Development of an application for
Wupper a PCIe Gen3 DMA for Virtex 7

Oussama el Kharraz Alami

29-1-2016

Studentnumber: 500639457
Course: Bachelor's Degree, Electrical Electronic and Communications Engineering
School: Amsterdam University of Applied Sciences
Supervisors:
Andrea Borga, Nikhef, Amsterdam, The Netherlands.
Frans Schreuder, Nikhef, Amsterdam, The Netherlands.
Wim Dolman, Amsterdam University of Applied Sciences, Amsterdam, The Netherlands.

Contents

1 Introduction

1.1 Wupper package

Approaching a development package bottom up, the Wupper core[1], is a module of the FELIX firmware and provides an interface for the Direct Memory Acces (DMA) in the Xilinx Virtex-7 FPGA hosted on the VC-709. This FPGA has a PCIe Gen3 hard block integrated in the silicon [1]. 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.

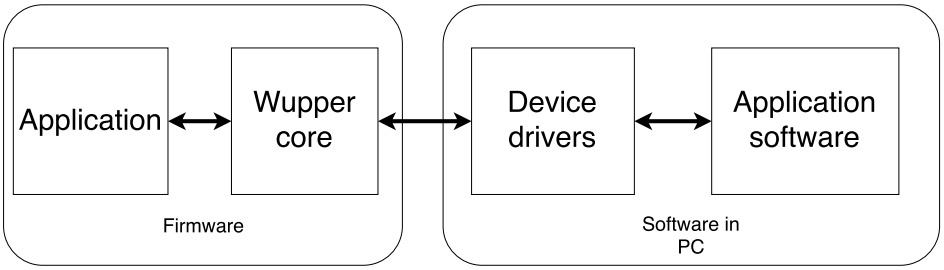


Figure 1: Wupper package overview

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 [6]. 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
- 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.

[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

2 Internship

2.1 Goal

Given the background provided in the previous chapter, my contribution to this project is to develop an example application that checks the health of the core in both directions. The application also checks whether the data that is written into the PC memory is valid. The development contains software (Wupper tools) and an HDL example application.. In addition, a GUI will be developed for the application. Besides those main activities, the device driver and tools developed for Wupper used in the FELIX application has to be ported and tested for the Wupper version published on OpenCores. Appendix B shows the global schedule of the activities I carried out during the development of the application.

2.2 Topics

As introduced in the previous paragraph, the aim of this internship is to develop a test application for Wupper. Its purpose is to benchmark the robustness and performance of the Wupper core. To reach this goal, at first the structure of the Wupper package needs to be understood. This requires grasping how to transfer data using the Wupper core and what is needed for controlling the FPGA using software. Each specific sub-task of the work carried out for this development is detailed in the following sub-paragraphs.

2.3 Drivers and tools

The drivers and tools are the low level software parts which control the logic of the Wupper core. A set of device drivers are used to: (i) initialize the FPGA PCIe card and control DMA transfers, (ii) perform I/O operations on registers inside the FPGA, (iii) allocate memory buffers in the host PC to be used as landing areas for data transfers. The Wupper-tools, a collection of tools which is made in the programming languages C and C++, are used to control the logic through the drivers.The Wupper-tools are intended to be a subset of the tools developed for Wupper in the framework of the FELIX project, meaningful for the OpenCores users. The key to implement the Wupper tools is to understand how the original tools work and which parts can be reused.

2.4 VHDL example application code

The purpose of the VHDL example application is to show the essentials of the DMA transfer function of Wupper. Prior to the development described in this report, there was only a simple 32-bit counter used to test the data flow in only one direction, i.e. from the FPGA to the PC. Understanding the Wupper core will lead to a renewed version which should transfer 256 bit data with high speed, both in the up and down direction.

2.5 Developing a GUI

Prior to this development operating the FPGA card was done via a terminal. There is a certain order to get it working which can be very complicated for the users. The solution is to design a Graphical User Interface (GUI) which can be run on Linux systems.

3 Wupper package

In this section, the firmware, drivers, and tools of Wupper (see Figure 2) together with its working principle are explained. For more detailed information about the internals and the core please refer to the official Wupper documentation [3].

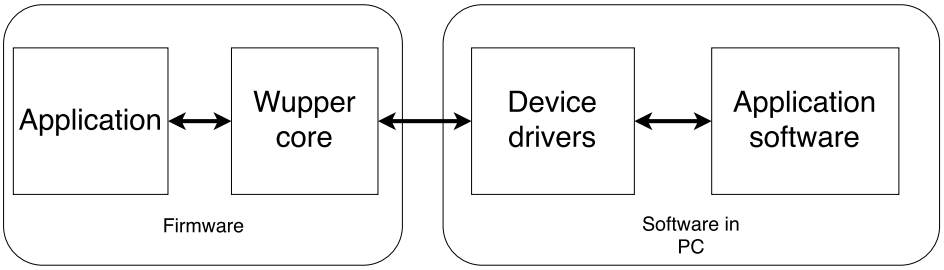


Figure 2: Wupper package overview

3.1 Wupper core

An Engine, like Wupper, moves data bidirectionally to a memory 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 3 shows a complete 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.

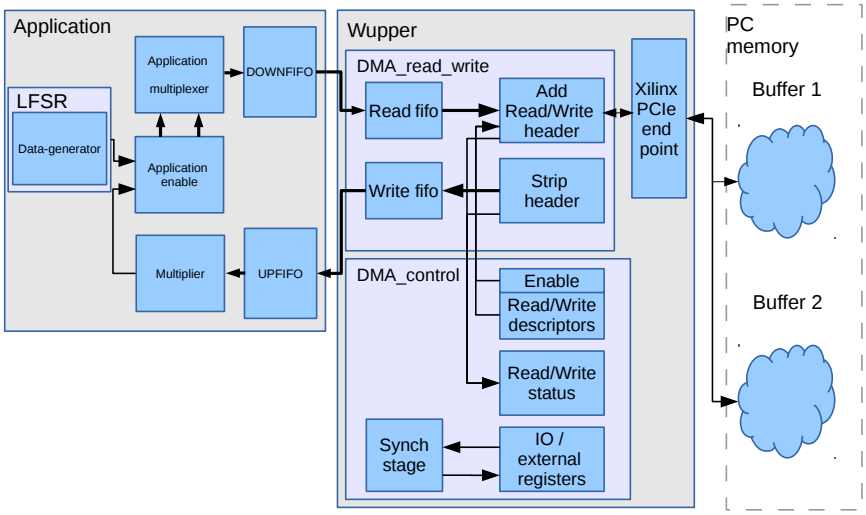


Figure 3: Overview of the HDL modules in the Wupper package

3.1.1 Xilinx PCIe End Point

The Virtex-7 XC7VX690T-2FFG1761C on the VC-709 board has an integrated endpoint for PCI Express Gen3 [1]. 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 4 shows a simplified model of the firmware stack. Configuration of the core is explained in section 3.2 of the official documentation of Wupper [4].

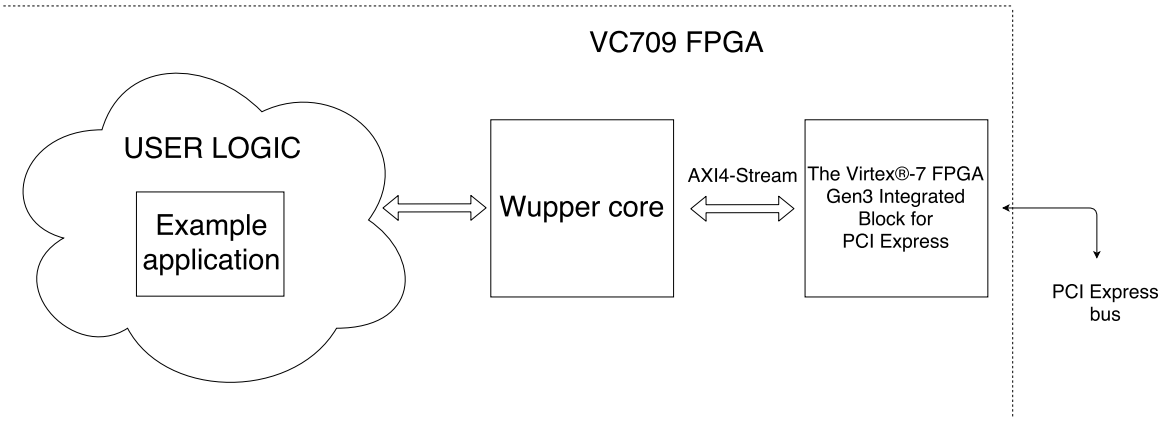


Figure 4: Block diagram of the logic in the VC-709 FPGA

3.1.2 Core control

The DMA control (DMA_control in Figure 3) 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 3 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 work done for this report defines and acts on registers in BAR2, as summarized in Appendix C.

As previously shown in Figure 3, the example application core consists of multiple function blocks which are attached to the register map. This makes it possible to control the benchmark application from the PC. A complete overview of the register map can be found in the official documentation of Wupper [3].

3.1.3 DMA read/write

The DMA read and write (DMA_read_write in Figure 3) module handles the transfer from the FIFO's according to the direction specified by the descriptors. If data shifts into the down FIFO, a non-empty flag will be asserted to start the DMA write process, this direction of the flow is defined as the "down link". This process reads the descriptors and creates a header with the information. The header is added when the data shifts out of the down 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 "up link". The information in the header will be parsed by the DMA control and the data fed to the up FIFO.

3.2 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. 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 2014.4 [2]. The example application is now part of the Wupper package on OpenCores.

3.2.1 Functional blocks

Figure 5 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 5).

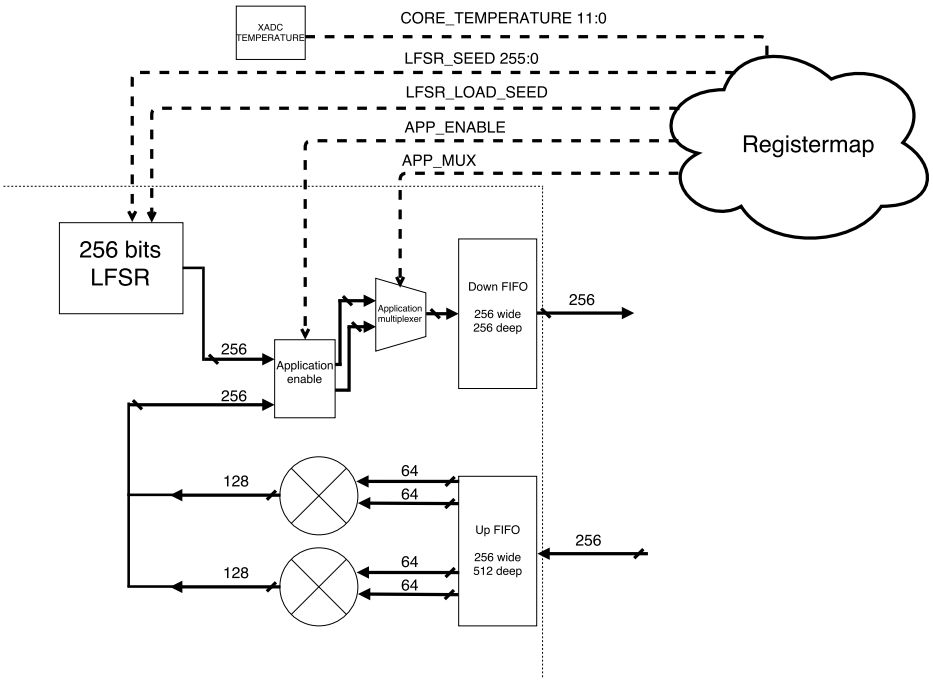


Figure 5: Overview of the example application

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 [7], 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 $2^n - 1$ [7]. 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 [9] 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.

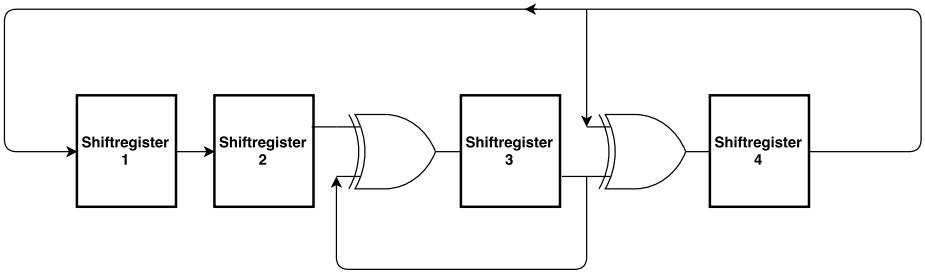


Figure 6: A 4 bit Linear Feedback Shift Register (LFSR)

For multiplication, the Xilinx multiplier IP block is used. The operations are based on the DSP48E1 [8] 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 [10] 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.

3.3 Device driver and Wupper tools

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 3). 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 section 6.1.2 en 6.1.3 of the official documentation of Wupper [5].

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. Along with the collection of tools derived from the FELIX tool suite, the Wupper-dma-transfer and Wupper-chaintest had been added as new features for the OpenCores' benchmark. As mentioned before, the purpose of those applications is to check the health of the Wupper core.

The Wupper tools collection comes with a readme [11], this explains how to compile and run the tools. Most of the tools have an -h option to provide helpful information. The table below shows a list of the tools derived from the original flxtools suite and their description.

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".

For the Wupper package on OpenCores two extra tools had been newly developed to target specific benchmark requirement for the generic example application: Wupper-dma-transfer and Wupper-chaintest. In the next paragraphs a detailed description of such tools and their operation is given.

3.3.1 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 1.

Listing 1: 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 2.

Listing 2: 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 3.

Listing 3: 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 4.

Listing 4: 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 5.

Listing 5: Performing a DMA read and DMA write

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

3.3.2 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 3). The tool returns the number of errors out of 65536 loops as shown in Listing 6.

Listing 6: 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
```

3.4 Wupper GUI

The concept of the Wupper GUI is based on the Wupper tools and has the same construction (see Figure 7). 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.

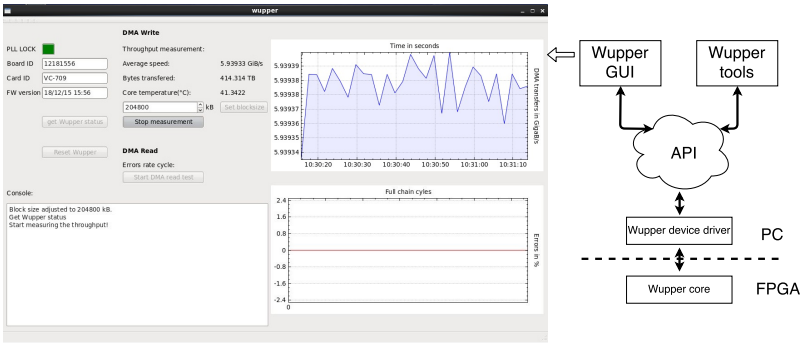


Figure 7: High and low level software overview block diagram.

3.4.1 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 8 shows an overview of the threaded programs in the Wupper GUI.

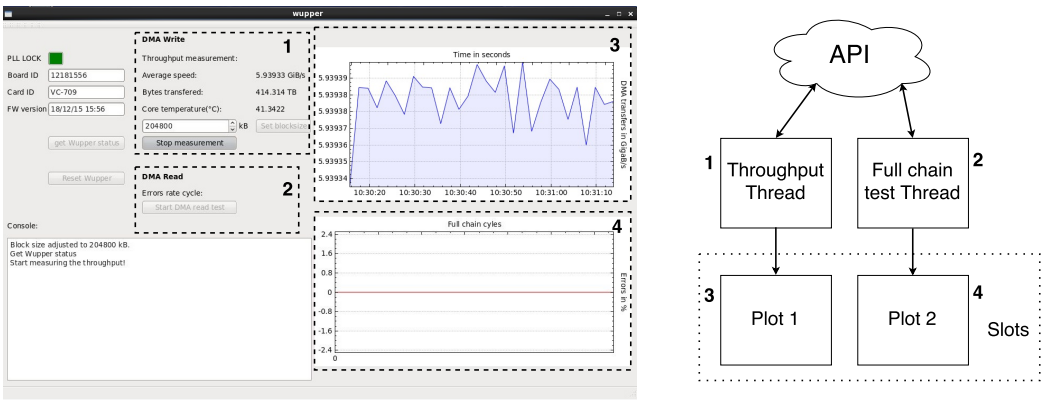


Figure 8: Threaded programs in the Wupper GUI

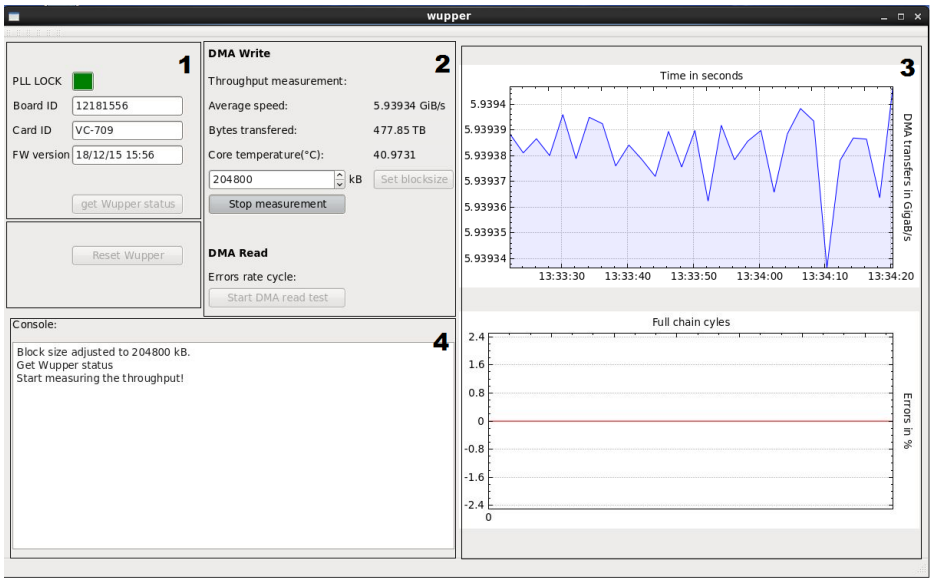
3.4.2 GUI operation

The GUI is separated in four regions (see Figure 9): 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 A. 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 9: Screenshot of the example application GUI

4 Verification

In this section the verification of the example application HDL modules is being discussed. During the development of the example application HDL modules, the functional blocks are simulated separately. The HDL blocks are simulated with Mentor Graphics Questasim using a scripting language called Tool Command Language (TCL) [15].

4.1 Randomness of the data generator

The heart of the example application HDL modules is the data generator. This provides data to the Wupper core. The data generator is based on the Linear Feedback Shift Register (LFSR). There are other techniques for generating (pseudo) random data such as the Linear Congruential Generator and Multiple Recursive Generators. The problem of these methods are the need of a lot of multiplication computing power [14]. This requires a lot of digital logic / DSP slices.

It is important that the output data is random. To check this pattern Questasim is used for simulation. The first signal in Figure 10 shows the output signal of the LFSR module using the approach by R.W. Ward and T.C.A Molteno [9]. Questasim can plot the data in the waveform viewer which gives a nice overview of the randomness.

Figure 10: Randomness of the data generator based on a 256-bits LFSR.

4.2 Verification flow

After an expected behaviour of the application HDL modules, the complete Wupper package needs to be verified. The expected behaviour of the full Wupper pacakage is that the output of the LFSR is first sent to a buffer in the PC memory. This buffer will later be transferred back into the FPGA by means of a DMA read cycle and fed into the input of the multipliers. Meanwhile, a second transfer is started simultaneously to transfer the multiplied data into a second buffer of PC memory. Simulating the behaviour of transactions to PC memory is possible but very complex. In this case it is efficient to test the behavior real-time with Vivado's Integrated Logic Analyzer (ILA) [16]. ILA allows monitoring signals in real-time. The ILA core uses RAM blocks inside the FPGA as storage elements for the data in between acquisitions and subsequent transfers to host via the JTAG interface. It is therefore obvious that a combination of monitored signals and the depth of an acquisition will impact the resource consumption in the FPGA when equipped with debug probes. It is therefore crucial to carefully select the signals that one wants to monitor and the depth, i.e. number of samples, one wants to get per acquisition.

When the probes are set properly and the triggers are armed, the next step in this process is operating the logic. This is done on the host PC via the Wupper-tools (described in Paragraph 3.3). The tools will activate the triggers and create an event. This event will acquire signal status which can be used for verifying the behaviour. This approach tests at the same time the HDL part and the low level software part. This approach and resources used during the verification is displayed below in Figure 11.

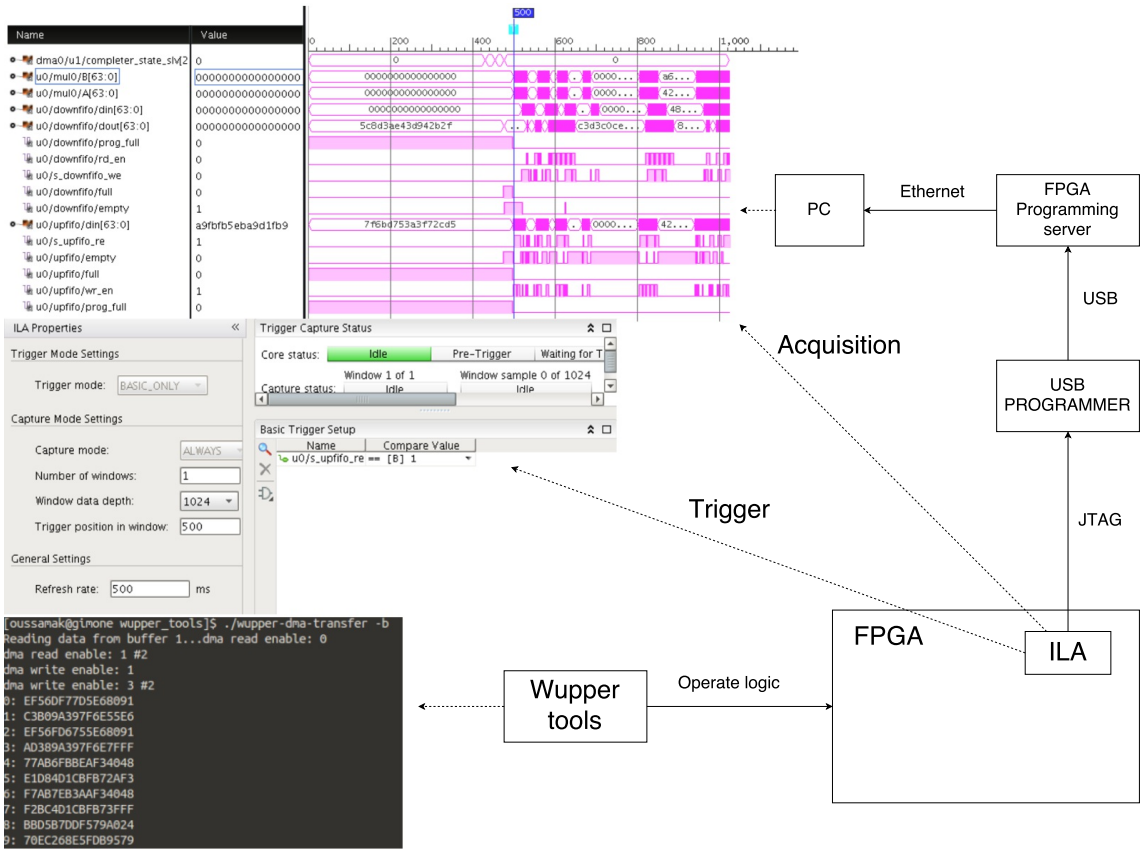


Figure 11: View of the verification flow

The output of the LFSR is fed back to the input of the multipliers. To verify the multiplication, the tool Wupper-chaintest is developed. The tool activates the flow from the data generator to PC memory and back to the PC memory through the multipliers. As described in Section 3.3.2, the tool reserves two buffers inside the target PC. The data that is stored in buffer 2, are the inputs of the multipliers. This data is multiplied and verified with the output of the data that is stored in buffer 1. This is compared by the tool and returns an overview of errors that occur.

The high level software has the same purpose as the Wupper-tools. This makes it easy to implement it in the Wupper GUI. As mentioned before, the tools are based on programming languages C and C++ while Qt is based on C++. Which makes it possible to port the tools into Qt.

5 Conclusion

The Wupper package is published on OpenCores and includes an example application, Wupper GUI and two Wupper tools. The example application have two functionality, the half-loop test and full loop test. The half loop test is a write only test. The random data generator sends data to the host via the Wupper core. This test can be fired using the Wupper-dma-transfer tool. The second functionality, the full loop test, reads data from the PC memory through Wupper, data is multiplied and writes back via Wupper to the PC memory. In combination with the Wupper-chaintest, the Wupper core can be tested on corrupt data. A random data generator based on a Linear Feedback Shift Register (LFSR) is used for data generation. This is used to test the Wupper core with every combination of data. The Wupper tools are command line tools and are hard to handle without the manual. Wupper GUI makes it possible to give a clear graphical view of the performance of the Wupper core. The verification shows the expected behaviour of the complete Wupper package. By controlling, monitoring and Read/Write transfers tests, the example application benchmarks the essentials of the Wupper core. The source code of the example application, Wupper tools and Wupper GUI are open source which makes the user easier to develop an application for the Wupper core.

Appendix A 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 12. 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).

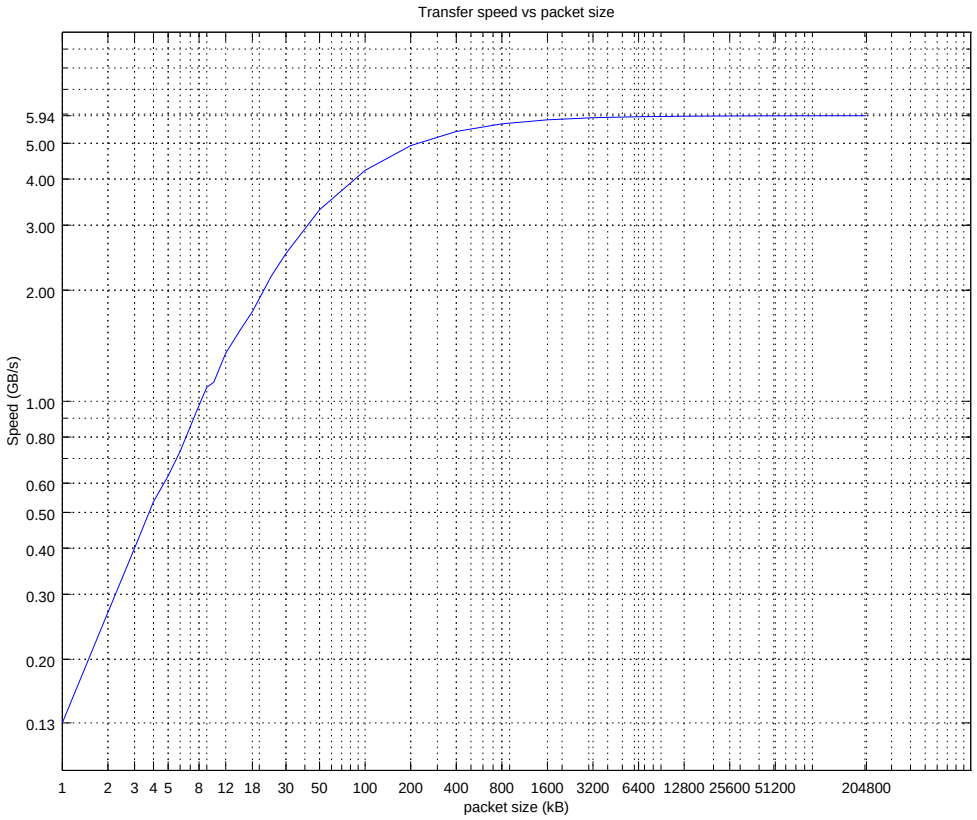


Figure 12: upfifo and downfifo are full.

Appendix B Problem solving: solution on different stream congestion and data corruption

This appendix shows the approach of the problem during verification of the example application. During a write action (FPGA $\rightarrow$ PC), data flows into the PC memory. But when a DMA read action (PC $\rightarrow$ FPGA) is performed, the Wupper tool got stuck. The result of debugging the tool shows that the strange behaviour occurs during a DMA_WAIT function. This function waits on the assertion of the descriptor completion signal from the Wupper core, this signal informs the host PC that the DMA action is completed. The signal comes from the Wupper core, this allows to take a closer look at the HDL module. The ILA core allows to fire an immediate trigger to capture the status of the signals. First thing that attract attention is that both FIFO's are full as shown below in Figure 13.

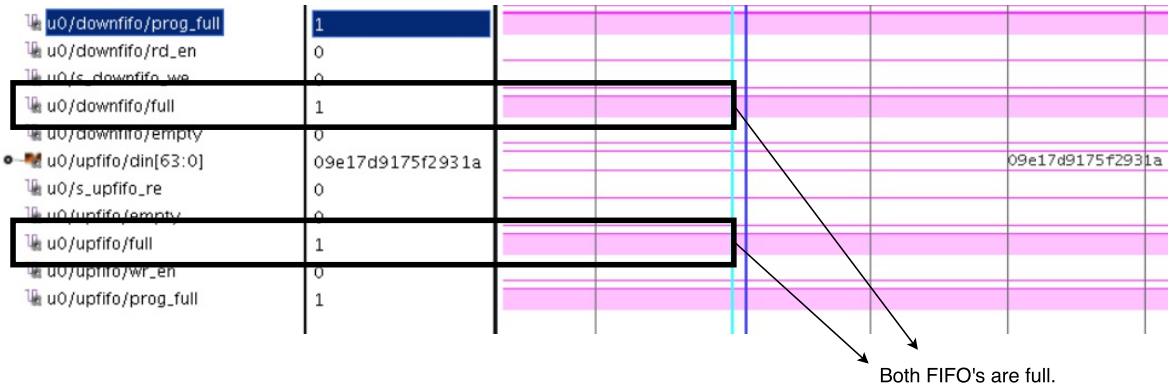


Figure 13: upfifo and downfifo are full.

In order to resolve this issue, three changes where required:

- Editing the DMA_WAIT function: the function contains a timer which stops executing itself after a certain period and returns an error message. This measure allows the user clarity.
- Increasing the depth of the Up FIFO: by expanding the Up FIFO, the example application have more capacity so more room for processing data.
- Adding application enable block in the example application: As described before in paragraph 3.2.1, to prevent data congestion in the example application.

Appendix C Application Base Address Region

The registers in BAR2, which is shown below in Table 1, is dedicated to user application.

Address	Name/Field	Bits	Type	Description
Bar2				
0x0000	REG_BOARD_ID	39:0	R	Board ID
0x0010	REG_STATUS_LEDS	7:0	R/W	Board GPIO Leds
0x0040	REG_CARD_TYPE	63:0	R	Card type information
Monitor Registers				
0x0300	REG_PLL_LOCK	19:0	R	PLL lock status
0x1060	INT_TEST_2	any	T	Fire a test MSIx interrupt #2
0x1070	INT_TEST_3	any	T	Fire a test MSIx interrupt #3
Example application register				
0x2000	REG_LFSR_SEED_0	63:0	R/W	Seed value 127:0
0x2010	REG_LFSR_SEED_1	63:0	R/W	Seed value 256:128
0x2020	REG_APP_MUX	1	R/W	Select of the application multiplexer
0x2030	REG_LFSR_LOAD_SEED	1	R/W	Initialize the seed value in to the data generator
0x2040	REG_APP_ENABLE	2:0	R/W	Enables the application
0x310	REG_CORE_TEMPERATURE	4:0	R	XADC temperature core

Table 1: Register map BAR2

References

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

[2] Xilinx Vivado Design Suite User Guide 2014.4
http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_4/ug973-vivado-release-notes-install-license.pdf

[3] Official Wupper documentation:
http://opencores.org/websvn,filedetails?repname=virtex7_pcie_dma&path=%2Fvirtex7_pcie_dma%2Ftrunk%2Fdocumentation%2Fpcie_dma_core.pdf

[4] Official Wupper documentation: Configuration of the core
http://opencores.org/websvn,filedetails?repname=virtex7_pcie_dma&path=%2Fvirtex7_pcie_dma%2Ftrunk%2Fdocumentation%2Fpcie_dma_core.pdf#subsection.3.2

[5] Official Wupper documentation: Loading the driver
http://opencores.org/websvn,filedetails?repname=virtex7_pcie_dma&path=%2Fvirtex7_pcie_dma%2Ftrunk%2Fdocumentation%2Fpcie_dma_core.pdf#subsection.6.1

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

[7] 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

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

[9] 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

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

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

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

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

[14] A note on random number generation
https://cran.r-project.org/web/packages/randtoolbox/vignettes/fullpres.pdf

[15] Scripting language TCL
https://en.wikipedia.org/wiki/Tcl

[16] PG172: ILA core
http://www.xilinx.com/support/documentation/ip_documentation/ila/v5_0/pg172-ila.pdf