esa

DOCUMENT

document title/ *titre du document*

PCI INTERFACE FOR LEONUMC

prepared by/*préparé par* Roland Weigand
European Space Agency

reference/*réference* TOS-ESM/001.03/RW
issue/*édition* 3
revision/*révision* leonumc
date of issue/*date d'édition* 12-Mar-2004
status/*état* User Information
Document type/*type de document* Technical Note
Distribution/*distribution* LEONUMC Processor Users

CHANGE LOG

reason for change /*raison du changement*	issue/*issue*	revision/*revision*	date/*date*
Based on Release 3.3c, updated to reflect the implementation in LEONUMC	3	leonumc	12-Mar-2004

CHANGE RECORD

ISSUE: **3** REVISION: **LEONUMC**

reason for change/*raison du changement*	page(s)/*page(s)*	paragraph(s)/*paragraph(s)*

TABLE OF CONTENTS

1 INTRODUCTION

This document contains **user information** (datasheet) for the PCI functionality of the LEONUMC processor. It references the non-public InSilicon documentation ([3]) concerning the PCI core. Most information however is also contained in the PCI standard ([1]), and the InSilicon documents are deemed unnecessary for the user.

1.1 AMBA implementation

The interface is connected to the on-chip AMBA bus implemented in LEON, according to the AMBA specification [4] and to ESA's AMBA-VHDL package [6]. It has two masters and one slave connected to the AMBA High-performance Bus (AHB). A slave on the AMBA Peripheral Bus (APB) allows access to configuration and status registers.

1.2 PCI implementation

The interface uses the Verilog PCI core revision 3.1b from Synopsys (former In-Silicon and Phoenix Technologies), according to reference [3]. The core has nevertheless been configured to the LEON-specific needs and was updated with patches. A block diagram of the PCI core is given in Figure 1 (in this document, the term 'PCI core' only to the functionality contained in the box named 'PCI controller'). The PCI core contains data and request FIFOS and state machines to schedule the PCI transactions, compliant to the PCI standard [1].
The PCI implementation has the following features:

- PCI 2.2 compliant configuration space
- 32 Bit 33 MHz PCI bus operation, independent of the processor clock (synchronising FIFOs)
- Initiator (Master) and Target operations (single word and burst mode)
- 8 words data FIFO per each of the 4 data paths (initiator/target, receive/transmit)
- 4 deep initiator request FIFO
- Target lock support
- Zero-latency Fast Back-to-Back transfers
- Zero wait state burst mode transfers
- Support for Memory Read Line/Multiple and Memory Write and Invalidate commands
- Use in both Host-Bridge and Satellite (add-in card) designs
- Delayed Read support
- Flexible error reporting by polling or LEON interrupt line

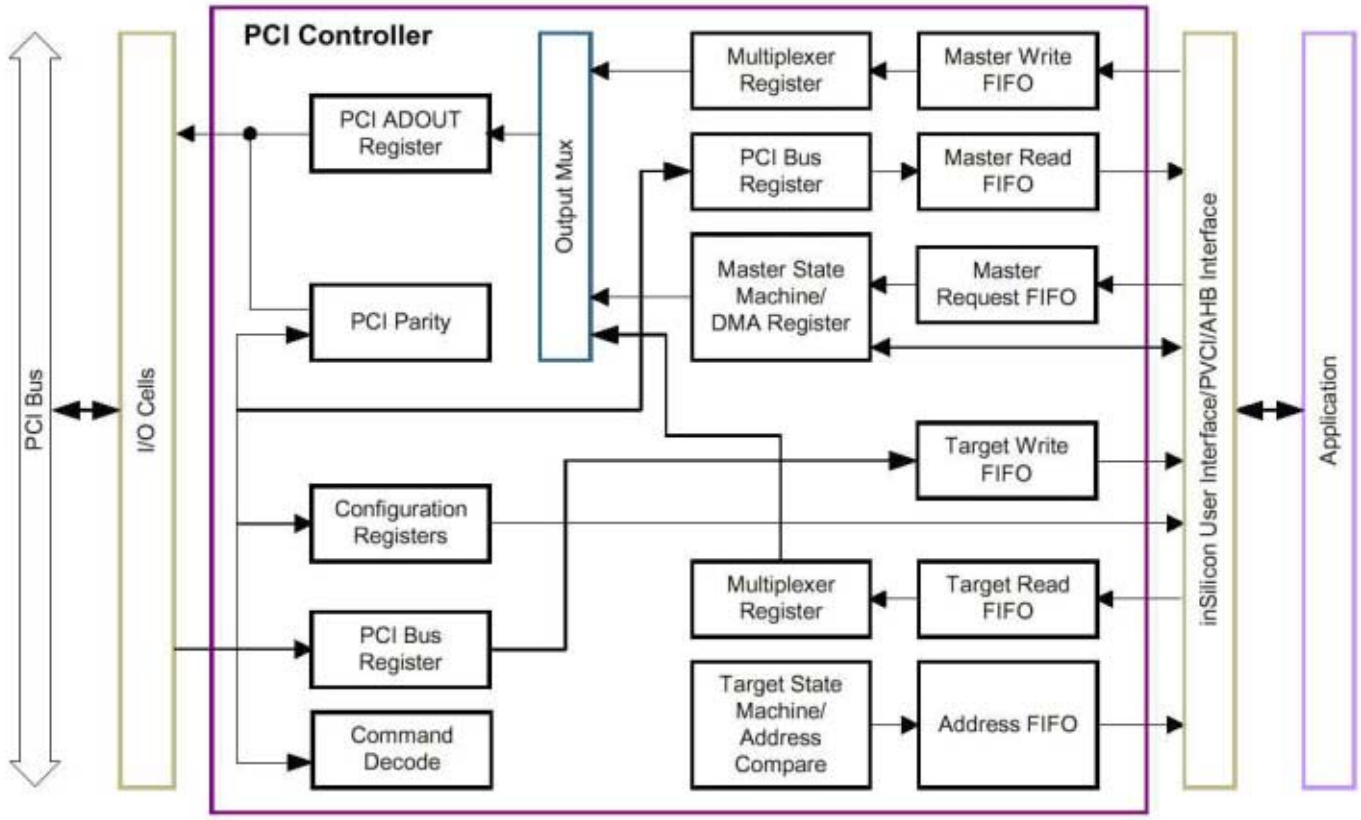

FIGURE 1: BLOCK DIAGRAM OF THE PCI CORE (SOURCE: [3])

1.3 Relating Documents and Links

Listed are the reference sources, which were used for this design, and which are relevant for the user of this interface. These documents are not necessarily available for free download; however, information may be obtained at the given Internet addresses.

[1] PCI Local Bus Specification, Rev. 2.2 and PCI Bus Power Management Interface Specification, Rev. 1.1, PCI Special Interest Group (for information: http://www.pcisig.com)

[2] LEON/AMBA source code and User's Manual, GR-LEON2-(FT)-1, Jiri Gaisler, Version 1.0.4, available from http://www.gaisler.com

[3] PCI Core Series Host-Bridge and Satellite Synthesizable Cores Asynchronous and Synchronous with FIFOs, Release Version 3.0x Cores, Rev. 4, Phoenix Technologies Ltd., relevant release and application notes up to Version 3.1b universal core and patches. The core is now distributed by Synopsys, for information please refer to: http://www.synopsys.com/products/designware/docs/ds/c/dwcore_pci.pdf.

[4] AMBA™ Specification Rev 2.0. Document ARM IHI 0011A, 13 May 1999, issue A, first release, ARM Limited. The document can be retrieved from http://www.arm.com. AMBA is a trademark of ARM Limited. ARM is a registered trademark of ARM Limited.
[5] The SPARC Architecture Manual, Version 8. Prentice Hall, 1992. ISBN 0-13-825001-4 (SPARC® is a registered trademark of SPARC International, Inc. For information: http://www.sparc.org, free download: http://www.sparc.com/standards/V8.pdf).
[6] VHDL package "amba", Version 0.5, published by ESA – Microelectronics Section, 30 – Aug. 2000, can be retrieved from http://www.estec.esa.int/microelectronics.
[7] The AMBA FAQ http://www.arm.com/support/amba?OpenDocument
[8] PCI to AMBA Bridge VHDL Model Datasheet, European Space Agency, 24-Jan-03
[9] LEON2-FT User's Manual (*leon2ft-1.0.4-configured.pdf*)
[10] LEON-PCI Verification Study, GR-TECH-021, Issue 1.3, Gaisler Research, February 2003

1.4 Terms and Acronyms

AHB: **A**MBA **H**igh **P**erformance Bus, see [4].

APB: **A**MBA **P**eripheral **B**us, see [4].

BAR: **B**ase **A**ddress **R**egister. By the number of bits writeable in its BAR(s), every PCI device indicates, how much address space it claims on the PCI bus. By writing to the BAR, the PCI address space is allocated to the different satellite devices by the host-bridge device at system boot-up time.

DAC: **D**ual **A**ddress **C**ycle. Allows specifying addresses > 32 bit on a 32-bit PCI bus within two successive bus cycles.

DMA: **D**irect **M**emory **A**ccess, a data transfer, which is initiated by the processor, but data is transferred directly from one peripheral to another. The PCI-initiator interface has a DMA controller. Initialised via APB, it transfers data between AHB and PCI. The PCI target interface is also a DMA port, initialised by the remote PCI initiator.

FIFOs: The PCI core has 4 data FIFOs of 8 words each and a master/initiator command FIFO for 4 requests. In the FIFOS, synchronisation between the 2 clock domains (PCI clock and processor clock) is done.
- MXMT: Master-transmit-FIFO (from AHB to PCI bus, for store instructions).
- MRCV: Master-receive-FIFO (from PCI bus to AHB, for load instructions).
- TXMT: Target-transmit-FIFO (from local memory to PCI bus, PCI-read).
- TRCV: Target-receive-FIFO (from PCI bus to local memory, PCI-write).
- Master request FIFO: Allows posting master read- and write-requests.

HDL: Hardware Description Language, means here both languages used for the present design – Verilog for the PCI core and VHDL for the AMBA wrapper.

HOST-BRIDGE and SATELLITE: The host-bridge connects the local bus of a processor (host) to the PCI bus. The PCI configuration registers of a host-bridge are accessible locally by the host, but not via PCI configuration cycles. The host-bridge initialises the other devices (satellites) through PCI configuration commands. The satellite is a PCI device, configurable via PCI configuration cycles and the idsel line, but not locally. A PCI system usually has one host-bridge, and several satellites. Do not confuse with the terms of master and target: PCI is a multi-master bus. Both, host-bridge and satellites can be initiator (master) and/or target on the bus. The present interface has universal functionality, allowing both operation modes configurable via a bootstrap pin.

MASTER and TARGET: A PCI master initiates a transaction on the PCI bus. It specifies the address and requests the number of data words to be read or written. Do not confuse with the terms of Host-Bridge or Satellite. A PCI target decodes all addresses specified by a master and responds, if the address corresponds to its address space. To avoid confusion with the AMBA masters, the PCI master is usually called INITIATOR in this document. However, especially in the context of the PCI core or the PCI standard, the term master is sometimes used.

PCI: **P**eripheral **C**omponent **I**nterconnect. See [1].

SPARC: **S**calable **P**rocessor **Arc**hitecture. See [5]. SPARC Data formats: SPARC allows accesses to byte (8 bit), half-word (16 bit), word (32 bit) or double-word (64 bit) formats. In a 32-bit architecture, double-word accesses are translated into 2 successive 32-bit accesses. SPARC is big-endian: The address always points to the most significant data byte in the memory. Addresses must be aligned according to the size: For half-word bit 0, for word bit 1 and 0, for double-word bit 2-0 must be zero. The same alignment rules are applicable on AHB. See [4] and [5]. In this document, a “word” refers to a 32-bit data word.

2 ARCHITECTURE AND FUNCTIONALITY

2.1 General Architecture

The block diagram of the AMBA to PCI Bridge is shown in Figure 2. This is the PCI block which appears in fig. 1 of [9] (LEON block diagram). The signals at the top of the figure are connected to the AMBA bus system, to clock, reset and interrupt signals. The signals at the bottom are the external pads.

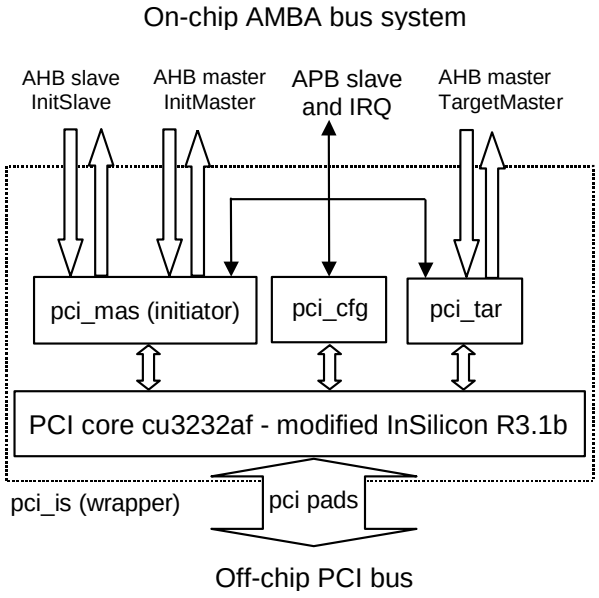

FIGURE 2: BLOCK DIAGRAM OF THE AMBA TO PCI INTERFACE

The interface is connected to the AHB by the means of one AHB slave and two AHB masters:

- **InitSlave**: AHB slave providing other AHB masters (i.e. the LEON processor) with a direct memory-mapped (initiator) access to the PCI bus. An AHB request, directed to InitSlave, will be translated by the interface into the appropriate PCI transactions. See section 2.2.
- **InitMaster**: AHB master of the DMA controller. When start addresses (local and remote) and transfer sizes are programmed through APB, the DMA controller initiates transactions on PCI as well as on the AHB and executes a data transfer between local AHB slaves and remote PCI targets. See section 2.3.
- **TargetMaster**: AHB master connecting the PCI target device to the AHB. It provides access for external PCI initiator devices to on-chip AHB slaves (for example the on-chip memory controller). The PCI-target data transfer takes place in background, without intervention of the LEON processor. However, the processor may monitor the status of the target interface, allow interrupts for error reporting, or reset the target interface through APB. See section 2.4.

The APB slave provides access to the configuration and status registers. The DMA controller is also programmed through the APB. The APB registers in general are described in section 4.

To enable PCI initiator data transfer (either via *InitSlave* for memory-mapped access or via *InitMaster* for DMA transactions), bit 0 of the mas_rs register must be set to 1 (see Table 5), **and** the PCI master bit 2 of the PCI-command register (see Table 4). To prevent data ordering errors, the operation of these two devices is mutually exclusive, i.e., a DMA transfer will not start until a slave transfer was finished, and any new slave request will be retried, while the DMA master is active. Care must be taken to avoid potential deadlock situations (one thread trying to program DMA, while another thread tries to access PCI via load/store instructions).

2.1.1 HOST-BRIDGE AND SATELLITE MODE

The PCI interface allows for two fundamentally different operating modes, called Host-Bridge and Satellite, selected by the bootstrap pin pci_host: '1' = host-bridge, '0' = satellite. The modes are defined in section 1.4, and further details are given in the following sections. Please note that a change of pci_host requires a reset and re-initialisation of the interface.

2.1.2 PCI ERROR PROCESSING

Error and status bits according to the PCI standard [1] are implemented in the PCI status register (see Table 3 and Table 4). In host-bridge (pci_host = '1') configuration, this allows an error detection by polling. Certain events and errors are also reported by the interface in the status bits of register int_st (Table 5). For each bit of this register, optionally, the (single) interrupt line can be asserted. The different interrupt causes are distinguished by a set of interrupt registers. Their functionality is documented in section 2.5.

2.2 PCI Initiator: AHB Slave

The AHB slave at the ports InitSlaveIn/Out allows other AHB masters, to become a master (initiator) on the PCI bus. The AHB request, directed to InitSlave, will be translated by the interface into the appropriate PCI transactions. This makes the PCI sequencing transparent for LEON programs. The PCI bus may be accessed by the same instructions as the main memory (just using the appropriate address range), even instructions can be fetched directly from PCI.

2.2.1 TRANSLATION OF MEMORY INSTRUCTIONS TO PCI

The SPARC instruction set foresees various load/store instruction types. The PCI bus foresees 32 bit wide transactions with byte-enables for each byte lane. These are translated to the PCI bus according to the following rules:

- **Transaction width:** Instructions of different width (byte, half-word, word, double) are translated as shown in Table 1 as a function of the 3 LSB of the SPARC address A[2:0]. The PCI byte enables are active low and are given here in the order CBE#[3:0]. Big-endian

mapping according to [5] is implemented: Byte-writing to A[1:0] = 00 results in the byte enable pattern 0111, hence, the msb byte lane (bits 31:24) of the PCI data bus is selected. The table gives the assembler instruction in the second line and the C data-type (singed or unsigned) to generate this instruction in the third line. For non-aligned accesses (in case they can occur at all), the byte enable pattern (1111) is issued on PCI, to avoid destroying data in the remote PCI target.

- **PCI command type:** The interface can issue the PCI commands memory-read/write (default), I/O-read/write, configuration-read/write, memory-read-line/write-invalidate. The command to be used is programmable through bits 6 and 7 of the mas_rs register (see Table 5). For memory commands (default), the address issued on PCI is a word address with bits (1:0) set to 00, indicating linear incrementing mode. For I/O and configuration commands, the full 32-bit address, as generated by the load/store instruction, is propagated to PCI.
- **PCI configuration cycles:** Configuration commands can be issued by setting mas_rs(7:6) to the appropriate value (see above). The PCI standard [1] suggests to connect the chip-select for configuration cycles (IDSEL#) directly to AD[31:16]. However, the PCI address space available via the AHB slave is limited, and therefore PCI devices whose IDSEL# pin is connected to or derived from AD[31:28] can not be addressed in this way. It is recommended either to use only AD[27:16] (allowing for 12 PCI devices), to decode them from AD[27:16], or to use the DMA controller for configuration cycles (section 2.3).
- **Direction:** store instructions result in PCI write transactions, loads are mapped to PCI read.
- **Burst handling:** By default, all requests are translated into single cycle PCI transactions (address phase, followed by a single data phase), with the following exceptions:
  - *Linear incrementing store-word sequences* are translated into undetermined length PCI write bursts (up to a maximum of 255 words). The PCI burst mode is then maintained as long as possible, i.e., read/write direction is unchanged and the address $A_{n+1} = A_n + 4$. When the sequence is discontinued, the PCI burst stops with a last data phase (byte enables 1111).
  - *Double word load/store requests* are optionally executed as 2 word bursts (one address phase, two data phases) on PCI, depending on the configuration bits mas_rs(1) for reads and mas_rs(2) for writes, see Table 5. The double word mode accelerates the transfer on the PCI side except in cases, where linear incrementing bursts are done by subsequent store-double instructions ($A_{n+1} = A_n + 8$), in this case mas_rs(2) should be set to 0. It is in general recommended to set both bits to 1 and to use the DMA to transfer large data blocks.
- **Write buffer:** Write requests are posted to the PCI core, even to non-consecutive addresses (the PCI core has a separate request FIFO). This allows the instruction to complete even before the PCI cycle.
- **Fast back2back cycles:** Can be issued by writing 1 to bit 12 of mas_wc. Note: The PCI core supports only fast back2back cycles to the same target, and the feature works only, if it is enabled in the status_command register and if the target(s) support it.
- **Address range:** Since the PCI interface is not the only slave on the AHB bus, its address range (on AHB and on the PCI bus) is limited to addresses between 0xA0000000 and 0xF0000000. No paging register is provided, to re-map the AHB address range to different PCI addresses. PCI addresses outside of this predefined range can be accessed only via DMA (section 2.3). Note also that no special Sparc address space identifier (ASI) is used to access the PCI.
- **Cacheability:** The PCI address range is non cacheable (hard-wired). Any access to these addresses results therefore in a PCI transaction. If PCI memory can be changed from outside

LEON and using a C-compiler, variables/pointers accessing the PCI addresses should be declared as 'volatile' to prevent the compiler from optimising away read accesses.

- **Fatal (abort) and address parity errors:** The interface flushes all the current and other buffer requests and reports a fatal error, mas_ferr in int_st(5). The next AHB request will restart the PCI core.
- **Data parity errors:** During load/read transactions, one PCI parity error is recoverable in hardware. If the mas_rs(3) bit is set, the interface ignores the erroneous PCI data and retries the request. However, if the data parity error persists at the same address, it is considered to be unrecoverable, an AHB error is produced and int_st(6) is set. Parity error is also impossible in cases where the transaction is already finished on AHB when the error is detected/reported on the PCI bus. The parity error is then reported in int_st(4) and error recovery must be done in software.

Width	8	16	32	64
Assembler	**ld[s|u]b, stb**	**ld[s|u]h, sth**	**ld, st**	**ldd, std**
C-datatype	**char**	**short**	**int**	**long long**
A[2:0] = 000	0111	0011	0000	0000, 0000 (burst)
A[2:0] = 100	0111	0011	0000	*not aligned*
A[2:0] = x01	1011	*not aligned*	*not aligned*	*not aligned*
A[2:0] = x10	1101	1100	*not aligned*	*not aligned*
A[2:0] = x11	1110	*not aligned*	*not aligned*	*not aligned*

TABLE 1 MAPPING OF AHB ADDRESS/SIZE TO PCI BYTE ENABLES CBE#[3:0]

2.3 PCI Initiator: DMA Interface = AHB Master

The DMA controller executes an autonomous data transfer between the local memory (or any other AHB slave, except the PCI-AHB-slave) and a remote target on the PCI bus. The LEON processor only intervenes for the initiation of the transfer. **Please note that trying to access the PCI-AHB slave with the PCI-DMA-AHB controller is not allowed and leads to undefined behaviour.** The DMA controller acts as a master on both busses, it reads from one bus and writes to the other bus. Therefore, the terms of write and read are ambiguous, and it is more appropriate to use "to PCI" for a read from AHB and write to PCI, and "from PCI" in the other case. The DMA AHB Master uses the ports InitMasterIn/Out.

2.3.1 PROGRAMMING OF THE DMA

To enable DMA, mas_rs(0) must be set along with status_command(2). Since DMA is executed in background, aerr and dma interrupts in int_en(6, 7) should also be enabled to synchronise the application with start and end of the transfer (requires an appropriate interrupt handler). Whereas the data itself is transferred through AHB, the DMA is initiated through APB. The DMA interface executes only word-size transactions with all 4 byte lanes enabled, the PCI command type is configurable. Please note that, though the AHB slave (see 2.2) is enabled together with the DMA, during DMA transfer any request to the AHB slave will be retried. **Each DMA sequence** must program PCI start address, PCI command type, number of words to be transferred and the start address in the local memory by **three APB writes in the specified order** (see example in the function pci_dma in **Appendix A**):

1. Write the PCI start address of burst to mas_address (re-write each time you initiate DMA, even if the address is identical to the previous DMA request).
2. Write together the PCI command to mas_wc(11:8) and the number of words to mas_wc(7:0). → Writing to mas_wc passes address, word count and command to the PCI core and initiates the transaction on the PCI bus. Note, that the appropriate PCI command (as specified in [1]) must be written, i.e., a write command for a "to PCI" transfer and a read command for a "from PCI" transfer.
3. Write the start address in the local memory map to dma_address. Now, data transfer is started in background. In the rare case, where the PCI core has not accepted the request, the DMA state controller remains locked and aerr in int_st(6) is reported. If the acceptance by the PCI core was just delayed, rewriting dma_address may succeed. If the problem persists, reset the interface by writing –1 (0xFFFFFFFF) to mas_rs. Please note that **a DMA transaction may never cross a 1 kByte border:** The value represented by dma_address(9:2) + mas_wc(7:0) must be less than 256. If this restriction is not respected, the data transfer stops at the 1 kByte border, the PCI core is flushed and simultaneously the dma and aerr bits int_st(7) and int_st(6) are asserted (+ interrupt if enabled).

When the specified number of words were successfully transferred, the interface asserts the dma bit int_st(7) (+ interrupt if enabled) and goes back to idle state. Note, that "to PCI" transactions are posted to the FIFO, therefore a "to PCI" transaction may still be ongoing (on the PCI bus), when the dma interrupt occurs (signalling termination on the AHB bus). However, a proper transaction ordering is assured, a subsequent (read) request is not executed before the FIFO data was written to the PCI target.

2.4 PCI Target: AHB Master

The PCI target interface receives requests originated by remote PCI initiators (masters) and transfers the data via its AHB-Master. The PCI target uses the AHB ports TargetMasterIn/Out. The interface works in a similar way, as the DMA controller described in section 2.3 above. In fact, this is considered as DMA in some documentation. The difference is, that the target transfer is initiated by a remote device, and not by the local processor; to avoid confusion with the PCI-Master-DMA, the term DMA is not used in this context.

2.4.1 TARGET INITIALISATION

Target data transfer is executed in background, without intervention of LEON. However, the target functionality first needs to be enabled, following the specifications the PCI standard. In host-bridge mode (pci_host = '1'), the target is configured by LEON itself via APB registers, whereas in satellite mode, the configuration is done by a remote device via PCI configuration commands. Some registers are available only in APB (accessible via the PCI bus with I/O commands, see below).

In either case, the target is configured in the following registers:
- status_command register (Table 4), bits 0/1 for memory and I/O command response, bits 6/8 for check of and response to data and address parity errors.
- base address registers (Table 3) mem_base_address, dac_address, iobar
- APB registers tar_pa, tar_sc(7) (Table 5)

2.4.2 ADDRESS CONFIGURATION

Two types of base address registers (BAR) are implemented, one for memory commands, and one for I/O commands. The address space occupied by a BAR on PCI is defined by the LSB of the BAR. For example, if a BAR implements bits 31:8 of the address, bits 7:0 are not decoded by the PCI core, and the BAR occupies 2^8 = 256 byte on PCI. The upper bits (31:8 in the example) are relocatable on the PCI bus and configured by the system boot routines. Since the PCI address map is unpredictable, and it is unwanted to move the target area in the local (AHB) address space along with the PCI area, the upper bits can not be used as local address and must be mapped to the local address space.

The address generation for PCI memory commands is illustrated in Figure 3. One memory BAR exists, occupying a (byte-) address space of 24 bit (16 MByte). Address, byte enables and the hit flag are propagated through the FIFO. The 8 MSB are then mapped in the 32 bit (local) AHB address space to the page address in the APB register tar_pa(31:24) (Table 5). The default value for tar_pa is 0x40000000, mapping the PCI target to the memory controller of LEON. The two LSB of the AHB address and the size information are generated from the byte enable pattern.

The BAR is defined as '64-bit registers', and 40-bit addressing with dual address cycles (DAC) is ensabled. Addressing with DAC cycles is not further explained in this document, please refer to [1].

Notes:

- Trying to access the PCI-AHB slave with the PCI-target AHB master (i.e. mapping the BAR to 0xA0xxxxxx – 0xFFxxxxxx) is not allowed and may lead to undefined behaviour.

The IO-BAR defines a 10 bit (= 1 kByte) I/O-address space. It is (hard-)mapped to the local address 0x80000000, and therefore provides access to all LEON (APB) registers for external (PCI) devices. In a multiprocessor system, availability is improved: After a failure of the local processor, another processor connected to the PCI bus can analyse the status of the faulty processor, or even take control over its peripherals through the PCI bus. Also all the APB registers of the PCI interface are accessible in this way, allowing (for example) a remote PCI-initiator to relocate the memory BAR by writing to tar_pa. On the other hand, this access bears the risk of access violations (a register configuration 'known' to the local processor being modified by a remote device etc.), and it should be handled with care. The access can be disabled by clearing the PCI status_command bit 0.

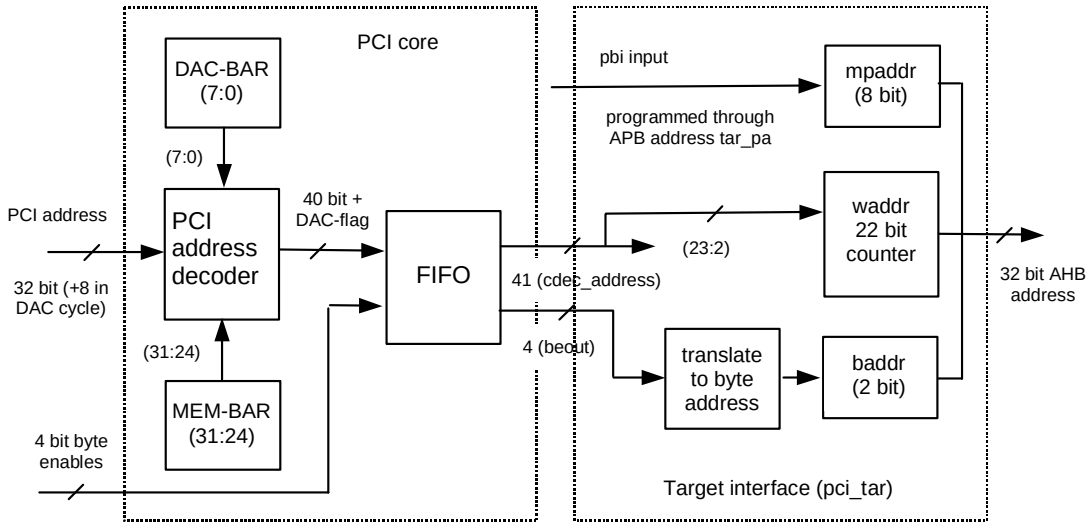

FIGURE 3: ADDRESS TRANSLATION FOR MEMORY COMMANDS IN THE PCI TARGET INTERFACE

2.4.3 ERROR REPORTING

PCI Error processing is implemented according to the PCI standard in the status_command registers (see Table 4, accessible locally via APB host-bridge mode, or remotely through PCI configuration cycles in satellite mode), and by the pin pci_perr and pci_serr. A few additional events covered by the status/interrupt APB registers (Table 5) in the bits 0..3.

- The interface translates the PCI write byte-enable patterns into an AHB byte address and AHB size info. The following byte enable patterns are supported: 0111 1011 1101 1110 (single byte), 0011 or 1100 (half word), or 0000. The byte enables 1111 are ignored, this pattern is frequently used as a dummy write cycle. Other byte enable combinations are ignored, and the berr status bit int_st(2) is set (optional interrupt).
- Received data parity errors are detected and reported in int_st(1), with optional interrupt. The target interface by default ignores these erroneous data. Nothing is written into memory, unless the configuration bit perr_ign = tar_sc(7) is set.
- AHB errors are reported by the aerr status bit int_st(3), with optional interrupt.
- PCI system errors (pci_serr by any device on the bus) are reported by the serr bit int_st(0), with optional interrupt.
- By writing ‘all-ones’ to tar_sc(6:0), the interface can be reset, the target FIFO’s are flushed.

2.4.4 TRANSACTION ORDERING

As specified in the PCI standard, delayed read functionality is implemented, obeying to the following rules:

- The interface stores one delayed read at a time. When a read request was retried (because local data not yet available), the interface remains locked for any other target read (targeting different addresses). The initiator of the original read has to repeat its request to the same address.

- A retried (delayed) read can be interrupted by one or more PCI write accesses. The PCI standard requires this write command to be processed first, to prevent a system lock-up.
- Meanwhile, the interface will prefetch read-data into the TXMT FIFO. After the (interfering) write, when the read request is repeated, and the requested data is available in the FIFO the delayed transfer completes normally.

All target read accesses are generally prefetching, also reads with I/O command. Once a start address is given, the interface prefetches up to 8 words into the TXMT FIFO. After the last required data word was transferred to PCI, the PCI core automatically flushes the FIFO to discard the unused prefetched data. The interface assumes the complete local address space to be 'prefetchable', defined here as the fact, that reading from an address does not alter the data. This behaviour is to be considered if non-prefetchable devices (for example the UART's) shall be read through the PCI target.

2.5 Interrupt Support

2.5.1 INTERRUPTS FROM SATELLITE DEVICES

Interrupt support according to PCI standard release 2.3 [1] is not implemented.

Only the 8-bit interrupt line register in the PCI CONFIG space exists and can be programmed by the driver for interrupt routing information.

The handling of (PCI-satellite) interrupts at board/system level is implementation dependent, and the interface does not provide further support. Any connection between PCI interrupts and LEON interrupts must be done outside the processor.

2.5.2 INTERRUPTS FROM THE PCI INTERFACE TO LEON

The PCI standard foresees a single parity check, by which bus-errors can be detected, but not corrected. Read data parity errors can eventually be retried by the hardware, as described in section 2.1.1. In other cases, recovery must be done in software. Therefore, events, which occur in the PCI interface or on the PCI bus, are saved in status bits, and optionally, the irq pin of *pci_is* is asserted. In LEON, this pin is connected to IRQ 14 of the interrupt-controller. Different events can be selected to assert the interrupt by a configuration register (int_en) and the interrupt handler can read the interrupting event (IE) in the status register (int_st). Furthermore, interrupts can be forced (for SW test purposes) by writing to int_test. Examples of how to use the error bits and interrupts are included in the SW test programs.

For generation of interrupts, retrieving and clearing the event, the following rules apply:
- To each IE, one particular bit is assigned in int_en, int_st and int_test, see Table 5.
- In any case, the IE is latched in its bit in int_st, even if the corresponding interrupt is disabled.
- A particular IE generates IRQ 14 only, if the corresponding bit in int_en is set to 1.
- The interrupt handler can retrieve the IE by reading int_st. The event is cleared by writing 1 to its bit in int_st. It is suggested to clear only those bits, which were 1, when int_st was read, to prevent the loss of IE, which may occur in the time span between reading and writing to int_st. This is easily done by re-writing exactly the read value.

- The interrupt can be forced by writing to int_test. For each bit of int_test, to which a 1 is written, the corresponding bit of int_st will be set.

2.6 Not Included Functionality

Since the AHB slaves of LEON do never issue retry, both masters in the current design do not support retry. The AHB masters do not use the master lock and the hprot features, and the AHB slave never issues a split response.

The APB address decoding is on the select signal PSEL and the bits (7:2) of PADDR. All other bits of PADDR are ignored. Hence, the APB master must take care that PSEL is asserted in a proper way.

PCI interrupt acknowledges, special cycle and memory read multiple commands can be initiated only via the APB-PCI interface. Initiation via the AHB devices (DMA or Slave) is not possible. Dual Address Cycles (DAC) are limited to 40 bit addresses (only the 8 LSB of the DAC register are implemented).

Power management: PCI power management is not implemented. The 'extended capabilities' bit 4 of the PCI status register is 0. However, the capabilities pointer (address 0x34) and the pmc register at address 0xDC are implemented as read-only registers.

2.7 Bugs and Application Notes

The following application notes/warnings should be considered:

- **Note on initiator flushing** (applicable to AHB slave, DMA or APB mode)**:** Some types of errors imply a FIFO flush, or the FIFO's can be flushed manually writing to the respective bits in the mas_rs register. In some cases the PCI core may resume a previously recorded (burst) transaction and fetch new data, the flush operation is then incomplete. It is therefore recommended after a flush (whether implicit or manual) to check carefully the idle state of the interface (should be 0x30 in the mas_st register), and eventually do a complete restart of the interface (write –1 = 0xFFFF to mas_rs and re-initialise mas_rs).

- **Note** that any target read request is retried (on PCI), if target write data remains in the TRCV FIFO. A sequence of write transactions to a target device (whether posted before or after a read request from the same device) may therefore block a read. The writes have priority (cf. to the PCI standard) and the read is retried, as long as there are new write transactions. This deadlock is observed in particular at low processor frequency (the backend has not enough time to write data to the memory), it can be avoided by reducing the frequency of write transactions posted to the target device.

- **Note** that **all** target reads (memory, IO etc.) are prefetched (up to the FIFO size) on the AHB bus, even if only one word is requested on PCI. This may result in data loss, if read-destructible registers (e.g. UART RX ports) are in proximity of the address to be accessed.

- Never try to do loop-back transactions on PCI, i.e. access the PCI-Initiator AHB-slave from either the PCI-Target port, or from the PCI-DMA AHB-master. This leads to undefined behaviour. Do not program the target page address to the range 0xA…0xF, and if you wish to do DMA from one remote PCI target to another, please first 'DMA' the data to local memory, then 'DMA' it back to PCI. *AHB error is generated in future code releases.*

The following problems have been reported ([10]):

- Though compliant to the PCI standard, the 64-bit BAR may pose compatibility problems with current PC BIOSes, making it impossible to configure the device with the plug-and-play boot routines. However, in an operating system which allows to re-scan the PCI bus (e.g. Linux), the configuration is possible. *Fixed in future code releases.*

- DMA read operations may fail if the dma_address was not written very closely after the mas_wc has been written (i.e. before the read was completed on PCI). *Fixed in future code releases.*

- The LEON SDRAM controller can not handle a burst access from the PCI interface. *Fixed in future code releases.*

3 THE PCI ARBITER

The PCI arbiter is not included in the PCI interface. It is part of the LEON model, and it is licensed and delivered together with LEON. This arbiter is totally independent from the PCI interface. In order to use it together with the PCI interface, one of the arbiter's pci_arb_[req|gnt]_n pairs must be connected externally (off-chip) to the pci_[req|gnt_in]_n pair of the interface.

The hardwired configuration options:

- Arbiter for 4 PCI agents, numbered 0..3
- Arbitration is by round-robin, nested in two different priority levels (0 = high, 1 = low). Assignment of the bus agents to the levels is APB programmable at the address 0x80000280. Exception is agent number 3, which is always in the lower priority level (1).

In the APB register (see Table 2), bits 0..2 indicate the priority of agents 0..2. Bit 3 is don't care at write and reads 1, all other bits are don't care and read 0. Reset value is a one for all bus agents, which means that by default all agents are in the same (lower) priority level.

Nested round-robin means, that the agents are grouped in two separate round-robin loops. In first place, the loop with level 0 is executed. At each complete round-turn in level 0 (or if no request of a device placed in level 0 is pending), one step is done in level 1. As an example, with agents 0 and 1 in level 0, and agents 2 and 3 in level 1, the sequence of would be 0 – 1 – 2 – 0 – 1 – 3 – 0 – 1 – 2 – 0 – 1 – 3 etc., if all agents are permanently requesting the bus. Hence, the following probabilities of access are implemented:

- All agents in the same level have equal probability
- All agents in level 1 together have the same probability of access as one agent in level 0.
- If no agent is in level 0 (default at reset), or no agent in level 0 has a request, all agents in level 1 are granted with equal probability

Re-arbitration occurs at the following conditions:

- Frame_n is asserted (low), and another master has requested the bus. Only one single re-arbitration takes place during a burst, and the handover only becomes effective, when the current owner deasserts frame_n. No latency control is performed by the arbiter, latency counters – by the PCI standard – are implemented in each PCI master.
- At bus-idle and no request pending, as soon as a new request occurs. A turnover cycle is required by the standard to prevent collisions with the previous owner. "idle" is assumed, when frame_n is high for more than 1 cycle.
- At "broken master" timeout (section 3.4.1 of the standard). Grant is removed from an agent, which has not started a cycle within 16 cycles after request (and grant). Reporting of such a 'broken' master is not implemented.

The bus is parked to agent 0 after reset, it remains granted to the last owner, until another agent requests the bus. When another request is asserted, re-arbitration occurs after one turnover cycle. The optional bus lock mentioned in the standard is not considered here and there are no special conditions to handle when lock_n is active.

Bit	31..4	3	2	1	0	Default value
Content	0x0000000 (read-only)	1 (read-only)	P_2 = priority of master 2	P_1 = priority of master 0	P_0 = priority of master 0	0x0000000F

TABLE 2 PCI ARBITER CONFIGURATION REGISTER AT APB-ADDRESS 0x80000280

4 REGISTER DESCRIPTION

All registers of the PCI interface are connected to the internal APB bus and mapped to the address 0x800001XX (except the PCI arbiter control register on 0x80000280, see Table 2). In the tables below, only the byte address (the 'XX') is specified. The register names given in this chapter correspond to the names defined by the C-structure pci_conf_regs in **Appendix A**.

Each PCI device has a mandatory set of configuration registers, which are specified in the PCI standard [1]. Their names and default values are summarised in Table 3. Details of the PCI status/command register at address 0x04 are given in Table 4. The PCI core allows writing to a subset of the four byte lanes (= 32 bit) through four active-low byte enable lines. Since APB only supports word writes, the byte enables must be written to the config_ben register (see Table 5), before writing to one of the registers of Table 3. It is suggested to clear config_ben immediately after a nonzero value was used. Identification constants, such as device_id, vendor_id, subsys_id, and rev_id are hard-wired as specified in the table. All not implemented registers are read as 0. Since power management capabilities are not implemented, bit 4 of the PCI status ("extended capabilities" = bit 20 of status_command) is 0. However, the cap_ptr and pmc register remain at addresses 0x34 and 0xDC as shown in the table in italics.

Note that in host-bridge configuration (pci_host = '1'), the registers of Table 3 are locally available to LEON at the addresses 0x80000100 to 0x80000140, whereas in satellite mode (pci_host = '0'), they are accessible via the PCI bus only (using configuration commands).

Note:

- A copy of the registers of Table 3 remains accessible even in satellite mode. **These registers have no impact on the functionality**, but they may be read to get constants like vendor-id etc.
- Even in host-bridge mode, the registers of Table 3 – like any other APB register - can be accessed remotely via PCI-IO commands, if the target is enabled for IO access.

Other registers of the PCI interface (not standardised in PCI [1]) are mapped to the APB addresses between 0x80000144 and 0x80000178. The registers are listed in Table 5. The address column gives the 8 LSB of the APB address. Bits (1:0) are supposed to be 0, but their actual value is ignored. Register bits not listed in this table are not-implemented and read as 0. Some registers of this block contain status information for debug purpose during hardware design and are not relevant for application programs. These are marked *reserved*.

Addr.	Register Name[1]	Description	Default value
00	device_id	Device- & vendor ID (ESA = 16E3). Read only.	0x1E0F16E3
04	status_command	According to PCI standard [1] Bit description see Table 4	0x02800000
08	class_revision	Processor device (0xB), Revision 1. Read only.	0xB01
0C	bist_head_ lat_cacheline	See PCI standard [1].	0
10	mem_base_address	Defines a 24 bit memory address space. Bits 23-0 are read-only.	0xC = prefetchable and 64 bit relocatable
14	dac_address	For dual address cycles. Dac_address(7:0) = bits 39-32 of the 40 bit PCI address in DAC mode.	0
18	iobar	Base register 10 bit IO address space. Bits 9-0 are read-only.	0x1 = IO space
1C - 28	reserved1[4]	*other registers, not implemented (read 0)*	0
2C	subsystem_id	Subsystem- and vendor ID (ESA = 16E3). Read only.	0x210316E3
30	exp_rom_base_addr	not implemented	0
34	*cap_ptr*	*index to the extended capabilities register. Read only*	*0xDC*
38	reserved2		0
3C	latency_interrupt	See PCI standard [1].	0
40	retry_trdy	Bits 15:8: Retry Timeout Value, sets the number of retries that the initiator will perform Bits 7:0: TRDY Timeout Value, sets the number of PCI clocks that the initiator will wait for TRDY	0x8080
DC	*Pmc*	*Power management capabilities: see* [1]*. Read only.*	*0x7E010001*

TABLE 3 PCI CONFIGURATION REGISTERS (SEE ALSO [1] AND [3])

Bit	PCI register	Description	RR, RW or RO[2]	Reset value
31	status 15	Parity error detected	RR	0
30	status 14	SERR asserted	RR	0
29	status 13	Master has terminated with master abort	RR	0
28	status 12	Master was terminated with target abort	RR	0
27	status 11	Target has signalled target abort	RR	0
26..25	status 10..9	Devsel timing	RO	01 (medium)
24	status 8	Master received/asserted PERR	RR	0
23	status 7	Target supports fast back2back	RO	1
22	status 6	User definable features	RO	0
21	status 5	66 MHz capability	RO	0
20	status 4	Power management capability	RO	0
19	status 3 interrupt status (optional, see section 2.5.1)	State of the interrupt in the device. Only when the Interrupt Disable bit (bit 10) in the command register is a 0 and this Interrupt Status bit is a 1, will the device's INTA# signal be asserted. Setting the Interrupt Disable bit to a 1 has no effect on the state of this bit.	RO	State at any determined by external input pcii.intr_status at any time.
18..16	status 2..0	reserved	RO	000
15..11	command 15..11	reserved	RO	000000
10	command 10 interrupt disable	0 enables, and 1 disables the assertion of the device's INTA# signal. Fed to external output pcio.intr_disable	RW (RO) (option, see 2.5.1)	0
9	command 9	Master can generate fast back2back	RW	0
8	command 8	Enable SERR driver	RW	
7	command 7	Address/data stepping on PCI bus	RO	0
6	command 6	Enable parity check	RW	0
5	command 5	VGA palette snooping	RO	0
4	command 4	Enable memory write and invalidate	RW	0
3	command 3	Enable special cycles	RO	0
2	command 2	Enable PCI master	RW	0
1	command 1	Enable target memory command response	RW	0
0	command 0	Enable target IO command response	RW	0

TABLE 4 PCI STATUS COMMAND REGISTER AT ADDRESS 0X04 (SEE ALSO [1])

[1] Names are identical to the names used in the C data structure pci_conf_regs defined in **Appendix A**
[2] RR = Read and Reset by writing 1, RW = Read Write, RO = Read Only

Addr.	Reg. Name[3]	Bit(s)	Description	Default value
44	config_ben	3-0	Byte enables for writes to the registers of Table 3. Cf. to [1], each of the 4 bits is assigned to 1 byte lane.	0 = all 4 byte lanes enabled
48	mas_address	31-0	PCI start address for PCI initiator transactions in DMA mode. First parameter of DMA/APB programming sequence.	X = no reset
4C	*Reserved*			
50	mas_wc	Specifies the PCI transaction in DMA mode. Writing to this register initiates the transaction on PCI. Second parameter in the DMA programming sequence.		
		12	Use back2back-mode: can be written to 1, if this transaction is to the same target, as the last one. Note: works only, if the core is enabled for back2back mode.	0
		11-8	PCI command (cf. to [1]) to be used in DMA mode	0
		7-0	Word count: number of words to be transferred.	0
54	*reserved*			
58	mas_rs	Initiator reset/set. Read/write config bits. Writing all-ones resets the interface: flush FIFOS, reset byte enables, and terminate the AHB-slave cycle. A DMA burst is terminated with an AHB error, int_st(6).		
		7-6	Specifies the 2 MSB of the PCI command, which is used by the AHB slave interface. The 2 LSB are generated internally (11 for writes and 10 for reads). The following commands can be selected by writing to bit 7-6: 00 = IO-read/write, 01 = mem-read/write, 10 = configuration-read/write, 11 = mem-read-line/write-invalidate	01 (mem-rd/wr)
		3	Perr-retry enable: if set, the AHB slave (word interface mode) generates one retry on reads with a parity error. If the error on the retried address persists, an AHB error is generated, to prevent deadlocks. LEON, when receiving the AHB error is trapped (trap 0x9, memory exception).	0
		2	LEON double word write: if set, double word writes (SPARC store double) execute as 2 word bursts on the PCI bus. Else double word writes execute as bursts of undefined length as long as the addresses are sequential.	0
		1	LEON double word read: if set, double word reads (SPARC load double) execute as 2 word bursts on the PCI bus. Else double word execute as 2 single reads.	0
		0	PCI initiator enable: 1 = AHB slave and DMA master are enabled, if also command 2 of the PCI status_command register is set to 1 (see Table 4).	0
5C	tar_pa	31..24	Target page address. Defines the 8 MSB of the memory page of 16 Mbytes, to which incoming PCI addresses are mapped (see section 2.4.2).	0x40 = start of the RAM area
		23..0	*reserved*	0
60	tar_sc	Target status and command register. Bit 7: Configuration, Bits 6..0: *status information for debug purpose*. By writing 111 1111 to bits 6:0 of this register, the target interface can be reset.		
		7	When this bit is set, TRCV data arriving with a parity error is written to the memory. Generation of the perr status bit and assertion of interrupt is not affected.	0 = do not write perr data
		6-0	Target status (011 0000) in idle state, write 111 1111 to reset the target interface and flush TXMT and TRCV FIFO	011 0000
64 68 6C	int_en int_st int_test	The PCI interface has one single interrupt output (IRQ14). Interrupts for different events can be enabled, by setting/clearing the corresponding bit in the int_en register. In any case (whether interrupt is enabled or not), interrupting events are saved in the corresponding bits in int_st. The status bits are cleared by writing 1. Writing to int_test, asserts a test interrupt (non maskable by int_en), and sets the corresponding bit in int_st.		
		7	DMA finished. If bit 7 and 6 are asserted simultaneously, the DMA transfer was aborted either due to a 1 kByte border (AMBA rule), or by a PCI fatal error, or by a general interface reset (by writing mas_rs = 'all-ones').	0
		6	PCI-initiator (master) interface AHB error has occurred	0
		5	PCI-initiator (master) has reported fatal error	0
		4	PCI-initiator has reported parity error (can occur on read and write transactions).	0
		3	AHB error occurred in a PCI target transaction	0
		2	Target byte enable error (invalid byte enable received from PCI bus)	0
		1	Target parity error (data with perr received from PCI bus)	0
		0	The system error (pci_serr) was asserted on the PCI bus.	0
70	*reserved*			
74	*reserved*			
78	dma_address	In DMA mode: start address of a DMA burst in local memory. Last parameter of DMA programming sequence.		

TABLE 5 APB REGISTERS OF THE PCI INTERFACE

[3] Names are identical to the names used in the C data structure pci_conf_regs defined in **Appendix A**

5 EXTERNAL INTERFACE DESCRIPTION

5.1 Pin Description

The external pins of the interface are summarised in Table 6. For a detailed description of these signals, please refer to [1]; the table gives the respective pin names of the standard.

Pin Name	Direction	Width	Name in PCI [1] standard	Remarks
pci_rst_in_n	in	1	RST#	Asynchronous Reset for PCI clock domain, rising edge synchronised internally to PCI clock
pci_clk_in	in	1	CLK	33 MHz PCI clock (no 66 MHz capability)
pci_gnt_in_n	in	1	GNT#	
pci_idsel_in	in	1	IDSEL	ignored in host-bridge mode
pci_lock_n	inout	1	LOCK#	used as input only (never drives)
pci_ad	inout	32	AD[31::0]	
pci_cbe_n	inout	4	C/BE[3:0]	
pci_frame_n	inout	1	FRAME#	
pci_irdy_n	inout	1	IRDY#	
pci_trdy_n	inout	1	TRDY#	
pci_devsel_n	inout	1	DEVSEL#	
pci_stop_n	inout	1	STOP#	
pci_perr_n	inout	1	PERR#	
pci_par	inout	1	PAR	
pci_req_n	inout	1	REQ#	functionally output only, but tristated at reset
pci_serr_n	inout	1	SERR#	open drain output
pci_host	in	1		bootstrap pin: ‘1’ = host-bridge, ‘0’ = satellite mode
pci_arb_req_n	in	4		PCI arbiter request inputs
pci_arb_gnt_n	out	4		PCI arbiter grant outputs

TABLE 6 EXTERNAL PINS OF PCI INTERFACE AND PCI ARBITER

5.2 Clocks, Timing and Resets

The PCI interface is connected to two independent clock domains: LEON **clk** and **pci_clk_in**. Synchronisation between these domains is done in the internal FIFOs. PCI clock frequency is nominally 33 MHz (no 66 MHz capability), and synchronisation is done in a way to allow the LEON clock to operate at a lower (< 33 MHz) as well as at a higher (> 33MHz) frequency with respect to the PCI frequency.

All the PCI pins are in the PCI clock domain. The electrical (AC and DC) characteristics of the PCI pins relative to the PCI clock is specified in [1].

The PCI interface uses two asynchronous active-low resets:

- **pci_rst_in_n** for the part connected to the PCI bus (including the PCI sequencer)
- **resetn,** the main LEON reset for the part connected to the AMBA bus

The following remarks should be considered:

- The two reset inputs are independent, for a complete reset, both lines should be activated.
- The PCI configuration registers of Table 3 are always reset with the PCI reset, even in the host-bridge (pci_host = '1') mode, where they are accessible via APB.
- The rising edges of both reset signals are synchronised internally to their respective clock domains.

APPENDIX A SOFTWARE EXAMPLE

The following C source code gives some programming examples for the PCI interface. Please note that **this code is UNTESTED and NO OPERATIONAL SOFTWARE!!!**

```
/*************************************************************************/
/* DEFINITION of PCI command types                                       */
/*************************************************************************/
#define IACKN_C        0
#define SPECIAL_CYC_C  1
#define IO_RD_C        2
#define IO_WR_C        3
#define MEM_RD_C       6
#define MEM_WR_C       7
#define CONF_RD_C      10
#define CONF_WR_C      11
#define MEM_RD_MUL_C   12
#define DAC_C          13
#define MEM_RD_LIN_C   14
#define MEM_WR_INV_C   15

/*************************************************************************/
/* PCI register structure                                                */
/*************************************************************************/
typedef struct pci_conf_regs
{

  volatile unsigned int device_id;                /* 00 */
  volatile unsigned int status_command;           /* 04 */
  volatile unsigned int class_revision;           /* 08 */
  volatile unsigned int bist_head_lat_cacheline;  /* 0C */
  volatile unsigned int mem_base_address;         /* 10 */
  volatile unsigned int dac_address;              /* 14 */
  volatile unsigned int iobar;                    /* 18 */
  volatile unsigned int reserved1[4];             /* 1C-28 */
  volatile unsigned int subsystem_id;             /* 2C */
  volatile unsigned int exp_rom_base_addr;        /* 30 */
  volatile unsigned int cap_ptr;                  /* 34 */
  volatile unsigned int reserved2;                /* 38 */
  volatile unsigned int latency_interrupt;        /* 3C */
  volatile unsigned int retry_trdy;               /* 40 */
  volatile unsigned int config_ben;               /* 44 */
  volatile unsigned int mas_address;              /* 48 */
  volatile unsigned int reserved3;                /* 4C */
  volatile unsigned int mas_wc;                   /* 50 */
  volatile unsigned int reserved4;                /* 54 */
  volatile unsigned int mas_rs;                   /* 58 master reset/set */
  volatile unsigned int tar_pa;                   /* page addr 5C */
  volatile unsigned int tar_sc;                   /* status+cmd 60 */
  volatile unsigned int int_en;                   /* interr. enable 64 */
  volatile unsigned int int_st;                   /* interr. status 68 */
  volatile unsigned int int_test;                 /* interr. test 6C */
  volatile unsigned int reserved5;                /* 70 */
  volatile unsigned int reserved6;                /* 74 */
  volatile unsigned int dma_address;              /* dma local address 78 */


} pci_apb_t;


pci_apb_t *pciregs = (pci_apb_t *) 0x80000100; /* PCI registers */
volatile pci_apb_t *rpciregs = (pci_apb_t *) 0xB0000100; /* Remote PCI registers */


/*********************************************************************/
/* MACROS FOR PCI-INTERRUPTS                                         */
/*********************************************************************/
#define PCI_SERR_INT  (1<<0)
#define TAR_PERR_INT  (1<<1)
#define TAR_BERR_INT  (1<<2)
#define TAR_AERR_INT  (1<<3)
#define MAS_PERR_INT  (1<<4)
#define MAS_FATAL_INT (1<<5)
#define MAS_AHB_INT   (1<<6)
#define MAS_DMA_INT   (1<<7)
```

```
/*********************************************************************/
/* Interrupt wait                                                    */
/*********************************************************************/
wait_interrupt(int_type) unsigned int int_type;
{
  /*
     put the appropriate system call here to suspend the process,
     until the respective PCI interrupt was asserted. Wake-up
     condition is irq 14 with bit no. int_type set in pciregs->int_st
  */
}

/*********************************************************************/
/* WRITE to the LOCAL PCI COMMAND REGISTER                           */
/* (without altering status, in HOST-BRIDGE MODE)                    */
/*********************************************************************/
write_pci_cmd_register(cfg) unsigned short cfg;
{
  pciregs->config_ben = 8 + 4; /* byte enables for command reg */
  pciregs->status_command = (unsigned int) cfg;
  pciregs->config_ben = 0; /* reset byte enables */
}

/*********************************************************************/
/* Start DMA by writing to DMA controller APB.                       */
/* The function returns the interface status after the transaction.  */
/*********************************************************************/
unsigned int pci_dma(paddr, laddr, size, cmd)
     unsigned int paddr, laddr, size, cmd;
{

  pciregs->mas_address = paddr; /* write start address on PCI */
  pciregs->mas_wc = (cmd << 8) | size; /* start PCI with length and command */
  pciregs->int_st = MAS_AHB_INT | MAS_DMA_INT;       /* clear AHB interrupt bit */
  pciregs->dma_address = laddr;     /* start AHB master with local address */

}

/*********************************************************************/
/* INITIALISE LOCAL MASTER INTERFACE in HOST-BRIDGE MODE             */
/* cfg = 0x254 to initialise fback2back, parity, mwinv, PCI master   */
/* Enables AHB slave and DMA master state machines                   */
/*********************************************************************/
init_local_master() {
      write_pci_cmd_register(0x254);
    pciregs->mas_rs &= ((MEM_RD_C << 4) | 0xF);
}

/*********************************************************************/
/* INITIALISE REMOTE TARGET INTERFACE, supposing that this is also   */
/* running on a LEON processor                                       */
/* This is also an example of how to use the DMA                     */
/* After this initialisation, when the IOBAR is set to 0xB...,       */
/* APB regs of the remote processor are accessible through PCI,      */
/* using f.ex. the pointer *rpciregs declared above.                 */
/*********************************************************************/
init_remote_target(devno) unsigned int devno;
{
  volatile unsigned int remcfg;
  volatile unsigned int rembar[3];

  remcfg = 0x143; /* target I/O + memory command response with serr/perr */
  pci_dma((1 << (16+devno)) + 4, &remcfg, 1, CONF_WR_C);
  wait_interrupt(MAS_DMA_INT);

  rembar[0] = 0xA0000000;  /* configure memory BAR */
  rembar[2] = 0xB0000000;  /* configure IO BAR */

  /* transfer 3 words with DMA */
  pci_dma((1 << (16+devno)) + 16, &rembar, 3, CONF_WR_C);
  wait_interrupt(MAS_DMA_INT);

  /* NOTE: configuration cycles can be issued with the AHB slave as well,
     however, the address space is limited then to addresses >= 0xA0000000,
     this means that address lines 28:31 can not be used for idsel decoding */
}
```