PLB-to-WB Bridge Specification

Author: Christian Haettich
feddischson@opencores.org

Rev. 0.1
August 14, 2010

Revision History

Ref.	Date	Author	Description
0.1	08.14.2010	Christian Haettich	First Draft

Contents

Nomenclature

List of Abbreviations

BFC	Bus Functional Compiler
BFL	Bus Functional Language
PLBFM	Processor Local Bus Functional Model
EDK	Embedded Development Kit
FPGA	Field-Programmable Gate Array
IP	Intelligent Peripheral
PLB2WB	PLB-to-WB
PLB	Processor Local Bus
SoC	Systems on Chip
WB	Wishbone Bus
XPS	Xilinx Platform Studio

1. Introduction

The intention of the project is the development of a bus bridge, which enables the usage of WB compliant IP cores in a system, which uses the PLB as system and peripheral bus. Figure 1 shows the intended system, which contains a PLB and WB. The **PLB-to-WB**

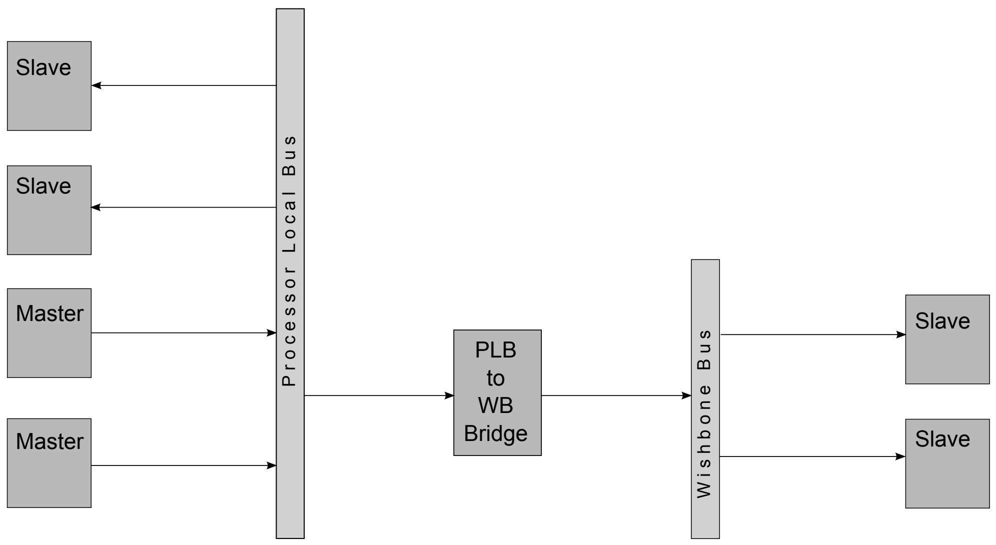


Figure 1.1.: System with a PLB and WB, connected via PLB2WB-Bridge.

(PLB2WB) Bridge enables the access to slaves on the WB side for masters on the PLB side. Such a bus bridge enables the usage of free[1] IP cores together with the proprietary MicroBlaze.

[1]Mostly distributed under LGPL licence.

1.1. Features

- separate clock domains for PLB and WB
- separate resets for PLB and WB possible
- PLB address pipelining (optional)
- PLB fixed length burst transfers (only words, optional)
- PLB line transfers (optional)
- WB B.3 classic cycles (block and single, block cycles are optional)
- flexible address offset
- handling of delayed write errors on WB side
- transfers interrupts to PLB side

2. IO Ports and Generics

2.1. Generics

Name	Description
C_BASEADDR	PLB base address
C_HIGHADDR	PLB high address
C_SPLB_AWIDTH	PLB address bus width
C_STATUS_BASEADDR	PLB base address of status registers
C_STATUS_HIGHADDR	PLB base address of status registers
C_SPLB_DWIDTH	PLB data bus width
C_SPLB_NUM_MASTERS	PLB Number of masters
C_SPLB_MID_WIDTH	Master ID bus width
C_SPLB_NATIVE_DWIDTH	Internal native data bus width
C_SPLB_SUPPORT_BUR_LINE	Defines if PLB burst and line transfers are supported
C_SPLB_SUPPORT_ADR_PIPE	Defines if PLB address pipelining is supported
WB_ADR_OFFSET	Address offset: is added to every address on WB side
WB_ADR_OFFSET_NEG	Defines if WB_ADR_OFFSET is added or subtracted
WB_PIC_INTS	Number of WB interrupt lines
WB_PIC_INT_LEVEL	Interrupts are active high or acrive low
WB_SUPPORT_BLOCK	Defines if WB block transfers are supported
WB_DAT_W	WB data bus width
WB_ADR_W	WB address bus width
WB_TIMEOUT_CYCLES	Watchdog timer cycles

2.2. PLB signals

Name	Description
SPLB_Clk	bus clock
SPLB_Rst	bus reset
PLB_ABus	address bus
PLB_UABus	upper address bus
PLB_PAValid	primary address valid indicator
PLB_SAValid	secondary address valid indicator
PLB_rdPrim	secondary to primary read request indicator
PLB_wrPrim	secondary to primary write request indicator
PLB_masterID	current master identifier
PLB_abort	abort request indicator
PLB_busLock	bus lock
PLB_RNW	read/not write
PLB_BE	byte enables
PLB_MSize	master data bus size
PLB_size	transfer size
PLB_type	transfer type
PLB_lockErr	lock error indicator
PLB_wrDBus	write data bus
PLB_wrBurst	burst write transfer indicator
PLB_rdBurst	burst read transfer indicator
PLB_wrPendReq	write pending bus request indicator
PLB_rdPendReq	read pending bus request indicator
PLB_wrPendPri	write pending request priority
PLB_rdPendPri	read pending request priority
PLB_reqPri	current request priority
PLB_TAttribute	transfer attribute
Sl_addrAck	slave address acknowledge
Sl_SSize	slave data bus size
Sl_wait	slave wait indicator
Sl_wrDAck	slave write data acknowledge

Sl_rearbitrate	slave re-arbitrate bus indicator
Sl_wrComp	slave write transfer complete indicator
Sl_wrBTerm	slave terminate write burst transfer
Sl_rdDBus	slave read data bus
Sl_rdWdAddr	slave read word address
Sl_rdDAck	slave read data acknowledge
Sl_rdComp	slave read transfer complete indicator
Sl_rdBTerm	slave terminate read burst transfer
Sl_MBusy	slave busy indicator
Sl_MWrErr	slave write error indicator
Sl_MRdErr	slave read error indicator
Sl_MIRQ	slave bus interrupt indicator (not used by xilinx)

2.3. WB signals

Name	Description
wb_clk_i	bus clock
wb_rst_i	bus reset
wb_dat_i	read data bus
wb_dat_o	write data bus
wb_adr_o	address bus
wb_sel_o	byte enables
wb_we_o	write enable ('0' when read)
wb_cyc_o	bus cycle indicator
wb_stb_o	strobe output
wb_ack_i	acknowledge input
wb_err_i	error input
wb_rty_i	retry input
wb_lock_o	bus lock

2.4. Other signals

Name	Description
PLB2WB_IRQ	slave interrupt out (PLB size)
wb_pic_int_i	interrupt input (WB side)

3. Registers

The PLB2WB-Bridge contains four status registers, which are listed in Table 3.1.

Name	Address	Width	Access	Description
WB_STAT	Base + 0x0	32 bit	R/W	Read access: WB status Write access: clear Interrupt
WB_DAT	Base + 0x4	32 bit	R/W	Read access: contains datum of failed write transfer Write access: retry and continue WB transfer
WB_ADR	Base + 0x8	32 bit	R/W	Read access: contains address of failed write transfer Write access: abort WB transfer
WB_IRQ	Base + 0xc	32 bit	R/W	Read access: WB interrupt source Write access: soft reset

Table 3.1.: List of all software accessible status registers

WB_STAT

Writing any value to this register clears the PLB2WB-Interrupt. Reading from this register returns a 32-bit value:

Bit 0	1	2	3 .. 31
WB interrupt	WB reset	WB write error	reserved

WB_DAT

If a WB write error occurs, this register provides the datum which failed to write. Writing any value to this register causes the bridge to retry and continue a failed WB write transfer.

WB_ADR

If a WB write error occurs, this register provides the address (without offset) of the datum which failed to write. Writing any value to this register causes the bridge to abort a failed WB write transfer.

WB_IRQ

If a WB interrupt occurs, this register holds the information about the periphery, which generated the interrupt. Writing any value to this register causes a soft reset.

4. Architecture

From the PLB side, the bridge looks like a slave device. On the WB side, the bridge implements a master device. Figure 4.1 shows the bridge with its main components. Because

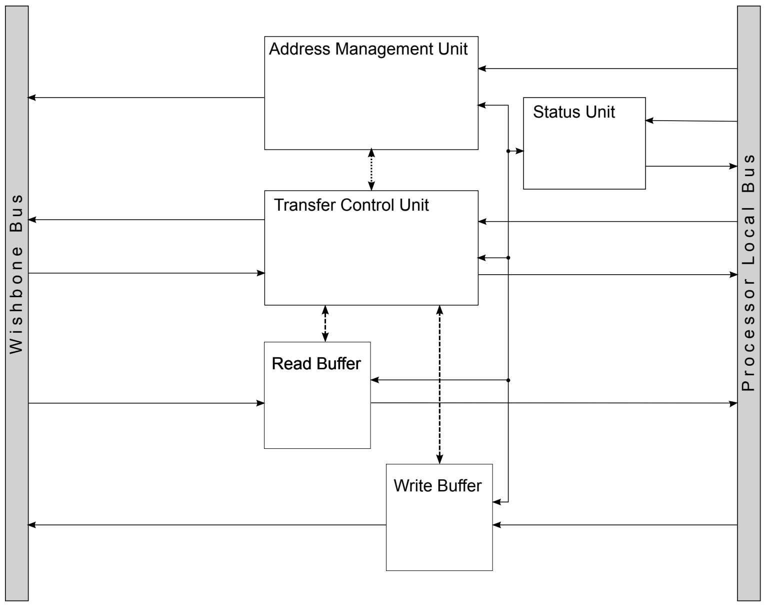
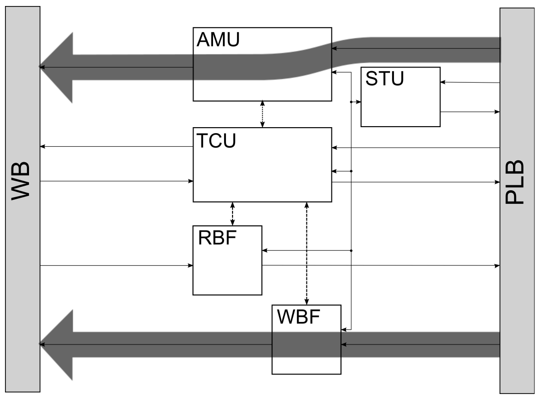


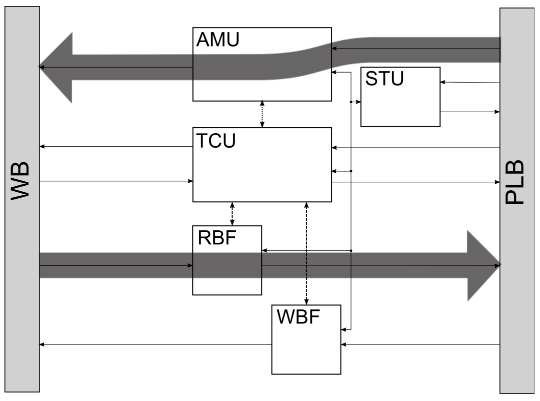
Figure 4.1.: Basic Bridge Architecture

of the two clock domains, the whole data flow is buffered with first in first out buffers (FIFOs). In a write transfer (Figure 4.2(a)), data crosses the bridge in one direction: An address together with a datum from the PLB to the WB side. In a read transfer (Figure 4.2(b)), the address crosses the bridge from the PLB to the WB side and after this, a datum crosses the bridge back from the WB to the PLB side. This has an huge impact for PLB masters. Writing through the bridge can be done very fast, because a PLB master don't have to wait until the transfer is finished. But a PLB master has to wait for the datum while reading through the bridge.

The bridge consists of two address spaces on the PLB side. One address space is used for status registers and another address space is mapped to the WB side. In addition, a negative or positive offset can be added to the address space, which is mapped to the WB

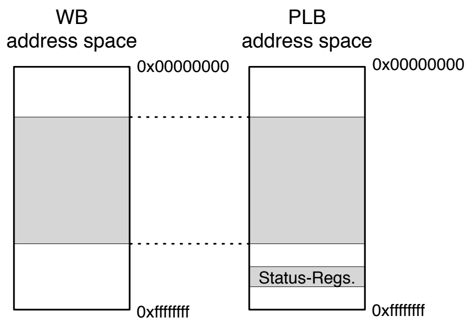


(a) Write transfer

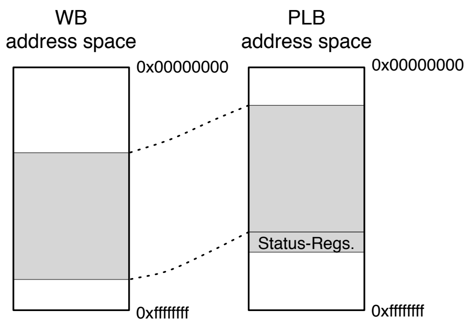


(b) Read transfer

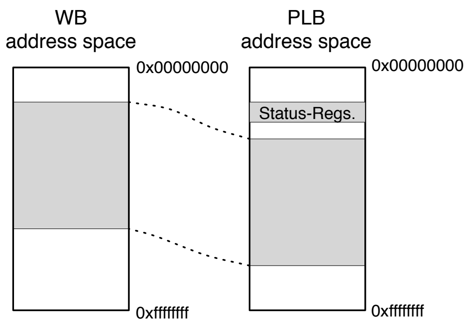
Figure 4.2.: Transfer flows



(a) No offset



(b) Positive offset



(c) Negative offset

Figure 4.3.: Address spaces examples with no offset, positive offset and negative offset

side. This offset allow a more flexible system design. Figure 4.3(a) shows a system with no offset, Figure 4.3(b) and Figure 4.3(c) show examples for a system with positive and negative offset.

4.1. PLB2WB-Bridge Components

4.1.1. Transfer Control Unit

The Transfer Control Unit (TCU) is connected to all components and both buses, in intention to interact with them. It controls whether something is read from a or written to a buffer, it controls the address and pipe-management and it is responsible for handling the most of the control signals of both buses.
Because of the two clock domains, the TCU is also divided into two clock domains. In addition, the PLB side of the TCU is divided into a separate read and write part. This necessary, because PLB write and PLB read transfers can be done overlapped. Under the line, there are three parts: a WB, PLB-Read and PLB-Write part. Each part is implemented in a state machine.

PLB Write-FSM

Table 4.1 shows all PLB write states. If the write and address buffer is not full, a single

State	Description
plb_widle	No transfer in process
plb_write	Single write or line write transfer in process
plb_burst_write	Burst transfer in process

Table 4.1.: PLB write state machine: states description

write transfer is done in one clock cycle, which means that the machine doesn't change the state. If one of the buffers is full and the arbiter requests a write transfer, the machine switches to **plb_write**.
If the PLB arbiter requests a line-write transfer, the machine switches to **plb_write**.
If the PLB arbiter requests a burst-write transfer, the machine switches to **plb_burst_write**.
After finishing a write transfer, the machine switches back to **plb_widle**.
In addition to the current state, the machine manages a counter (**plb_wtrans_state_type-.transfer_count**) and a size (**plb_wtrans_state_type.transfer_size**) value for burst and line transfers.
Because a write transfer to the internal status registers are handled different, an additional flag (**plb_wtrans_state_type.status_transfer**) in the state machine is used to determine, if

the target is the WB side or a internal status register.
In case of a pipelined write transfer, `PLB_wrPrim` goes high and the machine remembers this with **plb_wtrans_state_type.w_secondary**. This enables doing a write transfer directly after the current write transfer. These transitions are shown in Figure 4.4 with dashed lines.

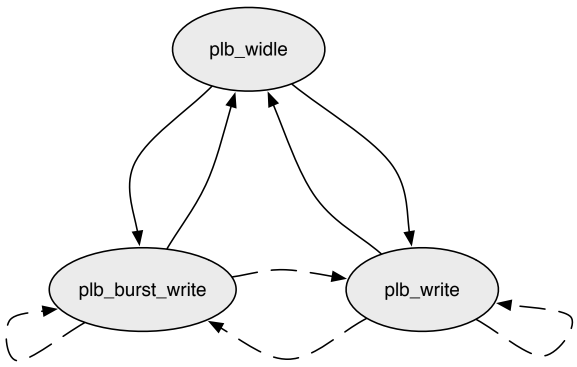


Figure 4.4.: PLB write states

PLB Read-FSM

The read machine is more complicated than the write machine. Table 4.2 lists all states of this machine. If the PLB arbiter requesta a read transfer, the address is added to

State	Description
plb_ridle	No transfer in process
plb_read	Single read transfer in process
plb_read_ack	Last cycle of a single read transfer
plb_line_read	Line read transfer in process
plb_wait_line_read	Wait cycle of a line read transfer
plb_line_read_ack	Last cycle of a line read transfer
plb_wait_burst_read	Wait cycle of a burst read
plb_burst_read	Burst read transfer in process
plb_burst_read_ack	Last cycle of a burst read transfer

Table 4.2.: PLB read state machine: states description

the address buffer and the machine switches from the **plb_ridle** to one of the the following states: **plb_wait_line_read**, **plb_wait_burst_read** or **plb_read**. The wait states (dotted states is Figure 4.5) are necessary and the PLB specification says: In case of a line read or burst read, the first datum must be assigned and acked at least two clock cycles after `Sl_addrAck`

was high. After one clock of a wait state, the machine continues with **plb_line_read** or **plb_burst_read**. Such a wait cycle is not necessary for a single read transfer, because only one ack is generated in **plb_read_ack**.
In a single transfer, the machine waits until the WB side has added the datum to the read buffer. If this is the case, the machine drives `Sl_rdComp` high and swiches from **plb_read** to **plb_read_ack**. The machine remains for one clock cycle in **plb_read_ack** and switches back to **plb_ridle** (it is assumed, that there is no pending secondary request).
In a line read or burst read transfer, the machine uses `RBF_almostEmpty` signal, which indicates, that two data is in the FIFO if the signal is low. This information is necessary, because after acknowledging the penultimate datum, the last datum must be acknowledged and read from the FIFO in the next clock cycle.
Because a read transfer to the internal status registers are handled different, an additional flag (**plb_rtrans_state_type.status_transfer**) in the state machine is used to determine, if the target is the WB side or a internal status register.
In case of a pipelined read transfer, `PLB_rdPrim` goes high and the machine remembers this with **plb_rtrans_state_type.r_secondary**. This enables doing a read transfer directly after the current read transfer. These transitions are shown in Figure 4.5 with dashed lines. The wait cycles are not used in this case (this is the benefit of pipelining).

WB State Machine

Table 4.3 lists the states of the WB state machine. Figure 4.6 shows a graph of the WB

State	Description
wb_idle	No transfer in process
wb_write	Write transfer in process
wb_read	Read transfer in process
wb_write_rty	Write was not successful: `wb_rty_i` was high and the machine switched for one cycle to this state.
wb_read_rty	Read was not successful: `wb_rty_i` was high and the machine switched for one cycle to this state.
wb_write_stall	Write was not successful: `wb_err_i` was high and the machine switched to this state. The machine remains in this state until the STU defines with the signals `STU_abort` and `STU_continue` what to do next.

Table 4.3.: WB state machine: states description

states. The machine switches from **wb_idle** to **wb_read** or **wb_write** if the AMU has a address for the WB side. The states **wb_write_rty** and **wb_read_rty** are used if the WB slave wants to have a retry. That means the machine switches to a retry state and switches

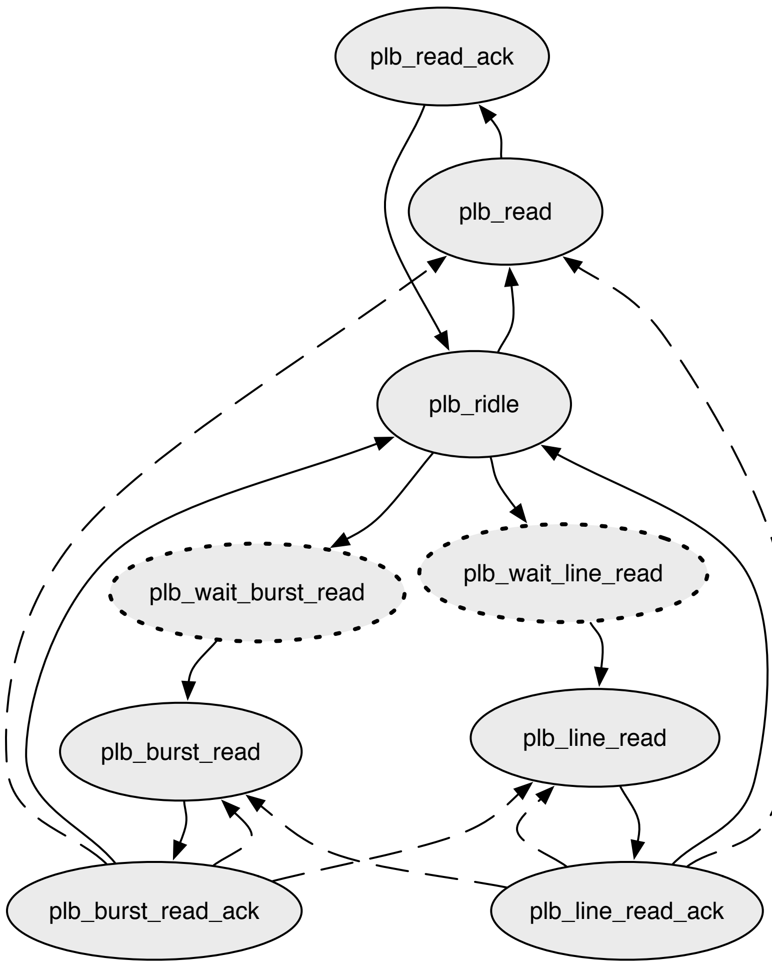


Figure 4.5.: PLB read states

back to **wb_read** or **wb_write** after one clock cycle.
If the WB slave indicates an error while doing a write transfer, the machine switches to **wb_write_stall**. The machine switches from **wb_write_stall** back to to **wb_write** if `STU_abort='1'` or `STU_continue='1'`. If `STU_abort='1'`, the machine reads and empties the write buffer without writing to the WB bus. IF `STU_continue='1'`, the machine continues reading from the write buffer and writing to the WB bus.
Such a stall state does not exist for read transfer. If a read transfer fails, a flag is set in the read buffer and the PLB read machine shows the PLB master, that the read transfer failed.

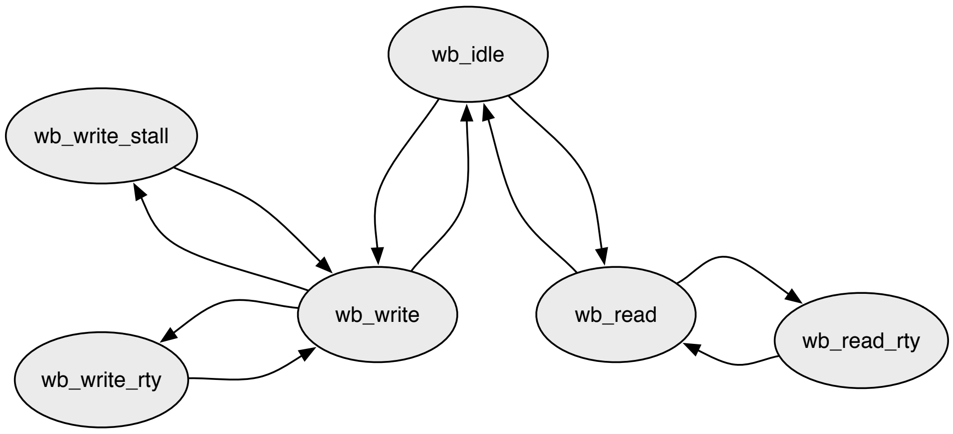


Figure 4.6.: WB state machine

4.1.2. Address Management Unit

The Address Management Unit (AMU) is responsible for the address management. This includes a pipelining mechanism, buffering the addresses and a simple address translation. This results to a design (Figure 4.7) which consists three FIFOs. One FIFO buffers the transfer qualifiers to the WB side. The read-signals are clocked with the WB clock and the write signals are clocked with the PLB clock. The other two FIFOs are used for pipe-lining. They are very small and completely clocked with the PLB clock.

Pipelines

The following signals are buffered with the pipelines:

- `PLB_ABus`
- `PLB_size`
- `PLB_BE`
- `PLB_masterID`

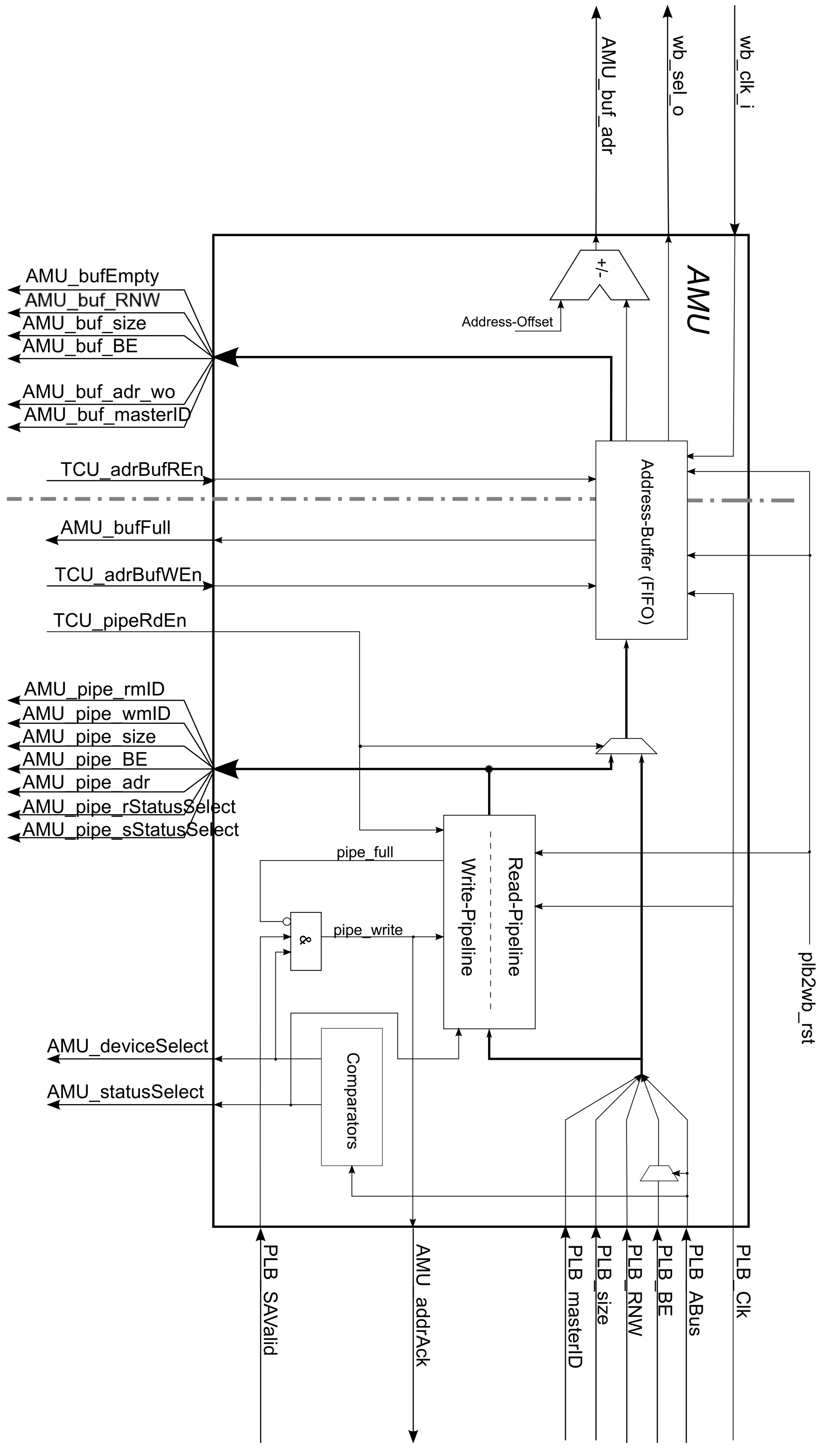


Figure 4.7.: Address Management Unit

- `statusSelect`

The signal `statusSelect` is generated internally and says, if this transfer addresses the WB-side (`statusSelect = '0'`) or the status registers (`statusSelect = '1'`).

Address Buffer

The following signals are buffered with the address buffer

- `PLB_ABus`
- `PLB_size`
- `PLB_BE`
- `PLB_masterID`

Note: `PLB_BE` is assigned to the WB select signals `wb_sel_o` if we don't have a burst transfer. The WB select signals `wb_sel_o` are all '1' if we have a burst transfer. (In this case, the signals `PLB_BE` define the length of the burst transfer.)

Operations

If the PLB arbiter initiates a transfer with driving `PLB_PAValid` high and the TCU wants to add the address to the address buffer, the PLB side of the TCU sets `TCU_adrBufWEn` to '1'. If the WB side of the TCU detects a `AMU_bufEmpty = '0'`, it starts with a WB transfer. After finishing the transfer, the WB side drives `TCU_adrBufREn` high to read from the address buffer.

If the AMU gets a `PLB_SAValid = '1'`, the pipe automatically adds the address to the FIFO. The PLB side of the TCU defines, when to read from the pipe. If the TCU wants to transfer the data from the pipe to the address buffer, it drivers `TCU_rpipeRdEn` or `TCU_wpipeRdEn` high and `TCU_adrBufWEn` high (this is the case, when a transfer addresses the WB side). If the TCU wants to only read from a pipe, it drivers `TCU_rpipeRdEn` or `TCU_wpipeRdEn` high (this is the case, when a transfer addresses status registers).

Address Offset of the WB Side

It is possible to add an optional positive or negative offset the addresses. This is configured statically with generics: `WB_ADR_OFFSET` defines the offset value and `WB_ADR_OFFSET_NEG` defines if the offset is positive `WB_ADR_OFFSET_NEG='0'` or negative `WB_ADR_OFFSET_NEG='1'`

4.1.3. Status Unit

The Status Unit has the following tasks:

- Detect interrupts on WB side and transfer information to the PLB side
- Transfer information about write errors to the PLB side

- Detect bus resets on the WB side and transfer this information to the PLB side

Figure 4.8 shows the internal structure of the STU. There are two FIFOs to transfer data from WB to PLB (Status-to-PLB-FIFO) or from PLB to WB (Status-to-WB-FIFO). In addition, the STU contains 4 software accessible registers (read only). In case of a write access to the status address space, the TCU indicates this with driving `TCU_stuWritePA` or `TCU_stuWriteSA` high (depending if it is a pipelined transfer or not). Some decoding is used to clear the interrupt (if the address is "00"), do a soft reset (if the address is "11") or to continue/abort a failed write transfer (address is "01" or "10").
In case of read access, the address is latched into a address-register. After latching the address, `STU_rdDBus` contains the required register value. On the WB side, `TCU_stat2plb_en` is used to add information about failed write transfer, resets and interrupts to the Status-to-PLB-FIFO. If something is in the Status-to-WB-FIFO, the value is automatically read and `STU_abort` or `STU_continue` is driven high.

4.2. FIFOs

There are seven FIFOs in the PLB2WB-Bridge. Table 4.4 shows the filename, in which part it is used and a description. The first five entries are FIFOs, which are used to separate the two clock domains and to buffer data and addresses. The the first five files

Number	Filename	Separate R/W Clock	Used in	Description
1	fifo_adr.vhd	✓	AMU	buffers address: flow from PLB to WB
1	fifo_rdat.vhd	✓	RBF	buffers data: flow from WB to PLB
1	fifo_wdat.vhd	✓	WBF	buffers data: flow from PLB to WB
1	fifo_stat2plb.vhd	✓	STU	buffers status information: flow from WB to PLB
1	fifo_stat2wb.vhd	✓	STU	buffers status information: flow from PLB to WB
2	plb2wb_fifo		AMU	buffers addresses: for pipelining

Table 4.4.: FIFO overview

in Table 4.4 are wrapper files. They are including device-dependet FIFOs, which must be created by the developer or system designer. If the target is a Xilinx FPGA, there exists

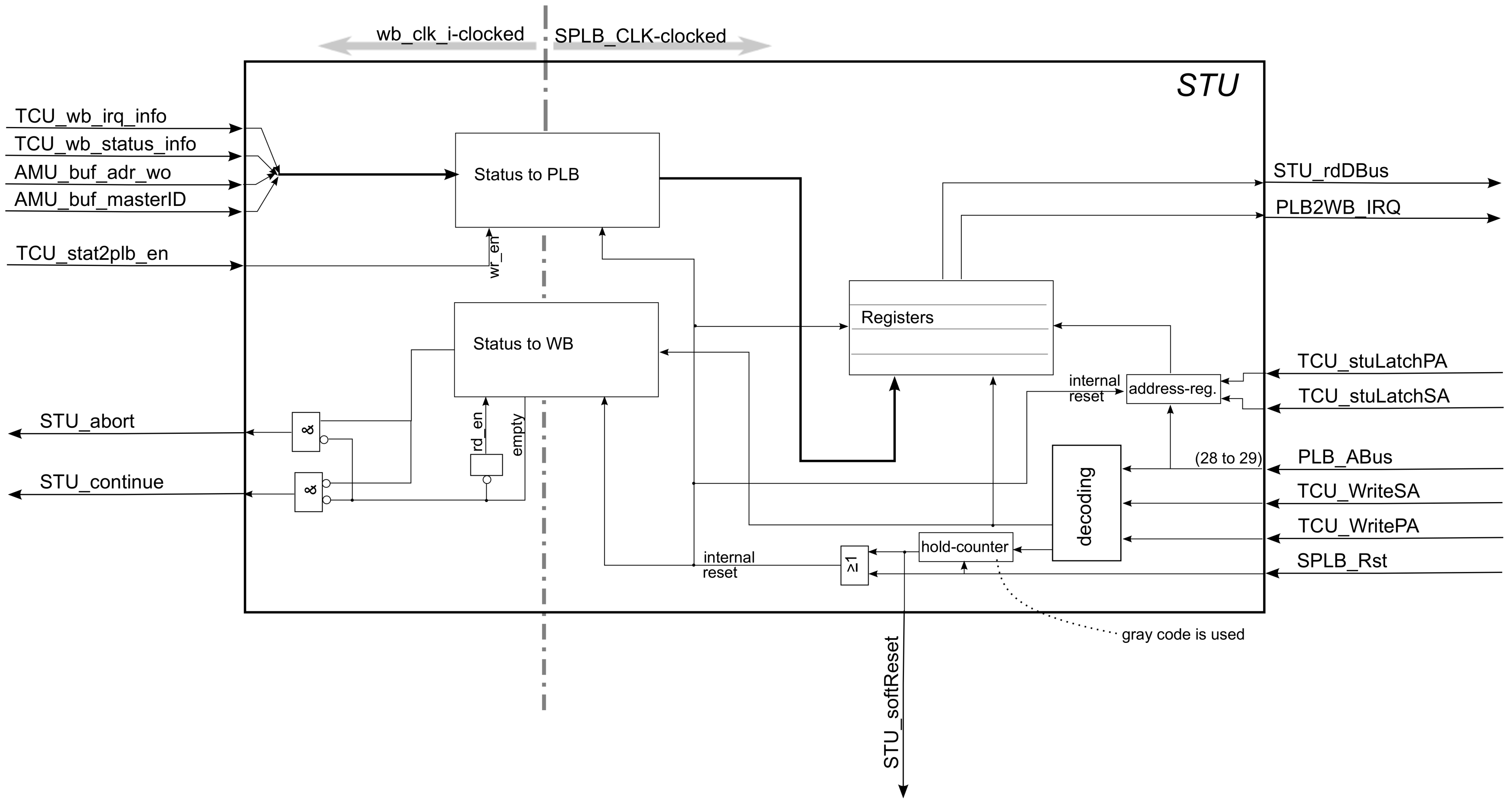


Figure 4.8.: Status Unit

a Fifo-Generator script (see A). The situation of the last entry in Table 4.4 is different. These two FIFOs are very small and they don't need to work with two independent clock domains which means, that they are implemented with some simple VHDL code.

4.3. Slice Logic Utilization

This section shows the slice logic utilization caused by the PLB2WB-Bridge. Four example setups are used to show, how the parameters impact the utilization. Table 4.5 shows the general setup, which is used for all systems.

Name	Value
Toolchain	Xilinx ISE 11
Device Family	virtex5
Device	xc5vlx50
Package	ff676
Speedgrade	-2
Optimization Goal	Area
Optimization Effort	Normal
FSM Style	LUT
FSM Encoding Algorithm	Auto
Place and Route Effort Level	High
Place and Route Mode	Route Only
PLB data bus width	128
WB data bus width	32
WB address bus width	32

Table 4.5.: General setup to test the slice logic utilization.

4.3.1. Small setup without burst transfers, line transfers and pipelining

Table 4.6 shows the setup and Table 4.7 the result after placing and routing.

Name	Value
Masters	1
Wishbone timeout cycles	4
Address offset	no offset
WB interrupts	0
Master id buffer size	16
Address buffer size	16
Read buffer size	16
Write buffer size	16
Status to WB buffer size	16
Status to PLB buffer size	16

Table 4.6.: Small setup without burst transfers, line transfer and pipelining.

	Synchron	Asynchron
LUTs	443	594
Flipflops	416	564
BRAMs/FIFOs	0	0

Table 4.7.: Result of small setup without burst transfers, line transfer and pipelining.

4.3.2. Small setup with burst transfers, line transfers and pipelining

Table 4.8 shows the setup and Table 4.9 the result after placing and routing.

Name	Value
Masters	1
Wishbone timeout cycles	4
Address offset	no offset
WB interrupts	0
Master id buffer size	16
Address buffer size	16
Read buffer size	16
Write buffer size	16
Status to WB buffer size	16
Status to PLB buffer size	16

Table 4.8.: Small setup with burst transfers, line transfer and pipelining.

	Synchron	Asynchron
LUTs	447	598
Flipflops	416	564
BRAMs/FIFOs	0	0

Table 4.9.: Result of small setup with burst transfers, line transfer and pipelining.

4.3.3. Medium setup with burst transfers, line transfers and pipelining

Table 4.10 shows the setup and Table 4.11 the result after placing and routing.

Name	**Value**
Masters	4
Wishbone timeout cycles	8
Address offset	with offset
WB interrupts	4
Master id buffer size	512
Address buffer size	512
Read buffer size	512
Write buffer size	512
Status to WB buffer size	16
Status to PLB buffer size	16

Table 4.10.: Medium setup with burst transfers, line transfer and pipelining.

	Synchron	**Asynchron**
LUTs	378	423
Flipflops	213	328
BRAMs/FIFOs	3	3

Table 4.11.: Results of medium setup with burst transfers, line transfers and pipelining.

4.3.4. Large setup with burst transfers, line transfers and pipelining

Table 4.12 shows the setup and Table 4.13 the result after placing and routing.

Name	Value
Masters	8
Wishbone timeout cycles	32
Address offset	with offset
WB interrupts	32
Master id buffer size	4096
Address buffer size	4096
Read buffer size	4096
Write buffer size	4096
Status to WB buffer size	64
Status to PLB buffer size	64

Table 4.12.: Large setup with burst transfers, line transfer and pipelining.

	Synchron	Asynchron
LUTs	485	572
Flipflops	400	433
BRAMs/FIFOs	13	13

Table 4.13.: Results of large setup with burst transfers, line transfers and pipelining.

5. RTL Simulation

The main development of the PLB2WB-Bridge is done with the Processor Local Bus Functional Model (PLBFM). The PLBFM is developed by IBM and Xilinx offers an adapted and a XPS-integrated version for developing PLB masters and slaves.
For this project, there exists one folder for developing a new functionality and one folder which contains several test-systems and test-cases. This chapter explains first the PLBFM and how it is used. The second part explains the test-structure for this project.

5.1. Bus Functional Model

The PLBFM contains three components

- PLBFM Master
- PLBFM Slave
- PLBFM Monitor

and a Bus Functional Compiler. Instances of these three components can be added to the system like normal IP cores in XPS. The PLBFM is used to generate bus stimulus without

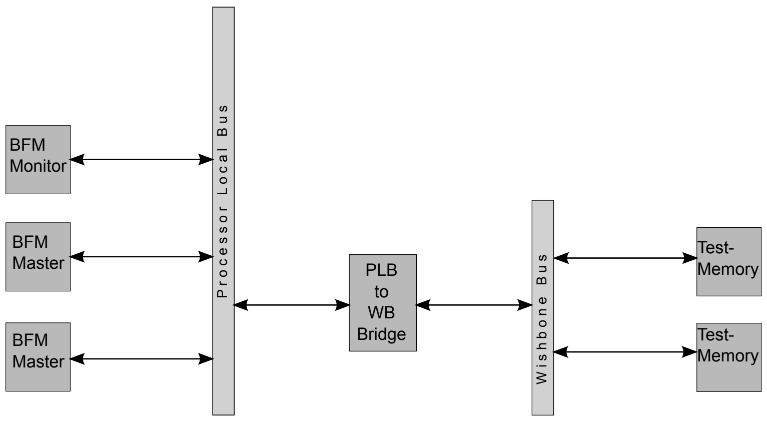


Figure 5.1.: Example for a PLB2WB-Bridge test system

simulating a processor and without writing any C or Assembler code. Instead of C or

Assembler code, some code with the Bus Functional Language (BFL) has to be written. So the PLBFM Master initiates the bus cycles and the PLBFM Slave can respond to it. The PLBFM Monitor is used to check the behavior of all PLB signals. If any signal of the PLB behaves wrong, an error message is printed.
Thereby, the whole PLBFM simplifies the development and verification of PLB peripheries like the PLB2WB-Bridge. The list of features of the PLBFM can be found in [plb01, Chapter 1.1/1.2].
A typical test setup is shown in Figure 5.1. The system contains two PLBFM Masters, a PLBFM Monitor and the PLB2WB-Bridge at the PLB. At the WB, there is the PLB2WB-Bridge and some instances of a WB Test-Memory, which is explained in Section 5.2. The PLBFM Masters generate read and write cycles to and from the WB Test-Memories, whereas the PLB2WB-Bridge must translate these transfers from the PLB to WB. In addition to these components, there exists a synchronization signal (not shown in Figure 5.1). This signal can be used for communication between PLBFM components. Because the PLB2WB-Bridge is not a PLBFM component, the testbench of this system interacts with this synchronization signal.
For example, the synchronization signal can be used to wait for something, like the slave has finished some processing. For testing the PLB2WB-Bridge, the synchronization signal is only used make breaks between bus transfers.

```
set_device(
  path=/system_tb
        /dut
        /plb_bfm_master_32
        /plb_bfm_master_32
        /master,device_type=plb_master)
configure(msize=00)

mem_update(addr=f2000000,data=01112233)
mem_update(addr=f2000004,data=44556677)
mem_update(addr=f2000008,data=8899aabb)
mem_update(addr=f200000c,data=ccddeeff)

mem_update(addr=f3000000,data=01112233)
mem_update(addr=f3000004,data=44556677)
mem_update(addr=f3000008,data=8899aabb)
mem_update(addr=f300000c,data=ccddeeff)

write(addr=f2000000,size=0001, be=0001)
write(addr=f3000000,size=1010, be=0011)
read (addr=f2000000,size=0000, be=1111)
read (addr=f3000000,size=0000, be=0011)
```

Listing 5.1: PLBFM example for a PLB Master

To setup and control the PLBFM peripherals, the BFL is used to describe their behavioral. The BFL source file is compiled with the BFC, which creates a script file. This script file has to be executed by the simulator before running the simulation. An example for a PLBFM master is shown in Listing 5.1. Line one to seven sets the path to the instance and makes some general configuration (in this case, only the size of the device is set: msize=00 $\hat{=}$32 bit). Line nine to 17 initializes the memory of the PLBFM Master device. This is used in line 19 to 22 to for some bus transfers. First, the data is written to the slave and after that, the data is read from the same address. If the data is not the same, which was written, an error message is printed. This allows to test the functionality of the PLB2WB-Bridge. If something goes wrong with a datum, which writing or reading to or from an address, an error message is printed. The type of the transfer is defined with the argument `size` and `be`. A detailed description of all BFL commands and all functionality is provided in [plb01].

5.2. WB Test-Memory (`Test-RAM`)

To simulate different devices with different behavior of a slave device at the WB, a WB Test-Memory was developed for this project. The following list shows the features, which can be configured via XPS:

- Memory initialization with a file
- Variable size
- Configurable read and write delay
- Simulation of WB retries
- Simulation of WB errors

This WB Test-Memory makes it easy to add several WB slaves with different behavior the WB without integrating a real IP core to XPS.

5.3. Test Structure

As mentioned above, there exists two folders. The first folder `proj_dir/systems/dev_-system` contains a development system. New functionality is developed with this system. The folder contains the XPS project files and a `simulation` folder, which contains all necessary files for simulating the system.
The second folder `proj_dir/systems/test_system_sim` contains several test systems, whereas each test system contains one or more than one test-cases. Table 5.1 gives an overview about the existing test systems with all test cases.

Test-System folder	System Description	Test-Cases	Test-Case Description
32bit_on_128bitPLB_asyn	3 PLBFM Masters, 1 PLBFM Monitor, 4 Test-Memories, PLB2WB-Bridge works asynchronous with block/burst transfers and pipelining	simple_burst_rw	Simple burst transfers without pipelining
		simple_line_rw	Simple line transfers without pipelining
		simple_read_write	Simple single transfers without pipelining
		stressfull_read_write	All types of transfers mixed with pipelining
32bit_on_128bitPLB_syn	3 PLBFM Masters, 1 PLBFM Monitor, 4 Test-Memories, PLB2WB-Bridge works synchronous with block/burst transfers and pipelining	simple_burst_rw	Simple burst transfers without pipelining
		simple_line_rw	Simple line transfers without pipelining
		simple_read_write	Simple single transfers without pipelining
		stressfull_read_write	All types of transfers mixed with pipelining
simple	3 PLBFM Masters, 1 PLBFM Monitor, 4 Test-Memories, PLB2WB-Bridge works asynchronous without block/burst transfers and without pipelining	stressfull_read_write	All types of transfers mixed with pipelining
wb_err_and_rst	3 PLBFM Masters, 1 PLBFM Monitor, 4 Test-Memories which generate errors, one Test-Memory has very long r/w delay, PLB2WB-Bridge works asynchronous with block/burst transfers and pipelining, testbech generates reset	errors_and_rst	Tests right behavior of bridge after WB errors and WB resets, interaction with status registers and PLB2WB IRQ
		timeouts	Tests the watchdog timer
wb_irqs	1 PLBFM Master, 1 PLBFM Monitor, 1 WB-Test Memory, IRQs are generated by test-bench	wb_irqs	Test handling of IRQs on WB side, interaction with status registers and PLB2WB_IRQ
wb_retries	1 PLBFM Master, 1 PLBFM Monitor, 3 WB-Test Memories which generate retries	simple_retries	Test right behavior of bridge after WB retries

Table 5.1.: All test systems with all test cases

5.3.1. Structure of a Test System

There exists a `common` folder, which contains a common makefile. This common makefile is used by all test systems to compile peripherals, which are used by all test systems. In addition, there is a makefile for every test system in the folder `simulation`. This makefile is dedicated for a test system and is responsible to create and compile the test system (which depends on `system.mhs` ect.). The test cases can be found in the folder `simulation/test_cases`. Every test case folder contains a makefile and some script files. The makefile generates the PLBFM script files and calls the simulator. And last but not least, there exists a top-level makefile in the `test_system_sim` folder which is responsible to call all makefiles and run all tests.
To run all tests, the user must run `make compile`, which causes to compile all test systems. After that, all test-cases can be run with `make all`. It is also possible to use the `-j` option of make to increase the performance on multicore workstations. An example for a Intel Core-i7 with 8 virtual cores is `make all -j 9`.
Note: It is not suggested to run `make compile` with the j option, because in some rare cases, the compilation fails.
After running `make all`, the three files `error.log` `simulation.log` and `assert.log` are generated which contains information about the simulation and the result.

6. C-Driver

There exists a C-Driver in
`systems/EDK_Libs/WishboneIPLib/drivers/plb2wb_bridge_v1_00_a/`
which is used for

- handling multiple PLB2WB-Bridge instances
- manage interrupt request handler table
- continue and abort failed write transfers
- software resets

This software interface is very similar to the software interface of the Xilinx XPS Interrupt Controller.

6.1. Functions

PLB2WB_Bridge_Initialize

```
int PLB2WB_Bridge_Initialize( PLB2WB_Bridge * InstancePtr, u16 DeviceId, XIntc* xintcInstancePtr, u8 irqID )
```

Arguments		
PLB2WB_Bridge*	InstancePtr	Instance of the bridge
u16	DeviceId	ID of the device (from xparamters.h)
XIntc*	xintcInstancePtr	Pointer to XIntc instance
u8	irqID	ID of the Interrupt (from xparamters.h)
Return value		
int		XST_DEVICE_NOT_FOUND if device was not found, else XST_SUCCESS

This function initializes the PLB2WB_Bridge instance. It needs a pointer the pointer *xintcInstancePtr* because the interrupt handler needs to acknowledge the interrupt in the XIntc. The *irqID* is the interrupt ID of the bridge.
In addition, the IRQ handler table is initialized with stub handlers.

PLB2WB_Bridge_Connect

```
int   PLB2WB_Bridge_Connect( PLB2WB_Bridge*,  u8 Id, XInterruptHandler Handler, void* CallBackRef );
```

Arguments		
PLB2WB_Bridge*	InstancePtr	Instance of the bridge
u8	Id	Interrupt ID
XInterruptHandler	Handler	Handler, which is connected
void*	CallBackRef	Reference to callback argument
Return value		
int		XST_SUCCESS

This function adds an interrupt handler to the IRQ handler table. If an interrupt with the Id *Id* occurs, the handler *Handler* with *CallBackRef* as argument is called.

PLB2WB_Bridge_Disconnect

```
void  PLB2WB_Bridge_Disconnect( PLB2WB_Bridge * InstancePtr, u8 Id );
```

Arguments		
PLB2WB_Bridge*	InstancePtr	Instance of the bridge
u8	Id	ID of the Interrupt, which is disconnected
Return value		
int		XST_SUCCESS

This function disconnects an interrupt handler with the id *Id* and overwrites the entry in the IRQ handler table with the stub handler.

PLB2WB_Bridge_Connect_WBWrErrHandler

```
int PLB2WB_Bridge_Connect_WBWrErrHandler( PLB2WB_Bridge* InstancePtr, XInterruptHandler Handler, void* CallBackRef );
```

Arguments		
PLB2WB_Bridge*	InstancePtr	Instance of the bridge
XInterruptHandler	Handler	Handler, which is connected
void*	CallBackRef	Reference to callback argument
Return value		
int		XST_SUCCESS

This function connects the interrupt handler *Handler* which is called with *CallBackRef* as argument, when a write error occurs.

It is important, that there exists such a handler, the stub handler never returns if a write error occurs.

PLB2WB_Bridge_Connect_WBRstHandler

```
int PLB2WB_Bridge_Connect_WBRstHandler( PLB2WB_Bridge* InstancePtr, XInterruptHandler Handler, void* CallBackRef );
```

Arguments		
PLB2WB_Bridge*	InstancePtr	Instance of the bridge
XInterruptHandler	Handler	Handler, which is connected
void*	CallBackRef	Reference to callback argument
Return value		
int		XST_SUCCESS

This function connects the interrupt handler *Handler* which is called with *CallBackRef* as argument, when a WB reset occurs.
It is important, that there exists such a handler, the stub handler never returns if a WB reset occurs.

PLB2WB_Bridge_WBContinue

```
void  PLB2WB_Bridge_WBContinue ( PLB2WB_Bridge* InstancePtr );
```

Arguments		
PLB2WB_Bridge*	InstancePtr	Instance of the bridge
Return value		
void		

This function continues a failed WB cycle.

PLB2WB_Bridge_WBAbort

```
void  PLB2WB_Bridge_WBAbort( PLB2WB_Bridge* InstancePtr );
```

Arguments		
PLB2WB_Bridge*	InstancePtr	Instance of the bridge
Return value		
void		

This function aborts a failed WB cycle.

PLB2WB_Bridge_SoftReset

```
void  PLB2WB_Bridge_SoftReset( PLB2WB_Bridge* InstancePtr );
```

Arguments		
PLB2WB_Bridge*	InstancePtr	Instance of the bridge
Return value		
void		

This function does a software reset of the PLB2WB-Bridge.

6.2. Example

Listing 6.1 shows, how the driver can be used.

```c
#include "xstatus.h"
#include "xbasic_types.h"
#include "xstatus.h"
#include "xio.h"
#include "xintc.h"
#include "plb2wb_bridge.h"

static   XIntc         intc_instance;
static   PLB2WB_Bridge plb2wb_bridge_instance;

static struct someData{
   int a;
   int b;
} handlerData;


void IRQ_HandlerOfAPeriphery( void* args )
{

   handlerData* data = (hanlderData*) args;
   // do something

   clear_irq_in_periphry();
}


void IRQ_wb_write_error( void* args )
{
   // do something, like
   PLB2WB_Bridge_WBContinue( plb2wb_bridge_instance );
   // or
   PLB2WB_Bridge_WBAbort( plb2wb_bridge_instance );
}
```

```
void IRQ_wb_reset( void* args )
{
   // do something, like
   PLB2WB_Bridge_SoftReset( plb2wb_bridge_instance );
}

int main( void )
{
   XStatus status;

   // initialize bridge and xintc instance
   if( ( status = XIntc_Initialize( &intc_instance, INTC_DEVICE_ID ) ) != XST_SUCCESS )
      return status;

   if( ( status = PLB2WB_Bridge_Initialize(
                     &plb2wb_bridge_instance,
                     PLB2WB_BRIDGE_DEVICE_ID,
                     &intc_instance,
                     PLB2WB_IRQ_ID )
         ) != XST_SUCCESS )
      return status;

   // connect bridge irq handler (-> xintc)
   status = XIntc_Connect( &intc_instance,
                           PLB2WB_IRQ_ID,
                           (XInterruptHandler)PLB2WB_Bridge_DeviceInterruptHandler,
                           (void*) &plb2wb_bridge_instance);
   if ( status != XST_SUCCESS )
      return status;

   // connect periphery irq handler (-> plb2wb_bridge)
   status = PLB2WB_Bridge_Connect( &plb2wb_bridge_instance,
                                   IRQ_ID_OF_A_PERIPHERY,
                                   (XInterruptHandler)IRQ_HandlerOfAPeriphery,
                                   (void*) &handlerData
                                );
   if ( status != XST_SUCCESS )
      return status;

   // connect wb write error handler (-> plb2wb_bridge)
   status = PLB2WB_Bridge_Connect_WBWrErrHandler( &plb2wb_bridge_instance,
                                                  IRQ_wb_write_error,
                                                  (void*)0
                                                  );
   if ( status != XST_SUCCESS )
      return status;

   // connect wb reset handler (-> plb2wb_bridge)
   status = PLB2WB_Bridge_Connect_WBRstHandler( &plb2wb_bridge_instance,
                                                IRQ_wb_reset,
                                                (void*)0
                                                );
   if ( status != XST_SUCCESS )
      return status;

   // enalbe the bridge irq (-> xintc)
```

```
    XIntc_Enable( &intc_instance, PLB2WB_IRQ_ID );

    // start irq controller (-> xintc)
    if( ( status = XIntc_Start(&intc_instance, XIN_REAL_MODE) ) != XST_SUCCESS )
      return status;

    // enable irq  (-> microblaze)
    microblaze_enable_interrupts( );

    while(1)
    {
      // do something
    }
}
```

Listing 6.1: Example usage of the bridge driver

A. FIFO-Generator Script

Because a lot of FIFOs are needed, the generation of this FIFOs might be done with a script[1]. Table A.1 shows all FIFOs, which are generated with this script. Because some text processing is needed, Ruby as modern highlevel programming languate is choosen. Due to the fact that the generation of the FIFOs depend on parameters, a setup file is necessary. The next section explains the usage and the functionallity of the FIFO-Generator script and Section A.2 explains the parameters and lists an example setup file.

A.1. fifo_generator.rb

The ruby script file `fifo_generator.rb` can be found in `proj_dir/coregen/fifo_generator/`.

```
   Usage: ruby fifo_generator [OPTIONS]... setup-file

 OPTIONS:

    -c --no-coregen         do not run coregen (Only creates *.xco command_files)
    -v --cp-vhdl-to-lib     copy vhdl-files to library folder
                            (path is set in 'setup-file')
    -n --cp-ngc-to_imp      copy netlist-files to implementation folder
                            (path is set in 'setup_file')
    -? --help               display this message
```

If you run the script without any option, all files (*.vhd files and *.ngc files) are created, but nothing more is done. There are two options to copy the *.vhd files and *.ngc files to the desired folders. The *.vhd files are only needed for simulations in contract to the *.ngc files, which are used in the synthesis.
If you have multiple systems, you can run the script one time without any options and then several times with the -c and -n flags with different paths. This copies the file into the desired folders.

[1]Only for Xilinx FPGAs

Filename	**Wrapper**	**Description**
fifo_adr_cc_1.vhd	fifo_adr.vhd	Common clocks, 1 master
fifo_adr_cc_2.vhd		Common clocks, 2-4 masters
fifo_adr_cc_3.vhd		Common clocks, 5-8 masters
fifo_adr_cc_4.vhd		Common clocks, 9-16 masters
fifo_adr_ic_1.vhd		Independet clocks, 1 master
fifo_adr_ic_2.vhd		Independet clocks, 2-4 masters
fifo_adr_ic_3.vhd		Independet clocks, 5-8 masters
fifo_adr_ic_4.vhd		Independet clocks, 9-16 masters
fifo_rdat_cc_32.vhd	fifo_rdat.vhd	Common clocks
fifo_rdat_ic_32.vhd		Independet clocks
fifo_stat2plb_cc_1.vhd	fifo_stat2plb.vhd	Common clocks, 1 master
fifo_stat2plb_cc_2.vhd		Common clocks, 2-4 masters
fifo_stat2plb_cc_3.vhd		Common clocks, 5-8 masters
fifo_stat2plb_cc_4.vhd		Common clocks, 9-16 masters
fifo_stat2plb_ic_1.vhd		Independet clocks, 1 master
fifo_stat2plb_ic_2.vhd		Independet clocks, 2-4 masters
fifo_stat2plb_ic_3.vhd		Independet clocks, 5-8 masters
fifo_stat2plb_ic_4.vhd		Independet clocks, 9-16 masters
fifo_stat2wb_cc.vhd	fifo_stat2wb.vhd	Common clocks
fifo_stat2wb_ic.vhd		Independet clocks
fifo_wdat_cc_32.vhd	fifo_wdat.vhd	Common clocks
fifo_wdat_ic_32.vhd		Independet clocks

Table A.1.: All FIFOs, which are generated by the FIFO-Generator script

A.2. setup file

Parameter name	**Range**	**Description**
Device_Family		The FPGA device family
Device		The FPGA device
Package		The FPGA package
Speedgrade		The speedgrade, used for synthesis
PLB2WB_Bridge_VHDL_DIR	16 ... 4194304	Destination directory for VHDL files
PLB2WB_Bridge_NGC_DIR	16 ... 4194304	Destination directory for NGC files
Address_Buffer_Size	16 ... 4194304	Size of the address buffer
Read_Buffer_Size	16 ... 4194304	Size of the read buffer
Write_Buffer_Size	16 ... 4194304	Size of the write buffer
Stat2WB_Buffer_Size	16 ... 4194304	Size of the status-to-plb buffer
Stat2PLB_Buffer_Size	16 ... 4194304	Size of the status-to-wb buffer
WB_Clk_Frequency		WB clock frequency
PLB_Clk_Frequency		PLB clock frequency
Address_Buffer_minDWidth	42 (constant)	Minimum width of the address buffer
Read_Buffer_DWidth	33 (constant)	Width of the read buffer
Write_Buffer_DWidth	32 (constant)	Width of the write buffer
Stat2PLB_Buffer_minDWidth	100 (constant)	Width of the status-to-plb buffer
Stat2WB_Buffer_DWidth	1 (constant)	Width of the status-to-wb buffer

Table A.2.: All parameters, which are used by the FIFO-Generator script

Listing A.1 shows an example, how a setup file for the FIFO-Generator script can look like. The first word in every line is the name of the option, followed by a equal sign and the value. It doesn't matter, how much spaces or tabs are between the names and equal signs and between the equal signs and values. The order of the parameters doesn't matter, too. Important is, that every parameter is defined with a valid value and the name is written right (case sensitive). It is also possible to use comments: everything behind a hash sign is ignored by the script. The parsing of this file is done with regular expressions.

```
Device_Family           = virtex5
Device                  = xc5vlx50
Package                 = ff676
Speedgrade              = -2
```

```
# Path to the vhdl and implementation directory
PLB2WB_Bridge_VHDL_DIR = ../../systems/EDK_Libs/WishboneIPLib/pcor\
  es/plb2wb_bridge_v1_00_a/hdl/vhdl/
PLB2WB_Bridge_NGC_DIR =../../systems/adv_system_fpga/implementation/

Address_Buffer_Size = 16
Read_Buffer_Size    = 16
Write_Buffer_Size   = 16
Stat2WB_Buffer_Size = 16
Stat2PLB_Buffer_Size = 16
WB_Clk_Frequency = 66
PLB_Clk_Frequency = 100

#################################
# Do not change below unless
# you know what you are doing
Address_Buffer_minDWidth = 42
Read_Buffer_DWidth      = 33
Write_Buffer_DWidth     = 32
Stat2PLB_Buffer_minDWidth = 100
Stat2WB_Buffer_DWidth   = 1
```

Listing A.1: Example of a setup file for the FIFO-Generator script

Table A.2 lists all parameters, which must be defined in the setup file

Bibliography

[plb01] *Processor Local Bus Functional Model Toolkit - User's Manual.* Version 4.3. Research Triangle Park, NC : IBM, 2001