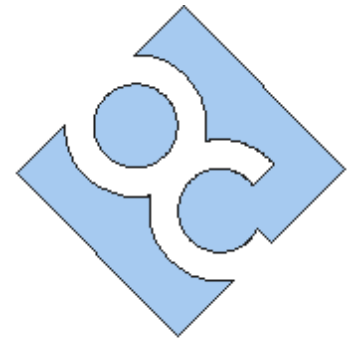
PCI_TARGET - Wishbone_MASTER INTERFACE MODULE (PCI-mini) Datasheet

IP core version - v2.0

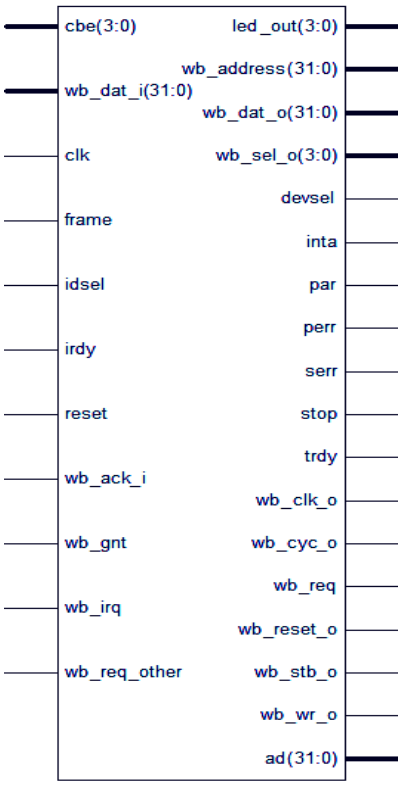


The original PCI module is from: Ben Jackson
http:www.ben.com/minipci/verilog.php

Redesigned for wishbone : Istvan Nagy, buenos@freemail.hu, istvan.nagy@peccorp.com
PEC Products, Industrial Technologies www.peccorp.com
2007

1.Introduction:

The core implements a 16MB relocable memory image. Relocable on the wishbone bus. The wb address = wb_baseaddr_reg + PCI_addr[23:2]
The wb_baseaddr_reg register: the upper 10 bits are implemented, the lower bits are zeroes, so it can specify, 4 Mdword banks.
Only Dword aligned Dword accesses allowed on the PCI. This way we can access to the 4GB wb-space through a 16MB PCI-window through address-translation. Translation is controlled by PCI-configuration write accesses.
The addressing on the wb-bus, is Dword addressing, while on the PCI bus, the addressing is byte addressing. A(pci)=A(wb)*4
The PCI address is increasing by 4, and we get 4 bytes. The wb address is increasing by 1, and we get 1 Dword (= 4 bytes also).
The wb_baseaddr_reg is the wb image relocation register, can be accessed at 50h address in the PCI configuration space.
Other bridge status and command is at the 54h and 58h addresses.
if access fails with timeout, then the address will be in the wb address will be stored in the failed_addr_reg at 5Ch address.
Source: one single verilog file, and some timing constraints for the system ucf file.
For writing a driver software, please read the chapter 3: “PCI compatibility” !
For connecting peripherals, please read the chapter 2: “wishbone compatibility” !



Test results:

Synthesis: 279 Slices on Xilinx Spartan-3 FPGA. (14.5% logic on SP3-200k)
Tested on hardware:
-PCI card (with SP2 FPGA) plugged into an old PC with Pentium-II CPU and VIA VT82C693A+VT82C596B chipset
-Custom motherboard developed by me, with the AMD Geode-LX processor, and Spartan-3 FPGA.
Test software: Hardware-Direct.
FPGA project: a peripheral block, consisting: AltiumDesigner Wishbone intercone module, Altium CAN controller, some custom peripherals, and the PCI2WB bridge.

Addressing:

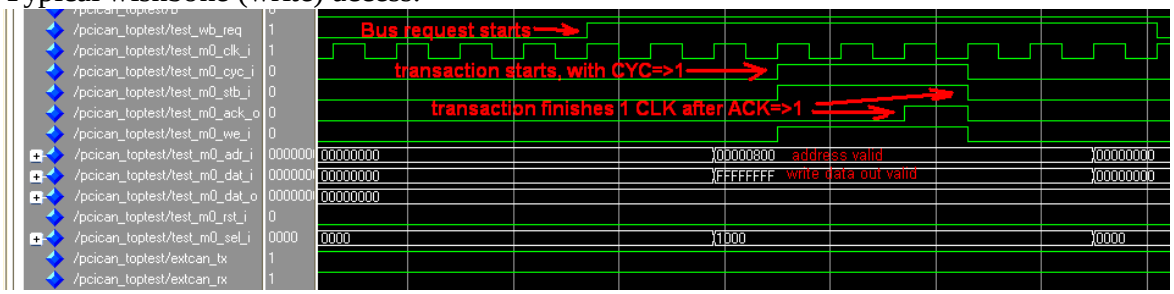
A(wishbone) = (A(pci) - BAR0) / 4 + wb_baseaddr_reg
A(pci) = (A(wishbone) - wb_baseaddr_reg) * 4 + BAR0

Addressing on PCI bus is byte-addressing, but on the WB-output address-bus, its DWORD addressing. So, only DWORD accesses are permitted.

2.Wishbone compatibility:

Wishbone signals: wb_address, wb_dat_o, wb_dat_i, wb_sel_o, wb_cyc_o, wb_stb_o, wb_wr_o, wb_reset_o, wb_clk_o, wb_ack_i.
Not implemented wb signals: error, lock, retry, tag-signals.
The peripheral has to response with ack in 16 clk cycles, otherwise the IP will Terminate the wishbone transaction, and read or write will be unsuccesful/corrupt.
The core has wishbone clk and reset outputs, just like a Syscon module.
The core generates single reads/writes. These are made of 4 phases, so dont write new data, until internal data movement finishes: about 300...500ns.
The connected peripherals have to be able to work together with the PCI interface, As its visible on the transaction waweforms.

Typical wishbone (write) access:



The transaction starts, when CYC, the STB, and the WE (if write access) is high (valid). And finishes when these signals return to zero. The address, write data, and SEL is valid one clock cycle before transaction starts. Finishing the transaction: When the peripheral responses with ACK, on the next clock cycle, the CYC/STB/WE signals change to zero, so transaction sinishes. There are no bursts, coming from the PCI interface IP core.

The ACK signal must be 1 clock cycle long. (not half), and it must arrive within 15 clock cycles, after the transaction starts. If we are using a multi-master system, then we can not use too slow peripherals: the ACK must arrive within 4 clock cycles, because of the aggressive bus arbitration. The PCI IP core implements this type of arbitration, otherwise we would have to implement retry cycles on the PCI bus, and it can freeze the computer.

3.PCI compatibility:

Only single DWORD reads/writes are supported. between them, the software has to wait 300...500nsec, to prevent data corrupting. STOP signaling is not implemented, so target terminations also not. And the Dword must be 4bytes loaction aligned, so the 2 LSBs of the Physical address are always zero.
Single Byte access is NOT supported! Bursts are also not! These may cause corrupt data.
The core uses INTA interrupt signal. There are some special PCI config registers, from 50h...60h config-space addresses.
PCI-parity: it generates parity, but doesnt check incoming parity.
Because of the PC chipset, if you read a value and write it back, the chipset will not write anything, because it can see the data is not changed. This is important at some peripherals, where you write, to control.

Device specific PCI configuration header registers:

name:	addr:	function:
wb_baseaddr_reg	50h	A(wb)=(A(pci)-BAR0)/4 + wb_baseaddr_reg
user_status_reg	54h	not used yet
user_command_reg	58h	not used yet
failed_addr_reg	5Ch	address, when timeout occurs on the wb bus.

Configurration header values, for device identification (standard registers):

```
DEVICE_ID = 16'h9500;
VENDOR_ID = 16'h10EE; //          16'h10EE=xilinx, beceuse we used a Xilinx chip.
DEVICE_CLASS = 24'h068000;     // Bridge device - other_bridge_type
DEVICE_REV = 8'h01;
SUBSYSTEM_ID = 16'h0001;       // Card identifier
SUBSYSTEM_VENDOR_ID = 16'hBEBE; // Card identifier
DEVSEL_TIMING = 2'b00;// Fast!
```

Write access:
This is a “posted write” type access, so during the PCI transaction, the data is writing into a buffer register, and after finishing the PCI transaction, a wishbone transaction starts, so the core writes the data into the correct location from the buffer.

Read access:
This is a “delayed read request/completion” type read, and it is made of 3 phases:

1. The host reads from the target (this module), then doesn’t use the data.
2. After the PCI transaction, a read transaction begins on the Wishbone bus, and reads the correct data, to a buffer register.
3. The host reads again, from the same address, but now the data is already correct, so the host can use it.

This way we have to read everything twice, to get a correct data. Between any accesses on the PCI bus, the host must wait 300…500 ns, for the completion of the wishbone transactions.
If we want to read a lot of data, then here is one trick:
Read the last data, throw it away, then read all the data in the correct order. Read everything into an array, then put the first element of the array to the end, and shift the others to the beginning. This wat there were N+1 accesses, and some processing.

We must be shure, that nor the chipset, nor the BIOS, nor the OS, nor the Driver will initiate burst transfers to this IP module. Its bossible, that one of them will be grouping the individual single read/write accesses into bursts. The software developers have to make shure in this!

If we want to implement Bursts transfers, then we have to implement STOP signaling on the PCI bus: Target terminations, like “Retry” when the wb state machine is busy on the wb bus, or “target disconnect with data”, when FIFO is full. Ans of course we need a read FIFO, a write FIFO, and FIFO control logic.

4.Local bus arbitration:

This is not really wishbone compatible, but needed for the PCI.
The method is: "brute force". it means if the PCI interface wants to be mastering on the local (wishbone) bus, then it will be mastering, so, the other master(s) must stop anything immediately. The req signal goes high when there is an Address hit on teh PCI bus. so the other master has few clk cycles to finish.

Restrictions:

-The peripherals have to be fast: If the other master starts a transaction before req goes high, the ack has to arrive before the PCI interface starts its own transaction. (max 4clk ACK delay)

- The other master or the bus unit must sense the req, and give bus mastering to the PCI-IF immediatelly, not just when the other master finished everything, like at normal arbitration schemes.

5. Buffering:

There is a single Dword buffering only. No FIFO memories are used.

6.The LED_out interface:

Only for system-debug: we can write to the LEDs, at any address. (in the same time there is a wishbone write also)

7. User sub-system control signals:

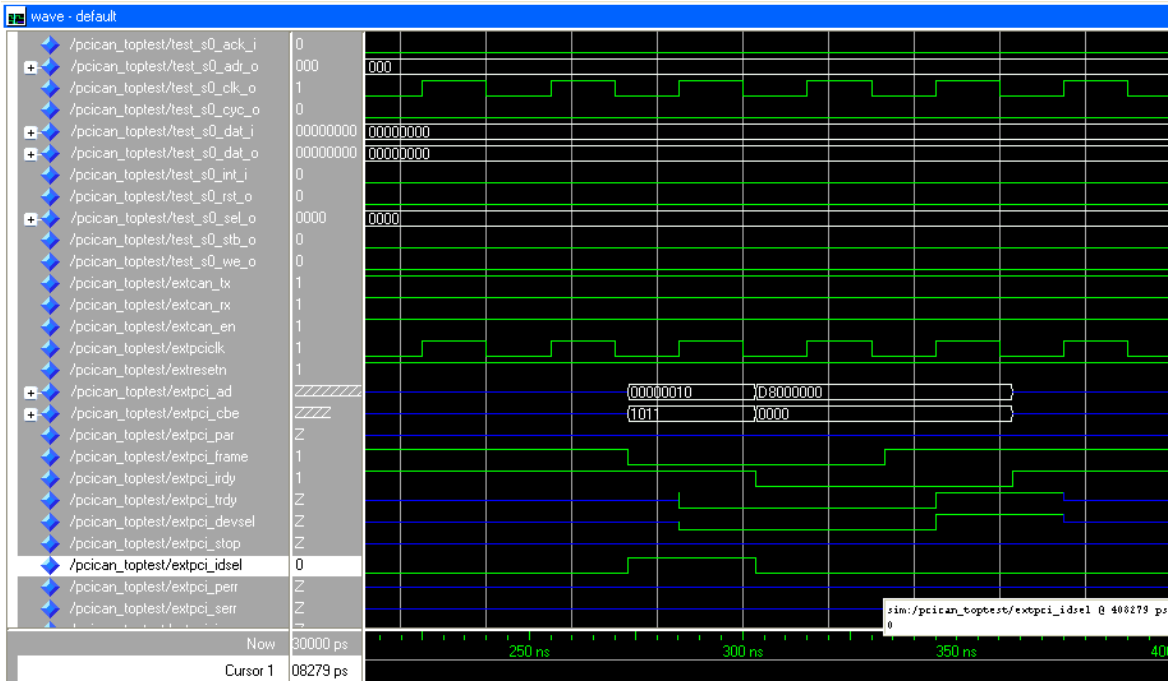
There is an additional 8-bit output bus: "`contr_o`" for any kinds of user subsystem control functions. For example resetting the user peripherals by software. It's a general-purpose IO, controlled by PCI configuration writes at the user command register:
"`contr_o = user_command_reg [7:0];`"

8.Changes since original version:

wishbone interface, bigger memory-image, parity-generation, interrupt handling, New registers, local-bus arbitration opportunity, Code size is 3x bigger.

9. Transaction waveforms:

Configuration write:



Memory write: