Wupper - a Xilinx Virtex-7 PCIe Engine

Frans Schreuder, Andrea Borga, Oussama el Kharraz Alami, Roel Blankers

27-09-2021

OpenCores
www.opencores.org

WISHBONE COMPATIBLE

Contents

Revision History

Revision	Date	Author(s)	Description
4.0	27-09-2021	F. P. Schreuder	Updated documentation with updated Wupper core and register map v2.0
3.0	17-05-2019	F. P. Schreuder	Updated some descriptions from the FELIX repository
2.4	06-11-2017	R. Blankers	Added Wishbone bus to the register map
2.3	14-04-2015	F.P. Schreuder	Updated register map, added description for drivers
2.1	14-04-2015	A.O. Borga	Uniformed documentatio naming convention (PCIe Engine)
2.0	21-01-2015	F.P. Schreuder	Updated register map
1.9	09-01-2015	A.O. Borga	Reviewed
1.8	07-01-2015	F.P. Schreuder	Modifications for OpenCores
1.7	29-10-2014	A.O. Borga	Major global revision
1.6	28-10-2014	A.O. Borga	Updated PCIe coregen figures, modified appearance of paths throughout the text, fixed typos, updated the simulation and testing sections, added the interrupt handling section, reversed the order of this table
1.5	23-10-2014	F.P. Schreuder	Updated register map, figures and pepo commands, some cosmetic improvements
1.4	23-09-2014	J.C. Vermeulen	Updated figures
1.3	23-09-2014	F.P. Schreuder	Updated pepo commands for memory allocation
1.2	19-09-2014	F.P. Schreuder	Added All pages of Xilinx core wizard
1.1	19-09-2014	F.P. Schreuder	Applied modifications after Andrea's review
1.0	16-09-2014	F.P. Schreuder	created

1 Supported tools

Wupper had been tested on the following platforms and tools:

1. Operating systems:
    - Scientific Linux CERN 6, kernel 2.6
    - CentOS Linux 7, kernel 3.10
    - Ubuntu Linux 20.04, kernel 5.11
2. Xilinx Vivado:
    - 2020.1: migrated 09-2021
    - 2018.1: migrated 05-2019
    - 2015.4: migrated 02-2016
    - 2014.4: initial version
3. Xilinx FPGA:
    - Virtex-7 690T (PCIe Gen3x8)
    - Kintex Ultrascale XCKU115 (PCIe Gen3x8, 8+8 lanes bifurcated)
    - Kintex Ultrascale XCKU040 (PCIe Gen3x8)
    - Virtex Ultrascale+ XCVU9P (PCIe Gen4x8, Gen3x16) - For Gen4 support requires Vivado version 2018.1 exactly
    - Virtex Ultrascale+ XCVU37P (PCIe Gen4x8, 8+8 lanes bifurcated)
    - Versal Prime XCVM1802 (PCIe Gen4x8, 8+8 lanes bifurcated)

2 Introduction

Wupper[1] is designed for the ATLAS / FELIX project [2], to provide a simple Direct Memory Access (DMA) interface for the Xilinx Virtex-7 PCIe Gen3 hard block and has later been ported to the Kintex Ultrascale, Virtex Ultrascale+ and Versal Prime series. The core is not meant to be flexible among different architectures, but especially designed for the 256 and 512 bit wide AXI4-Stream interface [5] of the Xilinx Virtex-7 and Ultrascale FPGA Gen3 Integrated Block for PCI Express, and the Ultrascale+ and Versal Prime Gen4 Integrated Block for PCI Express (PCIe) [6, 7, 8, 9].

The purpose of Wupper is therefore to provide an interface to a standard FIFO. This FIFO has the same width as the Xilinx AXI4-Stream interface (256 or 512 bits) and runs at 250 MHz. The user application side of the FPGA design can simply read or write to the FIFO; Wupper will handle the transfer into Host PC memory, according to the addresses specified in the DMA descriptors. Several descriptors can be queued, up to a maximum of 8, and they will be processed sequentially one after the other. The number of descriptors (NUMBER_OF_DESCRIPTORS generic) plays an important role, it determines the total number of descriptors, but also the number of FIFO interfaces in the ToHost direction. The last descriptor is always dedicated for FromHost (DMA memory read from the server) transactions, all other descriptors are dedicated for ToHost transfers (Memory writes from the FPGA into the server memory).

Another functionality of Wupper is to manage a set of DMA descriptors, with an $address$, a $read/\overline{write}$ flag, the $transfersize$ (number of 32 bit words) and an $enable$ line. These descriptors are mapped as normal PCIe memory or IO registers. Besides the descriptors and the enable line (one per descriptor), a status register for every descriptor is provided in the register map.

For synthesis and implementation of the Xilinx specific IP cores, it is recommend to use the latest Xilinx Vivado release as listed in section 1. The cores (FIFO, clock wizard and PCIe) are provided in the Xilinx .xci format, as well as the constraints file (.xdc) is in the Vivado Format.

For portability reasons, no Xilinx project files will be supplied with the core, but a bundle of TCL scripts has been supplied to create a project and import all necessary files, as well as to do the synthesis and implementation. These scripts will be described later in this document.

[1] The person performing the act of bongelwuppen, the Gronings version of the famous Frisian sport of the Fierljeppen (canal pole vaulting) https://nds-nl.wikipedia.org/wiki/Nedersaksische_sp%C3%B6llegies#Bongelwuppen

Xilinx has introduced the AXI4-Stream interface [5] for the PCIe EndPoint core: a simplified version of the ARM AMBA AXI bus [**?**]. This interface does not contain any address lines, instead the address and other information are supplied in the header of each PCIe Transaction Layer Packet (TLP). Figure 1 shows the structure of the Wupper_core design. The Wupper_core is divided in two parts:

1. **DMA Control:**
   This is the entity in which the Descriptors are parsed and fed to the engine, and where the Status register of every descriptor can be read back through PCIe. Depending on the address range of the descriptor, the pointer of the current address is handled by DMA Control and incremented every time a TLP completes. DMA Control also handles the circular buffer DMA if this is requested by the descriptor (See 2.2).

   DMA control contains a register map, with addresses to the descriptors, status registers and external registers for the user space register map.

2. **DMA Read Write:**
   This entity contains two processes:

   - *ToHost / Add Header:* In the first process the descriptors are read and a header according to the descriptor is created. If the descriptor is a ToHost descriptor, the payload data is read from the FIFO and added after the header. This process also takes care of switching to the next active DMA descriptor, which is leading for selecting the MUX on the output ports of the ToHostFifo's.
   - *FromHost / Strip Header:* In the second process the header of the received data is removed and the length is checked; then the payload is shifted into the FIFO.

   Both processes can fire an MSI-X type interrupt by means of the interrupt controller when finished.

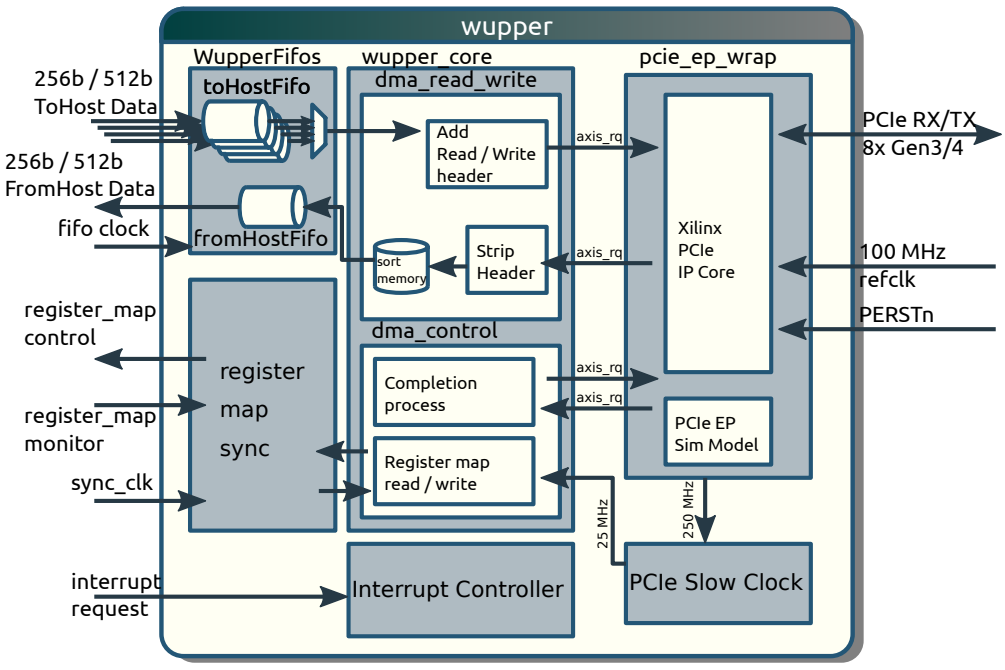

Figure 1: Structure of the Felix PCIe Engine

Figure 1 shows a synchronization stage for the IO and external registers, The user space registers are stored and processed in the 25 MHz clock domain in order to relax timing closure of the design. The synchronization stage synchronizes the register map again to the clock used in the application design (sync_clk).

The DMA Control process always responds to a request with a certain req_type from the server. It responds only to IO and Memory reads and writes; for all other request types it will send an unknown request reply. If the data in the payload contains more than 128 bits, the process will send a "completion abort" reply and go back to idle state. The maximum register size has been set to 128 bits because this is a useful maximum register size; it is also the maximum payload that fits in one 250 MHz clock cycle of the AXI4-Stream interface.

The add_header process selects the descriptor and sets the ToHostFifo MUX accordingly. Based on the descriptor content, it requests a read or write to/from the server memory. If the descriptor is set to ToHost, it also initiates a FIFO read and adds the data into the payload of the PCIe TLP (Transaction Layer Packet). When the descriptor is set to FromHost this process only creates a header TLP with no payload, to request a certain amount of data from the server memory that fits in one TLP.

The DMA FromHost process checks the size of the payload against the size in the TLP header, the data will be pushed into the FromHost FIFO.

2.1 DMA descriptors

Each transfer To and From Host is achieved by means of setting up descriptors on the server side, which are then processed by Wupper. The descriptors are set in the BAR0 section of the register map (see Appendix **??**). An extract of the descriptors and their registers is shown in Table 2 below. The register map in BAR0 has space for a maximum of 8 DMA descriptors, but the actual number of descriptors that are implemented is determined by the generic NUMBER_OF_DESCRIPTORS. The last active descriptor is always implemented with READ_WRITE set to 1 (read only) and the descriptors 0 to NUMBER_OF_DESCRIPTORS-2 are implemented as ToHost descriptors. The number of ToHost FIFOs is automatically determined by the same generic, as well as the ToHost FIFO depth. Setting NUMBER_OF_DESCRIPTORS to 5 (default in phase 2 FELIX) will result in 4 ToHost descriptors and FIFOs (descriptor 0..3) and a single FromHost descriptor / FIFO (descriptor 4).

Address	Name/Field	Bits	Type	Description
0x0000	DMA_DESC_0			
	END_ADDRESS	127:64	W	End Address
	START_ADDRESS	63:0	W	Start Address
0x0010	DMA_DESC_0a			
	RD_POINTER	127:64	W	server Read Pointer
	WRAP_AROUND	12	W	Wrap around
	READ_WRITE	11	R	1: FromHost/ 0: ToHost
	NUM_WORDS	10:0	W	Number of 32 bit words
...				
0x0200	DMA_DESC_STATUS_0			
	EVEN_PC	66	R	Even address cycle server
	EVEN_DMA	65	R	Even address cycle DMA
	DESC_DONE	64	R	Descriptor Done
	CURRENT_ADDRESS	63:0	R	Current Address
...				

0x0400	DMA_DESC_ENABLE	7:0	W	Enable descriptors 7:0. One bit per descriptor. Cleared when Descriptor is handled.

Table 2: DMA descriptors types

Every descriptor has a set of registers, with the following specific functions:

- DMA_DESC: the register containing the start ($start\_address$) and the end ($end\_address$) memory addresses of a DMA transfer; both handled by the server (software API).
- DMA_DESC_a: integrates the information above by adding (i) the status of the read pointer on the server side ($rd\_pointer$), (ii) the wrap around functionality enabling ($wrap\_around$, see Section 2.2 below), (iii) the FromHost ("1") and ToHost ("0") transfer direction bit ($read\_write$), and (iv) the number of 32 bits words to be transferred (num_words)
- DMA_DESC_STATUS: status of a specific descriptor including (i) wrap around information bits ($even\_pc$ and $even\_dma$), (ii) completion bit ($desc\_done$, (iii) DMA pointer current address ($current\_address$)
- DMA_DESC_ENABLE: the descriptors enable register (dma_desc_enable), one bit per descriptor

2.2 Endless DMA with a circular buffer and wrap around

In $single\,shot$ transfer, the DMA ToHost process continues sending data TLPs (Transaction Layer Packets) until the end address ($end\_address$) is reached. The server can check the status of a certain DMA transaction by looking at the $desc\_done$ flag and the $current\_address$. Another possible operation mode is the so- called $endless\ DMA$: the DMA continues its action and starts over (wrap-around) at start address ($start\_address$) whenever the end address ($end\_address$) is reached. The second mode is enabled by asserting the wrap-around ($wrap\_around$) bit. In this mode the server has to provide another address named server pointer ($PC\_read\_pointer$): indicating where it has last read out the memory. After wrapping around the DMA core will transfer To Host memory until the $PC\_read\_pointer$ is reached. The server read pointer should be updated more often than the wrap-around time of the DMA, however it should not be read too often as that would take up all the bandwidth, limiting the speed of the DMA transfer in progress. A typical rule of thumb to determine what "too often" means is that software should not update the pointer every clock cycle, but rather after processing a block of a few kB of data.

In order to determine whether Wupper is processing an address behind or in front of the server, Wupper keeps track of the number of wrap around occurrences. In the DMA status registers the even_cycle bits displays the status of the wrap-around cycle. In every even cycle (starting from 0), the bits are 0, and every wrap around the status bits will toggle. The $even\_pc$ bit flags a $PC\_read\_pointer$ wrap-around, the $even\_dma$ a Wupper wrap-around. By looking at the wrap-around flags the server can also keep track of its own wrap-arounds. Note that while in the $endless\ DMA$ mode ($wrap\_around$ bit set), the $PC\_read\_pointer$ has to be maintained by the server (software API) and kept within the start and end address range for Wupper to function correctly. Figure 2 below shows a diagram of the two pointers racing each other, and the different scenarios in which they can be found with respect to each other.

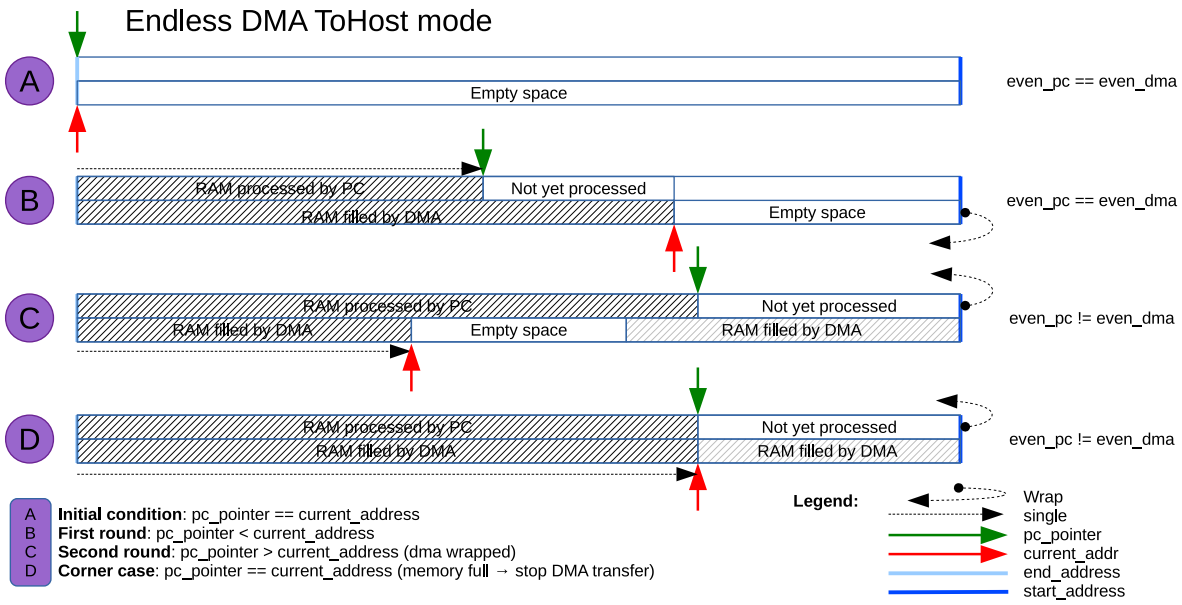

Figure 2: Endless DMA buffer and pointers representation diagram in ToHost mode

Looking at Figure 2 above, the following scenarios can be described:

- A : start condition, both the server and the DMA have not started their operation.
- B : normal condition, the PC_read_pointer stays behind the DMA's current_address
- C : normal condition, the DMA's current_address has wrapped around and has to stay behind the PC_read_pointer
- D : the server is reading too slow, the DMA is stalled because the server read pointer is not advancing fast enough, the DMA current_address has to stay behind.

If the DMA descriptor is set to FromHost, the comparison of the even bits is inverted, as the server has to fill the buffer before it is processed in the same cycle. In this mode the $pc\_read\_pointer$ is also maintained by the software API, however it is indicating the address up to where the server has filled the memory. In the first cycle the DMA has to stay behind the read pointer, when the server has wrapped around, the DMA can process memory up to $end\_address$ until it also wraps around.

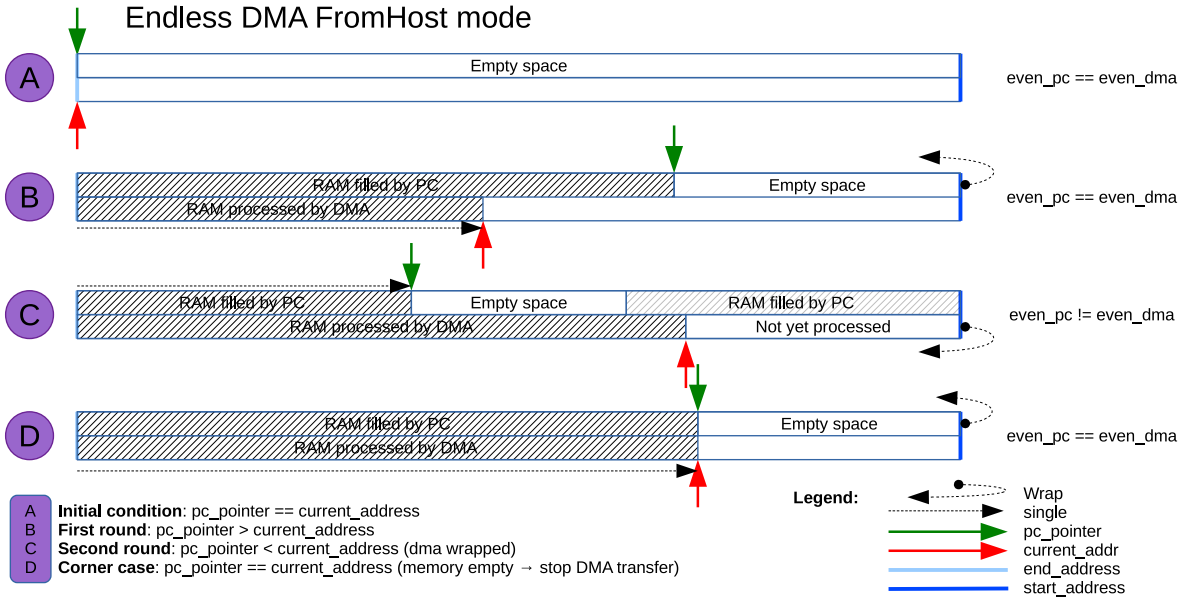

Figure 3: Endless DMA buffer and pointers representation diagram in FromHost mode

Looking at Figure 3 above, the following scenarios can be described:

- A : start condition, both the server and the DMA have not started their operation.
- B : normal condition, the DMA's current_address stays behind the PC_read_pointer
- C : normal condition, the PC_read_pointer has wrapped around and has to stay behind the DMA's current_address
- D : the server is writing too slow, the DMA is stalled because the server read pointer is not advancing fast enough, the DMA current_address has to stay behind.

2.3 Interrupt controller

Wupper is equipped with an interrupt controller supporting the MSI-X (Message Signaled Interrupt eXtended) as described in "Chapter 17: Interrupt Support" page 812 and onwards of [17]. In particular the chapter and tables in "MSI-X Capability Structure".

The MSI-X Interrupt table contains eight interrupts; this number can be extended by a generic parameter in the firmware. All interrupts are mapped to the data_available interrupt of the corresponding ToHost descriptor, formerly known as interrupt number 2. All the other interrupt sources have been removed since multiple ToHost descriptors were introduced. The interrupts are detailed in Table 3.

Table 3: PCIe interrupts

Interrupt	Name	Description
0	ToHost 0 Available	Fired when data becomes available in the ToHost FIFO 0 (falling edge of ToHostFifoProgEmpty)
1	ToHost 1 Available	Fired when data becomes available in the ToHost FIFO 1 (falling edge of ToHostFifoProgEmpty)
1	ToHost 2 Available	Fired when data becomes available in the ToHost FIFO 2 (falling edge of ToHostFifoProgEmpty)
3	ToHost 3 Available	Fired when data becomes available in the ToHost FIFO 3 (falling edge of ToHostFifoProgEmpty)
4	reserved	
5	reserved	
6	reserved	
7	reserved	

All Interrupts are fired when enough data has arrived in the ToHost fifo to fill at least one TLP of data. Once an interrupt has fired, it will not produce an additional interrupt until the SW_POINTER has been updated by the software.

All the interrupts can also be fired from the register INT_TEST, by setting the bitfield IRQ to the desired interrupt number. This write action will fire a single interrupt.

2.4 Sorting memory

In a bug report, it was made clear that some users (depending on the server hardware) experienced out of order FromHost memory transfers. For this reason a sorting memory has been added to the dma_read_write part of the Wupper core. This memory sorts pages of 4KB into the correct order before they are passed to the FromHostFifo.

2.5 Wishbone

The Wishbone protocol is a design method to connect IP cores with a common interface. The Wishbone can be used for soft, firm core or hard core IP and can be used with the VHDL language. The main purpose is to make an interconnection between IP cores and make it more compatible with each other.
The Wishbone bus is added because of a needed connection between the register map of the Wupper core and an external SLAVE. In this connection a Wishbone crossbar is added so that multiple SLAVEs can be attached. As a SLAVE example a 32 bits block memory was added. The memory has a data input to receive and a data output to send the data back to the crossbar.
The wupper_to_wb.vhd makes Wupper data Wishbone compatible. Also, two FIFOs are added to synchronize the Wupper clock with an external clock. One FIFO is to send data from Wupper to the crossbar. And one FIFO is to receive data from the crossbar to the Wupper.
The system controller makes the external clock and the external reset Wishbone compatible.

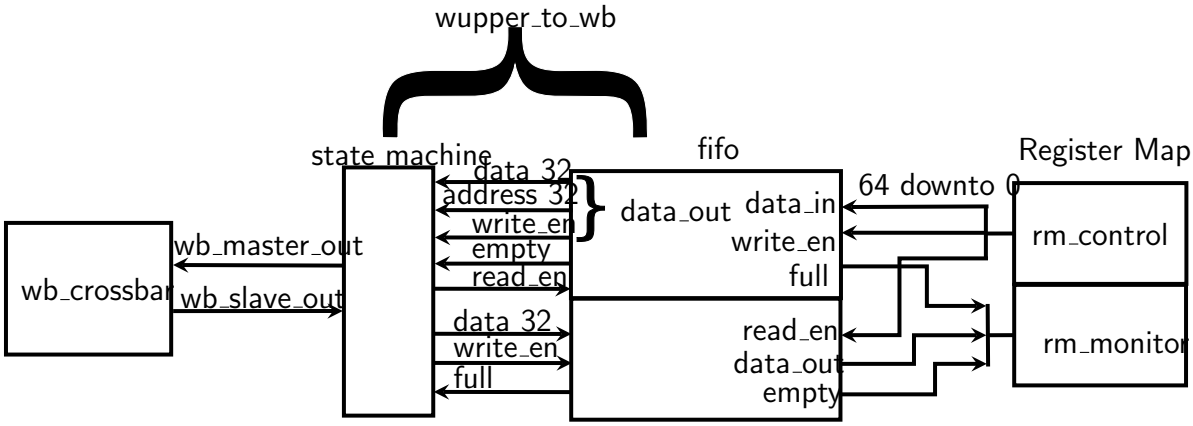

Figure 4: Block diagram of Wupper to Wishbone

3 Xilinx PCIe EndPoint Core

Wupper is based on the interface of the Virtex-7 FPGA Gen3 Integrated Block for PCI Express v3.0 [6]. This core is using a PCIe hard block in the Virtex-7 FPGA. The hard block is equipped with an AXI4-Stream interface.

3.1 Xilinx AXI4-Stream interface

The interface has the advantage that it has two separate bidirectional AXI4-Stream interfaces. The two interfaces are the requester interface, with which the FPGA issues the requests and the PC replies, and the completer interface where the PC takes initiative.

bus	Description	Direction
axis_rq	**R**equester re**Q**uest. This interface is used for DMA, the FPGA takes the initiative to write to this AXI4-Stream interface and the PC has to answer.	FPGA → PC
axis_rc	**R**equester **C**ompleter. This interface is used for DMA reads (from PC memory to FPGA), this interface also receives a reply message from the PC after a DMA write.	PC → FPGA
axis_cq	**C**ompleter re**Q**uest. This interface is used to write the DMA descriptors as well as some other registers.	PC → FPGA
axis_cc	**C**ompleter **C**ompleter. This interface is used as a reply inteface for register reads, as well as a reply header for a register write.	FPGA → PC

Table 4: AXI4-Stream streams

3.2 Configuration of the core

The Xilinx PCIe EndPoint core is configured as a PCI express Gen3 (8.0GT/s) End Point with 8 lanes and the Physical Function (PF0) max payload size is set to 1024 bytes. AXI-ST Frame Straddle is disabled and the client tag is enabled. All other options are set to default, the reference clock frequency is 100MHz and the only option for the AXI4-Stream interface is 256 bit at 250MHz, see Figures 5 to 15.

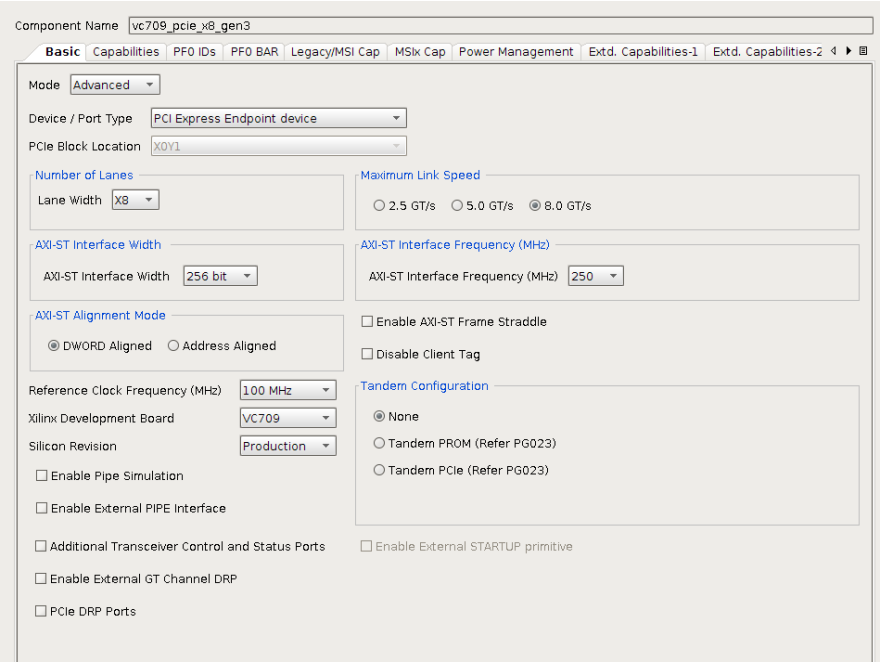

Figure 5: PCIe core configuration in Vivado [Basic]

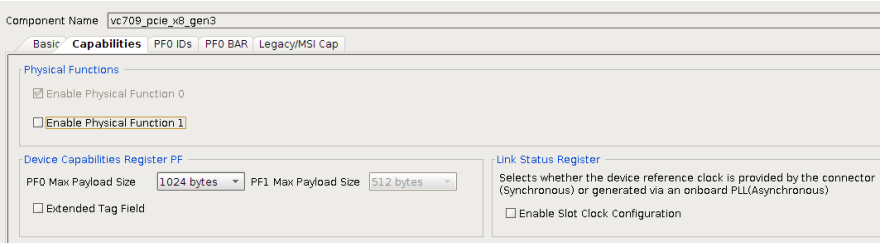

Figure 6: PCIe core configuration in Vivado [Capabilities]

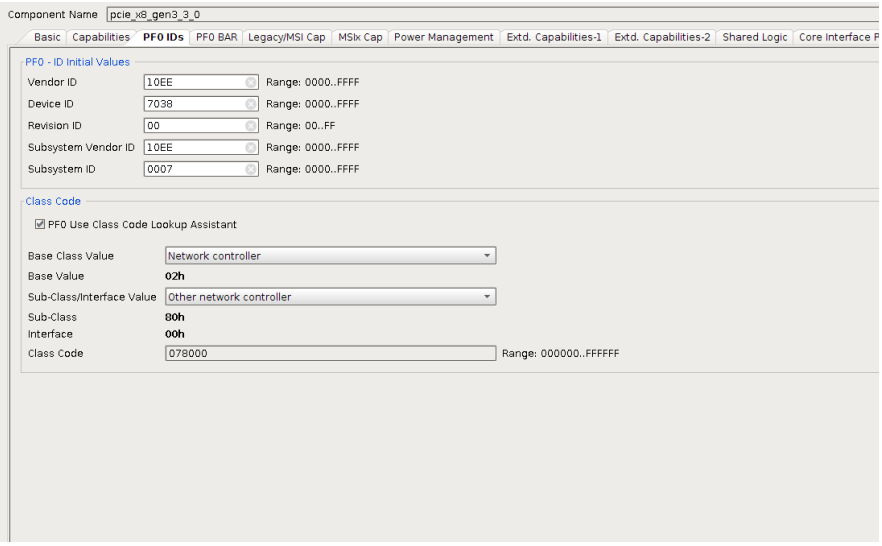

Figure 7: PCIe core configuration in Vivado [PF0 IDs]

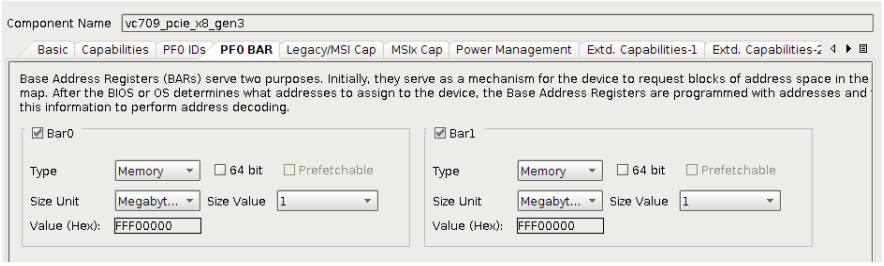

Figure 8: PCIe core configuration in Vivado [PF0 BAR]

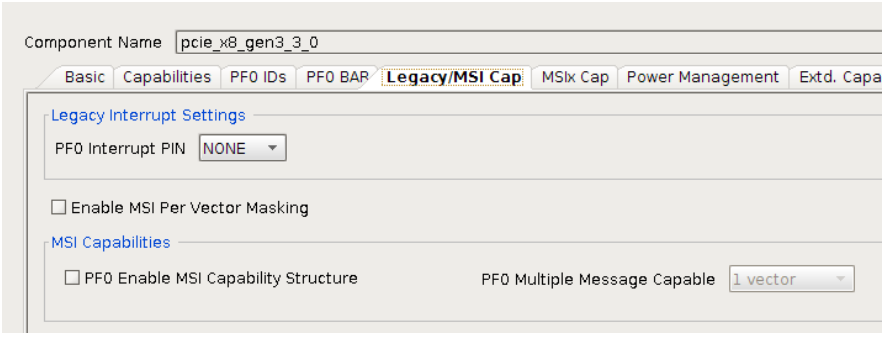

Figure 9: PCIe core configuration in Vivado [Legacy/MSI Cap]

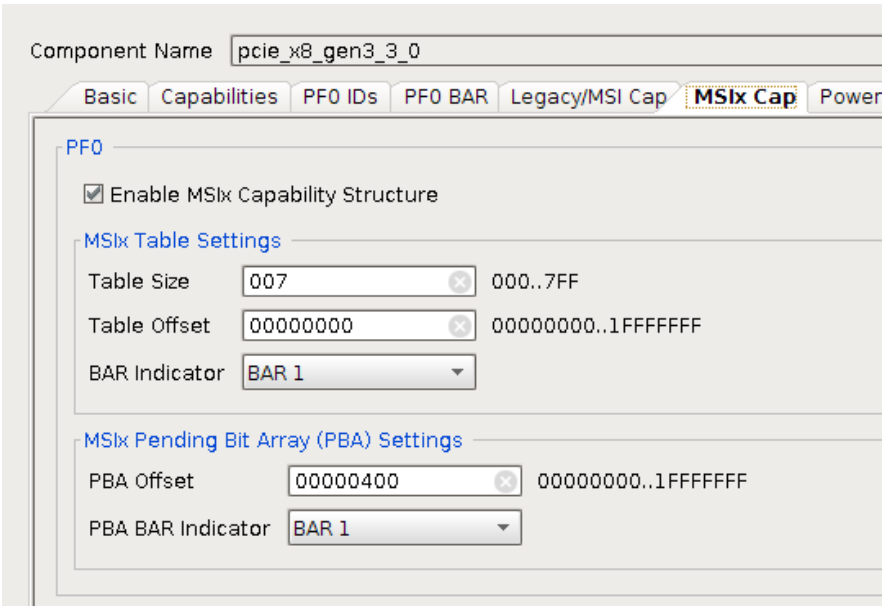

Figure 10: PCIe core configuration in Vivado [MSIx]

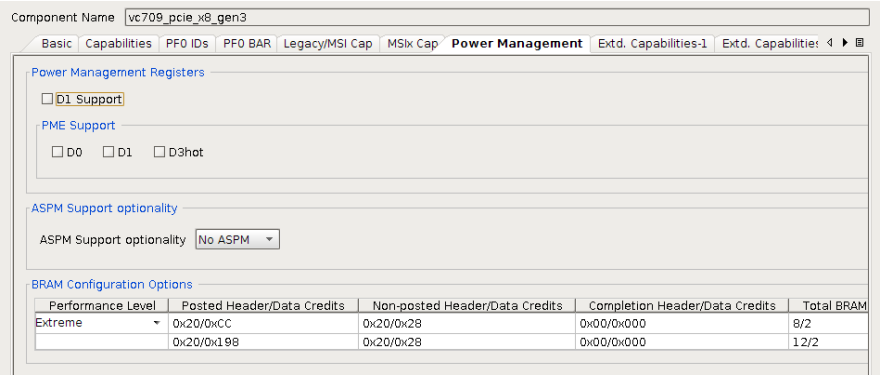

Figure 11: PCIe core configuration in Vivado [Power Management]

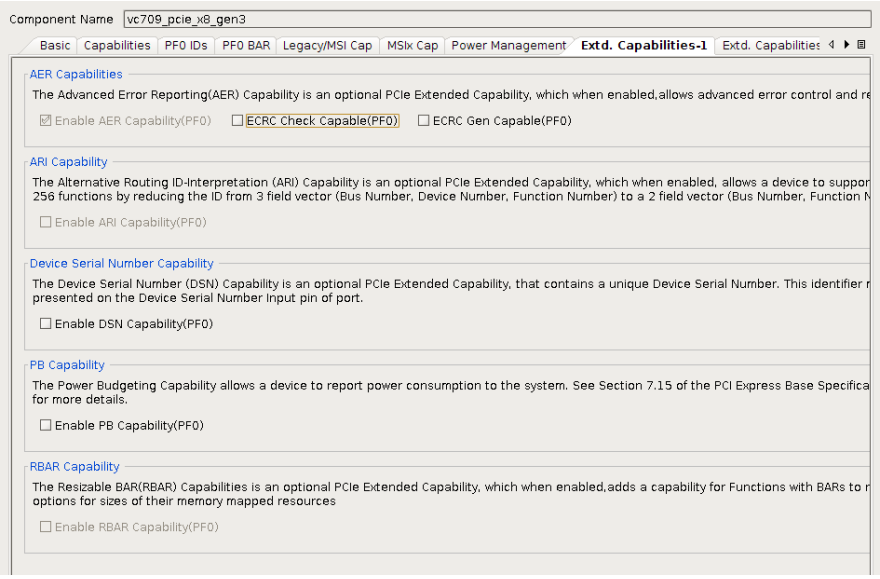

Figure 12: PCIe core configuration in Vivado [Extd. Capabilities 1]

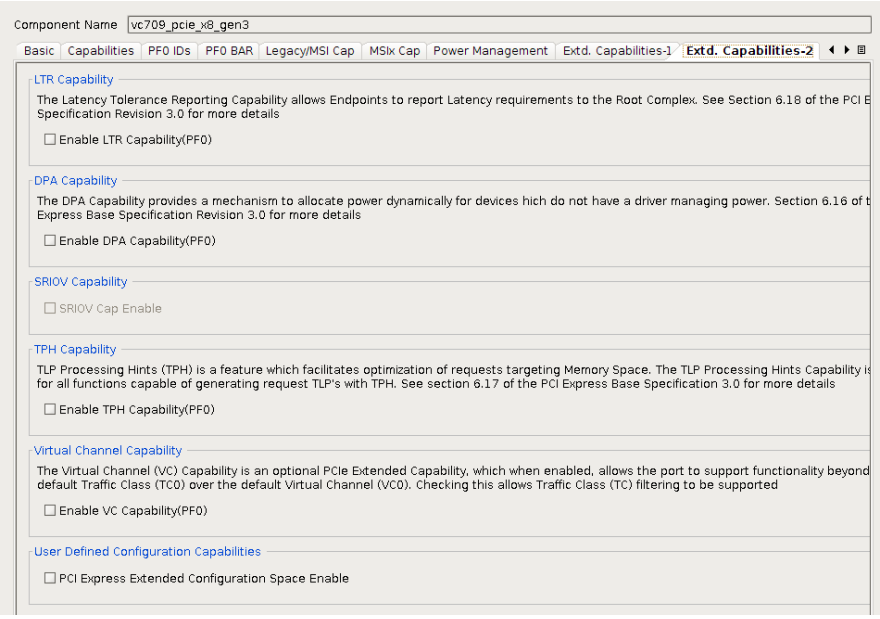

Figure 13: PCIe core configuration in Vivado [Extd. Capabilities 2]

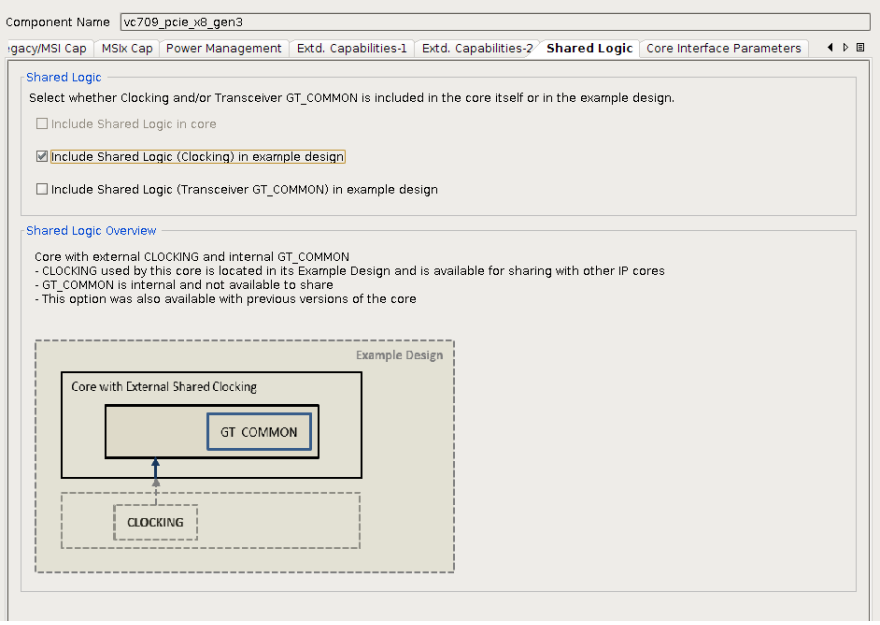

Figure 14: PCIe core configuration in Vivado [Shared LogicMSIx]

Figure 15: PCIe core configuration in Vivado [Core Interface Parameters]

4 Obtaining and building the PCIe Engine

The repository is divided in several directories:

directory	contents
firmware/constraints	Contains an XDC file with Vivado constraints including Chipscope ILA definitions, may differ over different commits
firmware/output	Empty placeholder where bit files will be generated
firmware/Projects	Empty placeholder where the Vivado projects will be generated
firmware/scripts/Wupper	This directory contains two scripts per hardware type to create the vivado project and to run synthesis and implementation, see later this chapter.
firmware/simulation/Wupper	Contains a Modelsim.ini project as well as the scripts ci.sh to run the simulation in Modelsim (or Questasim)
firmware/sources/pcie	This directory contains the files for the Wupper PCIe core
firmware/sources/packages	Contains a vhdl package with some type definitions, but more importantly the application specific register definitions.

Table 5: Directories in the repository

4.1 Clone the git repository

Before starting to work with this core, it is a good idea to clone the whole GIT repository, if you already have it, update to the latest revision.

Listing 1: git clone

```
git clone https://gitlab.nikhef.nl/franss/wupper.git
```

besides the firmware directory with the listing in the introduction of this chapter, you will find other directories:

- **documentation** contains this document as well as a doxygen script to document the firmware structure.
- **software**
  - **driver** contains the wupper and cmem driver, described in 6.1
  - **wupper_tools** contains several useful tools to control DMA, the registers and application specific example tools

4.2 Create the Vivado Project

The Vivado project is not supplied in the got tree, instead a .tcl script is provided to generate the project. To create the project, open Vivado without a project, then open the TCL console and run the following commands.

Listing 2: Create Vivado Project

```
cd /path/to/felix/firmware/scripts/Wupper/
source ./VC709_import_vivado
```

A project should now be created in *firmware/Projects/*. **beware that this script will overwrite and recreate the project if it exists already.**

4.3 Running synthesis and implementation

When the project has been created, you can simply press the buttons to run synthesis and implementation of the design, but a tcl script has been created to run these steps automatically. Additionally the script will create the bitfile in the *firmware/output* directory, as well as an .mcs file and an .ltx file, containing the ChipScope ILA probes. All those 3 files have a timestamp in their filename so any previous synthesis output will be maintained. The script can simply be executed if the project is open.

Listing 3: start synthesis / implementation

```
cd /path/to/felix/firmware/scripts/Wupper/
source ./do_implementation_VC709.tcl
```

5 Simulation

The directory *firmware/simulation/pcie_dma_top* contains all necessary files to run the simulation in Mentor Graphics Modelsim or Questasim [12].

5.1 Prerequisites

The directory contains a file modelsim.ini with some standard information, it is assumed that one have the Xilinx Unisim_VCOMPONENTS library compiled and the location is defined in the environment variable $XILLIB. Also the Library "work" has to be created in the project directory.

The simulation project also relies on a simulation model of the FIFOcore, which will be generated when the cores in the Vivado project are generated. The file that should be generated is *../../Projects/pcie_dma_top/pcie_dma_top.srcs/sources_1/ip/fifo_generator_0/fifo_generator_0_funcsim.vhdl*

5.2 Creating the project and running the simulation.

Like the Vivado project, also the Questasim project is generated and operated using .tcl scripts. To create and run the project execute the following commands from a bash console. Make sure that vsim is in the path, UVVM and xilinx_simlib are compiled in the firmware/simulation directory:

Listing 4: Run the simulation

```
cd firmware/simulation/Wupper/
./ci.sh Wupper
```

The project does not include the actual Xilinx PCIe core simulation model, but the AXI4-Stream interface is simulated by a behavioral simulation model.

6 Software and Device drivers

The Wupper tools communicate with the Wupper core through the wupper device driver. Buffers in the host PC memory are used for bidirectional data transfers, this is done by a part of the driver called cmem_rcc. This will reserve a chunk of contiguous memory (Up to several GB) in the host server. For the specific case of the example application, the allocated memory will be logically subdivided in two buffers. One buffer is used to store data coming from the FPGA (write buffer, buffer 1), the other to store the ones going to the FPGA (read buffer, buffer 2). The idea behind the logical split of the memory in buffers is that those buffers can be used to copy data from the write to read, and perform checks. The driver is developed for Scientific Linux CERN 6 but has been tested and used also under Ubuntu kernel version 5.11. Building and loading/unloading the driver is explained in 6.1.

In this chapter we assume that the card is loaded with the latest firmware, it has been placed in a Gen3 or Gen4 compatible PCIe slot and the PC is running Linux. Optionally a Vivado hardware server can be connected to view the Debug probes of the ILA cores, as specified in the constraints file. [13]

6.1 Building / Loading the drivers

The Drivers for Wupper consist of two parts. The first part is the cmem driver, this driver allocates a contiguous block of RAM in the PC memory which can be used for the DMA transfers.

The second part is the Wupper driver which allows access to the DMA descriptors and the registermap.

Listing 5: Building and Loading the driver

```
#build the driver
cd trunk/software/driver/src
./make
# load the driver
cd ../scripts
sudo drivers_wupper_local start
# see status of the driver
sudo drivers_wupper_local status
# unload the driver
sudo drivers_wupper_local stop
```

6.2 Driver functionality

Before any DMA actions can be performed, one or more memory buffers have to be allocated. The driver in conjunction with the wupper tools take this into account.

The application has to do two important tasks for a DMA action to occur.

- Allocate a buffer using the cmem_rcc driver
- Create and enable the DMA descriptor.

If the buffer is for instance allocated at address 0x00000004d5c00000, initialize bits 64:0 of the descriptor with 0x00000004d5c00000, and end address (bit 127:64) 0x00000004d5c00000

plus the write size. If a DMA Write is to be performed, initialize bits 10:0 of descriptor 0a with 0x40 (for 256 bytes per TLP, depending on the PC chipset) and bit 11 with '0' for write, then enable the corresponding descriptor enable bit at address 0x400. The TLP size of 0x40 (32 bit words) is limited by the maximum TLP that the PC can handle, in most cases this is 256 bytes, the Engine can handle bigger TLP's up to 4096 bytes.

Listing 6: Create a Write descriptor

```
#write descriptor 0
#BAR0 offset:  Contents:
0x0000         0x00000004d5c00000
0x0008         0x00000004d5c00400
#Set the length to 0x40 / Write
0x0010         0x040
#enable descriptor 0 to start the DMA Write
0x0400         1
```

If a DMA Read of 1024 bytes (0x100 DWords) from PC memory is to be performed at address 0x00000004d5d00000, initialize bits 64:0 of the descriptor with 0x00000004d5d00000, and bits [127:64] with 0x00000004d5d00400. Initialize bits 10:0 of descriptor 0 with 0x100 and bit 11 with '1' for read, then enable the corresponding descriptor enable bit at address 0x400. The TLP size of 0x100 is limited by the maximum TLP size of the Xilinx core, set to 1024 bytes, 0x100 words.

Listing 7: Create a Read descriptor

```
#write descriptor 1
#BAR0 offset:  Contents:
0x0020         0x00000004d5d00000
0x0028         0x00000004d5d00400
#Set the length to 0x100 / Read
0x0030         0x0900
#enable descriptor 1 to start the DMA Read
0x0400         2
```

6.3 Reading and Writing Registers and setting up DMA

The PCIe Engine has a register map with 128 bit address space per register, however registers can be read and written in words of 32, 64, 96 or 128 bits at a time. The addresses of the register have an offset with respect to a Base Address Register (BAR) that can be readout running: The PCIe Engine has 3 different BAR spaces all with their own memory map.

BAR0 is the memory area which contains registers that are related to DMA operations. The most important registers are the descriptors.

BAR1 is the memory area which contains registers that are related to Interrupt vectors.

BAR2 is the user memory area, it contains some example registers which can be implemented per the requirements for the user / application.

6.4 Wupper tools

The Wupper tools are a collection of tools which can be used to debug and control the Wupper core. These tools are command line programs and can only run if the device driver is

loaded. A detailed list and explanation of each tool is given in the next paragraphs. Some tools are specific to the example VHDL application, some other tools are more generic and can directly be used to control the Wupper DMA core, the Wupper-dma-transfer and Wupper-chaintest had been added as features for the OpenCores' benchmark example application. As mentioned before, the purpose of those applications is to check the health of the Wupper core.

The Wupper tools can be found in the directory hostSoftware/wupper_tools.

The Wupper tools collection comes with a readme [**?**], this explains how to compile and run the tools. Most of the tools have an -h option to provide helpful information.

Listing 8: Building Wupper Tools

```
cd trunk/software/wupper_tools
mkdir build
cd build
cmake ..
make
```

The build directory should now contain the following tools. All the tools come with a "-h" option to show a help message.

Tool	Description
wupper-info	Prints information of the device. For instance device ID, PLL lock status of the internal clock and FW version.
wupper-reset	Resets parts of the example application core. These functions are also implemented in the Wupper-dma-transfer tool.
wupper-config	Shows the PCIe configuration registers and allows to set, store and load configuration. An example is configuring the LED's on the VC-709 board by writing a hexadecimal value to the register.
wupper-irq-test	Tool to test interrupt routines
wupper-dma-stat	Displays information (addresses, size) of DMA descriptors
wupper-dma-transfer	Executes a DMA ToHost operation (-g) or a loopback operation (Both From and ToHost with internal loopback in the firmware) (-l)
wupper-throughput	The tool measures the throughput of the Wupper core.
wupper-dump-blocks	This tools dumps a block of 1 KB. The iteration is set standard on 100. This can be changed by adding a number after the "-n".
wupper-wishbone	This tool uses the wishbone registers in the register map to read and write values in the memory connected to the wishbone bus in the example.

6.4.1 Operating Wupper-dma-transfer

Wupper-dma-transfer sends data to the target PC via Wupper also known as half loop test. This tool operates the benchmark application and has multiple options. A list of such options is summarized in Listing 9.

Listing 9: Output of Wupper-dma-transfer -h

```
Usage: wupper-dma-transfer [OPTIONS]


Options:
-g              Generate data from internal counter in FPGA, to PC.
-l              Generate data from PC to PCIe and loopback to PC
-h              Display help.
```

The two executions of wupper-dma-transfer shown below show different operations. With -g, the internal 64-bit counter is incremented on every clock cycle. You will notice that the value is replicated 4 times (For Gen3x8 capable devices) because the internal FIFO interface is 256 bit, and in the example firmware the counter is replicated / concatenated 4 times to the same FIFO interface. In the loopback operation (-l) the FromHost buffer in the server is first filled with a 64-bit counter, sent towards the FPGA over DMA and then immediately looped back into a second buffer. This will result into an exact copy of the memory. The application displays only the first 10 elements of the memory.

Listing 10: Reset Wupper before a DMA Write action

```
$ ./wupper-dma-transfer -g
Starting DMA write
done DMA write
Buffer 1 addresses:
0: 0
1: 0
2: 0
3: 0
4: 1
5: 1
6: 1
7: 1
8: 2
9: 2
$ ./wupper-dma-transfer -l
Fill fromHost buffer with 64b counterm send to PCIe and read back
DONE!
Read back first 10 values:
0: 0
1: 1
2: 2
3: 3
4: 4
5: 5
6: 6
7: 7
8: 8
9: 9
```

7 Example application HDL entities

The example application, the user application inside the FPGA is a very simple example, implemented in wupper_oc_top.vhd. It has two modes of operation, controlled by the LOOPBACK register.

1. LOOPBACK=0: A 64-bit counter increments on every FIFO write into the ToHost memory, this is referred to as "write only" or "half loop" test.

2. LOOPBACK=1: The software allocates 2 buffers, then initializes the first buffer with some content. Two DMA operations are then initialized into both directions (ToHost and FromHost) and the two datastreams are connected to each other internally in the firmware example. The result is a copy of the memory into the second buffer, this is referred as "read and write" or "full loop" test.

8 Customizing the application

8.1 connection of the DMA FIFOs

Wupper comes with an example application that is described in 7. The toplevel file for the user application is wupper_oc_top.vhd

If you want to customize the example application for your own needs, the application can be stripped down by removing the connections to toHostFifo_* and fromHostFifo_* signals and connecting them as required to newly created entities in the design. The application can be controlled and monitored by the records "registermap_control" and "registermap_monitor". These records contain all the read/write register and the read only registers respectively, the registers are defined in the file pcie_package.vhd.

This file contains a read and a write port for a FIFO. This FIFO has a port width of 256 bit and is read or written at 250 MHz, resulting in a theoretical throughput of 60Gbit/s for PCIe Gen3x8 interfaces. For PCIe Gen4x8 or Gen3x16 interfaces (Virtex Ultrascale+, Versal Prime) the FIFO width is 512 bits and the theoretical throughput is around 120Gbit/s.

8.2 Application specific registers

Besides DMA memory reads and writes, the PCIe Engine also provides means to create a custom application specific register map. By default, the BAR2 register space is reserved for this purpose.

Listing 11: custom register types

```
  -- Control Record
type register_map_control_type is record
STATUS_LEDS                    : std_logic_vector(7 downto 0);     --
   Board GPIO Leds
I2C_WR                         : bitfield_i2c_wr_t_type;          -- House
    Keeping Controls and Monitors
I2C_RD                         : bitfield_i2c_rd_t_type;          -- House
    Keeping Controls and Monitors
INT_TEST                       : bitfield_int_test_t_type;        -- House
    Keeping Controls and Monitors
```

```
DMA_BUSY_STATUS                : bitfield_dma_busy_status_t_type;  --
    House Keeping Controls and Monitors
WISHBONE_CONTROL               : bitfield_wishbone_control_w_type;  --
    Wishbone
WISHBONE_WRITE                 : bitfield_wishbone_write_t_type;  --
    Wishbone
WISHBONE_READ                  : bitfield_wishbone_read_t_type;  --
    Wishbone
LOOPBACK                       : std_logic_vector(7 downto 0);    -- for
    every DMA descriptor
-- 0: Generate data from a counter value
-- 1: Loop back data from FromHost to ToHost DMA

end record;
```

The VHDL files containing the registermap are not supposed to be modified by hand. Instead WupperCodeGen can be used.

Inside the source tree you will find the directory WupperCodeGenScripts containing the YAML file with application specific registers, and a set of scripts to generate VHDL sources, C++ headers and Latex and HTML documentation.

- **registers-2.0.yaml:** This is the database of registers
- **build-doc.sh** Run this script to generate the table of registers in A
- **build-firmware.sh** Regenerate the firmware (pcie_control.vhd and pcie_package.vhd) from the yaml file
- **build-software.sh** Regenerate the sources in software/regmap from the yaml file.

For more information see the documentation in software/wuppercodegen/doc

References

[1] Cern Atlas Experiment
http://www.atlas.ch

[2] Atlas FELIX project
http://atlas-project-felix.web.cern.ch

[3] ARM AMBA AXI bus standard specification page
http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

[4] Xilinx website about the PCI Express core
https://www.xilinx.com/products/technology/pci-express.html

[5] UG761: Xilinx AXI Bus documentation
http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf

[6] PG023: The User guide for the Xilinx PCI Express core
https://www.xilinx.com/support/documentation/ip_documentation/pcie3_7x/v4_3/pg023_v7_pcie_gen

[7] PG156: The User guide for the Xilinx Ultrascale PCI Express core
https://www.xilinx.com/support/documentation/ip_documentation/pcie3_ultrascale/v4_4/pg156-ultrascale-pcie-gen3.pdf

[8] PG213: UltraScale+ Devices Integrated Block for PCI Express v1.3
https://www.xilinx.com/support/documentation/ip_documentation/pcie4_uscale_plus/v1_3/pg213-pcie4-ultrascale-plus.pdf

[9] PG343: Versal ACAP Integrated Block for PCI Express v1.0
https://www.xilinx.com/support/documentation/ip_documentation/pcie_versal/v1_0/pg343-pcie-versal.pdf

[10] M. Jackson, R. Budruk, *PCI Express Technology - Comprehensive Guide to Generations 1.x, 2.x and 3.0*, 1st Edition, MindShare Technology Series, 2012

[11] Official Wupper project (OpenCores version)
http://opencores.org/project,virtex7_pcie_dma

[12] Mentor Graphics Questasim
http://www.mentor.com/products/fv/questa/

[13] UG908: Programming and Debugging using Vivado
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug908-vivado-programming-debugging.pdf

[14] J. Corbet, G. Kroah Hartman, A. Rubini, *Linux Device Drivers*, 3rd Edition, O'Reilly, 2005
http://www.oreilly.com/openbook/linuxdrive3/book/index.html

[15] Xilinx Vivado Design Suite User Guide 2020.1
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug973-vivado-release-notes-install-license.pdf

[16] XADC Wizard v3.0 product guide
http://www.xilinx.com/support/documentation/ip_documentation/xadc_wiz/v3_0/pg091-xadc-wiz.pdf

[17] M. Jackson, R. Budruk, *PCI Express Technology - Comprehensive Guide to Generations 1.x, 2.x and 3.0*, 1st Edition, MindShare Technology Series, 2012

[18] Oussama el Kharraz Alami - Development of an application for Wupper a PCIe Gen3 DMA for Virtex 7
https://gitlab.nikhef.nl/franss/wupper/-/blob/master/documentation/example_application/internship-wupper.pdf

List of Figures

List of Tables

Appendix A WUPPER register map, version 2.0

Starting from the offset address of BAR0, BAR1 and BAR2, the register map for BAR0 expands from 0x0000 to 0x0430 for the PCIe control registers. BAR0 only contains registers associated with DMA. The offset for BAR0 is usually 0xFBB00000.

Address	PCIe	Name/Field		Bits	Type	Description
Bar0						
DMA_DESC						
0x0000	0,1	DMA_DESC_0				
			END_ADDRESS	127:64	W	End Address
			START_ADDRESS	63:0	W	Start Address
0x0010	0,1	DMA_DESC_0a				
			SW_POINTER	127:64	W	Pointer controlled by the software, indicating read or write status for circular DMA
			WRAP_AROUND	12	W	Wrap around
			FROMHOST	11	W	1: fromHost/ 0: toHost
			NUM_WORDS	10:0	W	Number of 32 bit words
...						
0x00E0	0,1	DMA_DESC_7				
			END_ADDRESS	127:64	W	End Address
			START_ADDRESS	63:0	W	Start Address
0x00F0	0,1	DMA_DESC_7a				
			SW_POINTER	127:64	W	Pointer controlled by the software, indicating read or write status for circular DMA
			WRAP_AROUND	12	W	Wrap around
			FROMHOST	11	W	1: fromHost/ 0: toHost
			NUM_WORDS	10:0	W	Number of 32 bit words
DMA_DESC_STATUS						
0x0200	0,1	DMA_DESC_STATUS_0				
			EVEN_PC	66	R	Even address cycle PC
			EVEN_DMA	65	R	Even address cycle DMA
			DESC_DONE	64	R	Descriptor Done
			FW_POINTER	63:0	R	Pointer controlled by the firmwarre, indicating where the DMA is busy reading or writing
...						
0x0270	0,1	DMA_DESC_STATUS_7				
			EVEN_PC	66	R	Even address cycle PC
			EVEN_DMA	65	R	Even address cycle DMA
			DESC_DONE	64	R	Descriptor Done
			FW_POINTER	63:0	R	Pointer controlled by the firmwarre, indicating where the DMA is busy reading or writing
0x0300	0,1	BAR0_VALUE		31:0	R	Copy of BAR0 offset reg.
0x0310	0,1	BAR1_VALUE		31:0	R	Copy of BAR1 offset reg.

Address	PCIe	Name/Field		Bits	Type	Description
0x0320	0,1	BAR2_VALUE		31:0	R	Copy of BAR2 offset reg.
0x0400	0,1	DMA_DESC_ENABLE		7:0	W	Enable descriptors 7:0. One bit per descriptor. Cleared when Descriptor is handled.
0x0410	0,1	DMA_FIFO_FLUSH		any	T	Flush (reset). Any write clears the DMA Main output FIFO
0x0420	0,1	DMA_RESET		any	T	Reset Wupper Core (DMA Controller FSMs)
0x0430	0,1	SOFT_RESET		any	T	Global Software Reset. Any write resets applications, e.g. the Central Router.
0x0440	0,1	REGISTER_RESET		any	T	Resets the register map to default values. Any write triggers this reset.
0x0450	0,1	FROMHOST_FULL_THRESH				
			THRESHOLD_ASSERT	22:16	W	Assert value of the FromHost programmable full flag
			THRESHOLD_NEGATE	6:0	W	Negate value of the FromHost programmalbe full flag
0x0460	0,1	TOHOST_FULL_THRESH				
			THRESHOLD_ASSERT	27:16	W	Assert value of the ToHost programmable full flag
			THRESHOLD_NEGATE	11:0	W	Negate value of the ToHost programmalbe full flag
0x0470	0,1	BUSY_THRESHOLD_ASSERT		63:0	W	Tohost or Fromhost busy will be asserted in circular DMA mode when the server PC buffer gets full (space below ASSERT threshold)..
0x0480	0,1	BUSY_THRESHOLD_NEGATE		63:0	W	Tohost or Fromhost busy will be negated in circular DMA mode when the server PC buffer gets less full (space above NEGATE threshold).
0x0490	0,1	BUSY_STATUS				
			FROMHOST_BUSY	1	R	A fromhost descriptor passed BUSY_THRESHOLD_ASSERT, busy flag set
			TOHOST_BUSY	0	R	A tohost descriptor passed BUSY_THRESHOLD_ASSERT, busy flag set

Address	PCIe	Name/Field	Bits	Type	Description
0x04A0	0,1	PC_PTR_GAP	63:0	W	This is the minimum value that the pc_pointer in a descriptor has to decrease in order to flip the evencycle_pc bit

Table 6: FELIX register map BAR0

BAR1 stores registers associated with the Interrupt vector. The offset for BAR1 is usually 0xFBA00000.

Address	PCIe	Name/Field		Bits	Type	Description
Bar1						
INT_VEC						
0x0000	0,1	INT_VEC_0				
			INT_CTRL	127:96	W	Interrupt Control
			INT_DATA	95:64	W	Interrupt Data
			INT_ADDRESS	63:0	W	Interrupt Address
...						
0x00F0	0,1	INT_VEC_15				
			INT_CTRL	127:96	W	Interrupt Control
			INT_DATA	95:64	W	Interrupt Data
			INT_ADDRESS	63:0	W	Interrupt Address
0x0100	0,1	INT_TAB_ENABLE		7:0	W	Interrupt Table enable Selectively enable Interrupts

Table 7: FELIX register map BAR1

BAR2 stores registers for the control and monitor of HDL modules inside the FPGA other than Wupper. A portion of this register map's section is dedicated for control and monitor of devices outside the FPGA; as for example simple SPI and I2C devices. The offset for BAR2 is usually 0xFB900000.

Address	PCIe	Name/Field		Bits	Type	Description
Bar2						
Generic Board Information						
0x0000	0	REG_MAP_VERSION		15:0	R	Register Map Version, 2.0 formatted as 0x0200
0x0010	0	BOARD_ID_TIMESTAMP		39:0	R	Board ID Date / Time in BCD format YYMMDDhhmm
0x0020	0	GIT_COMMIT_TIME		39:0	R	Board ID GIT Commit time of current revision, Date / Time in BCD format YYMMDDhhmm
0x0030	0	GIT_TAG		63:0	R	String containing the current GIT TAG
0x0040	0	GIT_COMMIT_NUMBER		31:0	R	Number of GIT commits after current GIT_TAG
0x0050	0	GIT_HASH		31:0	R	Short GIT hash (32 bit)
0x0060	0	STATUS_LEDS		7:0	W	Board GPIO Leds
0x0070	0	GENERIC_CONSTANTS				
			INTERRUPTS	15:8	R	Number of Interrupts
			DESCRIPTORS	7:0	R	Number of Descriptors

Address	PCIe	Name/Field	Bits	Type	Description
0x0080	0	CARD_TYPE	63:0	R	Card Type: - 105 (0x069): KCU-105 - 128 (0x080): VCU128 - 180 (0x0B4): VMK180 - 709 (0x2c5): VC-709 - 710 (0x2c6): HTG-710 - 711 (0x2c7): BNL-711 - 712 (0x2c8): BNL-712
0x0090	0	PCIE_ENDPOINT	0	R	Indicator of the PCIe endpoint on BNL71x cards with two endpoints. 0 or 1
0x00A0	0	NUMBER_OF_PCIE_ENDPOINTS	1:0	R	Number of PCIe endpoints on the card. The VCU128 cards have 2 endpoints
House Keeping Controls And Monitors					
0x1300	0	MMCM_MAIN_PLL_LOCK	0	R	Main MMCM PLL Lock Status
0x1310	0	I2C_WR			
		I2C_WREN	any	T	Any write to this register triggers an I2C read or write sequence
		I2C_FULL	25	R	I2C FIFO full
		WRITE_2BYTES	24	W	Write two bytes
		DATA_BYTE2	23:16	W	Data byte 2
		DATA_BYTE1	15:8	W	Data byte 1
		SLAVE_ADDRESS	7:1	W	Slave address
		READ_NOT_WRITE	0	W	READ/¡o¿WRITE¡/o¿
0x1320	0	I2C_RD			
		I2C_RDEN	any	T	Any write to this register pops the last I2C data from the FIFO
		I2C_EMPTY	8	R	I2C FIFO Empty
		I2C_DOUT	7:0	R	I2C READ Data
0x1330	0	FPGA_CORE_TEMP	11:0	R	XADC temperature monitor for the FPGA CORE for Virtex7 temp (C)= ((FPGA_CORE_TEMP* 503.975)/4096)-273.15 for Kintex Ultrascale temp (C)= ((FPGA_CORE_TEMP* 502.9098)/4096)- 273.8195
0x1340	0	FPGA_CORE_VCCINT	11:0	R	XADC voltage measurement VCCINT = (FPGA_CORE_VCCINT *3.0)/4096

Address	PCIe	Name/Field		Bits	Type	Description
0x1350	0	FPGA_CORE_VCCAUX		11:0	R	XADC voltage measurement VCCAUX = (FPGA_CORE_VCCAUX *3.0)/4096
0x1360	0	FPGA_CORE_VCCBRAM		11:0	R	XADC voltage measurement VCCBRAM = (FPGA_CORE_VCCBRAM *3.0)/4096
0x1370	0,1	FPGA_DNA		63:0	R	Unique identifier of the FPGA
0x1800	0	INT_TEST				
			TRIGGER	any	T	Fire a test MSIx interrupt set in IRQ
			IRQ	3:0	W	Set this field to a value equal to the MSIX interrupt to be fired. The write triggers the interrupt immediately.
0x1810	0	DMA_BUSY_STATUS				
			CLEAR_LATCH	any	T	Any write to this register clears TO-HOST_BUSY_LATCHED
			ENABLE	4	W	Enable the DMA buffer on the server as a source of busy
			TOHOST_BUSY_LATCHED	3	R	A tohost descriptor has passed BUSY_THRESHOLD_ASSERT in the past, busy flag was set
			FROMHOST_BUSY_LATCHED	2	R	A fromhost descriptor has passed BUSY_THRESHOLD_ASSERT in the past, busy flag was set
			FROMHOST_BUSY	1	R	A fromhost descriptor passed BUSY_THRESHOLD_ASSERT, busy flag set
			TOHOST_BUSY	0	R	A tohost descriptor passed BUSY_THRESHOLD_ASSERT, busy flag set
Wishbone						
0x2000	0	WISHBONE_CONTROL				
			WRITE_NOT_READ	32	W	wishbone write command wishbone read command
			ADDRESS	31:0	W	Slave address for Wishbone bus
0x2010	0	WISHBONE_WRITE				
			WRITE_ENABLE	any	T	Any write to this register triggers a write to the Wupper to Wishbone fifo
			FULL	32	R	Wishbone

Address	PCIe	Name/Field		Bits	Type	Description
			DATA	31:0	W	Wishbone
0x2020	0	WISHBONE_READ				
			READ_ENABLE	any	T	Any write to this register triggers a read from the Wishbone to Wupper fifo
			EMPTY	32	R	Indicates that the Wishbone to Wupper fifo is empty
			DATA	31:0	R	Wishbone read data
0x2030	0	WISHBONE_STATUS				
			INT	4	R	interrupt
			RETRY	3	R	Interface is not ready to accept data cycle should be retried
			STALL	2	R	When pipelined mode slave can't accept additional transactions in its queue
			ACKNOWLEDGE	1	R	Indicates the termination of a normal bus cycle
			ERROR	0	R	Address not mapped by the crossbar
Application						
0x3000	0, 1	LOOPBACK		7:0	W	for every DMA descriptor 0: Generate data from a counter value 1: Loop back data from FromHost to ToHost DMA

Table 8: FELIX register map BAR2

Appendix B Benchmark: block size versus write speed

During a ToHost DMA action (FPGA $\rightarrow$ PC), a transfer request is transferred from the host to the FPGA, therefore a write descriptor is setup. This descriptor contains information such as memory addresses, direction and the size of the payload, i.e. the amount of data to be transferred. The descriptor is then handled by Wupper, and the data transfer to host initiated. The size of the payload is in this case also the block size. For example when users choose to have a block size of 1 KB, the request gets completed after 1 KB of data had been transferred to host. Subsequently a new header will be created and repeated until the throughput measurement is stopped by the user. A plot of the block size versus write speed is shown below in Figure 16. One can clearly observe from this plot that the block size have effect on the write speed. This is somehow expected as there is an overhead due to the request of those blocks, hence the more data get transfer per request, the better the PCIe bandwidth is exploited. The bigger the block size is, the faster the write speed gets. The throughput obviously saturates at a level close to the theoretical maximum speed defined by an 8 lane PCIe Gen3 link (64 Gbps).

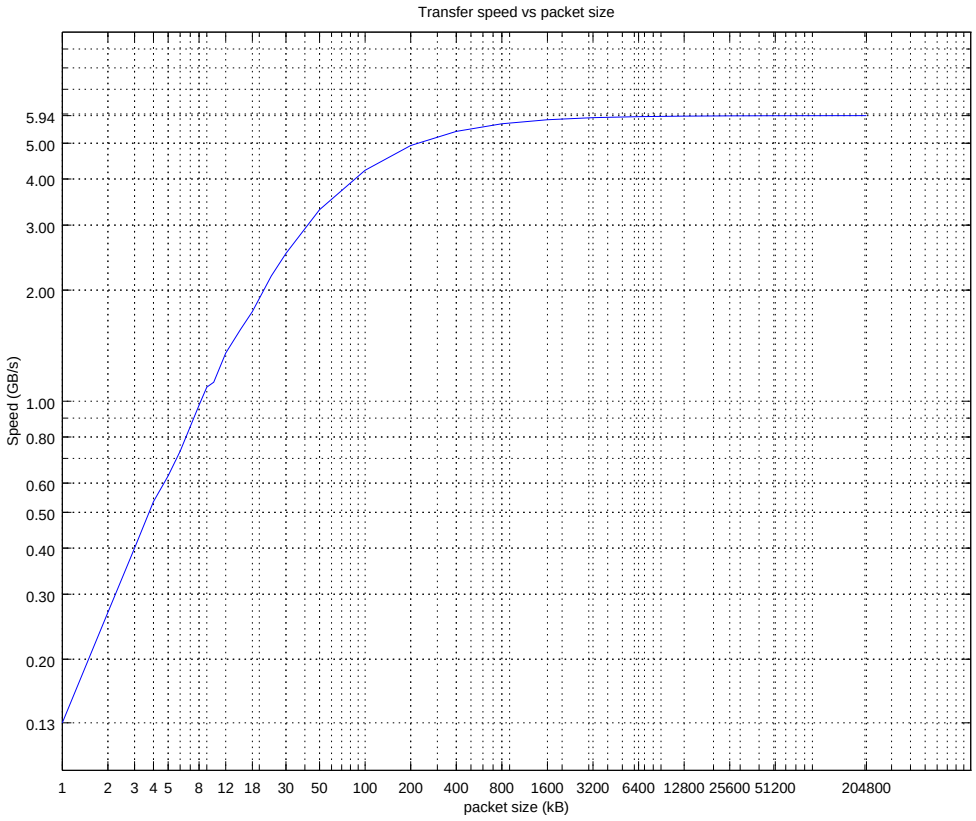

Figure 16: Transfer speed vs packet size