Wupper - a Xilinx Virtex-7 PCIe Engine

Frans Schreuder, Andrea Borga, Oussama el Kharraz Alami

04-10-2016

Contents

Revision History

Revision	Date	Author(s)	Description
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 Introduction

Wupper [1] was designed for the ATLAS / FELIX project [1], to provide a simple Direct Memory Access (DMA) interface for the Xilinx Virtex-7 PCIe Gen3 hard block and the . The core is not meant to be flexible among different architectures, but especially designed for the 256 bit wide AXI4-Stream interface [4] of the Xilinx Virtex-7 and Ultrascale FPGA Gen3 Integrated Block for PCI Express (PCIe) [3] [5] [6].

The purpose of the PCIe Engine is to provide an interface to a standard FIFO. This FIFO has the same width as the Xilinx AXI4-Stream interface (256 bits) and runs at 250 MHz. The application side of the FPGA design can simply read or write to the FIFO; the PCIe Engine will handle the transfer into Host PC memory, according to the addresses specified in the DMA descriptors.

With the PCIe Gen3 standard it is possible to reach a theoretical line rate of 8 GT/s; by using 8 lanes, it is therefore possible to reach a theoretical throughput of 64 Gb/s. The main purpose of Wupper is to handle data transfers from a simple user interface, i.e a FIFO, to and from the host PC memory. The other functionality supported by Wupper is the access to control and monitor registers inside the FPGA, and the surrounding electronics, via a simple register map. Figure 1 below shows a block diagram of the Wupper package.

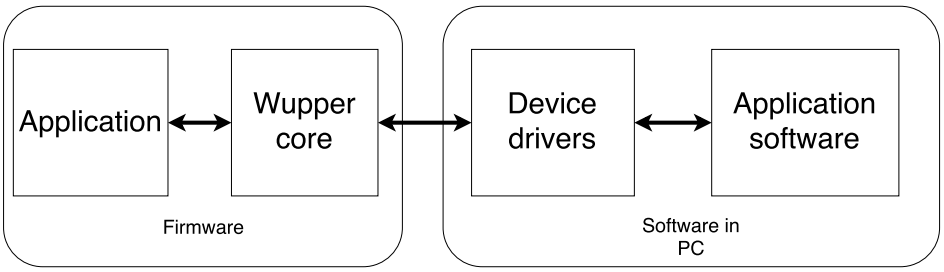


Figure 1: Wupper package overview

The Wupper core communicates to the host PC via the Wupper driver and is controlled by a set of, so called, Wupper tools. The Wupper driver through an Application Programming Interface (API) can also communicate to a Wupper Graphical User Interface (GUI). Wupper had been published under the LGPL license on Opencores.org [8]. As the developers firmly believe in the dissemination of knowledge through Open Source. Hence users can freely download, use and learn from the core and possibly provide feedback to further improve Wupper. The outcome of the development is the so called Wupper package: a suite of firmware and software components, which details will be given later in this report. On missing feature of the Wupper core published on OpenCores was a simple yet complete example application to study, test, and benchmark Wupper. To avoid confusion concerning name, a list is created to specify a name and description for all the parts of the Wupper project:

- Wupper core: firmware PCIe engine
- Wupper driver: software device driver

[1] A wupper is a person performing the act of bongelwuppen, the version from the Dutch province of Groningen of the Frisian sport Fierljeppen (canal pole vaulting). https://www.youtube.com/watch?v=Bre8DsQZqSs

- Wupper tools: software tools to operate the core
- Wupper GUI: a simple control and monitor panel
- Wupper package: the sum of the above packed for distribution on Open Cores.

For synthesis and implementation of the cores, it is recommend to use Xilinx Vivado 2015.4. 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 2015.4 Format.

For portability reasons, no Xilinx project files will be supplied with the Engine, 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.

2 Wupper package

In this section, the firmware, drivers, and tools of Wupper together with its working principle are explained.

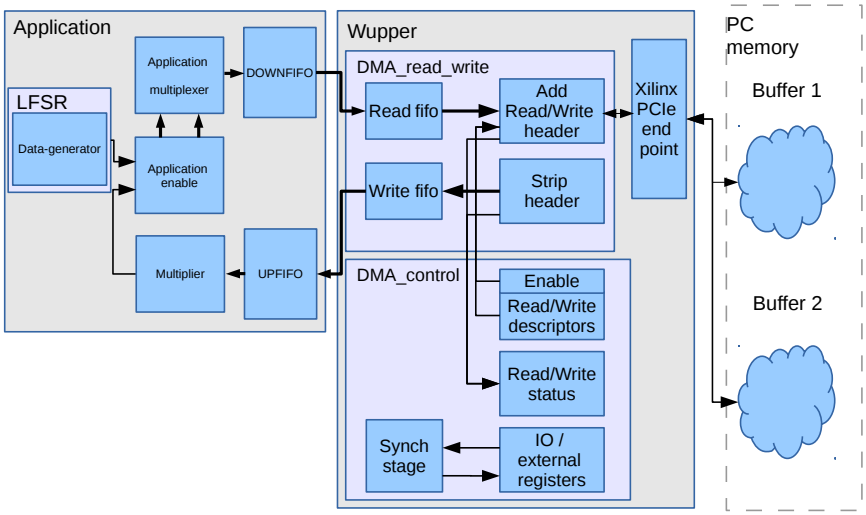


Figure 2: Overview of the HDL modules in the Wupper package

2.1 Wupper core

A DMA Engine, like Wupper, moves data bidirectionally to a memory buffer without CPU intervention. This efficient method is used for handling large amounts of data, which is crucial for throughput intensive applications. During a DMA tranfer, the DMA control core will take control according to the information provided by a DMA descriptor, and by flagging completion of operations in a per descriptor status register. By providing user data into the FIFO's, the core starts the DMA transfer over the PCIe lanes. Figure 2 shows a simplified diagram of the of the HDL modules of the Wupper package; including the HDL modules for the Wupper core and the example application, together with the host PC memory. Figure 3 shows a more detailed block diagram of the Wupper core.

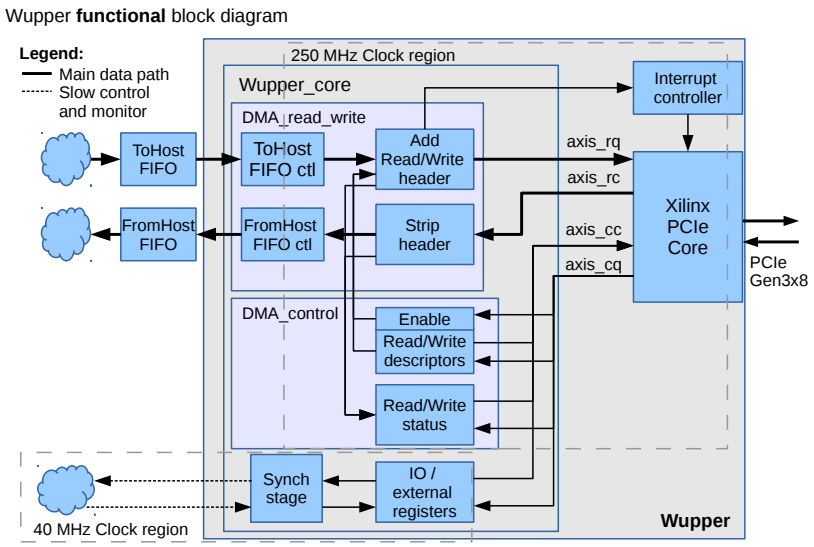


Figure 3: Structure of the PCIe Engine

Xilinx has introduced the AXI4-Stream interface [4] for the Virtex-7 PCIe core, this is a simplified version of the ARM AMBA AXI bus [2] which doesn't contain any address lines. Instead the Address and other information are supplied in the header of each PCIe package (TLP).

The Wupper Core is divided into 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. DMA control contains a register map, with addresses to the descriptors, status registers and external registers for the user application.

2. DMA Read Write
   This entity contains the processess to parse the DMA descriptors and transfer the data from the FIFO to the AXI stream bus and vise versa.

Figure 3 also shows the sync stage for the IO and external registers. The user space registers are synchronized to a slower clock (41.66MHz or 250MHz/6) in order to relax timing closure of the design.

2.1.1 DMA control

The DMA control (DMA_control in Figure 2) process consists of a register map which can be configured from a PC using the Wupper tools. The registermap is divided in three regions: BAR0, BAR1 and BAR2. BAR stands for Base Address Region. Every BAR has 1 MB of address space.

BAR0 contains registers associated with DMA like the DMA descriptors. The descriptors specify the addresses, transfer direction, size of the data and an enable line. Figure 2 shows that the information is fed to the DMA_read_write core.

BAR1 is reserved for the interrupt mechanism and consists of 8 vectors.

BAR2 is used for the benchmark application and is dedicated to user applications. The implemented registers are summarized in Appendix A.

The dma control module has the functionality to implement and manage the DMA descriptors, the interrupt vector and also the application specific registers.

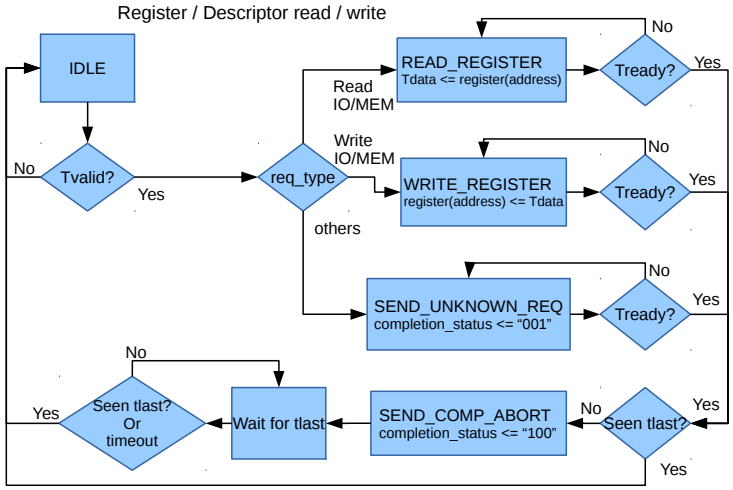


Figure 4: Flow of a Register / Descriptor Read or Write process

The DMA Control process always responds to a request with a certain req_type from the PC. It does only respond 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 only process the first 128 bits and go back to idle state. The maximum register size has been set to 128 bits because this is a convenient maximum register size; it is also the maximum payload that fits in one 250 MHz clock cycle of the AXI4-Stream interface. For most PC applications, a 64 bit register size is more practical, for that purpose, the registers can be read and written in parts of 32, 64 or 128 bits at the time.

2.1.2 DMA read/write

The DMA read and write (DMA_read_write in Figure 2) module handles the transfer from the FIFO's according to the direction specified by the descriptors. If data shifts into the ToHost FIFO, a non-empty flag will be asserted to start the DMA write process, this direction of the flow is defined as the "ToHost". This process reads the descriptors and creates a header with the information. The header is added when the data shifts out of the ToHost FIFO. For the reversed situation, the data with a header is read from the PC memory. This direction of the flow is then defined as "FromHost". The information in the header will be parsed by the DMA control and the payload fed to the FromHost FIFO.

The DMA Read / Write Module contains two important processes:

- Add Header
  In the first process the descriptors are read and a header is created according to the descriptor. If the descriptor is a write descriptor, the payload data is read from the FIFO and added after the header.

- Strip Header
  In the second process the header of the received data is checked against the tag of the request, and the payload is shifted into the FIFO.

Both processes can fire an MSI-X type interrupt through the interrupt controller when the processing of a descriptor is completed.

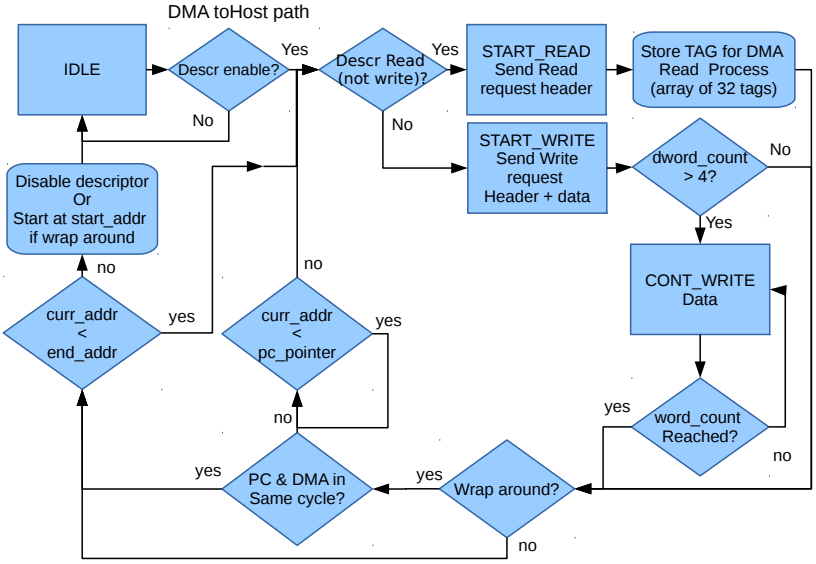


Figure 5: Flow of the DMA To Host action, or From Host request process

The DMA To Host process reads the current descriptor and requests a read or write to the PC memory. For a write it also initiates a FIFO read and adds the data into the payload of the PCIe packet. When initiating a read request, this process will additionally copy the current tag into an array of tags which will be used in the DMA read process in order to check a matching reply.

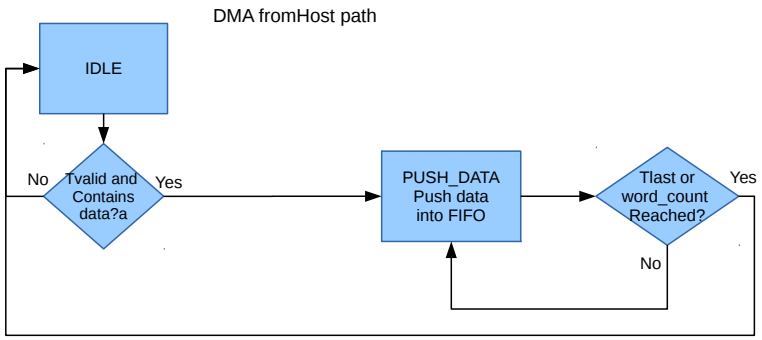


Figure 6: Flow of the DMA From Host process

The DMA From Host process checks the header for a valid tag, created by the DMA Write process, if this header and tag is valid, the data will be pushed into the FIFO.

2.1.3 Xilinx PCIe End Point

The Virtex-7 XC7VX690T-2FFG1761C which is used on one of the Wupper target platforms, the VC-709 board has an integrated endpoint for PCI Express Gen3 [5]. This black box handles the traffic over the PCI Express bus. Inside the Wupper core a DMA read/write process, sends and receives AXI4 commands over the AXI4-Stream bus. The black box translates this into differential electrical signals. Figure 7 shows a simplified model of the firmware stack.

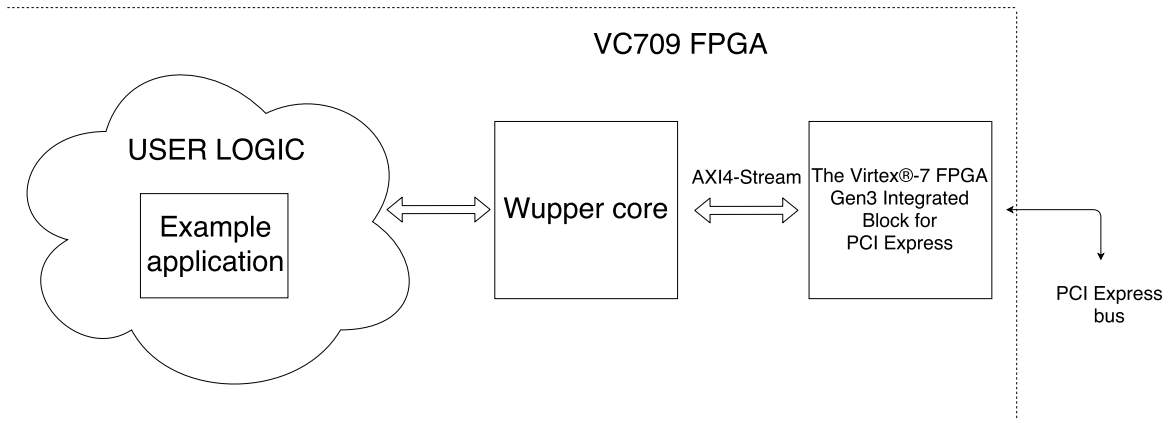


Figure 7: Block diagram of the logic in the VC-709 FPGA

The Configuration of the PCIe Gen3 endpoint IP Core from Xilinx is described in Appendix B

2.2 Xilinx AXI4-Stream interface

Wupper communicates through the AXI4 Stream interface with the Xilinx PCIe Gen3 endpoint. 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 2: AXI4-Stream streams

2.3 DMA descriptors

Each transfer To and From Host is achieved by means of setting up descriptors on the PC side, which are then processed by Wupper. The descriptors are set in the BAR0 section of the register map (see Appendix A). An extract of the descriptors and their registers is shown in Table 3 below.

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	PC Read Pointer
	WRAP_AROUND	12	W	Wrap around
	READ_WRITE	11	W	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 PC
	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 3: 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 PC (device driver).
- DMA_DESC_a: integrates the information above by adding (i) the status of the read pointer on the PC side ($rd\_pointer$), (ii) the wrap around functionality enabling ($wrap\_around$, see Section 2.4 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

2.4 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 PC 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 PC has to provide another address named PC 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 PC 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. In order to determine whether Wupper is processing an address behind or in front of the PC, 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 PC 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 PC (device driver) and kept within the start and end address range for Wupper to function correctly. Figure 8 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.

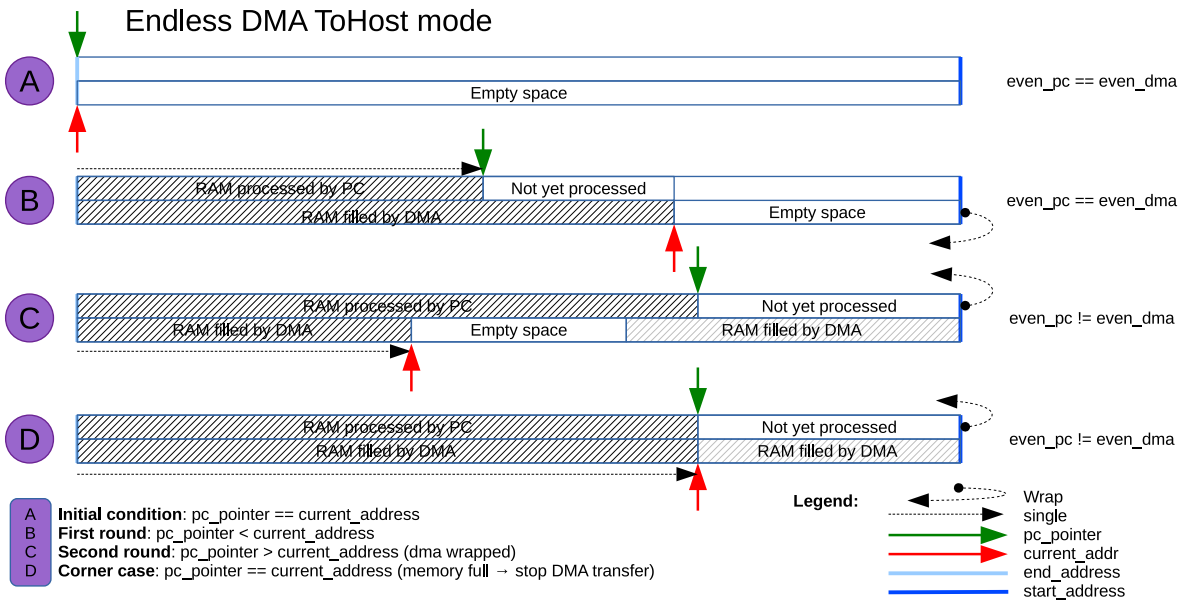


Figure 8: Endless DMA buffer and pointers representation diagram in ToHost mode

Looking at Figure 8 above, the following scenarios can be described:

- A : start condition, both the PC 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 PC is reading too slow, the DMA is stalled because the PC 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 PC 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 device driver, however it is indicating the address up to where the PC has filled the memory. In the first cycle the DMA has to stay behind the read pointer, when the PC has wrapped around, the dma can process memory up to $end\_address$ until it also wraps around.

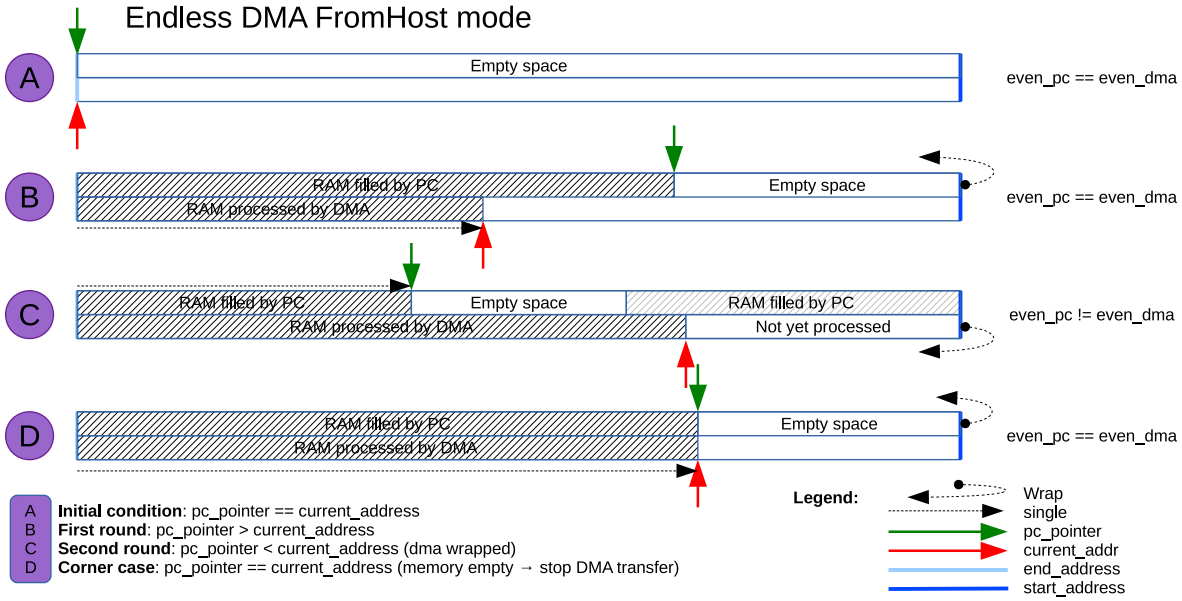


Figure 9: Endless DMA buffer and pointers representation diagram in FromHost mode

Looking at Figure 9 above, the following scenarios can be described:

- A : start condition, both the PC 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 PC is writing too slow, the DMA is stalled because the PC read pointer is not advancing fast enough, the DMA current_address has to stay behind.

2.5 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 [19]. In particular the chapter and tables in "MSI-X Capability Structure".

The MSI-X Interrupt table contains 8 interrupts, this number can be extended by a generic parameter in the firmware. 4 of the interrupts [0..3] are dedicated to Wupper, 4 interrupts [4..7] are called from CentralRouter:

Interrupt	Name	Description
0	FromHost wrap around	This interrupt is fired when the FromHost descriptor reaches the end address, or wraps around
1	ToHost wrap around	This interrupt is fired when the ToHost descriptor reaches the end address, or wraps around
2	ToHost Available	Fired when data becomes available in the ToHost fifo (falling edge of ToHostFifoProgEmpty)
3	FromHost Full	Fired when the FromHost fifo becomes full (rising edge of FromHostAppFifoProgFull)
4	Test interrupt #4	Fired when writing data to address BAR2 + 0x7800
5	Test interrupt #5	Fired when writing data to address BAR2 + 0x7810
6	ToHost Full	Fired when the ToHost fifo becomes full (rising edge of ToHostFifoProgFull)
7	GBT LOL	Currently reserved, but in the future this interrupt will be fired when a loss of lock occurs in one of the GBT channels.

Table 4: Interrupts

3 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/pcie_dma_top	This directory contains two scripts to create the vivado project and to run synthesis and implementation, see later this chapter.
firmware/simulation/pcie_dma_top	Contains a Modelsim.ini project as well as the scripts project.do, VSim_Functional.tcl and start.do to run the simulation in Modelsim (or Questasim)
firmware/sources/pcie	This directory contains the Vivado core (.xci) definition file for the PCIe core, as well the PCIe Engine files.
firmware/sources/shared	Contains a Vivado .xci file for the clock generator and the toplevel vhdl file.
firmware/sources/application	Contains an example vhdl file for a simple application
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

Please note that if changes to any of the core are made, a manual copy of the relevant .xci file in *firmware/Projects/pcie_dma_top/pcie_dma_top.srcs/sources_1/ip* should be made to the relevant folder in */firmware/sources*.

3.1 Check out the svn repository

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

Listing 1: svn checkout

```
svn co http://opencores.org/ocsvn/virtex7_pcie_dma/virtex7_pcie_dma/trunk
```

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.

- **hostSoftware**
  - **driver** contains the wupper and cmem driver, described in 5.1
  - **wupper_tools** contains several useful tools to control DMA, the registers and application specific example tools
  - **wupper_gui** contains an example application specific GUI application

3.2 Create the Vivado Project

The Vivado project is not supplied in the svn 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/svn/checkout/firmware/scripts/Wupper/
source ./vivado_import_virtex7.tcl
```

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

After the project is created you still have to generate the core's output products. Go to the project manager, in the IP Cores tab select all cores, right click one and select "generate output products".

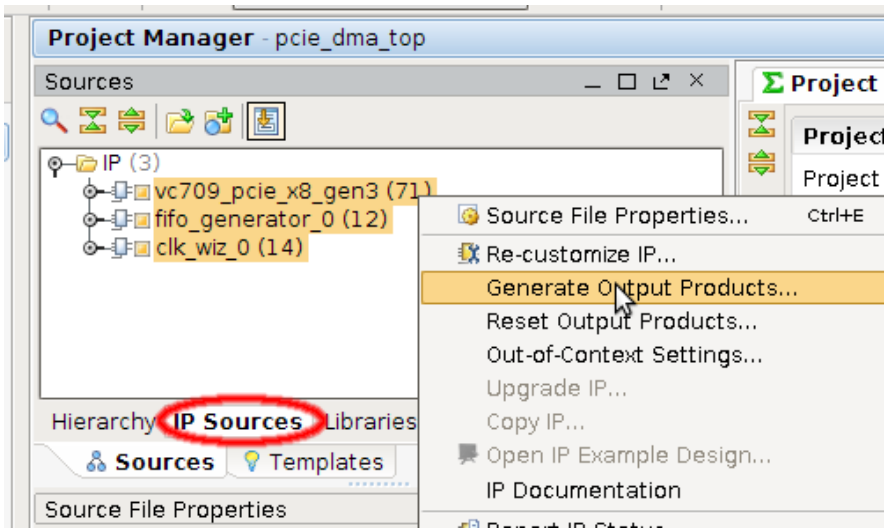


Figure 10: Generate IP Cores output products

3.3 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/svn/checkout/firmware/scripts/Wupper/
source ./do_implementation_VC709.tcl
```

4 Simulation

The directory *firmware/simulation/pcie_dma_top* contains all necessary files to run the simulation in Mentor Graphics Modelsim or Questasim [10].

4.1 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*

4.2 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 the Questasim console:

Listing 4: Run the simulation

```
cd firmware/simulation/Wupper/
#Create the project:
do project.do
#Start the simulation and load the waveforms:
do VSim_Functional.tcl
#Add stimuli to the AXI bus
do start.do
run 1us
```

The project does not include the actual Xilinx PCIe core simulation model, but the AXI4-Stream interface altered by stimuli in start.do, it can be edited according to your needs.

5 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. This will reserve a chunk of contiguous memory in the host. For the specific case of the example application, the allocated memory will be logically subdivided in two buffers (buffer 1 and buffer 2 in Figure 2). 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 3.13.0-44a. Building and loading/unloading the driver is explained in 5.1.

In this chapter we assume that the card is loaded with the latest firmware, it has been placed in a Gen3 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. [11]

5.1 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/hostSoftware/driver
./makedrivers
# load the driver
sudo scripts/drivers_wupper start
# see status of the driver
sudo scripts/drivers_wupper status
# unload the driver
sudo scripts/drivers_wupper stop
```

5.2 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 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
```

5.3 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.

5.4 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 [9], 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/hostSoftware/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-test	This tool transfers every second 1024 Byte of data and dumps it to the screen.
Wupper-throughput	The tool measures the throughput of the Wupper core. The method of computing the throughput is wrong, this is discussed in the section 3.4.2.
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".

5.4.1 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

```
daqmustud@gimone:$ ./wupper-dma-transfer -h

Usage: wupper-dma-transfer [OPTIONS]


This application has a sequence:
1 -Start with dma reset(-d)
2 -Flush the FIFO's(-f)
3 -Then reset the application (-r)


Options:
-l              Load pre-programmed seed.
-q              Load and generate an unique seed.
-g              Generate data from PCIe to PC.
-b              Generate data from PC to PCIe.
-s              Show application register.
-r              Reset the application.
-f              Flush the FIFO's.
-d              Disable and reset the DMA controller.
-h              Display help.
```

Before using the write function, make sure that the application is ready by resetting all the values, as shown in Listing 10.

Listing 10: Reset Wupper before a DMA Write action

```
daqmustud@gimone:$ ./wupper-dma-transfer -d
Resetting the DMA controller...DONE!
daqmustud@gimone:$ ./wupper-dma-transfer -f
Flushing the FIFO's...DONE!
daqmustud@gimone:$ ./wupper-dma-transfer -r
resetting application...DONE!
```

Before writing data into the PC, the data generator needs a seed to initialize the generator. There are two options available: load a unique seed or load a pre-programmed seed. The pre-programmed seed is always 256 bits, the unique seed value can be variable. The -s option displays the status of the register including the seed value. For a unique seed, replace the -l with -q, as shown in Listing 11.

Listing 11: Loading a pre-programmed seed in to the data generator.

```
daqmustud@gimone:$ ./wupper-dma-transfer -l
Writing seed to application register...DONE!
daqmustud@gimone:$ ./wupper-dma-transfer -s

Status application registers
----------------------------
LFSR_SEED_0A:         DEADBEEFABCD0123
LFSR_SEED_0B:         87613472FEDCABCD
LFSR_SEED_1A:         DEADFACEABCD0123
LFSR_SEED_1B:         12313472FEDCFFFF
APP_MUX:              0
LFSR_LOAD_SEED:       0
```

The -g option performs a DMA write to the PC memory. The data generator starts to fill the down FIFO and from the PC side, a DMA read action is performed. The size of the transfer is set to 1 MB by default, but the size is configurable. When the PC receives 1 MB of data, the transfer stops. It is possible that there is still some data left in the down FIFO, resetting the FIFO's can be done by the -f option, as shown in Listing 12.

Listing 12: Start generating data to the target.

```
daqmustud@gimone:$ ./wupper-dma-transfer -g
Starting DMA write
done DMA write
Buffer 1 addresses:
0: EED9733362A50D71
...
...
...
```

In a similar way a DMA read action from the FPGA can be performed by using the -b option. The output of the up FIFO is fed to a multiplier. The output of the multiplier is fed to the down FIFO with a destination to the PC memory as shown in Listing 13.

Listing 13: Performing a DMA read and DMA write

```
daqmustud@gimone:$ ./wupper-dma-transfer -b
Reading data from buffer 1...
DONE!
Buffer 2 addresses:
0: 24BBEC63B53F3BCC
...
...
...
```

5.4.2 Operating Wupper-chaintest

The Wupper-chaintest tool does in one shot a complete DMA Read and Write transfer. It checks if the multiplied data is done correctly. This is done by multiplying the data in buffer 2 and compare the output of the multiplier in buffer 1 (shown earlier in Figure 2). The tool returns the number of errors out of 65536 loops as shown in Listing 14.

Listing 14: Output of Wupper-chaintest

```
daqmustud@gimone:$ ./wupper-chaintest
Reading data from buffer 1...
DONE!
Buffer 2 addresses:
0: 49A5A89745420D34
...
...
...
9: 5D37679AE79FA7C2
0 errors out of 65536
```

5.5 Wupper GUI

The Wupper Gui is a dedicated application for the example vhdl application, as described in 6. The concept of the Wupper GUI is based on the Wupper tools and has the same construction (see Figure 11). The GUI is developed with Qt version 5.5 (C++ based) [12] and gives the user a visual feedback of the Wupper's status/health. The GUI can only run if the device driver is loaded.

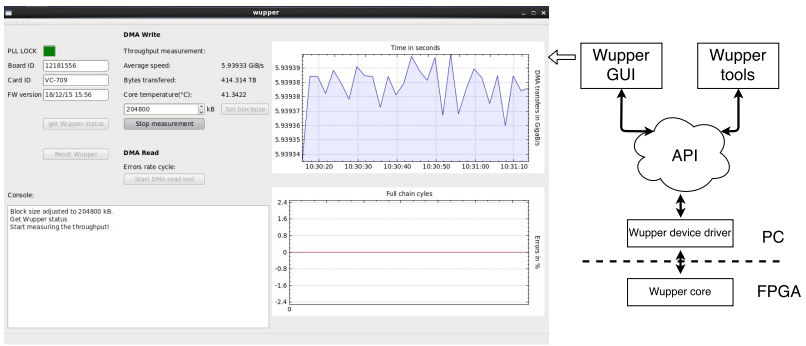


Figure 11: High and low level software overview block diagram.

5.5.1 Functional blocks and threaded programming

Multi-threading is used so functional blocks can run at the same time as the GUI. If multi-threading is not used, the GUI interface gets stuck. A thread starts a new process next to the main process. If another processor core is available, the thread will run on a separated core. By communicating via slots to the main process, the data is secured. There are two threads but only one of the threads can be used at the same time. The reason is that both threads use the same DMA ID, this will cause an error. The threads communicate with the Application Program Interface (API) to control and fetch the output of the logic. The output data communicate safely via a signal to the slots. Figure 12 shows an overview of the threaded programs in the Wupper GUI.

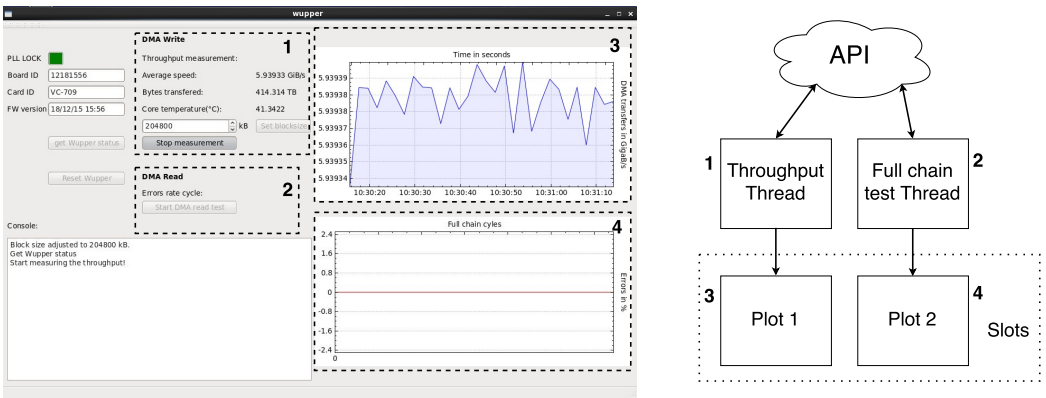


Figure 12: Threaded programs in the Wupper GUI

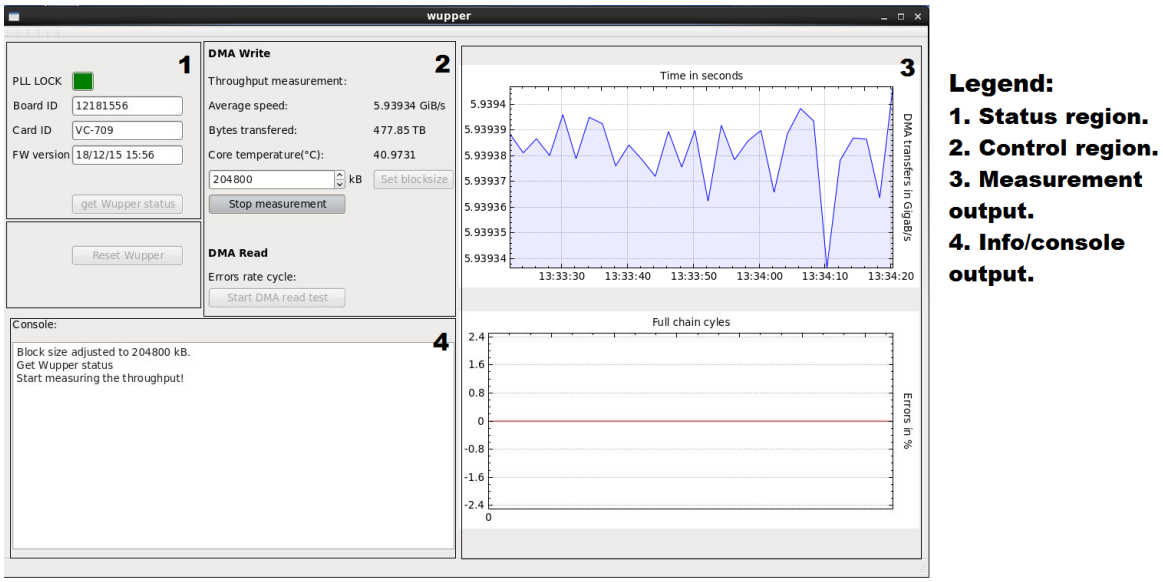
5.5.2 GUI operation

The GUI is separated in four regions (see Figure 13): status, control, measurement and an info region. The status region fetches the information about various parts of the FPGA on the VC-709 via the Wupper core, and about the core itself. When the user clicks on the "get Wupper status" button, it shows the internal PLL lock status, Board ID, Card ID and the firmware version.

The control region controls the logic inside Wupper through the API. The "Reset Wupper" button resets the application logic by resetting the DMA, flushing the FIFO's and reset the application values.

In the DMA Write section, the user can perform a DMA Write measurement. The user can configure the blocksize. The blocksize has effect on the speed, this is discussed in Appendix C. The measurement output is shown in the measurement region. The method of computing the throughput is different than the method of the Wupper-throughput tool. The fault is the wrong order of operations by misplacing brackets. The wrong method is $A/B * C = D$ instead of $A/(B * C) = D$.

In a similar way, the user can perform a DMA Read test and the output is shown in the plot in the measurement region. The info/console output region gives the user feedback of the application and the GUI.



Legend:
1. Status region.
2. Control region.
3. Measurement output.
4. Info/console output.

Figure 13: Screenshot of the example application GUI

6 Example application HDL modules

The example application, the user application inside the FPGA, replaces the counter with a pseudo-random data generator. Moreover the new feature in the application has the possibility to process data from the PC memory. A more detailed report about the example application can be found in [20]

The example application can be operated in two modes:

1. The random data generator directly sends data to the host via Wupper, this is referred to as "write only" or "half loop" test.

2. The content of the random data generator is wrote back to the FPGA, multiplied and sent to host again, this is referred as "read and write" or "full loop" test.

The example application is developed in VHDL, and the code is synthesized and implemented in Xilinx Vivado 2015.4 [13]. The example application is now part of the Wupper package on OpenCores.

6.1 Functional blocks

Figure 14 shows a detailed block diagram of the example application for Wupper. The Wupper core contains a list of addresses, this list is the register map. The values of the register map are implemented in the firmware as signals. The PC sees the signals as addresses. Wupper tools write values to these addresses which control the FPGA logic (see dashed lines in Figure 14).

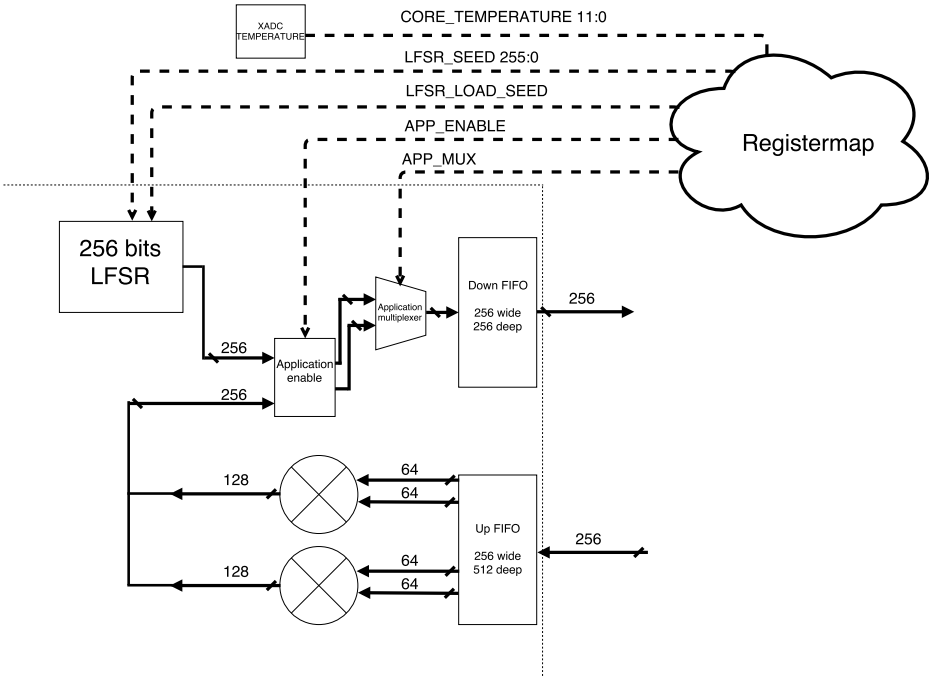


Figure 14: Overview of the example application

As introduced in the previous paragraph, one type of test possible with the example application is the "half loop": in such mode of operation, Wupper is fed by a random data generator based on a 256 bits Linear Feedback Shift Register (LFSR). An LFSR, as shown in Figure 15, consists of a number of shift registers which are fed back to the input. The feedback is manipulated by an XOR operation which creates a pseudo-random pattern. The ideal goal is to produce a sequence with a infinite length to prevent repetition. Repetition occurs by two factors, the feedback points/taps and the start value. The maximal length sequence can be approached by $2^n - 1$ [15]. Where the n is the number of shift registers. The 256 bits LFSR is a four stage Galois LFSR with taps at the registers 256, 254,251 and 246. The approach is explained in paper [16] by R. W. Ward and T.C.A. Molteno of the electronics group at the University of Otago. The software tools developed for the example application initialize the seed value by writing it to the register map thereafter the 1-bit $LFSR\_LOAD\_SEED$ signal is set to 1. This resets the LFSR process with a seed value.

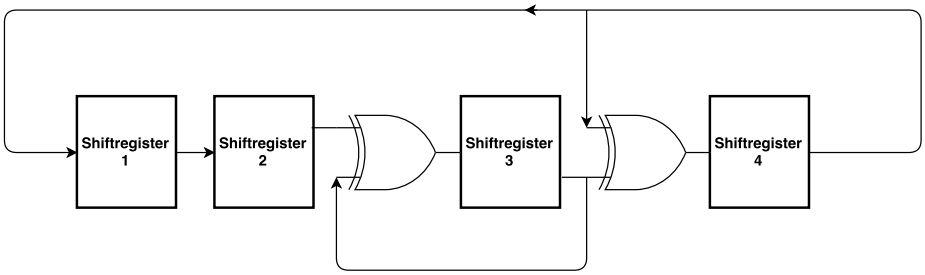


Figure 15: A 4 bit Linear Feedback Shift Register (LFSR)

For multiplication, the Xilinx multiplier IP block is used. The operations are based on the DSP48E1 [17] for the Virtex-7 series. There are two parallel multipliers used with two unsigned 64-bit inputs. To make the multiplier perform optimally at high clock rates, an 18 stage pipelining is used.

For monitoring the core temperature, a XADC IP block [18] is used. This is generated by Vivado's XADC wizard. The output signal of the block is connected to one register of the register map.

The 1-bit signal APP_MUX is attached to the select port of the application multiplexer. This enables the data flow to the down FIFO.

The signal APP_ENABLE enables the output of the LFSR and the multiplier. The 2-bits signal has three states:

- "00": No data flow, application is on standby.
- "01": Makes the example application enable 'high' causing data to flow only from the LFSR.
- "10": Makes the example application enable 'high' causing data to flow only from the multiplier.

The FIFO's are generated by Vivado's FIFO generator and using integrated common clock block RAMs. The clock is set to 250 MHz to reach the maximum theoretical throughput. The up FIFO is deeper to function as a buffer. This is an extra precaution. The reason is if the data is looped back in the application, both FIFO's can be full at the same time. If this occurs, the application stalls because of the loop back.

7 Customizing the application

7.1 connection of the DMA FIFOs

Wupper comes with an example application that is described in 6. The toplevel file for the user application is application.vhd

If you want to customize the example application for your own needs, the application can be stripped down to only containing the two FIFO cores. 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.

7.2 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 15: custom register types

```
type register_map_control_type is record
      STATUS_LEDS                    : std_logic_vector(7 downto 0);
            -- Board GPIO Leds
      LFSR_SEED_0                    : std_logic_vector(63 downto 0);
           -- Least significant 64 bits of the LFSR seed
      LFSR_SEED_1                    : std_logic_vector(63 downto 0);
           -- Bits 127 downto 64 of the LFSR seed
      LFSR_SEED_2                    : std_logic_vector(63 downto 0);
           -- Bits 191 downto 128 of the LFSR seed
      LFSR_SEED_3                    : std_logic_vector(63 downto 0);
           -- Bits 255 downto 192 of the LFSR seed
      APP_MUX                        : std_logic_vector(0 downto 0);
            -- Switch between multiplier or LFSR.
      --    * 0 LFSR
      --    * 1 Loopback

      LFSR_LOAD_SEED                 : std_logic_vector(64 downto 64);
          -- Writing any value to this register triggers the LFSR
         module to reset to the LFSR_SEED value
      APP_ENABLE                     : std_logic_vector(0 downto 0);
            -- 1 Enables LFSR module or Loopback (depending on APP_MUX)
      -- 0 disable application

      I2C_WR                         : bitfield_i2c_wr_t_type;
      I2C_RD                         : bitfield_i2c_rd_t_type;
      INT_TEST_4                     : std_logic_vector(64 downto 64);
          -- Fire a test MSIx interrupt #4
      INT_TEST_5                     : std_logic_vector(64 downto 64);
          -- Fire a test MSIx interrupt #5
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-1.0.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] ARM AMBA AXI bus standard specification page
http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

[3] Xilinx website about the PCI Express core
http://www.xilinx.com/products/intellectual-property/7_Series_Gen_3_PCI_Express.htm

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

[5] PG023: The User guide for the Xilinx Virtex7 PCI Express core
http://www.xilinx.com/support/documentation/ip_documentation/pcie3_7x/v3_0/pg023_v7_pcie_gen3.pdf

[6] PG156: The User guide for the Xilinx Ultrascale PCI Express core
http://www.xilinx.com/support/documentation/ip_documentation/pcie3_ultrascale/v4_2/pg156-ultrascale-pcie-gen3.pdf

[7] Avoiding repeating passwords for CVS and SVN - ATLAS
https://confluence.slac.stanford.edu/display/Atlas/Avoiding+repeating+passwords+for+CVS+and+SVN

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

[9] Official software readme
http://opencores.org/websvn,filedetails?repname=virtex7_pcie_dma&path=%2Fvirtex7_pcie_dma%2Ftrunk%2FhostSoftware%2Fwupper_tools%2FREADME.txt

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

[11] UG908: Programming and Debugging using Vivado
http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug908-vivado-programming-debugging.pdf

[12] Qt official Wiki page
http://wiki.qt.io/Main_Page

[13] Xilinx Vivado Design Suite User Guide 2015.4
http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug973-vivado-release-notes-install-license.pdf

[14] Official QThread documentation
http://doc.qt.io/qt-5/qthread.html

[15] Tutorial: Linear Feedback Shift Registers (LFSRs). An article abstracted from the book Bebop to the Boolean Boogie (An Unconventional Guide to Electronics)
http://www.eetimes.com|/document.asp?doc_id=1274550

[16] Table of Linear Feedback Shift Registers by R. W. Ward and T.C.A. Molteno of the electronics group at the University of Otago
http://www.physics.otago.ac.nz/reports/electronics/ETR2009-1.pdf

[17] Xilinx 7 Series DSP48E1 Slice: User guide
http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

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

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

[20] Oussama el Kharraz Alami - Development of an application for Wupper a PCIe Gen3 DMA for Virtex 7
http://opencores.org/websvn,filedetails?repname=virtex7_pcie_dma&path=%2Fvirtex7_pcie_dma%2Ftrunk%2Fdocumentation%2Fexample_application%2Finternship-wupper.pdf

List of Figures

List of Tables

Appendix A WUPPER register map, version 1.0

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				
			RD_POINTER	127:64	W	PC Read Pointer
			WRAP_AROUND	12	W	Wrap around
			READ_WRITE	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				
			RD_POINTER	127:64	W	PC Read Pointer
			WRAP_AROUND	12	W	Wrap around
			READ_WRITE	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
			CURRENT_ADDRESS	63:0	R	Current Address
...						
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
			CURRENT_ADDRESS	63:0	R	Current Address
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.
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)

Address	PCIe	Name/Field		Bits	Type	Description
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

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	64:0	W	Interrupt Address
...						
0x0070	0,1	INT_VEC_7				
			INT_CTRL	127:96	W	Interrupt Control
			INT_DATA	95:64	W	Interrupt Data
			INT_ADDRESS	64:0	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, 1.0 formatted as 0x0100
0x0010	0	BOARD_ID_TIMESTAMP		39:0	R	Board ID Date / Time in BCD format YYMMDDhhmm
0x0020	0	BOARD_ID_SVN		15:0	R	Board ID SVN Revision
0x0030	0	STATUS_LEDS		7:0	W	Board GPIO Leds
0x0040	0	GENERIC_CONSTANTS				
			INTERRUPTS	15:8	R	Number of Interrupts
			DESCRIPTORS	7:0	R	Number of Descriptors
0x0050	0	CARD_TYPE		63:0	R	Card Type: * 709 (0x2c5) VC709 * 710 (0x2c6) HTG710 * 711 (0x2c7) BNL711
Application Specific						
0x1000	0,1	LFSR_SEED_0		63:0	W	Least significant 64 bits of the LFSR seed
0x1010	0,1	LFSR_SEED_1		63:0	W	Bits 127 downto 64 of the LFSR seed
0x1020	0,1	LFSR_SEED_2		63:0	W	Bits 191 downto 128 of the LFSR seed

Address	PCIe	Name/Field	Bits	Type	Description
0x1030	0,1	LFSR_SEED_3	63:0	W	Bits 255 downto 192 of the LFSR seed
0x1040	0,1	APP_MUX	0:0	W	Switch between multiplier or LFSR. * 0 LFSR * 1 Loopback
0x1050	0,1	LFSR_LOAD_SEED	any	T	Writing any value to this register triggers the LFSR module to reset to the LFSR_SEED value
0x1060	0,1	APP_ENABLE	0:0	W	1 Enables LFSR module or Loopback (depending on APP_MUX) 0 disable application
House Keeping Controls And Monitors					
0x2300	0	MMCM_MAIN_PLL_LOCK	0	R	Main MMCM PLL Lock Status
0x2310	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¿
0x2320	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
0x2330	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
0x2340	0	FPGA_CORE_VCCINT	11:0	R	XADC voltage measurement VCCINT = (FPGA_CORE_VCCINT *3.0)/4096

Address	PCIe	Name/Field	Bits	Type	Description
0x2350	0	FPGA_CORE_VCCAUX	11:0	R	XADC voltage measurement VCCAUX = (FPGA_CORE_VCCAUX *3.0)/4096
0x2360	0	FPGA_CORE_VCCBRAM	11:0	R	XADC voltage measurement VCCBRAM = (FPGA_CORE_VCCBRAM *3.0)/4096
0x2370	0,1	FPGA_DNA	63:0	R	Unique identifier of the FPGA
0x2800	0	INT_TEST_4	any	T	Fire a test MSIx interrupt #4
0x2810	0	INT_TEST_5	any	T	Fire a test MSIx interrupt #5
Wishbone					
0x4000	0	WISHBONE_CONTROL			
		WRITE_NOT_READ	32	W	wishbone write command wishbone read command
		ADDRESS	31:0	W	Slave address for Wishbone bus
0x4010	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
		DATA	31:0	W	Wishbone
0x4020	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
0x4030	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

Table 8: FELIX register map BAR2

Appendix B Configuration of the core

The Xilinx PCIe core is configured as a PCI express Gen3 (8.0GT/s) core 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 disabled. The reference clock frequency is 100MHz and the only option for the AXI4-Stream interface is 256 bit at 250MHz. Detailed configuration parameters can be seen in Figure 16 to 26.

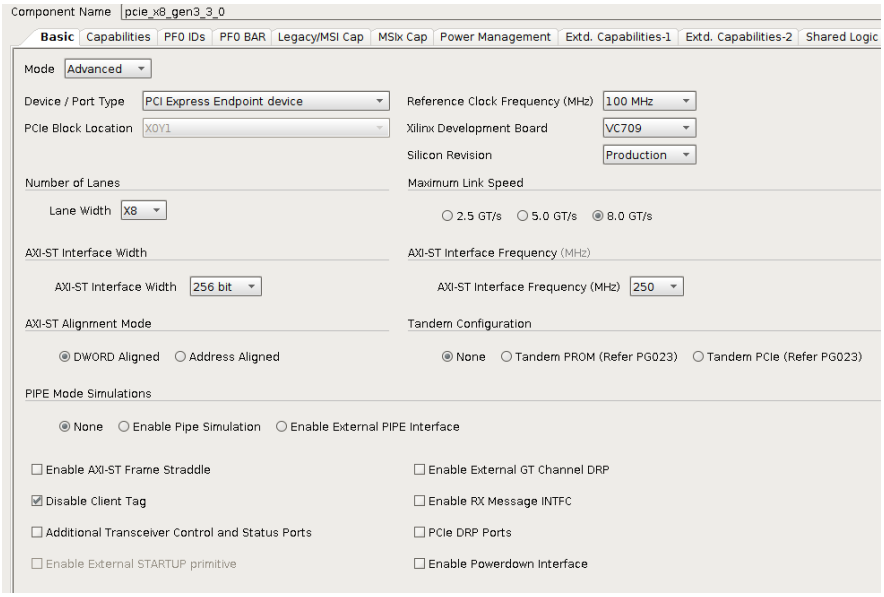


Figure 16: PCIe core configuration in Vivado [Basic]

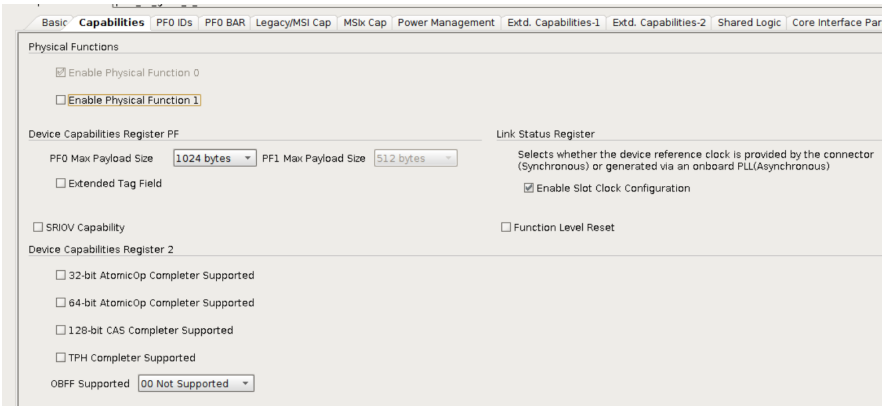


Figure 17: PCIe core configuration in Vivado [Capabilities]

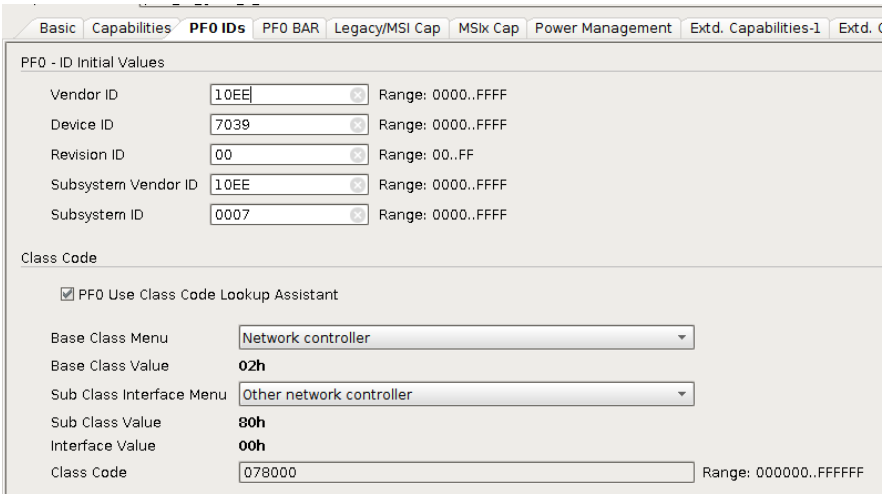


Figure 18: PCIe core configuration in Vivado [PF0 IDs]

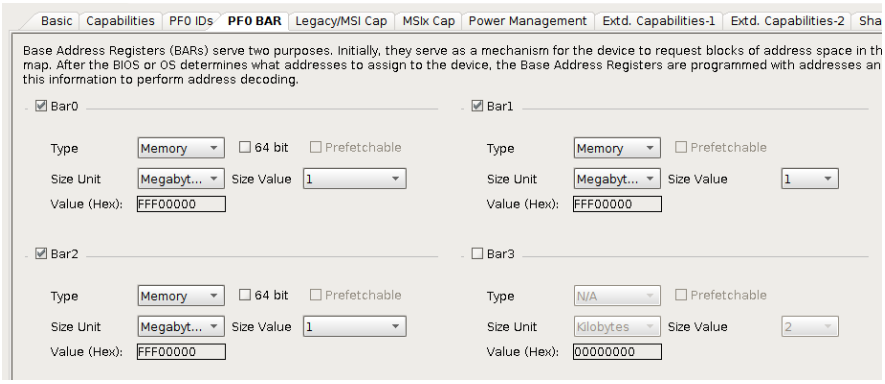


Figure 19: PCIe core configuration in Vivado [PF0 BAR]

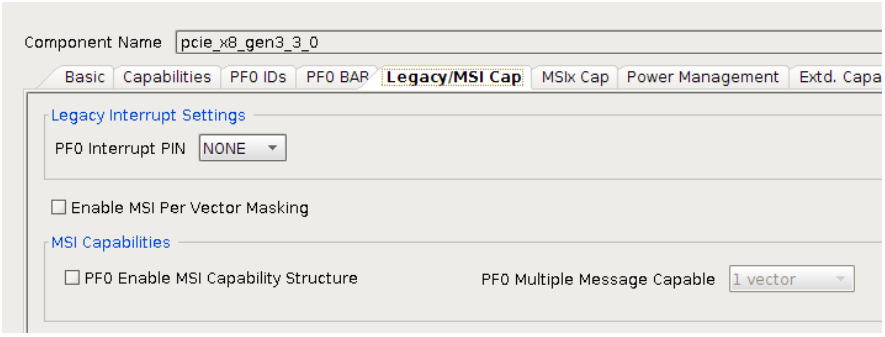


Figure 20: PCIe core configuration in Vivado [Legacy/MSI Cap]

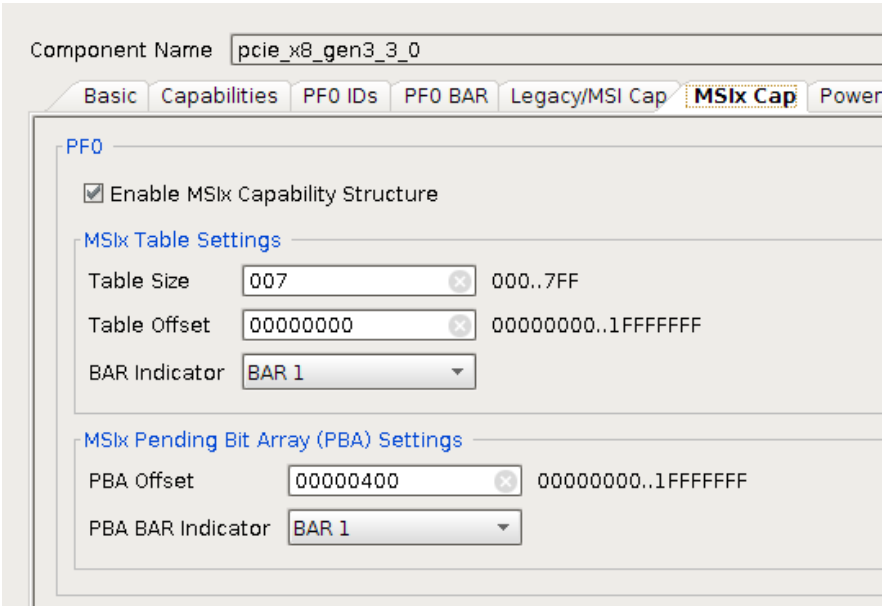


Figure 21: PCIe core configuration in Vivado [MSIx]

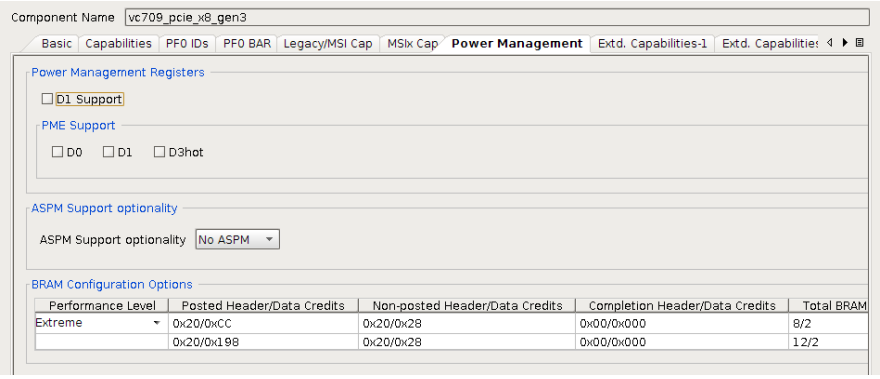


Figure 22: PCIe core configuration in Vivado [Power Management]

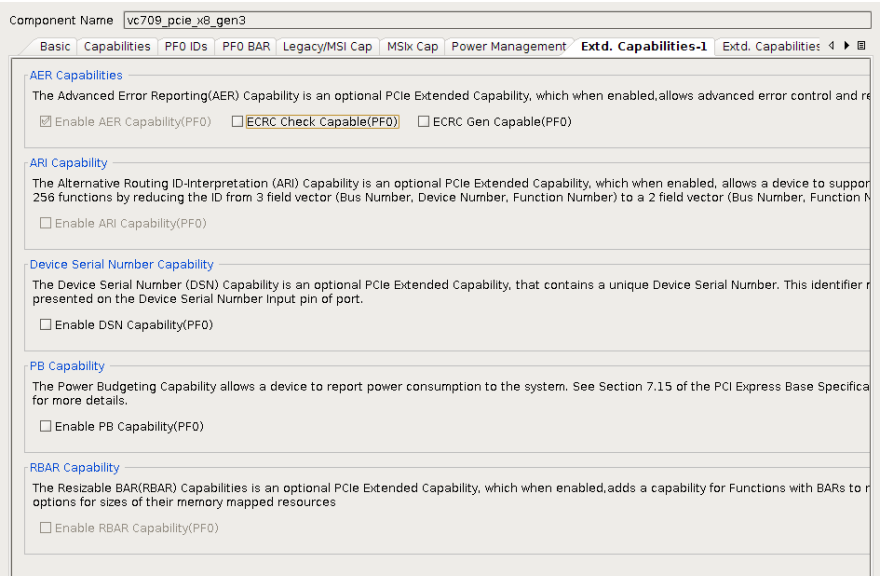


Figure 23: PCIe core configuration in Vivado [Extd. Capabilities 1]

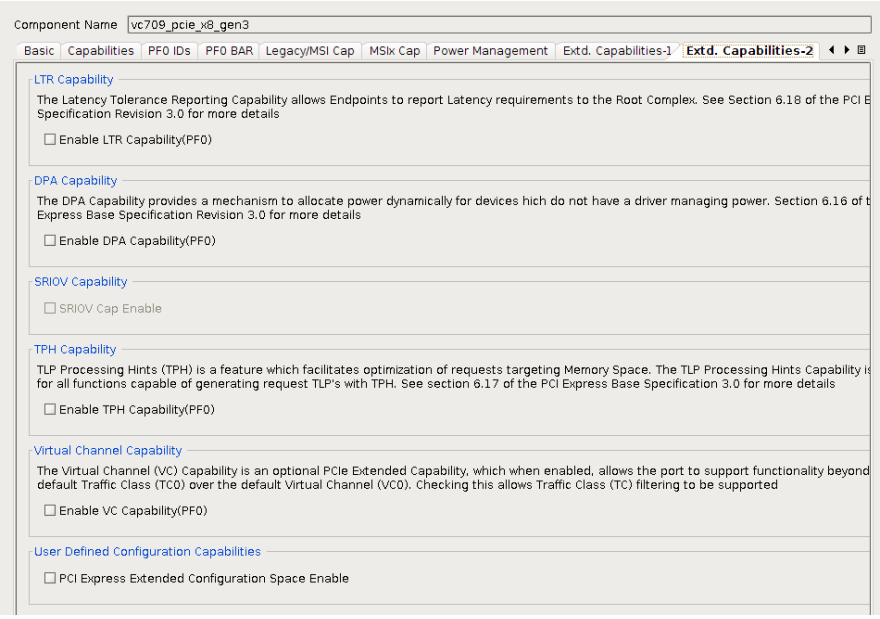


Figure 24: PCIe core configuration in Vivado [Extd. Capabilities 2]

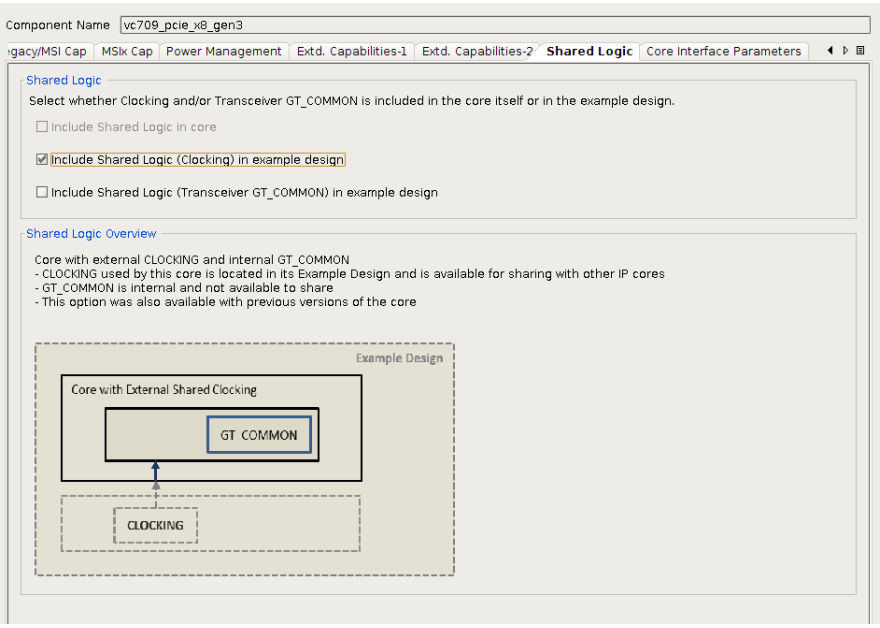


Figure 25: PCIe core configuration in Vivado [Shared LogicMSIx]

Figure 26: PCIe core configuration in Vivado [Core Interface Parameters]

Appendix C Benchmark: block size versus write speed

The Wupper GUI makes it possible for the users to configure the block size value. This appendix shows how much effect the block size have on the write speed.

During a DMA write 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 transfered. 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 27. 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).

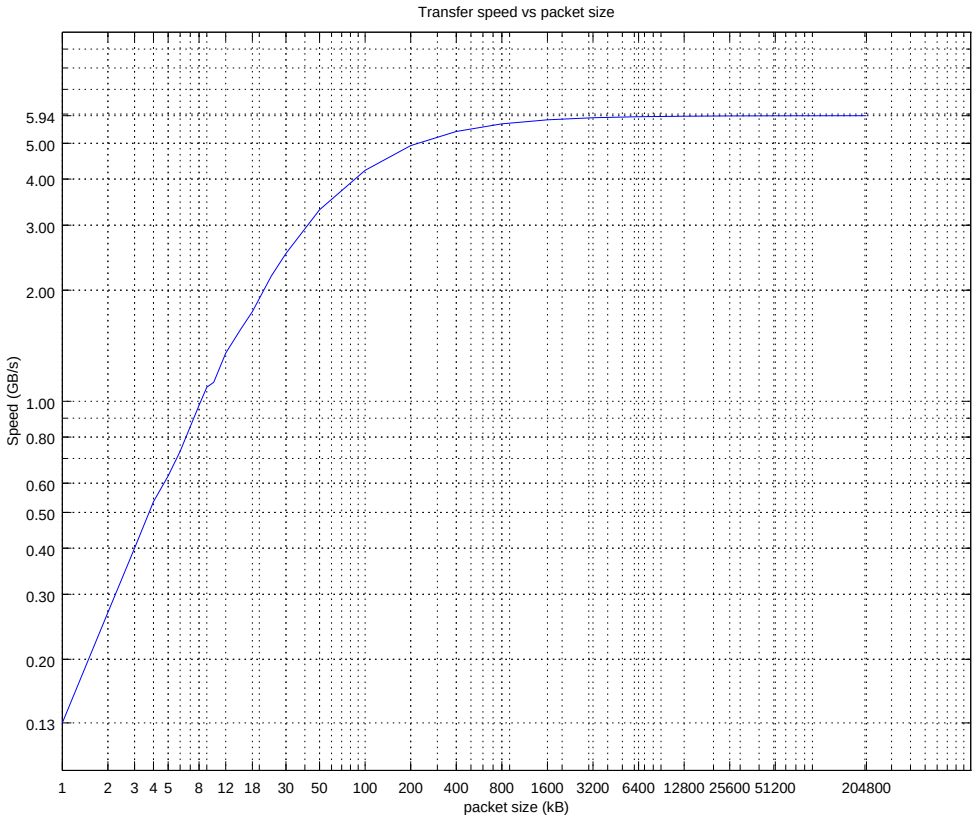


Figure 27: Transfer speed vs packet size