

Preface

This how-to is intended as a basic tutorial for the Opencore's PCI Bridge project, in order to

- 1) simulate and test the core on a complete open environment (Linux)
- 2) synthesize it on a PCI target board with a Xilinx FPGA
- 3) write a Linux driver for the resulting device.

The how-to is NOT intended as a replacement of the official documentation of the core but, given the complexity of the project, it wants to simplify the approach to its material.

It is assumed that the reader has a good knowledge of Verilog before reading parts 1-3, as well as a good practice with Linux operating system (I suggest the (K)Ubuntu distro, on which you can easily find all the material required by the tutorial)

In addition, it's required to have at least a basic knowledge of PCI and WISHBONE protocols, even if they are generally made transparent to the user by the bridge itself and by the operating system's API.

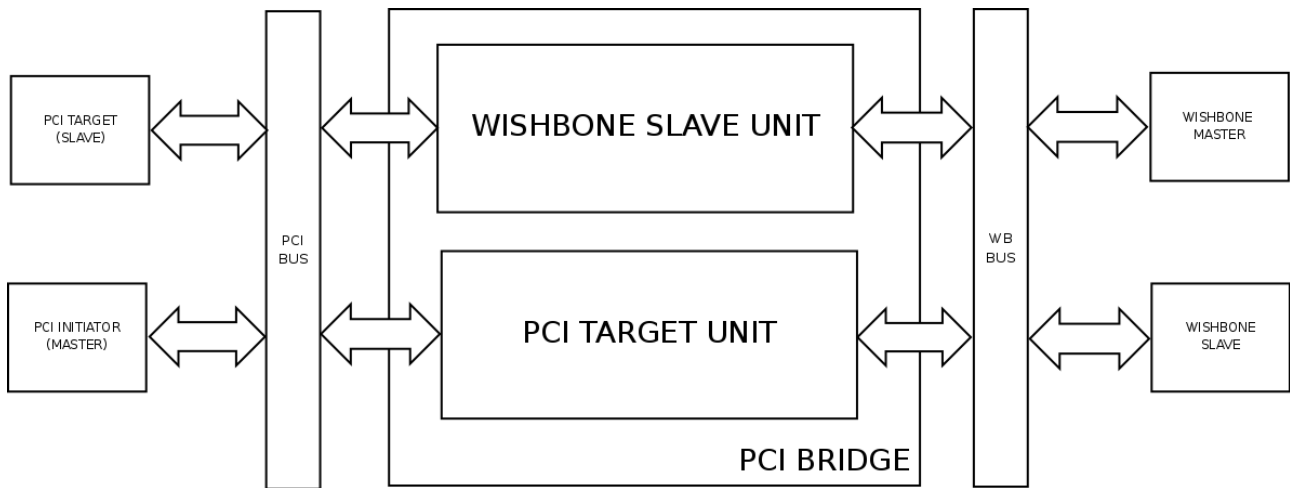
The WISHBONE bus

WB specification defines a protocol with which slave and master modules, all in the same chip (--> SOC), can communicate with each other.

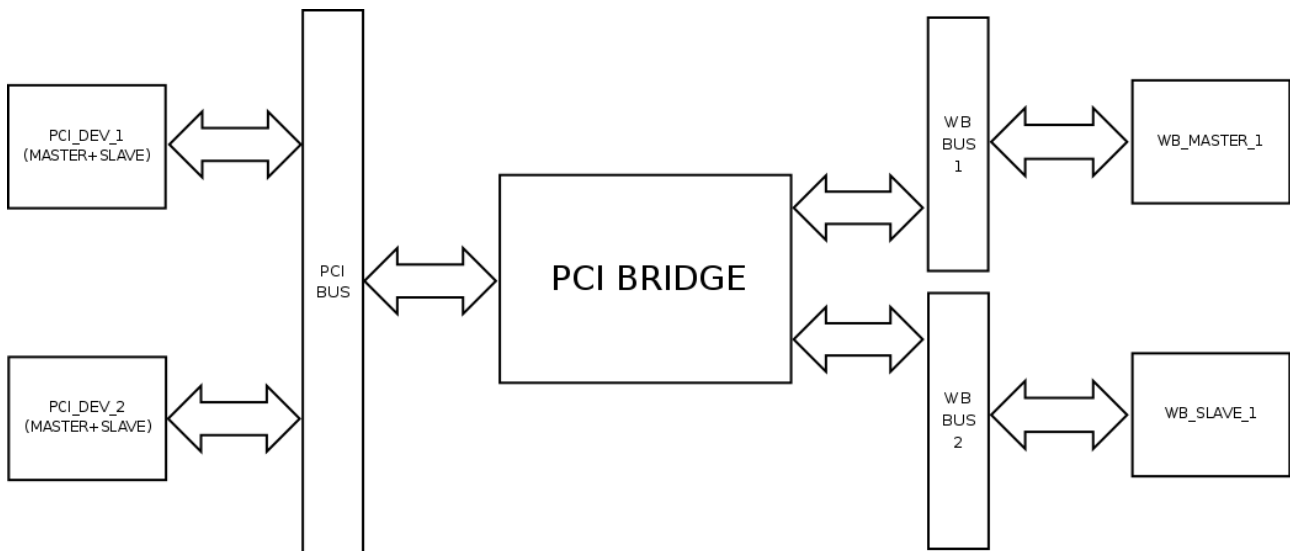
If all these modules are appropriately interconnected and they all respect protocol's specifications, we can figure a virtual common bus that we call Wishbone (WB) bus.

The communication between modules can be done with three types of cycles (or transactions): single read/write, block read/write, read-modify-write. The transaction's initiator is always a master module, which can read/write data (typically DWORDs) from/to a slave module

The PCI bridge is a normal PCI device which is composed of two independent macroblocks: an interface called "pci target unit", which allows the communication between a PCI master (initiator) and a WB slave, and an interface called "wishbone slave unit", which allows the communication between a WB master and a PCI slave (target):



Now, let's start from the following situation:



In the above figure:

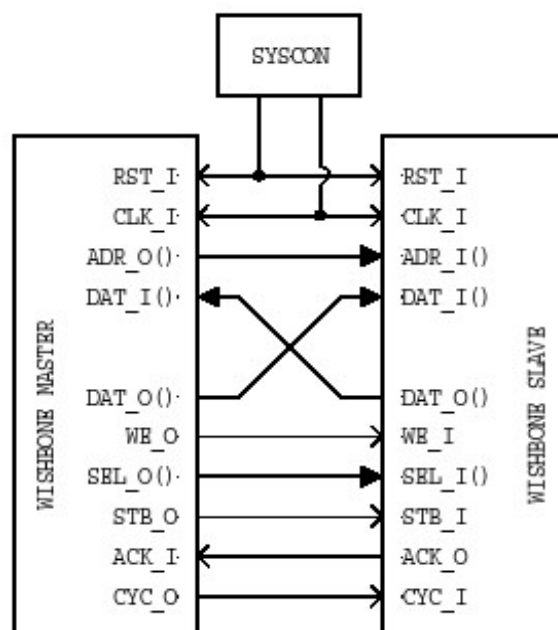
- WB_SLAVE_1 is a simple RAM which can be accessed R/W through a WB bus
- WB_MASTER_1 is a simple module which can read/write DWORDs from/to a WB bus
- PCI_DEV_1/2 are two PCI devices which can operate both as master, reading/writing DWORDs from/to the PCI bus, and slave (target), including a SRAM which can be accessed R/W through a PCI bus

Note from the above figure that PCI_DEV_1/2 and PCI_BRIDGE share the same PCI bus, while WB_SLAVE_1 and WB_MASTER_1 are connected to two independent WB buses:

WB_SLAVE_1 shares WB bus 1 with PCI_TARGET_UNIT, while WB_MASTER_1 shares WB bus 2 with WB_SLAVE_UNIT.

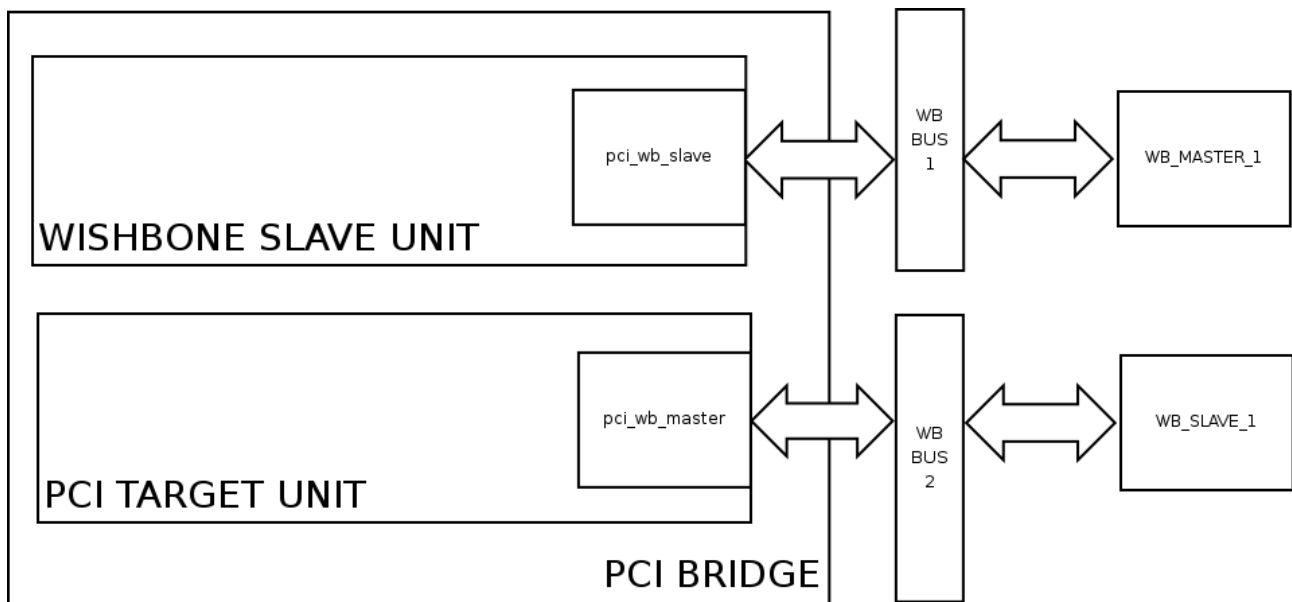
The choice of two independent WB buses, in our example, is due to the fact that, in this way, interconnections between modules are very simple; let's explain why.

As said before, a WB bus consists into multiple slave and master modules which are appropriately interconnected and respect the WB protocol; from [WISHBONE, Revision B.3 Specification](#) (Point to point interconnection, see pag. 96) we learn that if a WB bus has to be shared by a single master module and a single slave module, then the interconnection between them can be simply obtained by short-circuiting the master's I/Os with the slave's I/Os in this way (in addition: see the definition of SYSCON at page 25):



When, on the contrary, the same WB bus has to be shared by multiple master/slave modules, then we need some additional logic that makes the interconnections between them.

This said, if we observe in detail PCI TARGET UNIT's architecture, we note that the module is interfaced to the WB bus through a WB master module (pci_wb_master), which can drive a WB slave module, while WISHBONE SLAVE UNIT is interfaced to the WB bus through a WB slave module (pci_wb_slave) which can be driven by a WB master module



Our first target is to create a simple testbench which performs the following operations:

- 1) **PCI_DEV_1** WRITES 1 DWORD TO **WB_SLAVE_1**
- 2) **PCI_DEV_1** WRITES 6 DWORDs TO **WB_SLAVE_1**
- 3) **PCI_DEV_1** READS 6 DWORD FROM **WB_SLAVE_1**
- 4) **WB_MASTER_1** WRITES 1 DWORD TO **PCI_DEV_1**
- 5) **WB_MASTER_1** READS 1 DWORD FROM **PCI_DEV_1**
- 6) **WB_MASTER_1** WRITES 6 DWORDs TO **PCI_DEV_1**

Therefore, in order to limit as more as possible the additional logic, we will implement our testbench by interconnecting the four WB modules on two separate WB buses

Environment installation

- 1) Download the pci-bridge core:

<http://opencores.org/cvsget.cgi/pci>

- 2) Untar the project and create the environment var **\$PCI**, which must point to the base directory of the project. If, for example, the archive has been uncompressed in **/home/some_user**, in a debian based distro we can execute the following command:

```
paolo@paolo-laptop:~$ sudo cat PCI="/home/some_user/pci" >> /etc/environment
```

3) Download from THIS CVS the testbench for the above figure. The associated Verilog code is a simplification of the "official" pci-bridge's testbench; in addition to the figure it includes a pci_blue_arbiter, inherited from the previous testbench, which is used for arbitration on the PCI bus (see PCI IP Core Design Document, pg. 56); the resulting verilog code could appear still complex, but consider that all the PCI stuff will be removed and it will be replaced by a software pci initiator, after the core will be synthesized on a real FPGA:

4) Untar simple_testbench.tar.gz inside \$PCI dir

5) download from the below link the open source Verilog compiler/simulator that we are going to use: Icarus Verilog (IVerilog).

In some distros, this software is available as debian or rpm package, but some older version have bug that make the compiler/simulator crash with the pci-bridge core. Therefore it's preferable to download and manually compile/install (**./configure; make; sudo make install**) Icarus Verilog's sourcecode, in its last version, which works correctly:

<ftp://icarus.com/pub/eda/verilog/snapshots>

6) Create now a file wich will be used by Iverilog in order to compile the bridge and the testbench:

```
+incdir+$(PCI)/simple_bench/verilog
+incdir+$(PCI)/rtl/verilog
+libdir-nocase+$(PCI)/simple_bench/verilog
+libdir-nocase+$(PCI)/rtl/verilog
$(PCI)/simple_bench/verilog/system.v
```

Save the previous file as conf_file.txt in \$PCI/sim/rtl_sim/run/

7) Compile the testbench with the following command, so to produce an executable file (simple_testbench) wich will be used for the simulation (we can ignore the warning messages from the compiler):

```
paolo@paolo-laptop:~/pci/sim/rtl_sim/run$ iverilog -o simple_testbench -cconf_file.txt
/home/paolo/paolo/pci/simple_bench/verilog/paolo_behaviorial_device.v:206: warning: L-value
`pci_ext_idsel' is also an input port.
/home/paolo/paolo/pci/simple_bench/verilog/paolo_behaviorial_device.v:106: warning: input
pci_ext_idsel; is coerced to inout.
/home/paolo/paolo/pci/simple_bench/verilog/paolo_behaviorial_device.v:206: warning: L-value
`pci_ext_idsel' is also an input port.
/home/paolo/paolo/pci/simple_bench/verilog/paolo_behaviorial_device.v:106: warning: input
```

```
pci_ext_idsel; is coerced to inout.
```

Note: with the macros `define GUEST and `define HOST, in \$PCI/rtl/verilog/pci_user_constants.v, it is possible to choose to compile the bridge as HOST or as GUEST.

The document \$PCI/doc/pci_specification.pdf explains in detail all the differences between the two implementations (see paragraph 3.2); however, let's say briefly some considerations about that; the bridge has a global configuration space which is composed of the classic PCI configuration space:

//	31	24	23	16	15	8	7	0	
//	Device ID				Vendor ID				0x00
//	Status				Command				0x04
//	Class Code				Rev				0x08
//	BIST	HEAD		LTCY		CSize		0x0C	
//	Base Address 0				0x10				
//	Base Address 1				0x14				
//	Base Address 2				0x18				
//	Base Address 3				0x1C				
//	Base Address 4				0x20				
//	Base Address 5				0x24				
//	Cardbus Pointer				0x28				
//	SubSys ID				SubVnd ID				0x2C
//	Expansion ROM Pointer				0x30				
//	Reserved				Cap				0x34
//	Reserved				0x38				
//	MLat	MGnt	IPin	ILine	0x3C				

...and two configuration spaces which are respectively part of the WISHBONE SLAVE UNIT and of the PCI TARGET UNIT; both these areas form a region which is always implemented and which is pointed by a base address that will be assigned - during the bridge configuration (which we will examine shortly) - to the fifth (0x10) register of the PCI configuration space (see figure above). Now, if the bridge is implemented as HOST bridge, then:

- a) the WHISBONE SLAVE UNIT has R/W access to the configuration space whereas PCI TARGET UNIT has read-only access
- b) the bridge can scan the pci bus, determine all the connected devices and configure them

If the bridge is compiled as GUEST (which is the default implementation), then:

- a) the PCI TARGET UNIT has R/W access to the configuration space whereas WHISBONE SLAVE UNIT has read-only access

In our test we will use the default implementation (GUEST)

- 8) Install the waveform viewer that we are going to use in our

tests: GTKWave (in most distros you can find it as a .rpm or .deb package)

<http://home.nc.rr.com/gtkwave/>

9) Now, let's execute for the first time our `simple_testbench`; the execution will only inform us that a dumpfile (which we can open with GTKWave), called `Test.vcd`, has been created and nothing else, since all the code included in the main task (`run_tests`) has been intentionally commented out.

```
paolo@paolo-laptop:~/pci/sim/rtl_sim/run$ vvp simple_testbench
VCD info: dumpfile TestbenchResult.vcd opened for output.
** VVP Stop(0) **
** Current simulation time is 67584000 ticks.
** Interactive mode not supported, exiting simulation.
paolo@paolo-laptop:~/pci/sim/rtl_sim/run$
```

Configuration of PCI devices

In order to make a PCI device ready to use, it must be configured. As stated in advance, we assume that the reader knows at least the basic concepts of a PCI configuration task; here are some links which can be helpful for that purpose:

- 1) [PCI Local Bus Specification Revision 3.0](#)
- 2) http://en.wikipedia.org/wiki/Peripheral_Component_Interconnect#Auto_Configuration
- 3) http://en.wikipedia.org/wiki/PCI_Configuration_Space

Anyway, let's focus on the principal steps involved in this phase:

1) For each PCI device, at least one addresses region, which is used to access I/O registers of the device, must be enabled. This is achieved by assigning to the region a base address which has to be written in its respective register in the PCI configuration space

2) We can choose to implement the region as I/O or Memory space: the last solution is generally the saner one (see [PCI Local Bus Specification Revision 3.0](#), paragraph 3.2.2, and [Linux Device Drivers, Third Edition](#), pg. 316) and we will choose it for all our PCI devices

3) Command Register of PCI configuration space must be configured in order to set the basic level of functionality of the device (see again [PCI Local Bus Specification Revision 3.0](#), paragraph

6.2.2)

In our testbench we are going to simulate the above operations; a single base address will be assigned to PCI_DEV_1/2, while two base address will be assigned to the pci bridge (as stated above, bridge's configuration space is pointed by base address 0)

We don't need to go into detail of the tasks involved in this phase, since they will be replaced by the functions of a Linux device driver that we are going to write after executing the complete testbench; anyway, let's observe at least the principal steps involved during the configuration phase of a PCI device. Firstly, uncomment the following lines inside run_tests task:

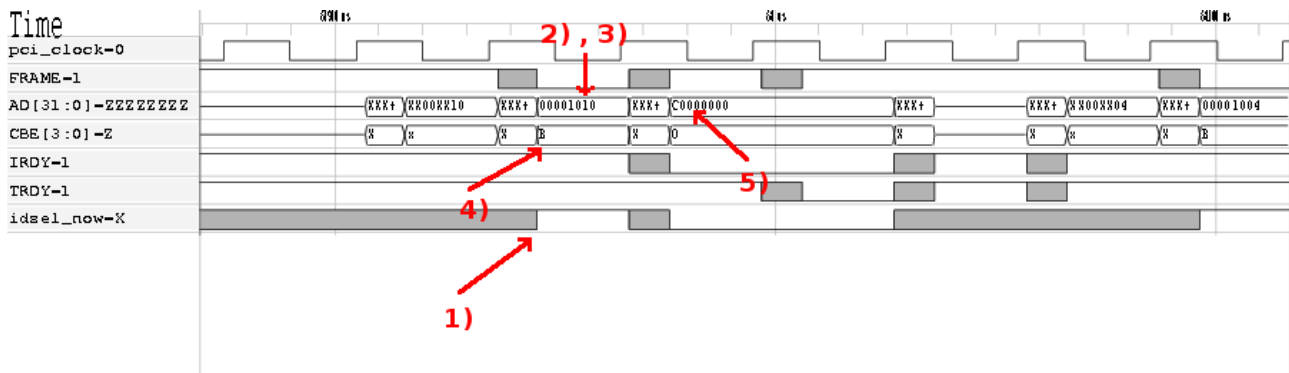
```
/****** file: system.v *****/
.
.
.
//
// CONFIGURATION
//

//CONFIGURE PCI_DEV_1
//configuration_cycle_write task has the following arguments: bus number, device number, //function
number, register number, type of configuration cycle, byte enables, data
@(posedge pci_clock) ;
$display(" PCI_DEV_1 - write PCI_DEV1 base addresses");
configuration_cycle_write(0, 1, 0, 4, 0, 4'hF, `BEH_TAR1_MEM_START ) ;
$display(" PCI_DEV_1- enable target's response, master and parity errors, disable system error");
configuration_cycle_write(0, 1, 0, 1, 0, 4'h3, 32'h00000047) ;

//CONFIGURE PCI_DEV_2
@(posedge pci_clock) ;
$display(" PCI_DEV_2 - write PCI_DEV2 base addresses");
configuration_cycle_write(0, 2, 0, 4, 0, 4'hF, `BEH_TAR2_MEM_START ) ;
$display(" PCI_DEV_2- enable target's response, master and parity errors, disable system error");
configuration_cycle_write(0, 2, 0, 1, 0, 4'h3, 32'h00000047) ;

//CONFIGURE pci bridge
@(posedge pci_clock) ;
$display(" pci bridge - Enabling master and target operation!");
configuration_cycle_write(0, 0, 0, 1, 0, 4'hF, 32'h00000007) ;
$display(" bridge target - Setting base address P_BA0 to 32'h1000_0000");
configuration_cycle_write(0, 0, 0, 4, 0, 4'hF, `TAR0_BASE_ADDR_0) ;
$display(" bridge target - Setting base address P_BA1 to 32'h2000_0000");
configuration_cycle_write(0, 0, 0, 5, 0, 4'hF, `TAR0_BASE_ADDR_1) ;
```

The above code is totally self-explanatory; let's compile now our testbench and start the simulation (iverilog -o simple_testbench -cconf file.txt; vvp simple_testbench). After the simulation is done, open Test.vcd file with GTKWave and examine the following signals, during the first configuration operation (PCI_DEV_1 - write PCI_DEV1 base addresses)



As you can see, according to [PCI Local Bus Specification Revision 3.0](#), paragraph 3.1.1:

- 1) In the address phase, `IDSEL` of `PCI_DEV_1` is asserted
- 2) `AD[1:0] == 2'b0`
- 3) `AD[7:2]` == base address 0's offset in PCI configuration space. Note also that `AD[7:0] == 8'h10 == 8'b00010000 --> AD[7:2] == 6'b000100`; this last value corresponds to offset 4 (fourth DWORD) of the configuration space
- 4) in the address phase, `C/BE[3:0] == 4'hB` (configuration write command)
- 5) In the data phase `AD` stores the value that must be written in the configuration register (`BEH_TAR1_MEM_START = c0000000`)

In addition you can note in the verilog code that two base registers (0 and 1) have been configured on the bridge. The first one allows us to access r/w the bridge's configuration area: all the registers in this area are memory mapped and they will be necessary later to enable/disable/configure some bridge's features

The second one is the base address from which the `pci_wb_master` can read/write data from/on the WB bus