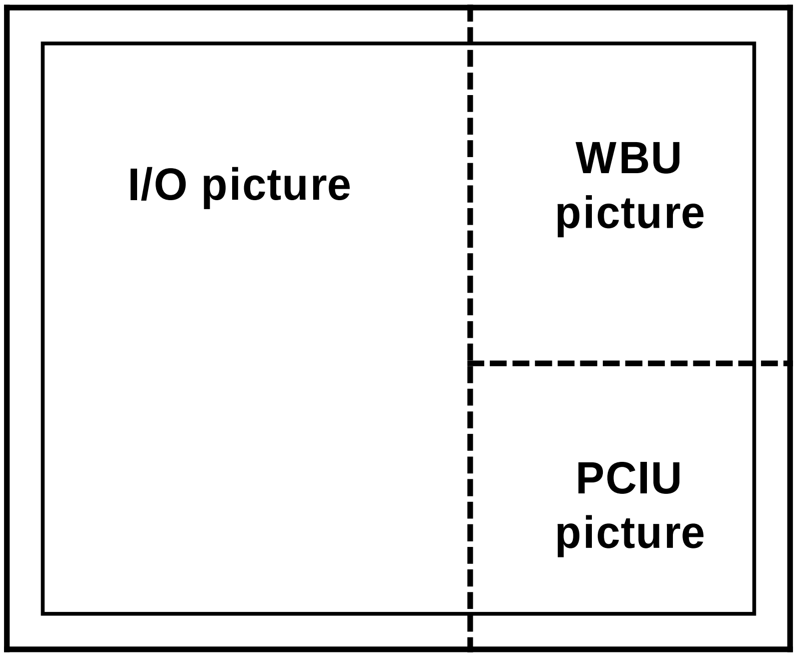


Figure 2-1: First picture - Distribution of modules hierarchy into three pictures

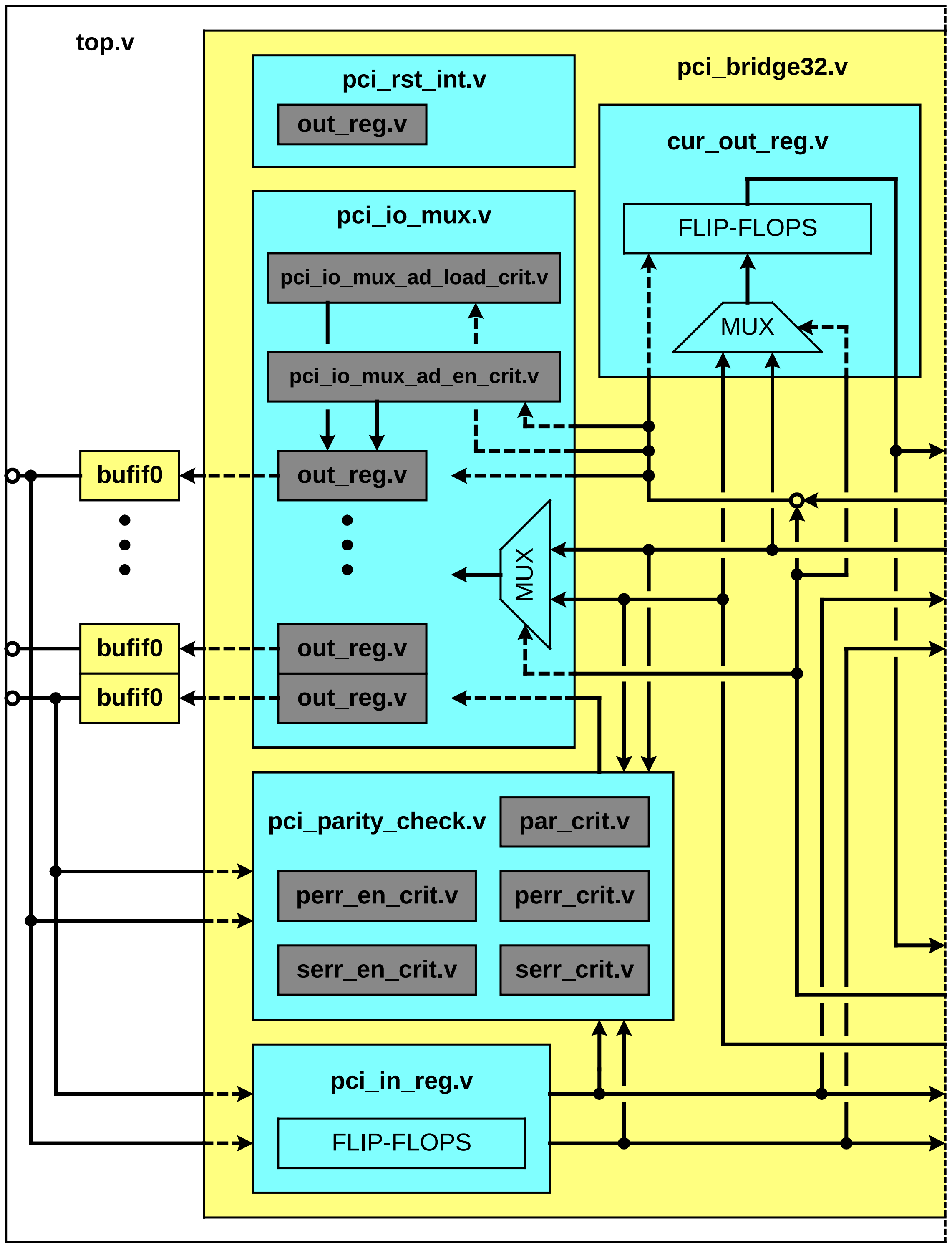


Figure 2-2: Second picture - Module hierarchy of PCI I/O interconnection

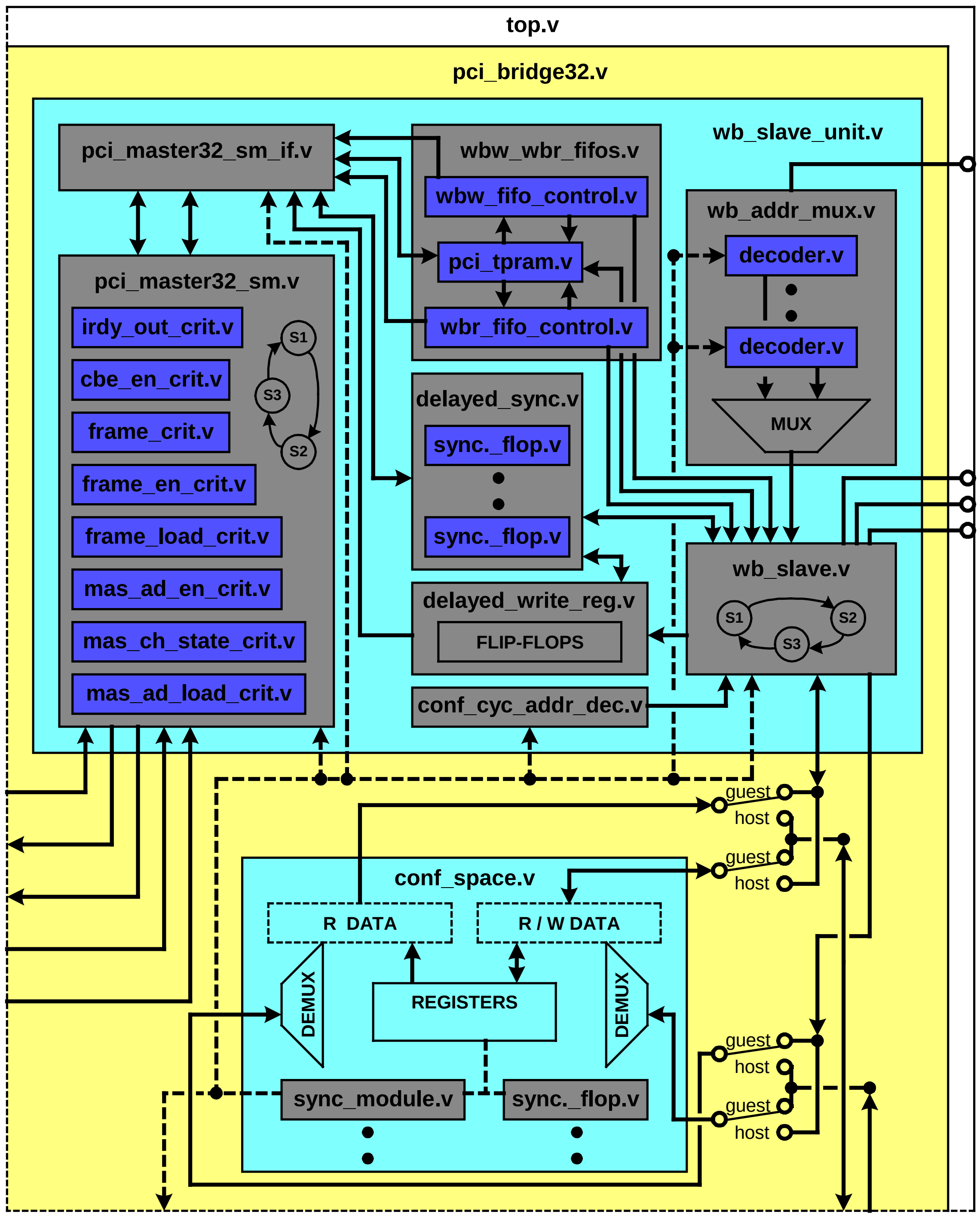


Figure 2-3: Third picture - Module hierarchy of WB slave Unit and Configuration space

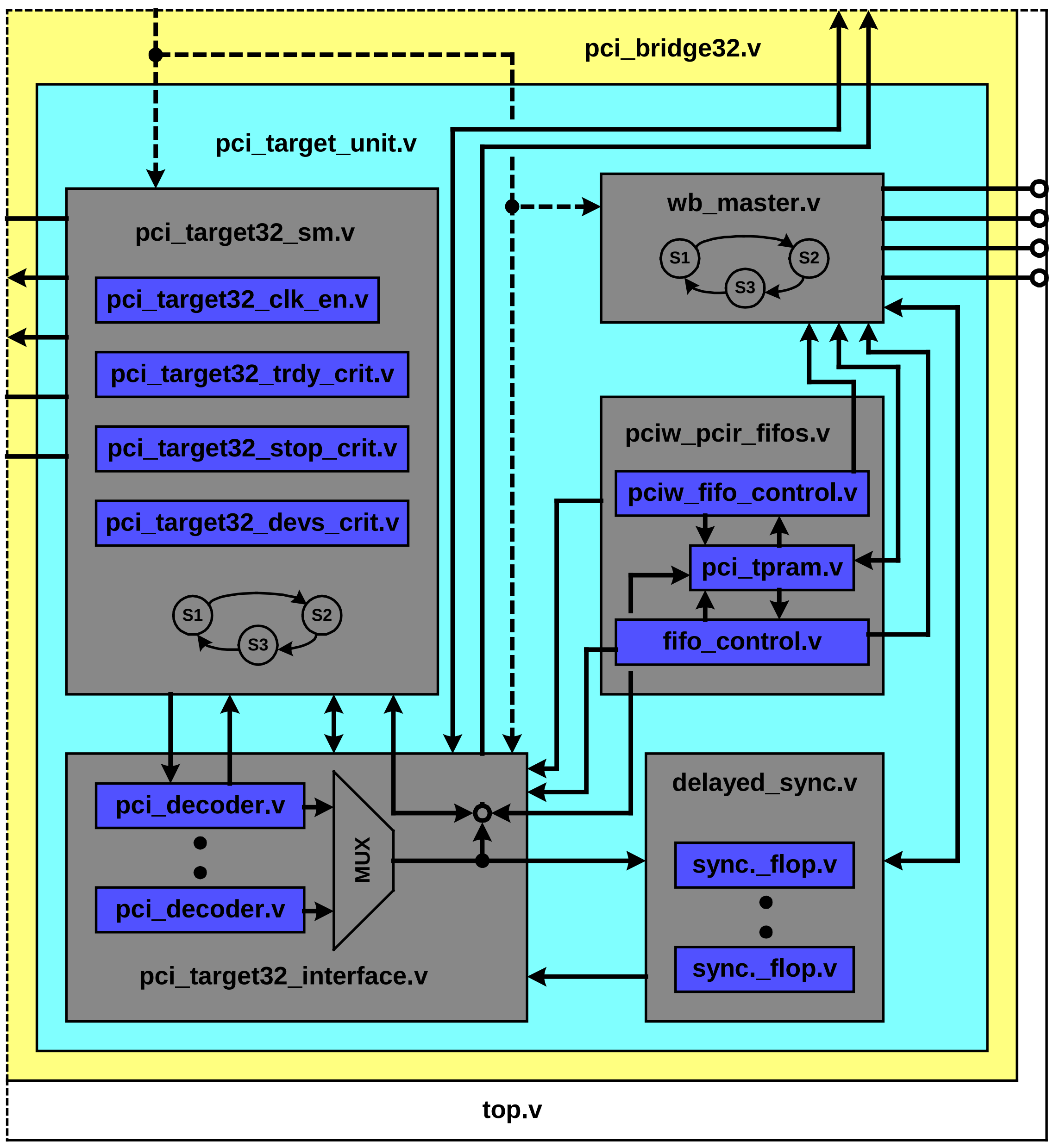


Figure 2-4: Fourth picture - Module hierarchy of PCI target Unit

2.3 Description of Core Modules

2.3.1 Description of general core submodules

The module ***top.v*** consists only of module ***pci_bridge32.v*** and Verilog instances ***bufif0*** for PCI Input and Output buffers.

The module ***pci_bridge32.v*** consists of all major submodules and is, like the module ***top.v***, without any logic. It includes some Verilog pre-compiler directives for connections with Configuration space, depending on the HOST or GUEST implementation of the core. If PCI bridge is implemented as HOST, then read/write connection is provided for WB slave Unit (accesses from WB bus) and read-only connection is provided for PCI target Unit (accesses from PCI bus). If PCI bridge is implemented as GUEST bridge, then read/write connection is provided for PCI target Unit and read-only connection for WB slave Unit.

2.3.2 Description of PCI I/O submodules

PCI I/O modules are shown on the Figure 2-2: Second picture - Module hierarchy of PCI I/O interconnection.

There are two sets of signals connected to PCI bus through I/O modules. One set of signals is from WB slave Unit and the other set of signals is from PCI target Unit. All output signals, connected to PCI bus, from WB slave and PCI target Unit, also have output enable signals.

2.3.2.1 pci_io_mux submodule

The module ***pci_io_mux.v*** is used for multiplexing output signals from WB slave and PCI target Unit before registering them in corresponding output Flip-Flops. The signal used as a select multiplexer signal is registered AD output enable from PCI target Unit, **tar_ad_en_reg_in** (from ***cur_out_reg.v***). This way the PCI master in the WB slave Unit can use the PCI output signals when Target is not operating. This is done to implement bus parking capability.

All output signals and their appropriate output enable signals are registered. Each output Flip-Flop and corresponding output enable Flip-Flop are implemented in **out_reg.v** file. In addition to data input, each of these Flip-Flops has an asynchronous reset and clock enable ports. AD output Flip-Flops are shared between Master and Target state machines, other outputs are controlled exclusively. Outputs from this Flip-Flops are driven into ***bufif0*** instances (for PCI I/O pins) in the top level module.

Loading (clock enable) of AD output Flip-Flops is in part calculated in **pci_io_mux.v**, while timing critical calculation is done in a **pci_io_mux_ad_load_crit.v**. There are four of these sub-modules instantiated, one for each byte, to reduce fan-out. Preserve hierarchy constraint can be selected on these modules, to prevent logic optimization in them and thus satisfy PCI input timing constraints.

Somewhat similar module and functionality is provided for AD output enable Flip-Flops driving. It is implemented in **pci_io_mux_ad_en_crit.v.**

2.3.2.2 cur_out_reg submodule

The opeartion of ***cur_out_reg.v*** submodule is the same as ***pci_io_mux.v***, so it consists of multiplexer and flip-flops and the same input signals. It is used for back-up information of signals driven on the PCI bus for WB slave and PCI target Units. This had to be done for FPGA implementation where IOB Flip-Flops are used – they can only have a fanout of 1, so register replication has to be done in order to use this information. This is also OK for ASIC implementation, since providing low fanout on output flip-flops reduces delay on output ports and maybe prevents buffer insertions.

2.3.2.3 pci_in_reg submodule

The module ***pci_in_reg.v*** is used for registering the input PCI bus signals. Registered inputs are used wherever and as much as possible, since stringent PCI bus input timing constraints don't allow free usage of unregistered inputs. Newertheless, PCI bus control signals must be used unregistered in some cases because of PCI bus protocol (eg: Master must deassert FRAME# on the same clock it detects STOP# asserted or Target must deassert DEVSEL#, STOP# and TRDY# on the same clock it detects FRAME# deasserted and IRDY# asserted).

2.3.2.4 pci_rst_int submodule

The module ***pci_rst_int.v*** is used for basic logic for reset and interrupt pins.

If PCI bridge is implemented as HOST bridge, then **RST_I** input from WB bus is recognized as a main asynchronous reset source. The module connects it to PCI core's internal reset signal and PCI bus **RST** signal. Soft Reset bit in configuration space also causes a reset on PCI bus, but does not have any effect on PCI Core's internal logic. Reset signal **RST_O** to WB bus is always inactive in this case. Some special design considerations must be taken into account when using Software Reset – PCI Core doesn't take care of Device Initialization time as specified in PCI Local Bus Specification v. 2.2. Software must take care of this by waiting the ammount of time required by the PCI Specification before starting any transactions on the PCI bus. This wouldn't influence PCI Core's logic, but could cause errors to occur or make PCI subsystem unstable. Refer to PCI Local Bus Specification v. 2.2., chapter 4.3.2. Special design considerations are also in order for power up reset which comes from **RST_I** signal on WB bus. This reset must be implemented in compliance with PCI Local Bus Specification!

If PCI Core is implemented as GUEST bridge, then **RST** input from PCI bus is recognized as a main asynchronous reset source. The module connects it to PCI core's internal reset signal and WB bus **RST_O** signal. Soft Reset bit in configuration space also causes a reset on WB bus, but does not have any effect on PCI Core's internal logic. When using Soft Reset bit, designer must take into consideration its application's reset implementation, to provide valid reset sequnce. For PCI bus reset operation, which is propagated to WB bus, refer to PCI Local Bus Specification v. 2.2., chapter 4.3.2.

If PCI Core is implemented as HOST bridge, then its **INTA#** PCI pin is configured as an input pin (output is always tri-stated). Its negated, synchronized to WB clock domain and passed to WB bus through **INT_O** pin, if interrupt propagation is enabled. (interrupt propagation enable bit of interrupt control register in ***conf_space.v***). Some software considerations come into account here – since interrupt is synchronized, it induces a certain latency to pass through the bridge. When software clears an interrupt request at its source, it should not immediately start accepting new interrupts. There are two possibilities:

- *Interrupt is cleared with a write to the source of interrupt:* Writes through the bridge are posted, therefore software cannot know and must not assume a time when this write will finish on PCI bus. A common thing a software would do would be to perform a write transaction to the source and immediately after that perform a read transaction. This would cause a write transaction to finish on PCI bus before read transaction can. When read transaction is finished on WB bus, software knows for sure, that all writes posted before a read are finished (except in case of an error). This should mean the source of interrupt was cleared, and WB agent can start accepting new interrupt requests.
- *Interrupt is cleared with a read from the source of interrupt:* When this read is finished, interrupt request should already be cleared – except in case of a source that also induces latency to clearing the interrupt. Software should take that into the account and wait appropriate number of cycles before accepting new requests.

If PCI Core is implemented as GUEST bridge, then its **INTA#** PCI pin is configured as an output pin. **INT_I** from WB bus is negated, synchronized to PCI clock domain and passed to PCI bus through **INTA#** pin, if interrupt propagation is enabled. (interrupt propagation enable bit of interrupt control register in ***conf_space.v***). Some software considerations come into account here – since interrupt is synchronized, it induces a certain latency to pass through the bridge. When software clears an interrupt request at its source, it should not immediately start accepting new interrupts. There are two possibilities:

- *Interrupt is cleared with a write to the source of interrupt:* Writes through the bridge are posted, therefore software cannot know and must not assume a time when this write will finish on WB bus. A common thing a software would do would be to perform a write transaction to the source and immediately after that perform a read transaction. This would cause a write transaction to finish on WB bus before read transaction can. When read transaction is finished on PCI bus, software knows for sure, that all writes posted before a read are finished (except in case of an error). This should mean the source of interrupt was cleared, and PCI agent can start accepting new interrupt requests.
- *Interrupt is cleared with a read from the source of interrupt:* When this read is finished, interrupt request should already be cleared – except in case of a source that also induces latency to clearing the interrupt. Software should take that into the account and wait appropriate number of cycles before accepting new requests.

Since PCI Core can also cause interrupt requests, **pci_rst_int.v** module handles passing those to appropriate output pins too. For HOST bridge implementation interrupts will be passed to **INT_O** pin on WB bus, for GUEST bridge implementation they will be passed to **INTA#** pin on PCI bus. Some internal synchronization is possible (depending on the source of interrupt), so software should make sure not to accept new interrupt requsts from the bridge for at least five cycles after interrupt status is cleared.

2.3.2.5 pci_parity_check submodule

The module ***pci_parity_check.v*** is used for generating/checking parity on PCI bus during address and data phases.

The module is implemented in such a way, that no special signals are needed from PCI Master or Target state machines. Its operation is based primarily on monitoring Master and Target output enable and PCI input signals.

When Master state machine is driving AD bus, parity calculation is based on AD and CBE outputs it provides. Appropriate parity value is driven to PCI bus on **PAR** pin one clock after corresponding AD and CBE values. **PERR#** signal is sampled on next clock. If it is sampled asserted, Parity Error Detected signal is generated and Master Parity Error is signaled, if Parity Error Response is enabled in configuration space. PCI Master reads are decoded using its FRAME and IRDY output enable signals. If those signals are enabled and AD bus is not, then Master Read is in progress. Operation now switches from generating value on **PAR** signal and monitoring **PERR#** signal, to monitoring **PAR** signal and generating **PERR#** signal. Valid parity is calculated from CBE signals provided by PCI Master state machine and AD signals provided by external Target. AD signals are monitored only on cycles on which **TRDY#** is sampled asserted. Value on **PAR** signal received on next clock cycle is xor –ed with parity calculated from CBE and

AD signals immediately. Result must be 0, otherwise Detected Parity Error is signalled. If Parity Error Response in Configuration space is enabled also, when this event occurs, then **PERR#** is asserted for one clock cycle and Master Data Parity Error signal is asserted to set appropriate status bits in Configuration space.

Reads from PCI bus through Target state machine are decoded with Target control signals and AD bus output enables. When Target state machine is driving control signals and AD bus, this means a read through Target is in progress. Appropriate parity value is calculated from Target – provided AD value and external Master-provided CBE value. It is then driven to PCI bus on **PAR** pin one clock after corresponding AD values. **PERR#** signal is sampled on next clock. If it is sampled asserted, Parity Error Detected signal is asserted to set appropriate bit in Configuration space. PCI Target writes are decoded using Target's TRDY output enable signal. If this signal is enabled and AD bus is not, then Target Write is in progress. Operation now switches from generating value on **PAR** signal and monitoring **PERR#** signal, to monitoring **PAR** signal and generating **PERR#** signal. Valid parity is calculated from external Master-provided CBE and AD signals. AD signals are monitored only on cycles on which **IRDY#** is sampled asserted. Value on **PAR** signal received on next clock cycle is xor –ed with parity calculated from CBE and AD signals. Result must be 0, otherwise Detected Parity Error is signalled. If Parity Error Response in Configuration space is enabled also, when this event occurs, then **PERR#** is asserted for one clock cycle.

Parity is also calculated on every Master initiated address phase – value is driven on **PAR** pin one clock cycle after the address phase.

Parity is also checked on any external Master's address phase. If invalid parity is detected, Parity Error Response and System Error Enable bits are set in configuration space, then **SERR#** is asserted for one clock cycle and Signaled System Error is signaled to Configuration space to set appropriate status bit. Detected Parity Error is also signaled to Configuration space in such an event, regardles of the state of Parity Error Response and System Error Enable bits. Parity error checking during external Masters' address phases is done same way as in writes through Target state machine, except in place of Target control signals, **FRAME#** input is monitored to decode address phase.

A few small submodules are included into ***pci_parity_check.v***. They are provided only because PCI timing constraints are hard to meet in FPGA implementation. “Preserve hierarchy” option can be selected over these modules in FPGA synthesis tool to prevent optimization of paths that include timing critical inputs. This kind of practice reduced logic levels on those inputs. Modules provided because of timing issues are:

- ***par_crit.v*** – provides output to PAR output Flip-Flop. Signal is generated directly from some timing critical PCI input signals during target operation.
- ***perr_crit.v*** – module feeding PERR output Flip-Flop.
- ***perr_en_crit.v*** – module feeding PERR output enable Flip-Flop.

- ***serr_en_crit.v*** – module feeding SERR output enable Flip-Flop
- ***serr_crit.v*** – module feeding SERR output Flip-Flop

2.3.3 Description of WB slave Unit submodules

wb_slave_unit.v is used for connecting lower level modules that implement unit's functionality. It also provides all connections needed between Configuration space and lower level modules (see Figure 2-3: Third picture - Module hierarchy of WB slave Unit and Configuration space).

2.3.3.1 wb_slave.v

Module implements WISHBONE slave state machine, which monitors and responds to cycles generated by external WB masters. It includes all logic needed for passing requests from WB bus to PCI bus or Configuration space. During WB write requests, logic checks whether or not all conditions for a write are satisfied. There are a few different ways writes are processed:

- WB state machine starts processing a write, when **CYC_I, STB_I** and **WE_I** WB signals are all active (1). WB address (**ADR_I**) passes through decoders implemented in ***wb_addr_mux.v***. Decoders compare input address with addresses stored in Configuration space and generate hit signals. WB slave state machine samples those signals at the rising edge of **CLK_I** while leaving IDLE or one of DECODE states. Hit signals are used in START state to decide where to go or what to do next. If not hit signal is sampled active, state machine doesn't respond and returns to IDLE state.
- **Configuration Writes** – configuration write starts, when decoder hit0 is active and lower twelve addresses don't match with Configuration Cycle data register offset. Configuration writes are accepted only as single writes to DWORD aligned addresses. If any of addresses [1:0] are non-zero or **CAB_I** signal is active during configuration write, WB slave state machine responds with an error and returns to idle state. WB slave state machine signals to Configuration space when write to register is to be done also (write enable signal to configuration space). In case of GUEST implementation of the bridge, configuration writes have no effect on Configuration space registers. A special case applies here – when bridge is defined as a GUEST and configuration image is not implemented, WB slave state machine does not respond to configuration writes at all, because decoder for Configuration space accesses is not implemented.
- **Configuration Cycle Writes** – this kind of writes is possible only in HOST bridge implementation. It is initiated when decoder hit0 is active and lower twelve addresses match with Configuration Cycle data register offset. State machine

handles these as Delayed Writes, so some additional restrictions apply. New Configuration Cycle Write Request can be accepted only if bridge has no other pending Delayed Transaction to be processed. That means no other Delayed Read/Write Request/Completion is pending. Pending Requests/Completions are signaled to WB slave from ***delayed_sync.v*** included in ***wb_slave_unit.v***. All of this conditions are sampled at the same time hits from decoders are sampled. If there is no Configuration Cycle Write Delayed Completion pending (signaled from ***delayed_sync.v***), then WB slave state machine responds with retry on WB bus and returns to idle state. If Completion is pending, WB slave state machine will end a cycle with error or acknowledge, depending on a kind of termination received on PCI bus and return to IDLE state. If neither Request or Completion is pending, then state machine signals Delayed Write Request to ***delayed_sync.v***, providing appropriate PCI bus command (Configuration Write) and address decoded by module in ***conf_cyc_addr_dec.v***.

- **Image Writes** – Image write is initiated when following conditions are satisfied:
    - WB Write Fifo (in ***wbw_wbr_fifos.v***) is not almost full nor full
    - Delayed Read or Write Requests are not pending
    - One of hits from decoders is active, except hit0

Image Write has two possible forms – I/O write or Memory Write. WB slave state machine decodes a form of Image Write by sampling map bits coming from Configuration space the same way hits from decoders are sampled. Each hit has its corresponding map bit. If map bit is 1, then I/O write is assumed, otherwise memory write is assumed. In case of I/O write, **CAB_I** signal is not allowed to be asserted (only single I/O writes possible) and **SEL_I**, **ADR_I[1:0]** combination must be valid (PCI IP Core Specification, Chapter 3.2.3). If **CAB_I** is asserted or invalid address/select combination is detected, WB slave state machine will not accept a write, respond with error and return to IDLE state. In case of memory write, **ADR_I[1:0]** must be zero, otherwise state machine doesn't accept a write, responds with an error and returns to IDLE state. When Image Write is accepted, state machine enables first write to WBW Fifo. During this write, address from ***wb_addr_mux.v*** is driven on WBW Fifo address/data output, PCI bus command is driven on command/byte enable output (PCI bus command is based on whether this is memory (Memory Write command) or I/O write (I/O Write command). Corresponding control bus value is also written to WBW Fifo during this write to mark an entry as an address/bus command entry. At this same time, state machine acknowledges the transfer and stores values from **SDAT_I** and **SEL_I** buses into intermediate register. State machine now switches WBW Fifo output to intermediate register data and negated selects sampled on previous clock edge. In case of I/O write, it asserts Fifo write enable, drives a value on Fifo control bus to mark an entry as last in a transaction and returns to IDLE state. In case of single memory write (**CAB_I** de-asserted) or when WBW Fifo status is ALMOST

FULL, the action taken is the same as for I/O writes. Otherwise, state machine continues to WRITE state used for burst writes. In this state, it keeps sampling data and selects on every clock cycle **STB_I** is sampled asserted, responds with acknowledge, and writes previously sampled data to Fifo, until it signals ALMOST FULL status or WB Master stops transferring (de-asserts **CYC_I** or **CAB_I**). When this happens, data sampled into intermediate register is written to Fifo and marked as last in a transaction by providing right value on Fifo control bus. State machine then returns to IDLE state.

During WB read requests, logic checks whether or not all conditions for a read are satisfied. There are a few different ways reads are processed:

- Reads are decoded in a same manner as writes, the only difference is that **WE_I** WB signal must be inactive (0) .
- **Configuration Reads** - configuration read starts, when decoder hit0 is active and lower twelve addresses don't match with Configuration Cycle data register or Interrupt Acknowledge Cycle register offset. Configuration reads are accepted only as single reads to DWORD aligned addresses. If any of addresses [1:0] are non-zero or **CAB_I** signal is active during configuration read, WB slave state machine responds with an error and returns to IDLE state. A special case applies when bridge is defined as a GUEST – if configuration image is not implemented, WB slave state machine does not respond to configuration reads at all, because decoder for Configuration space accesses is not implemented.
- **Interrupt Acknowledge Read** – can only be done when PCI Core is configured as HOST. It starts when decoder hit0 is active and lower twelve addresses match with Interrupt Acknowledge Cycle register offset. Since Interrupt Acknowledge cycle is generated as a Delayed Read Transaction, some additional rules to Configuration Reads' rules apply: There must be no other outstanding Delayed Request or Completion present in WB unit of the bridge. When this is true, Delayed Read Request with Interrupt Acknowledge PCI bus command is accepted, WB slave state machine responds with retry and returns to IDLE state. When Interrupt Acknowledge cycle is finished on PCI, status is transfered through ***delayed_sync.v*** and data through WB Read Fifo (in ***wbw_wbr_fifos.v***). When cycle is repeated by external WB Master, Delayed Completion pending is signaled from ***delayed_sync.v*** and PCI Write Fifo (***pciw_pcir_fifos.v*** in ***pci_target_unit.v***) is empty, data and appropriate status are signaled to requesting WB Master and state machine returns to IDLE state.
- **Configuration Cycle Read** - can only be done when PCI Core is configured as HOST. It starts when decoder hit0 is active and lower twelve addresses match with Configuration Cycle Data register offset. Since Configuration Read cycle is generated as a Delayed Read Transaction, some additional rules to Configuration Reads' rules apply: There must be no other outstanding Delayed Request or Completion present in WB unit of the bridge. When this is true, Delayed Read

Request with Configuration Read PCI bus command and decoded address from ***conf_cyc_addr_dec.v*** is accepted. WB slave state machine responds with retry and returns to IDLE state. When Configuration Cycle is finished on PCI, status is transfered through ***delayed_sync.v*** and data through WB Read Fifo (in ***wbw_wbr_fifos.v***). When cycle is repeated by external WB Master, Delayed Completion pending is signaled from ***delayed_sync.v*** and PCI Write Fifo (***pciw_pcir_fifos.v*** in ***pci_target_unit.v***) is empty, data and appropriate status are signaled to requesting WB Master and state machine returns to IDLE state.

- **Image Read –** starts when one of hit signals is set, except for hit0 (configuration hit). All image reads are processed as Delayed Transactions, so following rules apply: If no other outstanding Delayed Request or Completion is pending in WB slave unit, then new request can be accepted. In case of Memory Read, reads to DWORD aligned addresses are accepted only. In case of I/O Read, only single reads (**CAB_I** signal 0) with appropriate address/select line combinations are accepted (PCI IP Core specification, Chapter 3.2.3). I/O and Memory reads are differentiated by map bit that corresponds to current image hit. If map bit is 1, that means I/O Read, otherwise it is processed as Memory Read. When new Delayed Read Request is accepted, WB slave state machine responds with retry, stores address, select and bus command information and returns to IDLE state. I/O Reads are always processed the same way: as single reads with IO Read PCI bus command used. Memory reads however can be processed in a few different ways, depending on Core's configuration. Memory Read Line and Pre-fetch enable bits from an Image Control register that corresponds to current active hit input are sampled when state machine leaves IDLE or one of DECODE states. Invalid Cache Line Size register value masks these bits, so special commands can't be used and read bursts cannot be performed on PCI. Refer to PCI IP Core Specification, Chapter 3.2.4 to see how Delayed Memory Reads depend on the state of these bits. When new request can be accepted, WB slave state machine stores the address provided by ***wb_addr_mux.v***, **SEL_I** lines provided from external WB Master and decoded PCI bus command by signaling new request to ***delayed_sync.v*** included in ***wb_slave_unit.v***. When requested read transaction finishes on PCI bus, this status is signaled to WB slave state machine through ***delayed_sync.v***. The data fetched is stored in WB Read Fifo (***wbw_wbr_fifo.v***) and is stored there until external WB Master repeats the same request. On any occasion external WB Master starts a read cycle, address and select lines provided are compared with stored values and result of this operation sampled into registers, when WB slave state machine leaves IDLE or one of DECODE states. If decoded address and select lines are the same and PCI Write Fifo is empty, then WB slave state machine starts providing data from WB Read Fifo on **SDAT_O** lines on WB bus. It keeps sending out data and acknowledging transfers until Fifo is empty, data marked as last or error is fetched out from WB Read Fifo, or external WB master stops a transfer. If external WB master stops a transfer before Read Fifo is empty, state machine generates a flush signal, to get rid of stale data.

2.3.3.2 wb_addr_mux.v

Module provides address decoding functionality for WB Slave Unit. **ADR_I** bus from WISHBONE is connected directly to this module. Configuration space is connected to this module with all WISHBONE Base Addresses (WB_BAx), Translation Addresses (WB_TAx) and Address Masks (WB_AMx). Each triplet of those values is connected to its own decoder (in ***decoder.v***), while **ADR_I** bus is connected to all of them. Each implemented decoder generates independent address and hit signal output. Addresses from implemented decoders are multiplexed to one output address bus based on value of hit signals. Output address is connected to WB Slave state machine, where it is used in various functions. Each defined decoder instantiated in ***wb_addr_mux.v*** is implemented in ***decoder.v***. This module is responsible for comparing decoded number of address lines between provided base address and WISHBONE input address. If bus values are the same, decoder sets hit signal to 1. Address Translation Enable input provided from Configuration space is also checked. If it has a value of 1 and address translation is implemented, then translation is performed by changing decoded number of WISHBONE address bus input bits with decoded number of Translation Address inputs. Result is provided on address output bus.

2.3.3.3 wbw_wbr_fifos.v

Module is the main storage unit for data passing through WISHBONE Slave Unit. It instantiates synchronous dual port RAMs for each Fifo or one two port RAM used for both Fifos. It also inferes two counters – one for incoming transactions and one for outgoing transactions through WB Write Fifo. This is done to signal ***pci_master32_sm_if.v*** interface when at least one complete Write Transaction is in the Write Fifo and can be started on PCI bus. WB Read Fifo does not need such a counter, because complete transaction is signalled through ***delayed_sync.v***. The module is also responsible for multiplexing read and write addresses in PCI and WB clock domain when only one RAM is instantiated for both Fifos.

Data for WB Write Fifo is received from WB Slave state machine on multiplexed address/data bus. It's written to WB Write Fifo on rising WB clock edge, when Fifo is not full and WB Slave state machine asserts write enable signal. It's stored at write address provided by ***wbw_fifo_control.v***. If data witten to Fifo is marked as last on WB Write Fifo control bus (also provided by WB Slave state machine), then incoming transaction counter is incremented. When incoming and outgoing transaction counters are not equal, Transaction Ready signal is generated on next rising PCI clock edge. Comparison is done between Grey coded values, to provide glitch free comparator output. ***pci_master32_sm_if.v*** signals a read from WB Write Fifo. If on rising PCI clock edge

WB Write Fifo read enable is active, read address from ***wbw_fifo_control.v*** is applied to RAM interface, to provide next data at its data outputs. Address/data output from WB Write Fifo is connected to ***pci_master32_sm_if.v***. When data marked as last (determined by monitoring control bus output) is read from Fifo, outgoing transaction counter is incremented and Transaction Ready output is cleared. If there is another transaction ready in the Fifo, Transaction Ready output will be set on next rising PCI clock edge. ***wbw_fifo_control.v*** provides a module for generating synchronous RAM addresses for WB Write Fifo. It provides write side RAM address (in WB clock domain) and read side RAM address (in PCI clock domain). It also generates different statuses. Statuses are always determined by comparing Grey coded read and write addresses to provide glitch free comparator outputs. Status outputs are sampled in Flip-Flops in appropriate clock domain. For example – WB clock domain always writes to WB Write Fifo, so it is only interested in Fifo fullness. So Fifo's Almost Full and Full statuses are synchronized to WB clock domain. On the other hand, PCI clock domain always just reads from this Fifo and is therefore interested only in Fifo emptiness. Therefore statuses Two Left, Almost Empty and Empty are syncronized to PCI clock domain. Grey code pipeline is provided for generating statuses. Each clock domain has its own pipeline, values are compared between clock domains and results sampled at appropriate clocks as stated previously.

Data for WB Read Fifo is received from PCI Master state machine Interface. It is written to RAM address provided by ***wbr_fifo_control.v*** on rising PCI clock edge when WB Read Fifo Write Enable is asserted (received from PCI Master state machine Interface) and Fifo is not Full. WB Slave state machine performs a read on rising edge of WB clock, during which it holds WB Read Fifo Read Enable asserted. Read address provided from ***wbr_fifo_control.v*** is applied to RAM address inputs on a port clocked by WB clock, to provide next data and control signals to WB Slave state machine on Fifo's data and control outputs. ***wbr_fifo_control.v*** provides a module for generating synchronous RAM addresses for WB Read Fifo. It provides write side RAM address (in PCI clock domain) and read side RAM address (in WB clock domain). It also generates different statuses. Statuses are always determined by comparing Grey coded read and write addresses to provide glitch free comparator outputs. Status outputs are sampled in Flip-Flops in appropriate clock domain. For example – PCI clock domain always writes to WB Read Fifo, so it is only interested in Fifo fullness. So Fifo's Full status is synchronized to PCI clock domain. On the other hand, WB clock domain always just reads from this Fifo and is therefore interested only in Fifo emptiness. Therefore status Empty is syncronized to WB clock domain. Grey code pipeline is provided for generating statuses. Each clock domain has its own pipeline, values are compared between clock domains and results sampled at appropriate clocks as stated previously.

2.3.3.4 delayed_write_reg.v

Module is provided as additional storage for Delayed Write Transactions. Only Configuration Write Transactions are processed as Delayed Writes currently, so this register is only relevant in HOST bridge implementation. When WB Slave state machine signals a Delayed Write Request, this register stores provided data for transfer on PCI bus.

2.3.3.5 conf_cyc_addr_dec.v

Module provides address bus value for generation of Configuration Cycles. It is included only in HOST bridge implementation, since GUEST implementation cannot generate them. It receives value written in Configuration Cycle Address register in Configuration space. If value in this register marks Configuration Cycle as Type1 cycle, then address from register passes through the module unchanged. If a value in a register marks Configuration Cycle as Type0, then lower 11 bits pass through module unchanged. Next 5 bits in register value are decoded to provide 21 upper address bits. Refer to PCI IP Core Specification.

2.3.3.6 pci_master32_sm_if.v

Module is provided for passing requests from ***delayed_sync.v*** and ***wbw_wbr_fifos.v*** to ***pci_master32_sm.v***. When no transaction is in progress currently, which is determined by three request Flip-Flops, the interface monitors its inputs from WB Write Fifo (in ***wbw_wbr_fifos.v***) and ***delayed_sync.v***.

When Posted Write is prepared in WB Write Fifo, the interface starts preparing all necesary data for PCI Master state machine. It fetches the address into address counter and PCI bus command to its register first. These two registers are connected directly to address and bus command output buses, which go to ***pci_master32_sm.v***. Then it fetches two data entries with byte enables (if Posted Write is long enough) to additional registers provided. One register is needed to store the last data in a Write Transaction that was not transfered to PCI bus. This has to be done if Disconnect is signaled in the middle of transaction. The second register is used to provide next data and byte enable information to PCI Master state machine. When address and bus command are prepared and data registers are loaded, the interface signals a request to PCI Master state machine. It receives statuses from the state machine, to know when to provide next write data. Statuses received are:

- Wait – means state machine is not transfering yet – when wait is set to 1, all other statuses are ignored
- Registered Transfer – status means data was transfered on previous rising PCI clock edge.

- Wire Transfer – status signals transfer is going to happen on next rising PCI clock edge. Generation of this status uses unregistered PCI bus control inputs which are timing critical. Usage of this signal should be reduced to minimum to minimize fan-out and logic levels on timing critical PCI control inputs.
- Registered Retry – status signals that retry or dissconnect was signaled on previous rising PCI clock edge.
- Registered Error – status signals that Target Abort Termination was detected on previous rising PCI clock edge.
- Master Abort - status signals that Master Abort Termination was initiated on previous rising PCI clock edge. Master Abort Termination status is treated the same way as Target Abort, except during Configuration Cycles.

The interface is also responsible for signaling to PCI Master state machine if current data phase is also the last or if current data phase is one before last (state machine needs this information because of PCI bus protocol and registered PCI outputs). Posted Write Transaction will be repeated and retried, until all data in the transaction is transfered or error status is signalled. Write Transaction can be interrupted by an external Target in the middle of transfer, so address counter is implemented to always provide the right address from which transaction must continue. If error is detected in the middle of Posted Write Transaction (either Master or Target Abort), error status is signaled to Configuration space, with erroneous address, bus command, data, byte enable and error source information. Refer to PCI IP Core Specification. The interface also recovers from a Posted Write error, by pulling data out of WB Write Fifo, until last data of current transaction is detected (data marked as last on WB Write Fifo control bus input).

When no transaction is in progress, WB Write Fifo signals an Empty status and Delayed Request is signaled from ***delayed_sync.v***, the interface starts preparing everything data for PCI Master state machine. It fetches address and bus command provided from ***delayed_sync.v*** into address counter and bus command register. If this is a Delayed Write Request, then it also fetches data from ***delayed_write_reg.v***. If this is burst Delayed Read Request, a read counter is provided, to stop the reading when enough data is read. Single reads are stopped after single data is transfered from PCI bus. Refer to PCI IP Core Specification, regarding length of burst reads.

Delayed Writes are always single transactions. When Delayed Write is finished on PCI bus – PCI Master state machine signaling Registered Transfer or Error with Wait status zero, appropriate status is returned to ***delayed_sync.v*** (Delayed Request Complete and Error, if detected).

Delayed Reads can be done as burst or single transactions – Refer to PCI IP Core Specification. If delayed Read Request is decoded as a single read, then the interface will stop the request immediately after single data is stored in WB Read Fifo. Data can be marked as normal or error, depending on status returned by PCI Master state machine. Burst Read Request will be signaled to PCI Master state machine until all data required is stored in WB Read Fifo (determined by a state of read counter), external Target Disconnects or Master Abort or Target Abort is signaled. In later two cases, data entry will still be written to WB Read Fifo marked as error on WB Read Fifo control bus output (except on Master Abort during execution of Configuration Read command). Disconnects are determined with FIRST status Flip-Flop. If Registered Retry is received from PCI Master state machine and FIRST status is active, this means a normal Retry and transaction must be requested again. If FIRST status is inactive when Registered Retry is received, this means a Disconnect, which does not have to be repeated on PCI bus. The interface is responsible for signaling appropriate statuses to PCI Master state machine during Burst Delayed Reads also (last data phase, one before last data phase). This statuses are determined from a state of read counter. The interface also uses Cache Line Size register value from Configuration space for Burst Delayed Read Transactions in some cases.

2.3.3.7 pci_master32_sm.v

Module implements PCI Master state machine and control logic. State machine only signals current state, Wait status and data source to the outside world. Control logic calculates appropriate value for each PCI bus signal, controls loading of output and output enable Flip-Flops and supplies appropriate value for them. It also disables clock on PCI Master state machine, when it is not allowed to change the state it is in.

When state machine is in IDLE state, control logic monitors PCI control input signals. If PCI bus is idle (**FRAME#** and **IRDY#** both de-asserted) and **GNT#** is asserted, it just loads output enable Flip-Flops for **AD** and **CBE#** busses with active value (1). This implements bus parking functionality. Value of those buses is not important at this time. When Request and Ready signals from ***pci_master32_sm_if.v*** are both 1, control logic loads **REQ#** output Flip-Flop with active value (0). It also monitors state of PCI bus and **GNT#** input signal. If PCI bus is idle and **GNT#** is asserted (0), control logic enables load of new values to **AD** and **CBE** output buses, loads output enable Flip-Flops to active state for **FRAME#** signal and **AD** and **CBE** buses. It also enables clock for PCI Master State machine, which will proceede to ADDRESS state on next rising PCI clock edge. While state machine is in address phase (always for one cycle only), control logic keeps state machine clock enabled. It determines next value of **FRAME#** output signal – if data is marked as last from ***pci_master32_sm_if.v*** this would be 1, otherwise **FRAME#** output Flip-Flop is loaded with active value (0). It determines next value for **AD** output enable Flip-Flops. If this is a read transaction (determined from LS bit of PCI bus command), **AD** output enable Flip-Flops are loaded with inactive value, to disable driving of **AD** bus when state machine leaves an ADDRESS state. **IRDY#** output and output enable Flip-Flops are loaded with an active value. State machine is now in TRANSFER state. It is allowed to go out from this state only when **FRAME#** output is de-asserted. Control logic now calculates next values for output enables, loads and values for control signals by monitoring external Target response signals, statuses received from

pci_master32_sm_if.v and internal counters. It also signals its own statuses and the ones received from external PCI target to the interface. Two counters are implemented to satisfy PCI bus protocol: Decode Timeout and Latency Timeout. Decode Timeout occurs when state machine is in TRANSFER state and no external target has responded to transaction address by asserting **DEVSEL#** signal. This happens four PCI clock cycles after state machine leaves ADDRESS state. If **DEVSEL#** is not sampled asserted after these four clocks, Master Abort procedure is initiated. Latency Timer counter value is received from Configuration space. Timer counts all the time when state machine is in ADDRESS or TRANSFER state. When it is zero and **GNT#** is sampled de-asserted, Latency Timeout occurs. When this occurs, PCI Master state machine finishes the transfer and signals to the interface as if Disconnect was received. For Burst Posted Write Transaction this means it will continue next time PCI Master state machine is granted a bus, while Burst Read transaction is finished when this happens (since the interface stops requesting). When Latency Timeout, Master Abort, Target Abort, Retry, Disconnect or transfer of last data is decoded, state machine goes to TURN_AROUND state. At this time, **AD** and **CBE** buses and **FRAME#** output enable Flip-Flops are loaded with inactive value, **IRDY#** output is loaded with de-asserted value (1). When state machine leaves TURN_AROUND state, **IRDY#** output enable Flip-Flop is loaded with inactive value also.

Submodules instantiated in ***pci_master32_sm.v*** are used to resolve timing issues on PCI control signals used unregistered. Logic optimization was turned off for these modules during FPGA synthesis to reduce a number of logic levels on timing critical PCI input paths.

- ***frame_crit.v*** – module feeds FRAME PCI signal output Flip – Flop. **FRAME#** signal must be de-asserted on any clock cycle **STOP#** input is sampled asserted. Normal FRAME output calculation is performed in the ***wb_master32_sm.v*** and fed to this module in parallel with **STOP#** input. When this one is sampled asserted, module immediately blocks out **FRAME#** assertion regardless of the result of normal **FRAME#** calculation.
- ***frame_load_crit.v*** – module feeds FRAME output Flip-Flop's clock enable. Value of **FRAME#** output signal changes when state machine starts operation, when external PCI target is responding or when Master Abort termination is initiated. Master Abort termination and start of transaction are not critical to detect, since they are generated internally. The calculation of those is fed to this module in parallel with **TRDY#** and **STOP#** PCI inputs. New value of **FRAME#** output is loaded when internal logic forces it or, in the middle of transaction, when external target responds with **TRDY#** or **#STOP** asserted.
- ***irdy_out_crit.v*** – this module is provided for basically the same operation as previous one, except it feeds **IRDY#** output Flip – Flop. External logic signals when **IRDY#** must be asserted no matter what and state of current **FRAME#** output. When **FRAME#** output signal is de-asserted, module waits for **TRDY#** or

STOP# signals to be asserted and de-asserts **IRDY#** output on that same rising PCI clock edge.

- ***mas_ad_load_crit.v*** – module signals to ***pci_io_mux.v*** in ***pci_bridge32.v*** when to load next data to **AD[31:0]** output Flip-Flops. It receives a signal which forces output registers to be loaded, and signal that allows outputs to be loaded and **GNT#** input from PCI bus. Load is forced, when state machine is changing state from IDLE to ADDRESS, to provide the address received from ***pci_master32_sm_if.v*** on **AD** outputs, and when state machine is changing state from ADDRESS to TRANSFER, if current transaction is a write. When state machine is in TRANSFER state during a write, loading is allowed on every cycle transfer is signaled, which is decoded in **pci_io_mux.v**
- ***mas_ch_state_crit.v*** – module provides an internal signal for clock enable on state machine. State transition is controlled by requests received from ***pci_master32_sm_if.v***, state that state machine is currently in and PCI control input signals. For example – when state machine is in IDLE state, it can change state to ADDRESS state only when PCI bus is in idle state, ***pci_master32_sm_if.v*** request and ready signals are set and **GNT#** input is asserted. Or, when state machine is in TRANSFER state, state is allowed to change only when **FRAME#** output is de-asserted, Master Abort is in progress or external target is responding with **TRDY#** or **STOP#** asserted.
- ***mas_ad_en_crit.v*** – module is provided to feed AD bus output enable Flip-Flops. It is provided, because Master state machine must enable output buffers when PCI bus is IDLE and **GNT#** is asserted. **GNT#** is constrained PCI input, so calculation is done in separate module. Other internal, non-critical calculations are done outside this module.
- ***cbe_en_crit.v*** – module is provided for CBE output enable calculations that include timing critical PCI inputs. Since only PCI Master state machine can drive CBE outputs, this module resides in Master state machine and not in **pci_io_mux.v** as opposed to AD enable sub-modules, which are used in Master and Target operation.
- ***frame_en_crit.v*** – module feeds **FRAME#** output enable Flip-Flop. **FRAME#** output must remain enabled, until last data phase is complete or Master Abort termination occurs. Signals for the module provide information on when **FRAME#** must remain enabled, when it must be disabled and when it is allowed to be disabled. When disabling of **FRAME#** is allowed, module waits for **STOP#** or **TRDY#** to be asserted and disables **FRAME#** output immediately on that clock edge.

2.3.4 Description of PCI target Unit submodules

The module ***pci_target_unit.v*** is used for connecting lower level modules that implement unit's functionality which is accessing WB bus from PCI bus (see Figure 2-4: Fourth picture - Module hierarchy of PCI target Unit). It has three communication ports; one from PCI bus, one to access Configuration space and one to access WB bus. Ports from PCI bus and to access Configuration space operate on PCI clock, while port for accessing WB bus operates on WB clock.

2.3.4.1 pci_target32_sm.v & pci_target32_interface.v

Module ***pci_target32_sm.v*** implements PCI Target state machine and control logic. State machine only signals current state and decides a next state regarding the signals from control logic. Control logic calculates appropriate value for each PCI bus signal, controls loading of output and output enable Flip-Flops and supplies appropriate value for them. It also disables clock on PCI Target state machine, when it is not allowed to change the state it is in, and it signals to ***pci_target32_interface.v*** where to data must be loaded (FIFO or Configuration space) or where from data must be fetched. Module ***pci_target32_interface.v*** additionally checks when stored address from requested read is the same as currently translated address from PCI bus and other checking that influence, what kind of termination will be.

The control logic monitors PCI control input signals and control signals from ***pci_target32_interface.v***, since all storing registers, multiplexers and address decoders (***pci_decoder.v***) are implemented there. Internal control signals and ones from ***pci_target32_interface.v*** module, that influence on initial PCI Target response, are registered in ***pci_target32_sm.v*** module due to timing constraints in FPGA. This incorporates initial wait cycle for PCI Target response. Control signals from ***pci_target32_interface.v*** module are:

- **addr_claim_in** – ORed hits from PCI decoders (not registered before used)
- **same_read_in** – If PCI bus read command and translated address are the same as stored ones (registered due to critical timings)
- **pci_cbe_reg_in[0]** – Registered R/W signal from PCI bus command (registered due to large fanout)
- **read_completed_in** – If there is a delayed read request finished on a WB bus and all data are ready in PCI read FIFO – ***pciw_pcir_fifos.v*** (used in equation and registered due to critical timings)
- **read_processing_in** – If there is a delayed read request pending on a WB bus (used in equation and registered due to critical timings)
- **norm_access_to_config_in** – If there is a memory access to Configuration space (used in equation and registered due to critical timings)

- **disconect_wo_data_out** – If no data can be transferred any more. Signaled during data phases and used for disconnect without data (not registered before used).
- **disconect_w_data_out** – If no data can be transferred any more. Signaled during data phases and used for disconnect with data (not registered before used).
- **target_abort_out –** If a transaction must be aborted after address phase. Signaled at address phase only (not registered before used).
- **wbw_fifo_empty_in** – If WB write FIFO is empty – just connected through ***pci_target32_interface.v*** module (used in equation and registered due to critical timings)
- **pciw_fifo_full_out** – If PCI write FIFO has not enough space left (less then 3 locations, first one is for address) – ORed signals from FIFO **pciw_fifo_full_in**, **pciw_fifo_almost_full_in** and **pciw_fifo_two_left_in** (not registered before used).
- **pcir_fifo_data_err_out –** If corresponding control bit value, of the data read out of PCI read FIFO, signals that an data entry is erroneous, therefore read must be aborted. This is signaled when FIFO is selected and not Configuration space (not registered before used).

All transactions are initiated from PCI bus (from other PCI master). How PCI reads and writes are processed is discussed separately. There are a few different ways writes are processed:

- PCI Target state machine starts processing a write with address and one of write commands on **AD** and **CBE#** busses respectively, when **FRAME#** signal is active (0). Registered PCI address passes through PCI decoders (***pci_decoder.v***) implemented in ***pci_target32_interface.v*** (it is also stored in a register there). PCI decoders compare input address with addresses stored in Configuration space. Regarding the space (MEMORY or I/O) to which an image is assigned to (bit 0 in P_BA register), there must also be enabled MEMORY or I/O space (PCI Header - Command register, see PCI IP Core Specification, chapter 4.1.2) to allow PCI decoders to generate hit signals. Write can progress when there is address hit and: if there is no read request pending on a WB bus and PCI write FIFO is not full (full, almost full and two left), if there is read request completed on a WB bus and PCI write FIFO is not full (full, almost full and two left) or if there is single word write command to Configuration space.

- **Configuration Writes –** configuration write is meant as a write to Configuration space and they proceeds regardless of uncompleted read requests. It can be performed with MEM WRITE or CONF WRITE commands.

  Memory write to Configuration space starts, when decoder hit0 is active. Writes are accepted only as single writes (all MEM WRITE commands are DWORD aligned). If burst MEM WRITE is attempting, then control logic responds Disconnect With Data when data is transferred or Disconnect Without Data after data is transferred (**disconect_wo_data_in** and **disconect_w_data_in** signals from ***pci_target32_interface.v***).

Configuration write to Configuration space starts, when PCI signal IDSEL is active and CONF WRITE command is performed (all CONF WRITE commands are single writes and DWORD aligned).

Logic in PCI target state machine signals to Configuration space when write to register is to be done (**load_to_pciw_fifo_out** - write enable signal to configuration space). In case of HOST implementation of the bridge, writes have no effect on Configuration space registers. A special case applies here – when bridge is defined as a HOST and configuration image is not implemented or is changed into normal PCI-WB image, PCI target state machine does not respond to configuration writes at all, because decoder for Configuration space accesses is not implemented.

- **Image Writes** – Image write is initiated when following conditions are satisfied:
  - PCI Write Fifo (in ***pciw_pcir_fifos.v***) is not full nor almost full nor two left (**pciw_fifo_full_in** signal from ***pci_target32_interface.v***)
  - Delayed Read Request is not pending (**read_processing_in** signal from ***pci_target32_interface.v***), but can be completed (**read_completed_in** signal from ***pci_target32_interface.v***)
  - One of hits from decoders is active, except hit0 (see PCI IP Core Specification, chapter 3.3.2)

Image write has two possible forms – I/O write or Memory write with IO WRITE or MEM WRITE commands respectively. As mentioned, the appropriate space of an image must be enabled (MEMORY or I/O) for each kind of commands and also image must be assigned to appropriate space (MEMORY or I/O in a base address register – bit 0), otherwise PCI target state machine will not response at all. PCI decoders (***pci_decoder.v***) take care of this for each image.

Memory write can be single or burst (all MEM WRITE commands are DWORD aligned). The two LSBits of address (**AD[1:0]**) tell what kind of burst ordering is initiated. Two kinds are possible; linear incrementing and cacheline wrap, but only the first one is supported. If PCI master initiates the cacheline wrap burst ordering, the PCI target logic terminates the transaction with Disconnect With or Without Data and only single data phase transfers data (**disconect_wo_data_in** and **disconect_w_data_in** signals from ***pci_target32_interface.v***) as it reacts when there remains only two spare locations in PCI write FIFO during linear incrementing burst. The byte enables indicate the affected bytes within the DWORD for each data written.

I/O write is only single write. All 32 address bits are used to provide a full byte address. The PCI master is required to ensure that **AD[1:0]** indicate the least significant valid byte for the transaction. The byte enables indicate the size of the transfer and the affected bytes within the DWORD and must be consistent with **AD[1:0]** (see PCI IP Core Specification, chapter 3.3.3), otherwise PCI target logic terminates the transaction with Target-Abort (**target_abort_in** signal from ***pci_target32_interface.v***) and no data is transferred.

When Image Write is accepted, control logic in ***pci_target32_interface.v*** enables first write to PCIW Fifo. During this write, address stored in a register (address is stored from multiplexer, to which PCI decoders are connected) is driven on PCIW Fifo address/data output, PCI bus command stored in a register is driven on command/byte enable output. Corresponding control bus value is also written to PCIW Fifo during this write to mark an entry as an address/bus command entry. Control logic now switches registered data from PCI bus to PCIW Fifo. In case of I/O write, it asserts Fifo write enable, drives a value on Fifo control bus to mark an entry as last. In case of single memory write (**FRAME#** de-asserted before first data phase), the action taken is the same as for I/O writes. When there is a burst memory write, data are normally written, with corresponding control bus value written to mark an entry as a burst, until there remains only two spare locations in PCIW Fifo or PCI master stops writing. When this happens, registered data written to Fifo are marked as last with corresponding control bus value.

During PCI read requests, logic checks whether or not all conditions for a read are satisfied. There are a few different ways reads are processed:

- Reads are decoded in a same manner as writes, the only differences are READ commands instead of WRITE commands and that Read is requested on WB bus when there is address hit and there is no read request pending on a WB bus and no read request completed and it is not a read from Configuration space. Read from PCI bridge can progress when there is address hit and: if there is read request completed on WB bus and the same address as an address of completed read and WBW Fifo is empty (due to transaction ordering) or if there is single word read command from Configuration space.

- **Configuration Reads –** configuration read is meant as a read from Configuration space and they proceeds regardless of uncompleted or completed read requests. It can be performed with MEM READ or CONF READ commands.

  <u>Memory read from Configuration space</u> starts, when decoder hit0 is active. Reads are accepted only as single reads (all MEM READ commands are DWORD aligned). If burst MEM READ is attempting, then control logic responds Disconnect With Data when data is transferred or Disconnect Without Data after data is transferred (**disconect_wo_data_in** and **disconect_w_data_in** signals from ***pci_target32_interface.v***).

Configuration read from Configuration space starts, when PCI signal IDSEL is active and CONF READ command is performed (all CONF READ commands are single reads and DWORD aligned).

Logic in PCI target state machine just set the current address to Configuration space when read from register is to be done and set select signal on multiplexer for fetching the data from Configuration space (**sel_conf_fifo_out**). In case of HOST implementation of the bridge, just reads have effect on Configuration space registers. A special case applies here – when bridge is defined as a HOST and configuration image is not implemented or is changed into normal PCI-WB image, PCI target state machine does not respond to configuration reads at all, because decoder for Configuration space accesses is not implemented.

- **Image Reads** – Image read is accepted when following conditions are satisfied:
  - Delayed Read Request is not pending (**read_processing_in** signal from ***pci_target32_interface.v***) and Delayed Read Request is not completed (**read_completed_in** signal from ***pci_target32_interface.v***)
  - One of hits from decoders is active, except hit0 (see PCI IP Core Specification, chapter 3.3.2)

  – Image read is initiated when following conditions are satisfied:
  - Delayed Read Request is completed (**read_completed_in** signal from ***pci_target32_interface.v***)
  - One of hits from decoders is active, except hit0 (see PCI IP Core Specification, chapter 3.3.2)
  - Initiated address is the same as the starting address of completed Delayed Read Request (**same_read_in** signal from ***pci_target32_interface.v***)
  - WB Write Fifo (in ***wbw_wbr_fifos.v***) is empty (**wbw_fifo_empty_in** signal from ***wb_slave_unit.v***) due to transaction ordering

As mentioned, Delayed Read must first be accepted and requested to complete on WB bus. When new Delayed Read Request is accepted, PCI target state machine responds with retry and stores the translated address with its bus command. Image read has two possible forms – I/O read or Memory read with IO READ or MEM READ commands respectively. As mentioned, the appropriate space of an image must be enabled (MEMORY or I/O) for each kind of commands and also image must be assigned to appropriate space (MEMORY or I/O in a base address register – bit 0), otherwise PCI target state machine will not response at all. PCI decoders (***pci_decoder.v***) take care of this for each image. I/O Reads are always processed the same way: as single reads with IO Read PCI bus command used. The PCI master is required to ensure that **AD[1:0]** indicate the least significant valid byte for the transaction. The byte enables indicate the size of the transfer and the affected bytes within the DWORD and must be consistent with **AD[1:0]** (see PCI IP Core Specification, chapter 3.3.3), otherwise PCI target logic terminates the transaction with Target-Abort (**target_abort_in** signal from ***pci_target32_interface.v***) and no data is transferred. Memory reads however can be processed in a few different ways, depending on Core's configuration. Pre-fetch enable bits from an Image Control register that corresponds to current active hit input together with one of MEM READ commands and Cache Line Size register value influence to a number of data read on a WB bus (see PCI IP Core Specification, Chapter 3.2.4). When new request can be accepted, PCI target state machine stores the translated address and PCI bus command by signaling new request to ***delayed_sync.v*** included in ***pci_target_unit.v***. When requested read transaction finishes on WB bus, this status is signaled to PCI target state machine through ***delayed_sync.v*** and ***pci_target32_interface.v***. The data fetched is stored in PCI Read Fifo (***pciw_pcir_fifo.v***) and is stored there until external PCI Master repeats the same request. On any occasion external PCI Master starts a read cycle, translated address and PCI bus command provided are compared with stored values. If decoded address and bus command are the same and WB Write Fifo is empty, then PCI target state machine set select signal on multiplexer for fetching the data from Fifo and starts providing data from PCI Read Fifo to PCI bus. There is also one storing register between Fifo and output register, in order to have only one initial wait cycle and no subsequent wait cycles. PCI target state machine keeps sending out data and acknowledging transfers until Fifo is empty, data marked as last or error is fetched out from PCI Read Fifo, or external PCI master stops a transfer. If PCI master initiates the cacheline wrap burst ordering for MEM READ commands, the PCI target logic terminates the transaction with Disconnect With or Without Data and only single data phase transfers data (**disconect_wo_data_in** and **disconect_w_data_in** signals from ***pci_target32_interface.v***). If a transfer is stopped before Read Fifo is empty, state machine generates a flush signal, to get rid of stale data. The byte enables indicate the affected bytes within the DWORD for each single data read, while for burst data reads byte enables have no meaning.

Submodules instantiated in ***pci_target32_sm.v*** are used to resolve timing issues on PCI control signals used unregistered. Logic optimization was turned off for these modules during FPGA synthesis to reduce a number of logic levels on timing critical PCI input paths.

- ***pci_target32_clk_en.v*** – module feeds clock enable signal for state machine with inputs **FRAME#** and **IRDY#**.
- ***pci_target32_trdy_crit.v*** – module feeds output value for **TRDY#** signal.
- ***pci_target32_stop_crit.v*** – module feeds output value for **STOP#** signal.
- ***pci_target32_devs_crit.v*** – module feeds output value for **DEVSEL#** signal.

2.3.4.2 pciw_pcir_fifos.v

Module is the main storage unit for data passing through PCI Target Unit. It instantiates synchronous dual port RAMs for each Fifo or one two port RAM used for both Fifos. It also inferes two counters – one for incoming transactions and one for outgoing transactions through PCI Write Fifo. This is done to signal ***wb_master.v*** state machine when at least one complete Write Transaction is in the Write Fifo and can be started on WB bus. PCI Read Fifo does not need such a counter, because complete transaction is signalled through ***delayed_sync.v***. The module is also responsible for multiplexing read and write addresses in PCI and WB clock domain when only one RAM is instantiated for both Fifos.

Data for PCI Write Fifo is received from PCI Target state machine on multiplexed address/data bus. It's written to PCI Write Fifo on rising PCI clock edge, when Fifo is not full and PCI Target state machine asserts write enable signal. It's stored at write address provided by ***pciw_fifo_control.v***. If data witten to Fifo is marked as last on PCI Write Fifo control bus (also provided by PCI Target state machine), then incoming transaction counter is incremented. When incoming and outgoing transaction counters are not equal, Transaction Ready signal is generated on next rising WB clock edge. Comparison is done between Grey coded values, to provide glitch free comparator output. ***wb_master.v*** signals a read from PCI Write Fifo. If on rising WB clock edge PCI Write Fifo read enable is active, read address from ***pciw_fifo_control.v*** is applied to RAM interface, to provide next data at its data outputs. Address/data output from PCI Write Fifo is connected to ***wb_master.v***. When data marked as last (determined by monitoring control bus output) is read from Fifo, outgoing transaction counter is incremented and Transaction Ready output is cleared. If there is another transaction ready in the Fifo, Transaction Ready output will be set on next rising WB clock edge. ***pciw_fifo_control.v*** provides a module for generating synchronous RAM addresses for PCI Write Fifo. It provides write side RAM address (in PCI clock domain) and read side RAM address (in WB clock domain). It also generates different statuses. Statuses are always determined by comparing Grey coded read and write addresses to provide glitch free comparator outputs. Status outputs are sampled in Flip-Flops in appropriate clock domain. For example – PCI clock domain always writes to PCI Write Fifo, so it is only interested in Fifo fullness. So Fifo's Almost Full and Full statuses are synchronized to PCI clock domain. On the other hand, WB clock domain always just reads from this Fifo and is therefore interested only in Fifo emptiness. Therefore statuses Two Left, Almost Empty and Empty are syncronized to WB clock domain. Grey code pipeline is provided for generating statuses. Each clock domain has its own pipeline, values are compared between clock domains and results sampled at appropriate clocks as stated previously.

Data for PCI Read Fifo is received from WB Master state machine. It is written to RAM address provided by ***fifo_control.v*** on rising WB clock edge when PCI Read Fifo Write Enable is asserted (received from WB Master state machine) and Fifo is not Full. PCI Target state machine performs a read on rising edge of PCI clock, during which it holds

PCI Read Fifo Read Enable asserted. Read address provided from ***fifo_control.v*** is applied to RAM address inputs on a port clocked by PCI clock, to provide next data and control signals to PCI Target state machine on Fifo's data and control outputs. ***fifo_control.v*** provides a module for generating synchronous RAM addresses for PCI Read Fifo. It provides write side RAM address (in WB clock domain) and read side RAM address (in PCI clock domain). It also generates different statuses. Statuses are always determined by comparing Grey coded read and write addresses to provide glitch free comparator outputs. Status outputs are sampled in Flip-Flops in appropriate clock domain. For example – WB clock domain always writes to PCI Read Fifo, so it is only interested in Fifo fullness. So Fifo's Full status is synchronized to WB clock domain. On the other hand, PCI clock domain always just reads from this Fifo and is therefore interested only in Fifo emptiness. Therefore status Empty is syncronized to PCI clock domain. Grey code pipeline is provided for generating statuses. Each clock domain has its own pipeline, values are compared between clock domains and results sampled at appropriate clocks as stated previously.

2.3.4.3 *wb_master.v*

Module implements WISHBONE master state machine and is provided for passing requests from ***delayed_sync.v*** and ***pciw_pcir_fifos.v*** to external WB slaves.

When Posted Write is prepared in PCI Write Fifo, the state machine fetches the address into address counter. These register is connected directly to address output bus. Then it fetches each data with byte enables (if Posted Write) and put it on WB bus until last data is fetched from PCIW Fifo, while address counter is incremented for each data phase, when data are terminated with **ACK_I**. If write is terminated with **RTY_I** signal, the last data transfer is repeated for the value of define WB_RTY_CNT_MAX. If there is no response on WB bus, internal signal (**set_retry**) is active after 8 WB clock periods, and is used as there would be a termination with **RTY_I** signal. If retry counter reaches maximum value, address, data, bus command and control signals (cause of the error, **error_source_out** and **write_rty_cnt_exp_out**) are written into ERR registers if this is enabled. The rest of posted write is emptied from PCIW Fifo until last data (more Posted Writes can be in PCIW Fifo). If write is terminated with **ERR_I** signal, then action is the same as when retry counter reaches maximum value.

When there is a delayed read request, it can not be processed until PCIW Fifo is empty (due to transaction ordering). Logic first determine how long a read should be performed. This is calculated from PCI bus command, pre-fetch enable bit of image control register and cache line size register (must be aligned to 4 DWORDs – 2 LSBits are zero). Then state machine fetches the address into address counter and starts the transaction. Address counter is incremented for each data phase, when data are terminated with **ACK_I**. If read is terminated with **RTY_I** signal, the last data transfer is repeated for the value of define WB_RTY_CNT_MAX. If there is no response on WB bus, internal signal (**set_retry**) is active after 8 WB clock periods, and is used as there would be a termination with **RTY_I** signal. If retry counter reaches maximum value because of **RTY_I** termination, then state machine stop reading, otherwise if retry counter reaches maximum value because of no response, then state machine write last “data” into PCIR Fifo with error control bit, the same as it would be terminated with **ERR_I**.

2.3.5 Description of Configuration space submodules

The module ***conf_space.v*** is also referenced as Configuration space (see Figure 2-3: Third picture - Module hierarchy of WB slave Unit and Configuration space). It consists of PCI Configuration Header and Device specific registers for WB slave and PCI target Units. It has some sub-modules used for synchronization, ***sync_module.v*** and ***synchronizer_flop.v***.

It has two ports for access registers by their address. One port is Read-Only, the other is Read/Write. Both ports have theirs own address and read data busses, but only Read/Write port has write data bus. Each port has its own address decoder for accessing appointed registers. Reading is asynchronous, while writing is synchronous. To have only one address decoder for Read/Write port, we set address decoded select signals **w_reg_select_dec[55:0]** in address decoder for reading from Read/Write port and then we use those selects for writing into individual registers.

Some register bits are constant and cannot be changed. Almost all others can be set to desired values. Only exceptions are status bits, which are set to logic ‘1’ when there occurs e.g. error and are deleted with writing ‘1’ into them.

If PCI bridge is implemented as HOST bridge, then Configuration access port from ***wb_slave.v*** (in WB slave Unit) is connected to Read/Write port of Configuration space and Configuration access port from ***pci_target32_interface.v*** (in PCI Target Unit) is connected to Read-Only port. For GUEST bridge implementation it is vice-versa.

If PCI bridge is implemented as HOST bridge with all 6 PCI images (PCI_IMAGE6) or if there are less images and disabled Read-Only access (NO_CNF_IMAGE), then there is no Read-Only port connection between Configuration space and PCI Target Unit. If PCI bridge is implemented as GUEST bridge and disabled Read-Only access (NO_CNF_IMAGE), then there is no Read-Only port connection between Configuration space and WB Slave Unit.

Read-Only port consists of inputs: **r_conf_address_in** and **r_re** (read enable); and of outputs: **r_conf_data_out**. Read/Write port consists of inputs: **w_conf_address_in**, **w_conf_data_in**, **w_byte_en**, **w_re** (read enable) and **w_we** (write enable); and of outputs: **w_conf_data_out**.

Basically all register bit outputs (control, status, address etc.) are connected directly to the modules, which action depends on them. Registers themselves are well described in PCI IP Core Specification.

OUTPUTS form PCI Configuration Header registers:

- **serr_enable**, **perr_response** (bits 8 and 6 of Command register): connected to ***pci_parity_check.v*** of PCI I/O
- **pci_master_enable** (bit 2 of Command register): connected to ***wb_slave.v*** of WB slave Unit
- **memory_space_enable**, **io_space_enable** (bits 1 and 0 of Command register): connected to ***pci_target32_interface.v*** of PCI raget Unit
- **cache_line_size_to_pci** (Cache Line Size register): connected to ***pci_master32_sm_if.v*** of WB slave Unit
- **cache_line_size_to_wb** (Cache Line Size register): connected to ***wb_master.v*** of PCI target Unit
- **cache_lsize_not_zero_to_wb**: connected to ***wb_slave.v*** of WB slave Unit and to ***wb_master.v*** of PCI target Unit
- **latency_tim** (Latency Timer register): connected to ***pci_master32_sm.v*** of WB slave Unit

OUTPUTS from PCI Image registers:

- **pci_base_addr*** (bits 31-**12** of P_BA* register): connected to ***pci_decoder.v*** of PCI target Unit
- **pci_memory_io*** (bit 0 of P_BA* register): connected to ***pci_decoder.v*** of PCI target Unit
- **pci_addr_mask*** (bits 31-**12** of P_AM* register): connected to ***pci_decoder.v*** of PCI target Unit
- **pci_tran_addr*** (bits 31-**12** of P_TA* register): connected to ***pci_decoder.v*** of PCI target Unit
- **pci_img_ctrl*** (bits 2-1 of P_IMG_CTRL* register): connected to ***pci_decoder.v*** (bit 2) and ***pci_target32_interface.v*** of PCI target Unit

* Stands for number from 0 to N (N is a number of implemented PCI images), see PCI IP Core Specification document.

12 This value of P_BA, P_AM and P_TA registers is determined according to PCI_Number_of_Decoded_Address_Lines (20 is maximum – 31:12, 1 is minimum – 31:31).

OUTPUTS from WB Image registers:

- **wb_base_addr**** (bits 31-**12** of W_BA** register): connected to ***decoder.v*** of WB slave Unit
- **wb_memory_io*** (bit 0 of W_BA* register): connected to ***wb_slave.v*** of WB slave Unit
- **wb_addr_mask*** (bits 31-**12** of W_AM* register): connected to ***decoder.v*** of WB slave Unit
- **wb_tran_addr*** (bits 31-**12** of W_TA* register): connected to ***decoder.v*** of WB slave Unit

- **wb_img_ctrl*** (bits 2-0 of W_IMG_CTRL* register): connected to ***decoder.v*** (bit 2) and ***wb_slave.v*** of WB slave Unit

* Stands for number from 1 to N (N is a number of implemented WB images).

** Stands for number from 0 to N (N is a number of implemented WB images), see PCI IP Core Specification document.

12 This value of W_BA, W_AM and W_TA registers is determined according to WB_Number_of_Decoded_Address_Lines (20 is maximum – 31:12, 1 is minimum – 31:31).

OUTPUTS from device specific control registers:

- **config_addr** (CNF_ADDR register): connected to ***conf_cyc_addr_dec.v*** of WB slave Unit if there is a HOST bridge implementation
- **icr_soft_res** (bit 31 of ICR register): connected to ***pci_rst_int.v*** of PCI I/O
- **int_out** (ORed bits 4-0 of ISR register):connected to ***pci_rst_int.v*** of PCI I/O

INPUTS to PCI Configuration Header register:

- **perr_in**, **serr_in** and **master_data_par_err** (bits 15, 14 and 8 of Status register): connected from ***pci_parity_check.v*** of PCI I/O
- **master_abort_recv** and **target_abort_recv** (bits 13 and 12 of Status register): connected from ***pci_master32_sm_if.v*** of WB slave Unit
- **target_abort_set** (bit 11 of Status register): connected from ***pci_target32_sm.v*** of PCI target Unit

INPUTS to PCI Error registers:

- **pci_error_be**, **pci_error_bc**, **pci_error_rty_exp** , **pci_error_es** and **pci_error_sig** (bits 31-28, 27-24, 10, 9 and 8 of P_ERR_CS register): connected from ***wb_master.v*** of PCI target Unit
- **pci_error_addr** (P_ERR_ADDR register): connected from ***wb_master.v*** of PCI target Unit
- **pci_error_data** (P_ERR_DATA register): connected from ***wb_master.v*** of PCI target Unit

INPUTS to WB Error registers:

- **wb_error_be** (bits 31-28 of W_ERR_CS register): connected from ***cur_out_reg.v*** of PCI I/O
- **wb_error_bc**, **wb_error_es** and **wb_error_sig** (bits 27-24, 9 and 8 of W_ERR_CS register): connected from ***pci_master32_sm_if.v*** of WB slave Unit
- **wb_error_addr** (W_ERR_ADDR register): connected from ***pci_master32_sm_if.v*** of WB slave Unit
- **wb_error_data** (W_ERR_DATA register): connected from ***cur_out_reg.v*** of PCI I/O

INPUTS to Interrupt Status register:

- **isr_sys_err_int** and **isr_par_err_int** (bits 4 and 3 of ISR register): connected from ***pci_master32_sm_if.v*** of WB slave Unit
- **pci_error_sig** (bit 2 of ISR register): connected from ***wb_master.v*** of PCI target Unit

- **wb_error_sig** (bit 1 of ISR register): connected from ***pci_master32_sm_if.v*** of WB slave Unit
- **isr_int_prop** (bit 0 of ISR register): connected from ***pci_rst_int.v*** of PCI I/O

Since access and monitoring of Configuration space is from two clock domains, some synchronization need to be done, where necessary (fixed values of course don't need to be synchronized, e.g. WB_CONF_SPC_BAR). Access from Read-Only and Read/Write ports is straight forward, because of system point of view, most of registers don't need to be synchronized. If PCI bridge is implemented as HOST, then WB device must initialize the Configuration space, otherwise (if GUEST) PCI device (the HOST) must initialize the Configuration space. Following registers are set only during initialization phase and therefore don't need to be synchronized:

- Interrupt Line register (PCI Header)
- P_IMG_CTRL0 - P_IMG_CTRLn (n is a number of implemented PCI images, see also PCI IP Core Specification document)
- P_BA0 - P_BAn (n is a number of implemented PCI images)
- P_AM0 - P_AMn (n is a number of implemented PCI images)
- P_TA0 - P_TAn (n is a number of implemented PCI images)
- W_IMG_CTRL1 - W_IMG_CTRLn (n is a number of implemented WB images, see also PCI IP Core Specification document)
- W_BA1 - W_BAn (n is a number of implemented WB images)
- W_AM1 - W_AMn (n is a number of implemented WB images)
- W_TA1 - W_TAn (n is a number of implemented WB images)

If the initialization device needs to change any of this registers when system already normally operate, then it must prevent all devices on the opposite bus to access through bridge and stop the opposite bridge unit.

Following registers are used only when PCI bridge is implemented as HOST and also don't need to be synchronized, because they are used for specific cycles totally by WB slave Unit:

- CNF_ADDR (for configuration cycles, see also PCI IP Core Specification document)
- CNF_DATA (for configuration cycles)
- INT_ACK (for interrupt acknowledge cycles)

Following registers are somehow linked with synchronization (their outputs, their deleting etc.), except specific bits. therefore they are described separately.

- Bits **15-11** and **8** of Status register (PCI Header) are synchronized, when PCI bridge is implemented as HOST. This bits are always set on a PCI CLK, and in a case of HOST bridge, they are read and deleted on WB CLK. The value of status bit is also blocked the next WB CLK period, when delete bit is written, so there is like immediate response of status. The value of status bit itself is also synchronized (with two flip-flops on WB CLK) to WB CLK. See Figure 2-5: SET and DELETE synchronization for status reporting bit.

- Bits **8, 6, 1** and **0** of Command register (PCI Header) are synchronized, when PCI bridge is implemented as HOST. Their values are always used on PCI CLK (in ***pci_parity_check.v*** and ***pci_decoder.v***). This bits are set in a case of HOST bridge on WB CLK. Their values are therefore synchronized (with two flip-flops on PCI CLK) to PCI CLK.
- Bit **2** of Command register (PCI Header) is synchronized, when PCI bridge is implemented as GUEST. Its value is always used on WB CLK (in ***wb_slave.v***). This bit is set in a case of GUEST bridge on PCI CLK. Its value is therefore synchronized (with two flip-flops on WB CLK) to WB CLK.
- Cache Line Size register (PCI Header) is synchronized.
  Outputs **cache_line_size_to_pci** are always used on PCI CLK (***pci_master32_sm_if.v***) and are synchronized, when PCI bridge is implemented as HOST. This register is set in a case of HOST bridge on WB CLK. Its outputs are therefore synchronized (with two flip-flops on PCI CLK) to PCI CLK.
  Outputs **cache_line_size_to_wb** are always used on WB CLK (***wb_master.v***) and are synchronized, when PCI bridge is implemented as GUEST. This register is set in a case of GUEST bridge on PCI CLK. Its outputs are therefore synchronized (with two flip-flops on WB CLK) to WB CLK.
- Latency Timer register (PCI Header) is synchronized, when PCI bridge is implemented as HOST. Its outputs are always used on PCI CLK (in ***pci_master32_sm.v***). This register is set in a case of HOST bridge on WB CLK. Its outputs are therefore synchronized (with two flip-flops on PCI CLK) to PCI CLK.
- Bit **0** (ERR_EN) of P_ERR_CS and W_ERR_CS registers is not synchronized, because it is only the enable for error pulse on error signal.
- Bit **8** (ERR_SIG) of P_ERR_CS register is synchronized, when PCI bridge is implemented as GUEST. This bit is always set on WB CLK if bit **0** (ERR_EN) of P_ERR_CS register is set. Its value is used in a case of GUEST bridge on PCI CLK (software on other PCI device). The value of status bit is also blocked the next PCI CLK period, when delete bit is written, so there is like immediate response of status. The value of status bit itself is also synchronized (with two flip-flops on PCI CLK) to PCI CLK. See Figure 2-5: SET and DELETE synchronization for status reporting bit.
- Bit **8** (ERR_SIG) of W_ERR_CS register is synchronized, when PCI bridge is implemented as HOST. This bit is always set on PCI CLK if bit **0** (ERR_EN) of W_ERR_CS register is set. Its value is used in a case of HOST bridge on WB CLK (software on other WB device). The value of status bit is also blocked the next WB CLK period, when delete bit is written, so there is like immediate response of status. The value of status bit itself is also synchronized (with two flip-flops on WB CLK) to WB CLK. See Figure 2-5: SET and DELETE synchronization for status reporting bit.

- Bits **31-28** (BE), **27-24** (BC), **10** (RTY_EXP only for P_ERR_CS) and **9** (ES) of P_ERR_CS and W_ERR_CS registers are only status information bits and are not synchronized, because they are valid when bit **8** (ERR_SIG) is valid (this bit is synchronized). If software uses error poling, it must respond only to bit **8** (ERR_SIG)!
- Bit **31** (SW_RST) of ICR register is not synchronized, since reset is asynchronously propagated. If PCI bridge is implemented as HOST, then value of this bit is propagated to PCI bus, otherwise it is propagated to WB bus.
- Bits **4** (SERR_INT_EN) and **3** (PERR_INT_EN) of ICR register are implemented only when bridge is implemented as HOST (GUEST bridges only trigger interrupts to PCI bus; see also PCI IP Core Specification). This bits are not synchronized because they are only enables for interrupt pulses on interrupt signals.
- Bits **2-0** (PCI_EINT_EN, WB_EINT_EN and INT_PROP_EN) of ICR register are not synchronized because they are only enables for interrupt pulses on interrupt signals.
- Bits **4** (SERR_INT) and **3** (PERR_INT) of ISR register are synchronized and are implemented only when bridge is implemented as HOST (GUEST bridges only trigger interrupts to PCI bus; see also PCI IP Core Specification). This bits are always set on PCI CLK if bits **4** (SERR_INT_EN) and **3** (PERR_INT_EN) respectively, of ICR register, are set. Theirs values are used on WB CLK (software on other WB device). The values of status bits are also blocked the next WB CLK period, when each delete bit is written, so there is like immediate response of status. The values of status bit them-self are also synchronized (with two flip-flops on WB CLK) to WB CLK. See Figure 2-5: SET and DELETE synchronization for status reporting bit.
- Bit **2** (PCI_EINT) of ISR register is synchronized, when PCI bridge is implemented as GUEST. This bit is always set on WB CLK if bit **2** (PCI_EINT_EN) of ICR register and bit **0** (ERR_EN) of P_ERR_CS register are set. Its value is used in a case of GUEST bridge on PCI CLK (software on other PCI device). The value of status bit is also blocked the next PCI CLK period, when delete bit is written, so there is like immediate response of status. The value of status bit itself is also synchronized (with two flip-flops on PCI CLK) to PCI CLK. See Figure 2-5: SET and DELETE synchronization for status reporting bit.
- Bit **1** (WB_EINT) of ISR register is synchronized, when PCI bridge is implemented as HOST. This bit is always set on PCI CLK if bit **1** (WB_EINT_EN) of ICR register and bit **0** (ERR_EN) of W_ERR_CS register are set. Its value is used in a case of HOST bridge on WB CLK (software on other WB device). The value of status bit is also blocked the next WB CLK period, when delete bit is written, so there is like immediate response of status. The value of status bit itself is also synchronized (with two flip-flops

on WB CLK) to WB CLK. See Figure 2-5: SET and DELETE synchronization for status reporting bit.

- Bit **0** (INT) of ISR register is synchronized.
  When PCI bridge is implemented as HOST, is this bit set on PCI CLK if bit **0** (INT_PROP_EN) of ICR register is set. Its value is used in a case of HOST bridge on WB CLK (software on other WB device). The value of status bit itself is also synchronized (with two flip-flops on WB CLK) to WB CLK.
  When PCI bridge is implemented as GUEST, is this bit set on WB CLK if bit **0** (INT_PROP_EN) of ICR register is set. Its value is used in a case of GUEST bridge on PCI CLK (software on other PCI device). The value of status bit itself is also synchronized (with two flip-flops on PCI CLK) to PCI CLK.

There is also a synchronization, before interrupts are connected to **int_out** signal.

All interrupts are logically ORed and then synchronized. When PCI bridge is implemented as HOST, the synchronization is done (with two flip-flops on WB CLK) to WB CLK, otherwise the synchronization is done (with two flip-flops on PCI CLK) to PCI CLK.

Signal, which sets bit 0 (INT) of ISR register, is connected to OR logic directly, before it is synchronized to status bit in ISR. All other 4 bits (in case of HOST bridge, otherwise there are 2 bits) are connected to OR logic from AND gates after flip-flops, which are set directly from interrupt signals, if they are synchronized, otherwise they are connected from flip-flops.

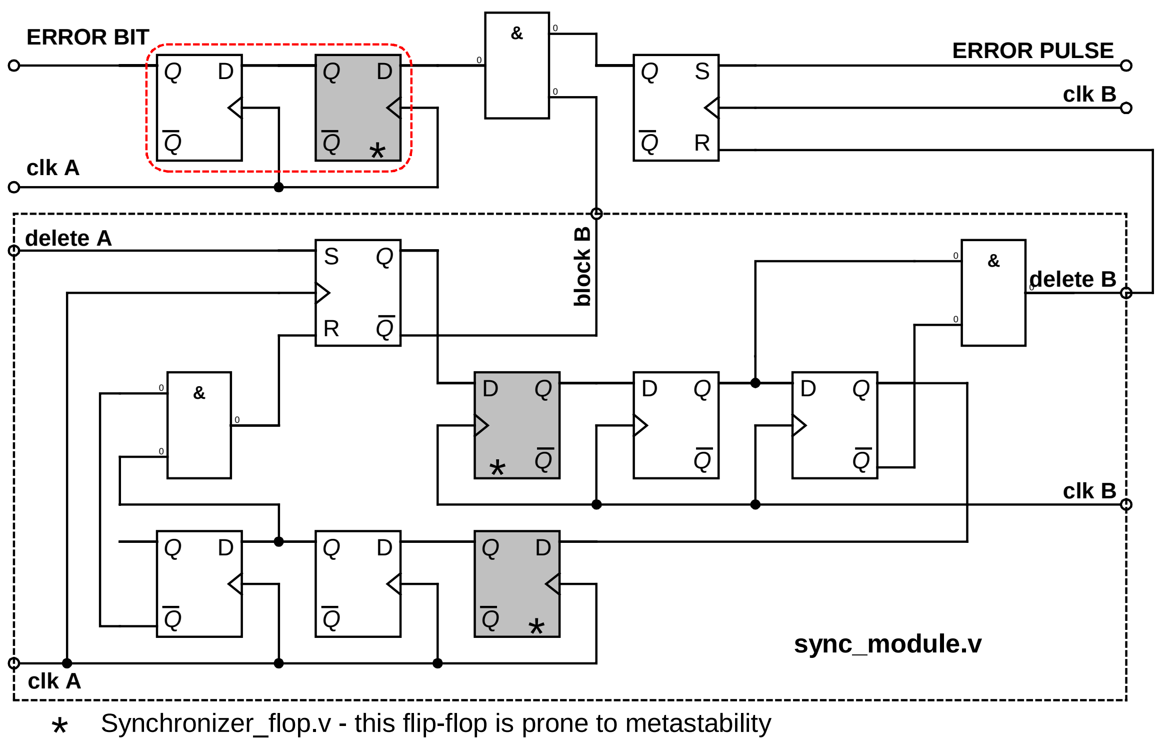


Figure 2-5: SET and DELETE synchronization for status reporting bit

2.4 Changeable Core Constants

PCI IP Core user changeable constants are defined in **pci_user_constants.v** in project's **rtl/verilog** subdirectory.

2.4.1 Fifo size constants

Values in following defines indirectly define sizes of implemented PCI IP Core Fifos:

- WBW_ADDR_LENGTH – number defined here defines WB Write Fifo's size. Size is calculated as 2^^WBW_ADDR_LENGTH. Note that Fifo's control logic is such, that one location in RAM is always empty, so usable Fifo size is (2^^WBW_ADDR_LENGTH) – 1. Any value equal to or larger than 3 is valid here – the only restriction is the size of RAMs instantiated for Fifo storage.

- WBR_ADDR_LENGTH – same as WBW_ADDR_LENGTH, except this applies to WB Read Fifo implementation.
- PCIW_ADDR_LENGTH – same as WBW_ADDR_LENGTH, except this applies to PCI Write Fifo implementation.
- PCIR_ADDR_LENGTH – same as WBW_ADDR_LENGTH, except this applies to PCI Read Fifo implementation.

2.4.2 Fifo RAM instantiation Constants

RAMs are instantiated in **pci_tpram.v** and **wb_tpram.v**, each used in PCI Target Unit and WB Slave Unit respectively.

Following defines in **pci_user_constants.v** influence RAM – storage implementation for all Fifos in the PCI IP Core:

- RAM_DONT_SHARE – if defined, than each Fifo in PCI IP Core has storage space implemented in its own RAM instance. That would mean 12 block RAM instances in XILINX FPGA or 4 RAM instances in ASIC. This instances can be dual port (write and read port in different clock domains). If this is not defined, RAM sharing between Fifos is performed. Two Fifos use one RAM instance for their storage. That would mean 6 block RAM instances used in XILINX FPGA or 2 RAM instances in an ASIC. This RAM instances must be two port (read and write port in different clock domains). When RAM sharing is implemented, Fifos split the RAM address space in half. One half is used for one Fifo, other half for the other.
- FPGA – if FPGA is defined, then some restrictions on RAM instantiating apply.
    - XILINX - Currently only Xilinx block and distributed RAM instances are supported, which are included into design by defining XILINX in addition to FPGA. Xilinx Block RAMs used are configured as RAMB4_S16_S16, which is dual 16 bit port RAM. It can store 256 entries, so address for this RAM is 8 bit in length (no exceptions). For smaller Fifo sizes (8 or 16), Xilinx distributed RAM instances can be included into the implementation by defining WB_XILINX_DIST_RAM or PCI_XILINX_DIST_RAM. PCI_FIFO_RAM_ADDR_LENGTH
      and WB_FIFO_RAM_ADDR_LENGTH must be set to 4 in this case.
- FPGA not defined – user must generate appropriate RAM instances with RAM generators provided by various ASIC library vendors. RAM has to be 40 bits wide and a size equal to or larger than 8 for Fifo implementations that do not share RAM instances, and equal to or larger than 16 for implementations that share RAMs between FIFOs. Designer is also responsible for defining appropriate RAM address lengths.

  - PCI_FIFO_RAM_ADDR_LENGTH – must be equal to address length of instantiated RAM in **pci_tpram.v**.
  - WB_FIFO_RAM_ADDR_LENGTH – must be equal to address length of instantiated RAM in **wb_tpram.v**.

2.4.3 PCI I/O Pads constants

Depending on PCI I/O buffers used, designer may choose how PCI IP Core generates its PCI bus signals output enables.

- ACTIVE_HIGH_OE – selects active high output enables generated by the core
- ACTIVE_LOW_OE - selects active low output enables generated by the core

Designer must define only one of these two options for given implementation – never both at the same time!

2.4.4 HOST / GUEST implementation selection

- HOST – if defined, core will be implemented or simulated with HOST bridge features enabled
- GUEST – if defined, core will be implemented or simulated with GUEST bridge features enabled

These two defines are mutually exclusive. PCI IP Core Specification throughout contains all information regarding different behavior of the Core, if implemented as HOST or GUEST.

2.4.5 Image implementation constants

PCI IP Core can be implemented with various number of images for accessing WISHBONE bus address space from PCI bus and vice – versa.

2.4.6 Optional Read-Only Configuration image implementation

- NO_CNF_IMAGE – constant definition prevents Read-Only configuration image to be implemented. Read-Only Configuration space access can be provided through PCI image 0 for HOST implementation of the Core, and through WB image 0 for GUEST implementation. If NO_CNF_IMAGE is defined, then this image is not implemented. This saves one address decoder and one multiplexer in overall design. It is recommended to define this constant, when the Core is used in such an application that does not need this Read-Only access to Configuration space. PCI Bridge implemented as GUEST with CRT controller connected to WB Slave interface is an example application. Although CRT controller has WB Master interface to fetch its own pixel data from PCI bus, it will never need or use access to Bridge's Configuration space. So by defining NO_CNF_IMAGE, some additional space is saved.

- ADDR_TRAN_IMPL – if defined, address translation functionality is added to decoders for both, PCI and WISHBONE accesses. Address translation implementation is useful when application uses fixed address map, while PCI address map is configurable. If address translation is not needed, this define can be commented out to allow faster decode timing.

2.4.7 PCI images' implementation constants

Constants' definitions described in this chapter control the number and minimum address range of PCI Target images. PCI image 1 is always implemented, without any exceptions, so no define is provided for it. Image implementation means that all required registers for each image are implemented in Configuration space and each image needs its own decoder. Image constants' definitions have quite an impact on Core's size and speed.

- PCI_NUM_OF_DEC_ADDR_LINES – number defined here is used for controlling implementation of PCI images' decoders. It defines how many address lines are used for decoding PCI Target accesses and therefore defines what minimum image size can be. Maximum number allowed is 20 (4KB minimum image size) and minimum is 1 (2GB minimum image size – this value implies that more than two images cannot be enabled at the same time). The number of decoded address lines also influences the speed of the Core.
- PCI_IMAGE0 – this define only has meaning when HOST and NO_CNF_IMAGE are defined also. This enables usage of all 6 PCI Target images for accessing WISHBONE bus address space from PCI address space. Otherwise, PCI_IMAGE0 does not needs to be defined, since it is always used for accessing Configuration space.
- PCI_IMAGE2 – if defined, PCI Target image 2 is implemented
- PCI_IMAGE3 – if defined, PCI Target image 3 is implemented
- PCI_IMAGE4 – if defined, PCI Target image 4 is implemented
- PCI_IMAGE5 – if defined, PCI Target image 5 is implemented
- PCI_AM0, PCI_AM1, PCI_AM2, PCI_AM3, PCI_AM4, PCI_AM5 – values defined with this macros are initial (reset) values of PCI address masks'

registers. These are very important if the Core is implemented as GUEST, since configuration is done via PCI Target state machine. If the designer wants an implemented PCI Target image to be detected by device independent software at system power-up, he has to set initial masks to enabled state – MS bit has to be 1. Other bits can have a value of 1 or zero, depending on what size of an image has to be presented to the software. The masks can be set inactive also, but device independent software won't detect implemented PCI Target images and therefore not configure them. Device specific software will then have to jump in to configure images with inactive initial masks defined, which also means that it will probably have to rebuild PCI address space map.

- PCI_BA0_MEM_IO, PCI_BA1_MEM_IO, PCI_BA2_MEM_IO, PCI_BA3_MEM_IO, PCI_BA4_MEM_IO, PCI_BA5_MEM_IO – Those are initial values of PCI Base Address registers' bits 0. If the Core is configured as HOST, this initial values can later be changed by writing appropriate value to PCI Base Address register x. If the core is GUEST, than this values are hardwired, because device independent software must know in advance where to map each PCI Base Address.

2.4.8 WISHBONE images' implementation constants

Constants' definitions described in this chapter control the number and minimum address range of WISHBONE Slave images. WISHBONE image 1 is always implemented, without any exceptions, so no define is provided for it. Image implementation means that all required registers for each image are implemented in Configuration space and each image needs its own decoder. Image constants' definitions have quite an impact on Core's size and speed.

- WB_NUM_OF_DEC_ADDR_LINES – number defined here is used for controlling implementation of WISHBONE images' decoders. It defines how many address lines are used for decoding WISHBONE Slave accesses and therefore defines what minimum image size can be. Maximum number allowed is 20 (4KB minimum image size) and minimum is 1 (2GB minimum image size – this value implies that more than two images cannot be enabled at the same time). The number of decoded address lines also influences the speed of the Core.
- WB_IMAGE2 – if defined, WB Slave image 2 is implemented
- WB_IMAGE3 – if defined, WB Slave image 3 is implemented
- WB_IMAGE4 – if defined, WB Slave image 4 is implemented
- WB_IMAGE5 – if defined, WB Slave image 5 is implemented

2.4.9 WISHBONE Slave specific constants

- WB_DECODE_FAST, WB_DECODE_MEDIUM, WB_DECODE_SLOW – these are all mutually exclusive definitions. They define how many cycles WISHBONE Slave state machine takes to decode an access. WB Slave state machine samples decode information on same clock edge cycle is started when WB_DECODE_FAST is defined. It samples decode data one clock after cycle is started if WB_DECODE_MEDIUM is defined. If WB_DECODE_SLOW is defined, state machine samples decode data two clocks after cycle is started. This defines can or should be taken into account during synthesis to provide information about multicycle paths.
- WB_CONFIGURATION_BASE – defines 20 bit value for WISHBONE configuration image address. Those bits are compared to 20 MS bits of WISHBONE Slave address to decode Configuration accesses from WISHBONE bus. This is constant value and cannot be changed after the Core is implemented, since WISHBONE bus does not provide any special mechanism for device configuration.
- REGISTER_WBS_OUTPUTS – this define is used mostly when the Core is included into top level application as precompiled macro, because timings cannot be optimized during top level synthesis. WISHBONE Slave state machine registers all outputs when this macro is defined and introduces one wait cycle after every transfer. It also registers all interfaces with core internals to release a burden on WISHBONE Slave inputs also.

2.4.10 PCI specific constants

- PCI33, PCI66 – these two mutually exclusive defines are used for simulation purposes (PCI clock speed) and to set 66MHz Capable bit in PCI Device Status register, if PCI66 is defined. There are no other features dependent on those defines.
- HEADER_VENDOR_ID – each PCI bus compatible hardware vendor gets its 16 bit hexadecimal ID from PCI SIG organization. It should be specified in this define. This value shows up in Vendor ID register of PCI Type0 Configuration Header.
- HEADER_DEVICE_ID – Device ID is vendor specific, 16 bit hexadecimal value. It shows up in Device ID register of PCI Type0 Configuration Header.
- HEADER_REVISION_ID – Also vendor specific, 8 bit hexadecimal value, that shows up in Revision ID register of PCI Type0 Configuration Header.

2.4.11 *WISHBONE Master specific constants*

- REGISTER_WBM_OUTPUTS – this define is used mostly when the Core is included into top level application as precompiled macro, because timings cannot be optimized during top level synthesis. WISHBONE Master state machine registers all outputs when this macro is defined and introduces one wait cycle after every transfer. It also registers input data and some interfaces with core internals to release a burden on WISHBONE Master control inputs also.
- WB_RTY_CNT_MAX – this define is used to prevent deadlock in WB Master state machine for maximum counting value of RTY terminations on WB bus, before ACK or ERR terminations. The last two terminations reset the counter. This counter is also used, when no WB device responds (e.g. if accessing to unused memory locations). In that case internal **set_retry** signal is set every 8 WB clock periods and counter counts to maximum value defined.

2.5 Changeable constants dependencies

This chapter describes what the user should be aware of when defining / not defining some of the changeable Core constants.

- **WBW_ADDR_LENGTH, WBR_ADDR_LENGTH** – numbers defined cannot have value less than 3, because of Fifos control logic implementation. **WBW(R)_ADDR_LENGTH** constant value also depends on value of **WB_FIFO_RAM_ADDR_LENGTH** and definition of **WB_RAM_DONT_SHARE**. If **WB_RAM_DONT_SHARE** is defined, then **WBW(R)_ADDR_LENGTH** can be defined as any number between 3 and **WB_FIFO_RAM_ADDR_LENGTH**. Otherwise, **WBW(R)_ADDR_LENGTH** can be defined as any number between 3 and (**WB_FIFO_RAM_ADDR_LENGTH – 1**).
- **PCIW_ADDR_LENGTH, PCIR_ADDR_LENGTH** – numbers defined cannot have value less than 3, because of Fifo control logic implementation. **PCIW(R)_ADDR_LENGTH** constant value also depends on value of **PCI_FIFO_RAM_ADDR_LENGTH** and definition of **PCI_RAM_DONT_SHARE**. If **PCI_RAM_DONT_SHARE** is defined, then **PCIW(R)_ADDR_LENGTH** can be defined as any number between 3 and **PCI_FIFO_RAM_ADDR_LENGTH**. Otherwise, **PCIW(R)_ADDR_LENGTH** can be defined as any number between 3 and (**PCI_FIFO_RAM_ADDR_LENGTH – 1**).

- **WB_FIFO_RAM_ADDR_LENGTH, PCI_FIFO_RAM_ADDR_LENGTH** – depend on other constants only if **FPGA** is defined. If **XILINX** is defined, user can select between two possible RAM instances – Block Select RAM+ or Distributed RAM. If **WB(PCI)_XILINX_RAMB4** is defined, this selects Block Select RAM+ for WB (PCI) Read and Write Fifos implementation. **WB(PCI)_FIFO_RAM_ADDR_LENGTH** can only be defined to 8 in this case. If **WB(PCI)_XILINX_DIST_RAM** is defined, then **WB(PCI)_FIFO_RAM_ADDR_LENGTH** constant can only be defined to a value of 4. User can use Distributed and Block Select RAM+ in the same design, by defining Block RAM for PCI(WB) Fifos and Distributed RAM for WB(PCI) Fifos. When the Core is to be implemented in an ASIC, user must include appropriate RAM instances into **wb_tpram.v** and **pci_tpram.v** under appropriate define and edit these two constants to reflect proper information about instantiated RAM blocks.
- **WB(PCI)_XILINX_RAMB4** – user must define **FPGA** and **XILINX** also, to use Xilinx Block RAMs.
- **WB(PCI)_XILINX_DIST_RAM –** user must define **FPGA**, **XILINX** and corresponding **(WB)PCI_RAM_DONT_SHARE**. Distributed RAM can't be shared between Fifos, because it only has one write port.
- **PCI_NUM_OF_DEC_ADDR_LINES** – maximum value can be 20 and minimum 1, which implements minimum PCI image size of 4KB or 2GB respectively. This number also depends on number of implemented PCI images – bridge cannot decode accesses to 6 different images by decoding only 1 bit of address. If 1 or 2 PCI images are implemented, then minimum number of decoded address lines can be 1, if 3 or 4 are implemented, at least two address lines must be decoded and if 5 or 6 images are implemented, at least 3 address lines must be decoded. PCI images are implemented by defining **PCI_IMAGE2** through **PCI_IMAGE5** and **PCI_IMAGE0** for **HOST** bridge implementation with **NO_CNF_IMAGE** defined. Note that PCI image 1 is always implemented. PCI image 0 is implemented when bridge is configured as **GUEST** or when it is configured as **HOST** and **NO_CNF_IMAGE** is not defined or **PCI_IMAGE0** is defined.
- **WB_NUM_OF_DEC_ADDR_LINES** – rules described for PCI number of decoded address lines apply to WISHBONE number of decoded address lines also, except the number depends on WISHBONE Images implemented. WISHBONE Image 1 is always implemented. WISHBONE Image 0 is implemented in **HOST** implementation of the bridge or in **GUEST** implementation of the bridge when **NO_CNF_IMAGE** is not defined. Other WISHBONE Images are selected by defining **WB_IMAGE2** through **WB_IMAGE5**.

- **WB_DECODE_FAST** – fast decoding in WISHBONE Slave state machine is possible only when **REGISTER_WBS_OUTPUTS** is not defined. If **REGISTER_WBS_OUTPUTS** is defined, then medium decode is implemented in WISHBONE Slave state machine, unless **WB_DECODE_SLOW** is defined.

2.6 Unchangeable Core Constants

Constants defined in file **pci_constants.v** in project's **rtl/verilog** subdirectory are not to be changed by the end user of the core, so they are just described here briefly for reference.

2.6.1 Fifo constants

- WBW_DEPTH – size of WB Write Fifo calculated from user constant WBW_ADDR_LENGTH
- WBR_DEPTH – size of WB Read Fifo calculated from user constant WBR_ADDR_LENGTH
- PCIW_DEPTH – size of PCI Write Fifo calculated from user constant PCIW_ADDR_LENGTH
- PCIR_DEPTH – size of PCI Read Fifo calculated from user constant PCIR_ADDR_LENGTH
- *_BIT – Fifo Control bus bit positions – used internally by the core for decoding what kind of data or transaction is on top of Fifo.

2.6.2 WISHBONE Slave constants

- WB_AM0 – 20 bit address mask value for WISHBONE configuration image. Used for decoding WISHBONE Configuration accesses and can't be changed.

2.6.3 Configuration registers' address constants

- *_ADDR – these defines specify the offset of each individual register residing in Configuration space. Defines are also used in testbench for accessing and checking value of registers.

2.6.4 66Mhz Capable bit constant

- HEADER_66MHz – specifies a value for 66MHz Capable bit in Device Status register depending on PCI33 and PCI66 defines.

3 PCI Bridge Testbench

3.1 Overview

PCI Bridge testbench consists of a whole environment for testing PCI Bridge including PCI and WISHBONE bus models with bus monitors and test-cases which use those models to stimulate transactions through PCI Bridge. Those transactions are checked when Wishbone Slave Unit and PCI Target Unit operates separately and simultaneously.

3.2 Testbench File Hierarchy

The hierarchy of modules in the Testbench of the PCI Bridge core is shown here with file tree. Each file here implements one module in a hierarchy source files of the Testbench are in the **pci\bench\verilog** subdirectory.

```
system.v
.       top.v
.       wb_bus_monitor.v*
.       wb_master_behavioral.v
.       .       wb_master32.v
.       wb_slave_behavioral.v
.       pci_bus_monitor.v
.       pci_blue_arbiter.v
.       pci_behavioral_device.v*
.       .       pci_behaviorial_master.v
```

- - **pci_behaviorial_target.v**
- - **delayed_test_pad.v***
- **pci_behavioral_iack_target.v**
- **pci_unsupported_commands_master.v**

* Files are used within one module more than once: **wb_bus_monitor.v** is used within system.v for 2 times, **pci_behavioral_device.v** is used within system.v for 2 times, **delayed_test_pad.v** is used within pci_behavioral_device.v for 9 times.

3.2.1 Testbench Module Hierarchy

Module hierarchy is shown in detail in the following picture. Description of modules and their connections is in the chapter 3.3, Description of Testbench Modules.

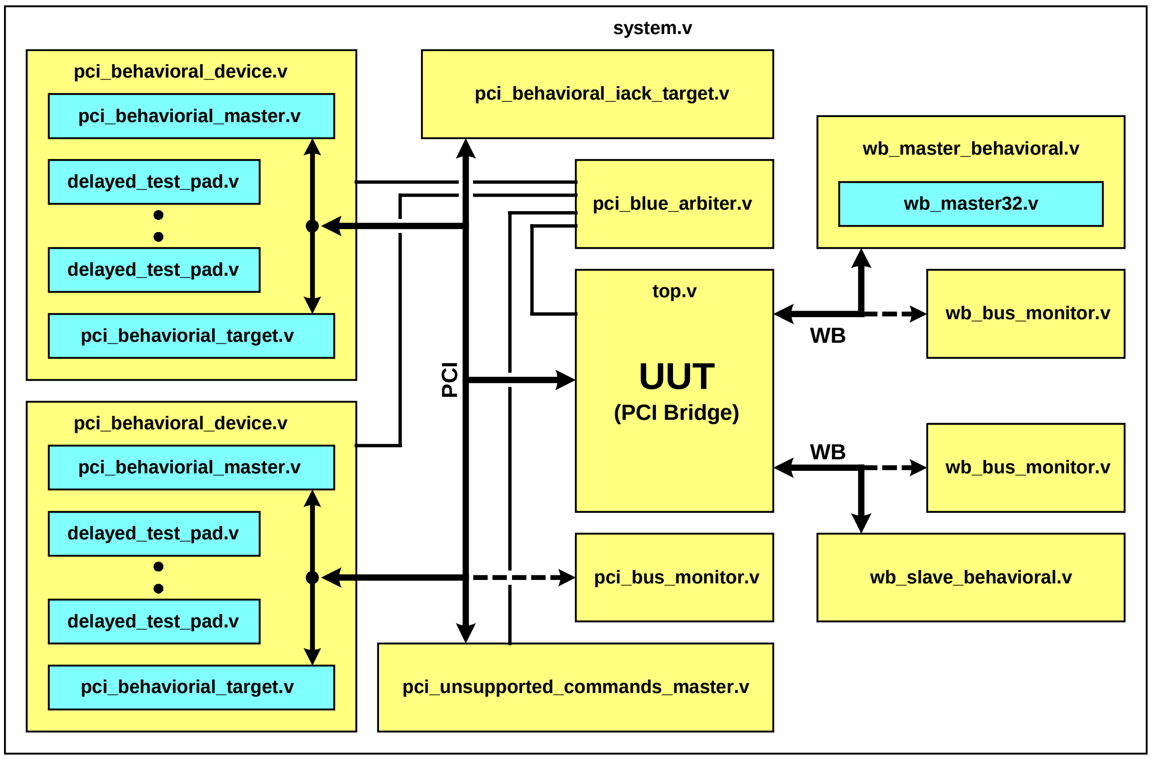


Figure 3-1: Testbench module hierarchy

3.3 Description of Testbench Modules

The module system.v is used as testing environment and it incorporates beside all test submodules, functions and tasks also Unit Under Test (PCI Bridge). Description of tasks is covered in chapter Description of Testcases, while all test submodules are described in the following chapters.

3.3.1 Description of PCI submodules

3.3.1.1 pci_bus_monitor submodule

The module ***pci_bus_monitor.v*** monitors the PCI Bus and tries to see PCI Protocol Errors. This module also has access to the individual PCI Bus OE signals for each behavioral interface (either through extra output ports or through "." notation), and it can see when more than one interface is driving the bus, even of the values are the same.

Author of this module is Blue Beaver.

3.3.1.2 pci_blue_arbiter submodule

The module ***pci_blue_arbiter.v*** is used for arbitration on the PCI bus. It has 4 external PCI Request/Grant pairs and one internal Request/Grant Pair. All 4 external PCI Request/Grant pairs are used for 2 ***pci_behavioral_device*** submodules, ***pci_unsupported_commands_master*** submodule and for UUT - PCI Bridge.

Author of this module is Blue Beaver.

3.3.1.3 pci_behavioral_device submodule

The module ***pci_behavioral_device.v*** is used as PCI behavioral interface and includes two submodules for its operation; ***pci_behavioral_master.v*** and ***pci_behavioral_target.v***.

The module ***pci_behavioral_master.v*** accepts commands from top-level stimulus task. Regarding the arguments passed to this submodule, it generates appropriate bus command. All bus cycles must terminate as expected by top-level task. It also arbitrates for the bus, sends data when writing, and compares returned data when reading. Comparing of read data is sometimes switched off, e.g. when NO data was written into a location from which a device is reading. Read data is also provided to the top-level (Note: providing codes of operation on AD lines to the PCI Target device is disabled).

The module ***pci_behavioral_target.v*** receives PCI commands from the PCI bus. Its termination of cycle is controlled by top-level (Note: providing codes of operation on AD lines to the PCI Target device is disabled). This Target contains Configuration registers and a 256 byte scratch SRAM. It responds with data when it is given a Read command, and checks data when it is given a Write command.

Author of this module is Blue Beaver.

3.3.1.4 pci_behavioral_iack_target submodule

The module ***pci_behavioral_iack_target.v*** is Target-only module. It is used to respond only to interrupt acknowledge commands, because other models don't respond to that command. When it should respond and the value of the interrupt vector is controlled by top-level.

3.3.1.5 pci_unsupported_commands_master submodule

The module ***pci_unsupported_commands_master.v*** is Master-only module. It is used to generate PCI bus commands, that are not supported by PCI Bridge Target device and those commands can not be generated by ***pci_behavioral_device.v*** module. It is also used for parity checking with ability to generate the address parity error on first and/or on second address phase of the DUAL ADDRESS CYCLE bus command. This bus command is unsupported, but parity checker must check for parity errors.

This PCI Master module waits for master-abort termination of the initiated bus cycle.

3.3.2 Description of WB submodules

3.3.2.1 wb_bus_monitor submodule

The module ***wb_bus_monitor.v*** monitors the WB Bus and tries to see WB Protocol Errors. There are two point-to-point WB buses (WB master from PCI Target Unit and WB slave from WB Slave Unit), and two WB bus monitors, one for each WB bus.

3.3.2.2 wb_master_behavioral submodule

The module ***wb_master_behavioral.v*** is used to initiates WB cycles to WB Slave in the PCI bridge. That is controlled by top-level. This module also includes a submodule ***wb_master32.v***, which is used to generate proper WB cycles. The length and type of each cycle is controlled by ***wb_master_behavioral.v*** module.

3.3.2.3 wb_slave_behavioral submodule

The module ***wb_slave_behavioral.v*** responds to cycles initiated by WB Master in the PCI bridge. When to respond and a type of cycle termination is controlled by top-level. This module also incorporates a block of SRAM.

3.4 Description of Testcases

There are some tasks not used as testcases (e.g. **fill_memory**, **do_reset** and specific tasks for initializing some registers, see chapters 3.4.1 and 3.4.2), but are significant for proper working of testbench. But there are some tasks used as basic tasks for all testcases (e.g. **DO_REF** – located in **system.v** module, **wishbone_master.wb_single_write** – located in **wb_master_behavioral.v** module, and other **wishbone_master.wb_…** tasks).

All Testcases are in **system.v** file and are sometimes combined with more tasks or are just a part of one task. Testcases are controlled in a task **run_tests**; some system parameters (behavioral master and target wait cycles – tb_init_waits, tb_subseq_waits; behavioral target response time – tb_target_decode_speed) are used with all possible combinations while running all testcases (most of testcases are used more than one time). In the following chapters basic descriptions for groups of testcases are presented. Each testcase has its name with a test type meaning. For deeper explanations of testcases see the comments in the **system.v** file.

Testbench also provide log files as a result of all tests into **pci/sim/rtl_sim/log** directory. File **pci_tb.log** has all testcases results written as SUCCESSFULL or FAIL. Files **pci_mon.log**, **wbu_mon.log** and **pciu_mon.log** report any wrong and suspicious activities on PCI and both WB buses.

3.4.1 Description of WBU Testcases

Before running testcases for WBU, some initialization must be done.

With task **configure_bridge_target**, some registers in PCI Bridge Target are initialized (master and target are enabled, base address register 0 is set and enabled).

With task **configure_target** each behavioral PCI target is initialized.

Task **find_pci_devices** executes only if PCI Bridge is implemented as HOST. It is not referenced in any log file as a testcase, since it is like a part of initialization. It checks if all PCI devices are connected correctly in the testbench with PCI Master configuration cycles.

3.4.1.1 Testcases for Configuration

Testcases described here are fundamental for all tests to pass successful. They set the configuration of the WBU of the PCI Bridge, but they don't test WB to PCI accesses. Because of that, they are not listed in the **pci_tb.log** file.

Following testcases are for configuring normal writes and reads through IMAGEs. Task location for testcases is: **run_tests –> test_wb_image**.

"WB IMAGE CONFIGURATION"

"CONFIGURE ADDRESS TRANSLATION FOR WISHBONE IMAGE"

"CHANGE WB IMAGE BASE ADDRESS"

"ENABLE WB IMAGE ADDRESS TRANSLATION"

"ENABLING/DISABLING IMAGE'S FEATURES"

"DISABLING WB IMAGE"

Following testcases are for configuring erroneous WB writes and reads through IMAGEs. Task location for testcases is: **run_tests –> wb_slave_errors**.

"CONFIGURING IMAGE FOR WB SLAVE ERROR TERMINATION TESTING"

"RECONFIGURING IMAGE TO I/O MAPPED ADDRESS SPACE"

"DISABLE IMAGE"

Following testcases are for configuring erroneous PCI writes and reads through IMAGEs. Task location for testcases is: **run_tests –> wb_to_pci_error_handling**.

"CONFIGURING BRIDGE FOR PCI ERROR TERMINATION RESPONSE BY WB SLAVE TESTING"

"SETUP BRIDGE FOR TARGET ABORT HANDLING TESTS"

Following testcases are for configuring parity checking during writes and reads through IMAGEs. Task location for testcases is: **run_tests –> parity_checking**.

"CONFIGURE BRIDGE FOR PARITY CHECKER FUNCTIONS TESTING"

"CLEARING PARITY ERROR STATUSES"

"CLEARANCE OF PARITY INTERRUPT STATUSES"

"DISABLE USED IMAGES"

Following testcases are for configuring transactions during writes and reads through IMAGEs. Task location for testcases is: **run_tests –> wb_to_pci_transactions**.

"BRIDGE CONFIGURATION FOR TRANSACTION TESTING"

"RECONFIGURE PCI MASTER/WISHBONE SLAVE"

"DISABLE_IMAGE"

3.4.1.2 Testcases for IMAGEs Tests

Following testcases are for testing normal writes and reads through IMAGEs. Task location for testcases is: **run_tests –> test_wb_image**.

"NORMAL SINGLE MEMORY WRITE THROUGH WB IMAGE TO PCI"

"NORMAL SINGLE MEMORY READ THROUGH WB IMAGE FROM PCI"

"NORMAL SINGLE MEMORY WRITE THROUGH WB IMAGE TO PCI WITH ADDRESS TRANSLATION"

"CAB MEMORY WRITE THROUGH WB SLAVE TO PCI"

"CAB MEMORY READ THROUGH WB SLAVE FROM PCI"

"CAB MEMORY READ THROUGH WB IMAGE WITH READ BURSTING ENABLED"

"SINGLE MEMORY READ THROUGH WB IMAGE WITH READ BURSTING ENABLED"

"I/O WRITE TRANSACTION FROM WB TO PCI TEST"

"I/O READ TRANSACTION FROM WB TO PCI TEST"

Following testcases are for testing erroneous WB writes and reads through IMAGEs. Task location for testcases is: **run_tests –> wb_slave_errors**.

"SINGLE ERRONEOUS MEMORY WRITE TO WB SLAVE"

"SINGLE ERRONEOUS MEMORY READ TO WB SLAVE"

"ERRONEOUS CAB MEMORY WRITE TO WB SLAVE"

"ERRONEOUS CAB MEMORY READ TO WB SLAVE"

"ERRONEOUS I/O WRITE TO WB SLAVE"

"ERRONEOUS I/O READ TO WB SLAVE"

"CAB I/O WRITE TO WB SLAVE"

"CAB I/O READ TO WB SLAVE"

"ERRONEOUS WB CONFIGURATION WRITE ACCESS"

"ERRONEOUS WB CONFIGURATION READ ACCESS"

"WB CAB CONFIGURATION WRITE ACCESS"

"WB CAB CONFIGURATION READ ACCESS"

Following testcases are for testing erroneous PCI writes and reads through IMAGEs. Task location for testcases is: **run_tests –> wb_to_pci_error_handling**.

"MASTER ABORT ERROR HANDLING DURING WB TO PCI WRITES"

"MASTER ABORT ERROR HANDLING FOR WB TO PCI READS"

"MASTER ABORT ERROR DURING CAB READ FROM WB TO PCI"

"TARGET ABORT ERROR ON SINGLE WRITE"

"TARGET ABORT ERROR ON CAB MEMORY WRITE"

"TARGET ABORT TERMINATION ON SECOND DATA PHASE OF BURST WRITE"

"TARGET ABORT DURING SINGLE MEMORY READ"

"TARGET ABORT ERROR DURING SECOND DATAPHASE OF BURST READ"

"TARGET ABORT ERROR DURING LAST DATAPHASE OF BURST READ"

"ERROR REPORTING FUNCTIONALITY FOR I/O WRITE"

Following testcases are for testing parity during PCI writes and reads through IMAGEs. Task location for testcases is: **run_tests –> parity_checking**.

"RESPONSE TO TARGET ASSERTING PERR DURING MASTER WRITE"

"RESPONSE TO TARGET ASSERTING PERR DURING MASTER WRITE WITH PARITY ERROR RESPONSE ENABLED"

"MASTER WRITE WITH NO PARITY ERRORS"

"BRIDGE'S RESPONSE TO PARITY ERRORS DURING MASTER READS"

"NO PERR ASSERTION ON MASTER READ WITH PAR. ERR. RESPONSE DISABLED"

"MASTER READ TRANSACTION WITH NO PARITY ERRORS"

"NO SERR ASSERTION AFTER ADDRESS PARITY ERROR, SERR DISABLED AND PAR. ERR. RESPONSE ENABLED"

"ADDRESS PARITY ERROR ON FIRST DATA PHASE OF DUAL ADDRESS CYCLE - SERR DISABLED, PAR. RESP. ENABLED"

"ADDRESS PARITY ERROR ON SECOND DATA PHASE OF DUAL ADDRESS CYCLE - SERR DISABLED, PAR. RESP. ENABLED"

"ADDRESS PARITY ERROR ON BOTH DATA PHASES OF DUAL ADDRESS CYCLE - SERR DISABLED, PAR. RESP. ENABLED"

"ADDRESS PARITY ERROR RESPONSE WITH SERR AND PARITY ERROR RESPONSE ENABLED"

"ADDRESS PARITY ERROR ON FIRST DATA PHASE OF DUAL ADDRESS CYCLE - SERR ENABLED, PAR. RESP. ENABLED"

"ADDRESS PARITY ERROR ON SECOND DATA PHASE OF DUAL ADDRESS CYCLE - SERR ENABLED, PAR. RESP. ENABLED"

"ADDRESS PARITY ERROR ON BOTH DATA PHASES OF DUAL ADDRESS CYCLE - SERR ENABLED, PAR. RESP. ENABLED"

"NO SERR ASSERTION ON ADDRESS PARITY ERROR WITH SERR ENABLED AND PAR. ERR. RESPONSE DISABLED"

"ADDRESS PARITY ERROR ON FIRST DATA PHASE OF DUAL ADDRESS CYCLE - SERR ENABLED, PAR. RESP. DISABLED"

"ADDRESS PARITY ERROR ON SECOND DATA PHASE OF DUAL ADDRESS CYCLE - SERR ENABLED, PAR. RESP. DISABLED"

"ADDRESS PARITY ERROR ON BOTH DATA PHASES OF DUAL ADDRESS CYCLE - SERR ENABLED, PAR. RESP. DISABLED"

"EXTERNAL WRITE WITH NO PARITY ERRORS"

"INVALID PAR ON WRITE REFERENCE THROUGH BRIDGE'S TARGET - PERR RESPONSE ENABLED"

"PARITY ERROR HANDLING ON TARGET READ REFERENCE"

Following testcases are for testing transactions during writes and reads through IMAGEs. Task location for testcases is: **run_tests –> wb_to_pci_transactions**.

"SINGLE POSTED WRITE TRANSACTION PROCESSING ON PCI"

"SINGLE POSTED WRITE FROM WISHBONE TO PCI RETRIED FIRST TIME"

"SINGLE POSTED WRITE FROM WISHBONE TO PCI RETRIED SECOND TIME"

"SINGLE POSTED WRITE FROM WISHBONE TO PCI DISCONNECTED"

"BURST LENGTH 2 POSTED WRITE TRANSACTION PROCESSING ON PCI"

"BURST LENGTH 2 POSTED WRITE STARTS WITH RETRY"

"BURST LENGTH 2 POSTED WRITE RETRIED SECOND TIME DISCONNECTED ON FIRST DATAPHASE"

"BURST LENGTH 2 POSTED WRITE NORMAL COMPLETION AFTER DISCONNECT "

"BURST LENGTH 2 POSTED WRITE DISCONNECTED AFTER FIRST DATAPHASE FIRST TIME"

"BURST LENGTH 2 POSTED WRITE DISCONNECTED WITH SECOND DATAPHASE SECOND TIME"

"BURST LENGTH 2 POSTED WRITE TRANSACTION PROCESSING ON PCI WITH NORMAL COMPLETION"

"BURST LENGTH 3 POSTED WRITE TRANSACTION PROCESSING ON PCI"

"BURST LENGTH 3 POSTED WRITE TRANSACTION DISCONNECT ON FIRST DATAPHASE FIRST TIME"

"BURST LENGTH 3 POSTED WRITE TRANSACTION DISCONNECT ON SECOND DATAPHASE SECOND TIME"

"BURST LENGTH 3 POSTED WRITE DISCONNECTED ON SECOND FIRST TIME"

"BURST LENGTH 3 POSTED WRITE DISCONNECTED ON FIRST SECOND TIME"

"BURST LENGTH 3 POSTED WRITE TRANSACTION WITH NORMAL COMPLETION"

"BURST LENGTH OF WISHBONE FIFO DEPTH POSTED MEMORY WRITE"

"FULL WRITE FIFO BURST RETRIED FIRST TIME"

"FULL WRITE FIFO BURST DISCONNECT WITH FIRST SECOND TIME"

"FULL WRITE FIFO BURST DISCONNECT AFTER FIRST THIRD TIME"

"FULL WRITE FIFO BURST DISCONNECT WITH SECOND FOURTH TIME"

"REMAINDER OF FULL WRITE FIFO BURST NORMAL COMPLETION FIFTH TIME"

"READ DATA BURSTED TO TARGET BACK AND CHECK VALUES"

"SINGLE READ TRANSACTION PROCESSING ON PCI"

"SINGLE MEMORY READ RETRIED FIRST TIME"

"SINGLE MEMORY READ DISCONNECTED WITH FIRST SECOND TIME"

"FILL TARGET MEMORY WITH DATA"

"SINGLE READ TO PUSH WRITE DATA FROM FIFO"

"BURST READ WITH DISCONNECT ON FIRST"

"BURST READ WITH DISCONNECT AFTER FIRST"

"BURST READ WITH DISCONNECT ON SECOND - TAKE OUT ONLY ONE"

"BURST READ WITH NORMAL TERMINATION"

"NORMAL BURST READ WITH NORMAL COMPLETION, MEMORY READ LINE DISABLED, PREFETCH ENABLED, BURST SIZE 4"

"SINGLE READ WITH FUNNY BYTE ENABLE COMBINATION"

"BURST READ WITH NORMAL COMPLETION FILLING FULL FIFO - MRL AND PREFETCH BOTH ENABLED"

"SINGLE CAB READ FOR FLUSHING STALE READ DATA FROM FIFO"

"WB BURST READ WHEN CACHE LINE SIZE VALUE IS INVALID"

"WB BURST READ WHEN CACHE LINE SIZE VALUE IS ZERO"

"LATENCY TIMER OPERATION ON PCI MASTER WRITE"

"BURST WRITE DATA DISCONNECTED BY LATENCY TIMEOUT"

"LATENCY TIMER OPERATION DURING MASTER READ"

Following testcases are for testing interrupt acknowledge cycles. Task location for testcases is: **run_tests –> iack_cycle**.

"INTERRUPT ACKNOWLEDGE CYCLE GENERATION WITH MASTER ABORT"

"INTERRUPT ACKNOWLEDGE CYCLE GENERATION WITH NORMAL COMPLETION"

"INTERRUPT ACKNOWLEDGE CYCLE GENERATION WITH NORMAL COMPLETION AND FUNNY BYTE ENABLES"

3.4.1.3 Testcases for Verification

Following testcase is for testing normal writes and reads through IMAGEs. Task location for testcase is: **run_tests –> test_wb_image**.

"CHECK MAXIMUM IMAGE SIZE"

Following testcases are for testing erroneous PCI writes and reads through IMAGEs. Task location for testcases is: **run_tests –> wb_to_pci_error_handling**.

"CHECKING ERROR REPORTING FUNCTIONS AFTER MASTER ABORT ERROR"

"CHECKING INTERRUPT REQUESTS AFTER MASTER ABORT ERROR"

"CHECKING PCI DEVICE STATUS REGISTER VALUE AFTER MASTER ABORT"

"CHECKING MASTER ABORT ERROR HANDLING ON CAB MEMORY WRITE"

"CHECKING ERROR REPORTING REGISTERS' VALUES AFTER MASTER ABORT ERROR"

"CHECKING INTERRUPT REQUESTS AFTER CLEARING THEM"

"CHECK NORMAL WRITING/READING FROM WISHBONE TO PCI AFTER ERRORS WERE PRESENTED"

"CHECKING ERROR STATUS AFTER MASTER ABORT ON READ"

"CHECK PCI DEVICE STATUS REGISTER VALUE AFTER TARGET ABORT ON DELAYED READ"

"CHECK PCI DEVICE STATUS REGISTER AFTER READ TERMINATED WITH MASTER ABORT"

"NORMAL SINGLE MEMORY WRITE IMMEDIATELY AFTER ONE TERMINATED WITH TARGET ABORT"

"NORMAL SINGLE MEMORY READ AFTER WRITE TERMINATED WITH TARGET ABORT"

"WB ERROR CONTROL AND STATUS REGISTER VALUE CHECK AFTER WRITE TARGET ABORT"

"INTERRUPT STATUS REGISTER VALUE CHECK AFTER WRITE TARGET ABORT"

"PCI DEVICE STATUS REGISTER VALUE CHECK AFTER WRITE TARGET ABORT"

"WB ERROR CONTROL AND STATUS REGISTER VALUE CHECK AFTER WRITE TARGET ABORT"

"WB ERRONEOUS ADDRESS AND DATA REGISTERS' VALUES CHECK AFTER WRITE TARGET ABORT"

"PCI DEVICE STATUS VALUE CHECK AFTER WRITE TARGET ABORT"

"WB ERROR CONTROL AND STATUS REGISTER VALUE CHECK AFTER CLEARING ERROR STATUS"

"INTERRUPT REQUEST ASSERTION AFTER ERROR REPORTING TRIGGER"

"PCI DEVICE STATUS REGISTER VALUE CHECK AFTER TARGET ABORTED MEMORY WRITE"

"INTERRUPT STATUS REGISTER VALUE CHECK AFTER CLEARING STATUS BITS"

"WB ERROR STATUS REGISTER VALUE CHECK AFTER CLEARING STATUS BIT"

"WB ERROR STATUS REGISTER VALUE CHECK AFTER READ TERMINATED WITH TARGET ABORT"

"PCI DEVICE STATUS REGISTER VALUE CHECK AFTER READ TERMINATED WITH TARGET ABORT"

"INTERRUPT STATUS REGISTER VALUE CHECK AFTER READ TERMINATED WITH TARGET ABORT"

"CHECK NORMAL BURST READ AFTER TARGET ABORT TERMINATED BURST READ"

"WB ERROR STATUS REGISTER VALUE AFTER MASTER ABORTED I/O WRITE"

"WB ERRONEOUS ADDRESS AND DATA REGISTERS' VALUES AFTER MASTER ABORTED I/O WRITE"

"INTERRUPT STATUS REGISTER VALUE AFTER MASTER ABORTED I/O WRITE"

"PCI DEVICE STATUS REGISTER VALUE AFTER MASTER ABORTED I/O WRITE"

Following testcases are for testing parity during PCI writes and reads through IMAGEs. Task location for testcases is: **run_tests –> parity_checking**.

"CHECK PCI DEVICE STATUS REGISTER VALUE AFTER PARITY ERROR DURING MASTER WRITE"

"PCI DEVICE STATUS REGISTER VALUE AFTER PARITY ERROR DURING MASTER WRITE - PAR. ERR. RESPONSE ENABLED"

"PCI DEVICE STATUS REGISTER VALUE AFTER NORMAL MEMORY WRITE"

"INTERRUPT REQUEST ASSERTION AFTER PARITY ERROR"

"PCI DEVICE STATUS REGISTER VALUE AFTER MASTER READ PARITY ERROR"

"INTERRUPT STATUS REGISTER AFTER MASTER READ PARITY ERROR"

"INTERRUPT REQUEST CHECK AFTER READ PARITY ERROR WITH PARITY ERROR RESPONSE DISABLED"

"INTERRUPT STATUS REGISTER VALUE AFTER MASTER READ PARITY ERROR WITH PAR. ERR. RESPONSE DISABLED"

"PCI DEVICE STATUS REGISTER VALUE AFTER NORMAL READ"

"INTERRUPT STATUS REGISTER VALUE AFTER NORMAL READ"

"PCI DEVICE STATUS REGISTER VALUE AFTER ADDRESS PARITY ERROR AND SERR DISABLED"

"INTERRUPT REQUEST AFTER ADDRESS PARITY ERROR"

"INTERRUPT REQUEST AFTER ADDRESS PARITY ERROR WAS PRESENTED ON PCI"

"PCI DEVICE STATUS REGISTER VALUE AFTER ADDRESS PARITY ERROR ON PCI"

"INTERRUPT STATUS REGISTER VALUE AFTER ADDRESS PARITY ERROR ON PCI"

"INTERRUPT REQUEST AFTER ADDR. PARITY ERROR WITH PERR RESPONSE DISABLED"

"PCI DEVICE STATUS REGISTER VALUE AFTER ADDR PERR WITH PERR RESPONSE DISABLED"

"INTERRUPT STATUS REGISTER VALUE AFTER ADDR PERR WITH PERR RESPONSE DISABLED"

"PCI DEVICE STATUS REGISTER VALUE AFTER ADDRESS PARITY ERROR AND PERR DISABLED"

"INTERRUPT REQUESTS AFTER NORMAL EXTERNAL MASTER WRITE"

"PCI DEVICE STATUS REGISTER VALUE AFTER NORMAL EXTERNAL MASTER WRITE"

"INTERRUPT STATUS REGISTER VALUE AFTER NORMAL EXTERNAL MASTER WRITE"

"INTERRUPT REQUESTS AFTER TARGET WRITE REFERENCE PARITY ERROR"

"PCI DEVICE STATUS REGISTER VALUE AFTER PARITY ERROR ON TARGET REFERENCE"

"INTERRUPT STATUS REGISTER VALUE AFTER PARITY ERROR ON TARGET REFERENCE"

"INTERRUPT REQUESTS AFTER PERR ON READ REFERENCE THROUGH BRIDGE'S TARGET"

"PCI DEVICE STATUS REGISTER VALUE AFTER PARITY ERROR ON TARGET READ REFERENCE"

"INTERRUPT STATUS REGISTER VALUE AFTER PARITY ERROR ON TARGET READ REFERENCE"

3.4.2 Description of PCIU Testcases

Before running testcases for PCIU, some initialization must be done.

With task **configure_bridge_target_base_addresses**, PCI Bridge Target is initialized (master and target are enabled, all implemented base address registers are set and enabled).

3.4.2.1 Testcases for Configuration

Testcases described here are fundamental for all tests to pass successful. They set the configuration of the PCIU of the PCI Bridge, but they don't test PCI to WB accesses. Because of that, they are not listed in the **pci_tb.log** file.

Following testcases are for configuring normal writes and reads through MEMORY IMAGEs. Task location for testcases is: **run_tests –> test_pci_image –> test_normal_wr_rd**.

"PCI IMAGE SETTINGS"

"CONFIGURE BRIDGE FOR NORMAL TARGET READ/WRITE"

Following testcases are for configuring erroneous WB writes and reads through MEMORY IMAGEs. Task location for testcases is: **run_tests –> test_pci_image –> test_wb_error_wr**.

"CONFIGURE BRIDGE FOR ERROR TERMINATED WRITES THROUGH PCI TARGET UNIT"

"CLEARING STATUS BITS AFTER WRITE TERMINATED WITH ERROR ON WB"

"INTERRUPT REQUEST FINISHED AFTER PCI ERROR INTERRUPT STATUS IS CLEARED"

"CONFIGURE BRIDGE FOR ERROR TERMINATED READS THROUGH PCI TARGET UNIT"

"DISABLE IMAGE"

Following testcases are for configuring normal writes and reads through I/O IMAGEs. Task location for testcases is: **run_tests –> test_pci_image**.

"PCI IMAGE SETTINGS"

"ENABLE/DISABLE ADDRESS TRANSLATION"

Following testcases are for configuring erroneous writes and reads through I/O IMAGEs. Task location for testcases is: **run_tests –> test_wb_error_rd**.

"ENABLE/DISABLE ADDRESS TRANSLATION"

"ENABLE/DISABLE ERROR REPORTING"

"ENABLE/DISABLE PCI ERROR INTERRUPTS"

"CLEAR ERROR STATUS"

"CLEARING INTERRUPT STATUS"

"CONFIGURE BRIDGE FOR ERROR TERMINATED READS THROUGH PCI TARGET UNIT"

"DISABLE IMAGE"

Following testcases are for configuring fast back-to-back writes and reads through IMAGEs. Task location for testcases is: **run_tests –> target_fast_back_to_back**.

"CONFIGURE TARGET FOR FAST B2B TESTING"

"DISABLING MEM IMAGE"

"DISABLING IO IMAGE"

Following testcases are for configuring target disconnects on PCI writes and reads through IMAGEs. Task location for testcases is: **run_tests –> target_disconnects**.

"CONFIGURE TARGET FOR DISCONNECT TESTING"

"DISABLING MEMORY IMAGE"

"DISABLING IO IMAGE"

Following testcases are for configuring target aborts on PCI writes and reads through I/O IMAGEs. Task location for testcases is: **run_tests –> test_target_abort**.

"CONFIGURE TARGET FOR TARGET ABORT TESTING"

"DISABLE IMAGE"

Following testcases are for configuring transaction ordering during writes and reads through WBU and PCIU simultaneously. Task location for testcase is: **run_tests –> transaction_ordering**.

"BRIDGE CONFIGURATION FOR TRANSACTION ORDERING TESTS"

3.4.2.2 Testcases for IMAGEs Tests

Following testcases are for testing normal writes and reads through MEMORY IMAGEs (singles and bursts). Task location for testcases is: **run_tests –> test_pci_image –> test_normal_wr_rd**.

"NORMAL POSTED WRITE THROUGH PCI TARGET UNIT"

"NORMAL MEMORY READ THROUGH PCI TARGET UNIT"

Following testcases are for testing erroneous WB writes and reads through MEMORY IMAGEs (singles and bursts). Task location for testcases is: **run_tests –> test_pci_image –> test_wb_error_wr**.

"POSTED WRITE THROUGH PCI TARGET ERROR TERMINATION ON WB ON FIRST TRANSFER"

"POSTED WRITE THROUGH PCI TARGET ERROR TERMINATION ON WB ONE BEFORE LAST TRANSFER"

"POSTED WRITE THROUGH PCI TARGET ERROR TERMINATION ON WB ON LAST TRANSFER"

"SINGLE READ THROUGH PCI TARGET TERMINATED WITH ERROR ON WISHBONE"

"BURST READ THROUGH PCI TARGET TERMINATED WITH ERROR ON WISHBONE ON FIRST DATAPHASE"

"BURST READ THROUGH PCI TARGET TERMINATED WITH ERROR ON WISHBONE ON SECOND DATAPHASE"

"BURST READ THROUGH PCI TARGET TERMINATED WITH ERROR ON WISHBONE ON LAST DATAPHASE"

"BURST READ THROUGH PCI TARGET TERMINATED WITH ERROR ON WISHBONE ON ONE BEFORE LAST DATAPHASE"

"FULL FIFO BURST READ THROUGH PCI TARGET TERMINATED WITH ERROR ON WISHBONE ON LAST DATAPHASE"

"FULL FIFO BURST READ THROUGH PCI TARGET TERMINATED WITH ERROR ON WISHBONE BEFORE LAST DATAPHASE"

"BURST READ THROUGH PCI TARGET TERMINATED WITH ERROR ON WISHBONE, ERROR NOT PULLED OUT ON PCI"

Following testcases are for testing normal writes and reads through I/O IMAGEs. Task location for testcases is: **run_tests –> test_pci_image –> test_target_io_wr_rd**.

"BYTE ADDRESSABLE WRITES THROUGH TARGET IO IMAGE"

"READ BY WORDS IO DATA PREVIOUSLY WRITTEN BY BYTES"

Following testcases are for testing erroneous writes and reads through I/O IMAGEs. Task location for testcases is: **run_tests –> test_pci_image –> test_target_io_err_wr**.

"POST IO WRITE THAT WILL BE TERMINATED WITH ERROR ON WISHBONE"

"SINGLE I/O READ THROUGH PCI TARGET TERMINATED WITH ERROR ON WISHBONE"

Following testcases are for testing NO target response (master abort) on unsupported PCI bus commands. Task location for testcases is: **run_tests –> test_pci_image –> target_unsupported_cmds**.

"MASTER ABORT WHEN ACCESSING TARGET WITH UNSUPPORTED BUS COMMAND - IACK"

"MASTER ABORT WHEN ACCESSING TARGET WITH UNSUPPORTED BUS COMMAND - SPECIAL"

"MASTER ABORT WHEN ACCESSING TARGET WITH UNSUPPORTED BUS COMMAND - RESERVED0"

"MASTER ABORT WHEN ACCESSING TARGET WITH UNSUPPORTED BUS COMMAND - RESERVED1"

"MASTER ABORT WHEN ACCESSING TARGET WITH UNSUPPORTED BUS COMMAND - RESERVED2"

"MASTER ABORT WHEN ACCESSING TARGET WITH UNSUPPORTED BUS COMMAND - RESERVED3"

"MASTER ABORT WHEN ACCESSING TARGET WITH UNSUPPORTED BUS COMMAND - DUAL_ADDR_CYC"

Following testcases are for testing fast back-to-back writes and reads through IMAGEs. Task location for testcases is: **run_tests –> target_fast_back_to_back**.

"FAST BACK TO BACK THROUGH TARGET - FILL WRITE FIFO, CHECK RETRY ON FAST B2B WRITE"

"FAST BACK TO BACK THROUGH TARGET - BOTH WRITES SMALL ENOUGH TO PROCEEDE THROUGH FIFO"

"FAST BACK TO BACK THROUGH TARGET - FIRST WRITE FULL FIFO, THEN READ BACK"

"FAST BACK TO BACK THROUGH TARGET - TWO SINGLE IO WRITES"

"FAST BACK TO BACK THROUGH TARGET - FIRST I/O WRITE, THEN READ BACK"

Following testcases are for testing target disconnects on PCI writes and reads through IMAGEs. Task location for testcases is: **run_tests –> target_disconnects**.

"TARGET DISCONNECT ON BURST WRITE TO CONFIGURATION SPACE"

"TARGET DISCONNECT ON BURST READ FROM CONFIGURATION SPACE"

"TARGET DISCONNECT WHEN WRITE FIFO FILLED DURING BURST WRITE"

"TARGET DISCONNECT WHEN READ FIFO IS EMPTIED DURING BURST READ"

"TARGET DISCONNECT ON WRITES WITH UNSUPPORTED WRAPING MODES"

"TARGET DISCONNECT ON READS WITH UNSUPPORTED WRAPING MODES"

"TARGET DISCONNECT ON BURST WRITE TO IO SPACE"

"TARGET DISCONNECT ON BURST READ TO IO SPACE"

Following testcases are for testing target aborts on PCI writes and reads through I/O IMAGEs. Task location for testcase is: **run_tests –> test_target_abort**.

"TARGET ABORT SIGNALING ON I/O ACCESSES WITH INVALID ADDRESS/BYTE ENABLE COMBINATION"

Following testcases are for testing transaction ordering during writes and reads through WBU and PCIU simultaneously. Task location for testcases is: **run_tests –> transaction_ordering**.

"SIMULTANEOUS WRITE REFERENCE TO WB SLAVE AND PCI TARGET"

"SIMULTANEOUS WRITE REFERENCE TO PCI TARGET AND WB SLAVE"

"SIMULTANEOUS MULTI BEAT WRITES THROUGH WB SLAVE AND PCI TARGET"

"SIMULTANEOUS MULTI BEAT WRITE REFERENCE TO PCI TARGET AND WB SLAVE"

"ORDERING OF TRANSACTIONS MOVING IN SAME DIRECTION - WRITE THROUGH WB SLAVE, READ THROUGH PCI TARGET"

"ORDERING OF TRANSACTIONS MOVING IN SAME DIRECTION - WRITE THROUGH PCI TARGET, READ THROUGH WB SLAVE"

3.4.2.3 Testcases for Verification

Following testcase is for testing normal writes and reads through IMAGEs. Task location for testcase is: **run_tests –> test_pci_image –> test_normal_wr_rd**.

"PCI ERROR STATUS AFTER NORMAL WRITE/READ"

Following testcases are for testing erroneous PCI writes and reads through MEMORY IMAGEs. Task location for testcases is: **run_tests –> test_pci_image –> test_wb_error_wr**.

"PCI ERROR CONTROL AND STATUS REGISTER VALUE AFTER WRITE TERMINATED WITH ERROR"

"PCI ERRONEOUS DATA REGISTER VALUE AFTER WRITE TERMINATED WITH ERROR ON WB"

"PCI ERRONEOUS ADDRESS REGISTER VALUE AFTER WRITE TERMINATED WITH ERROR ON WB"

"INTERRUPT ASSERTION AND STATUS AFTER WRITE TERMINATED WITH ERROR ON WB"

"PCI DEVICE STATUS REGISTER VALUE AFTER TARGET ABORT"

"PCI ERROR CONTROL AND STATUS REGISTER VALUE AFTER READ THROUGH TARGET TERMINATED WITH TARGET ABORT"

"ALL PCI DEVICE STATUSES AFTER ERRONEOUS TARGET READS"

Following testcases are for testing erroneous writes and reads through I/O IMAGEs. Task location for testcase is: **run_tests –> test_pci_image –> test_target_io_err_wr**.

"INTERRUPT REQUEST ASSERTION AFTER ERROR TERMINATED WRITE ON WISHBONE"

"ERROR STATUS REGISTER VALUE CHECK AFTER ERROR TERMINATED POSTED IO WRITE ON WISHBONE"

"ERRONEOUS ADDRESS AND DATA REGISTERS' VALUES CHECK AFTER WRITE TERMINATED WITH ERROR ON WISHBONE"

"INTERRUPT STATUS REGISTER VALUE AFTER ERROR TERMINATED WRITE ON WISHBONE"

"INTERRUPT REQUEST DEASSERTION AFTER CLEARING INTERRUPT STATUS"

"PCI DEVICE STATUS REGISTER VALUE AFTER TARGET ABORT"

"PCI ERROR CONTROL AND STATUS REGISTER VALUE AFTER READ THROUGH TARGET TERMINATED WITH TARGET ABORT"

Following testcases are for testing target aborts on PCI writes and reads through I/O IMAGEs. Task location for testcase is: **run_tests –> test_target_abort**.

"PCI DEVICE STATUS REGISTER VALUE AFTER TARGET ABORT"

"ERROR CONTROL AND STATUS REGISTER VALUE CHECK AFTER TARGET ABORTS"

3.5 Testbench Constants

There are three files with constants for whole testbench. Two of them are for behavioral models of Blue Beaver, **pci_blue_options.vh** and **pci_blue_constants.vh**. There were some additional lines added into this two files so we can use all features needed for testing PCI Bridge. The third file, for testbench and other behavioral models, is **pci_testbench_defines.v**.

All constants in all three files are set to test "all" possible combinations on WB and/or PCI buses. So none of those constants should be changed (see chapter 3.5.2 Unchangeable Testbench Constants), except those mentioned below (see chapter 3.5.1 Changeable Testbench Constants). All constants described are from **pci_testbench_defines.v** file.

3.5.1 Changeable Testbench Constants

Constants written in a **pci_testbench_defines.v** file under section 'Changeable testbench defines (constants)' are described herein.

Following define affects the execution of the testbench:

- STOP_ON_FAILURE – if defined, testbench will stop when a testcase fails.

Value in the following define affects a ratio of WB to PCI clocks:

- WB_FREQ – number defined here defines WB clock frequency in GHz (e.g. 0.025 = 25 MHz). PCI clock is fixed (33 MHz).

Following constants can be changed, but must be very carefully set. IMAGEs are NOT allowed to interleave on a PCI bus. See PCI IP Core Specification document for more details on how to set IMAGEs.

Values in following defines define base addresses of implemented PCI IP Core IMAGEs:

- TAR0_BASE_ADDR_0, TAR0_BASE_ADDR_1, TAR0_BASE_ADDR_2, TAR0_BASE_ADDR_3, TAR0_BASE_ADDR_4, TAR0_BASE_ADDR_5 – numbers defined here define 32 bit values for each PCI IMAGE's base address. They are written into PCI Bridge during initialization.

  Which base addresses are implemented and the number of valid MSBits depend on changeable constants (NO_CNF_IMAGE, PCI_IMAGE0, PCI_IMAGE2 – 5 and PCI_NUM_OF_DEC_ADDR_LINES) described in a chapter 2.4.7 PCI images' implementation constants.

Values in following defines define address masks of implemented PCI IP Core IMAGEs:

- TAR0_ADDR_MASK_0, TAR0_ADDR_MASK_1, TAR0_ADDR_MASK_2, TAR0_ADDR_MASK_3, TAR0_ADDR_MASK_4, TAR0_ADDR_MASK_5 – numbers defined here define 32 bit values for each PCI IMAGE's address mask. This defines the size of each IMAGE. They are written into PCI Bridge during initialization.

Which address masks are implemented and the number of valid MSBits depend on changeable constants (NO_CNF_IMAGE, PCI_IMAGE0, PCI_IMAGE2 – 5 and PCI_NUM_OF_DEC_ADDR_LINES) described in a chapter 2.4.7 PCI images' implementation constants.

Values in following defines define translation addresses of implemented PCI IP Core IMAGEs:

- TAR0_TRAN_ADDR_0, TAR0_TRAN_ADDR_1, TAR0_TRAN_ADDR_2, TAR0_TRAN_ADDR_3, TAR0_TRAN_ADDR_4, TAR0_TRAN_ADDR_5 – numbers defined here define 32 bit values for each PCI IMAGE's translation address. This defines how base address is translated from PCI to WB side. They are written into PCI Bridge during initialization.

  Which translation addresses are implemented and the number of valid MSBits depend on changeable constants (NO_CNF_IMAGE, PCI_IMAGE0, PCI_IMAGE2 – 5 and PCI_NUM_OF_DEC_ADDR_LINES) described in a chapter 2.4.7 PCI images' implementation constants.

Values in following defines define starting and ending base addresses of behavioral PCI target IMAGEs:

- BEH_TAR1_MEM_START, BEH_TAR1_MEM_END – numbers defined here define 32 bit values for starting and ending base addresses of PCI MEMORY IMAGE. They are written into behavioral PCI device 1 during initialization.
- BEH_TAR1_IO_START, BEH_TAR1_IO_END – numbers defined here define 32 bit values for starting and ending base addresses of PCI IO IMAGE. They are written into behavioral PCI device 1 during initialization.
- BEH_TAR2_MEM_START, BEH_TAR2_MEM_END – numbers defined here define 32 bit values for starting and ending base addresses of PCI MEMORY IMAGE. They are written into behavioral PCI device 2 during initialization.
- BEH_TAR2_IO_START, BEH_TAR2_IO_END – numbers defined here define 32 bit values for starting and ending base addresses of PCI IO IMAGE. They are written into behavioral PCI device 2 during initialization.

3.5.2 Unchangeable Testbench Constants

Constants written in a **pci_testbench_defines.v** file under section 'User-unchangeable testbench defines (constants)' are mentioned herein. Basic descriptions can be found in a file where they are defined.

Index

This section contains an alphabetical list of helpful document entries with their corresponding page numbers.