Virtex-7 PCIe Engine

Frans Schreuder, Andrea Borga

17-04-2015

Contents

Revision History

Revision	Date	Author(s)	Description
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

The PCIe Engine is 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. 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 FPGA Gen3 Integrated Block for PCI Express (PCIe) [3] [5].

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.

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

For synthesis and implementation of the cores, it is recommend to use Xilinx Vivado 2014.2. 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 2014.2 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 Core architecture

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. Figure 1 shows the structure of the PCIe Engine design. The PCIe Engine is divided in two parts:

1. DMA Control
   This is the entity in which the descriptors are parsed and fed to the engine, and where the status register of every descriptor can be read back through PCIe. 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 two 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.

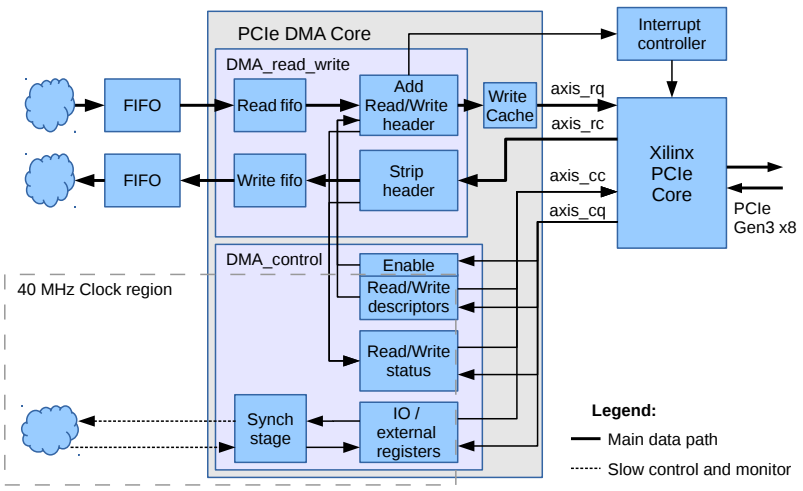

Figure 1: Structure of the PCIe Engine

Figure 1 also shows the sync stage for the IO and external registers. The user space registers are synchronized to a slower clock (e.g. 40 MHz) in order to relax timing closure of the

design. To make sure no glitches or latch-up will occur, the synchronization stage does not update during a write cycle of the DMA Control process.

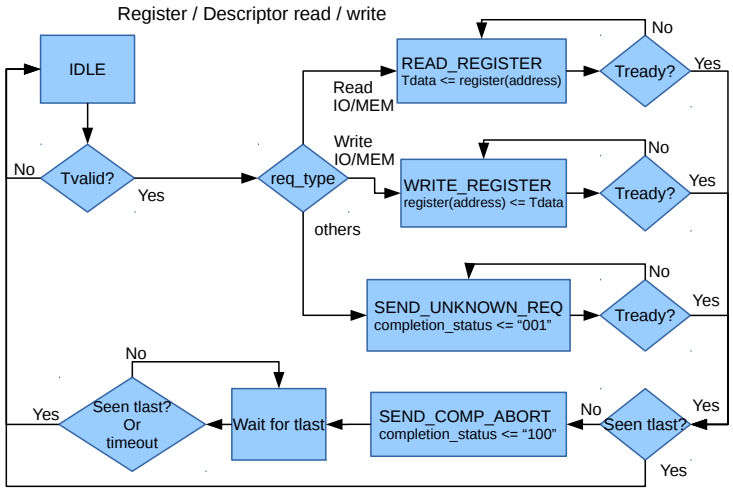

Figure 2: Flow of a Register / Descriptor Read or Write process

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 send a "completion abort" reply and go back to idle state. The maximum register size has been set to 128 bits because this is a convenient maximum register size; it is also the maximum payload that fits in one 250 MHz clock cycle of the AXI4-Stream interface.

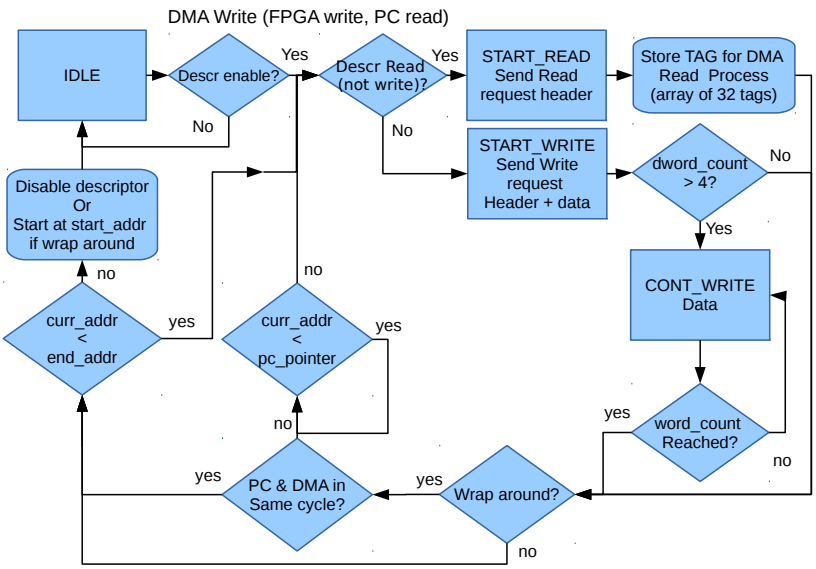

Figure 3: Flow of the DMA Write, or Read request process

The DMA write 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.

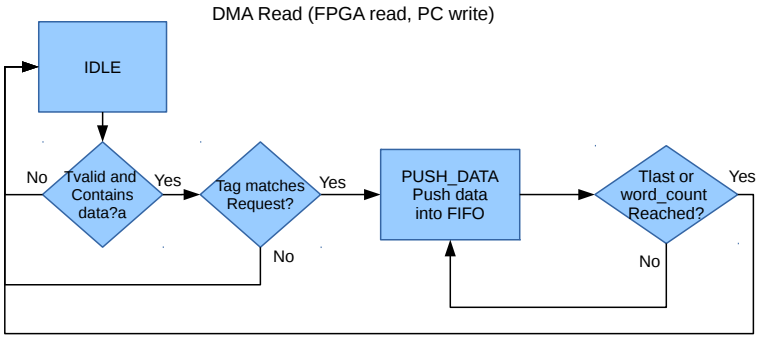

Figure 4: Flow of the DMA Read process

The DMA read 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.

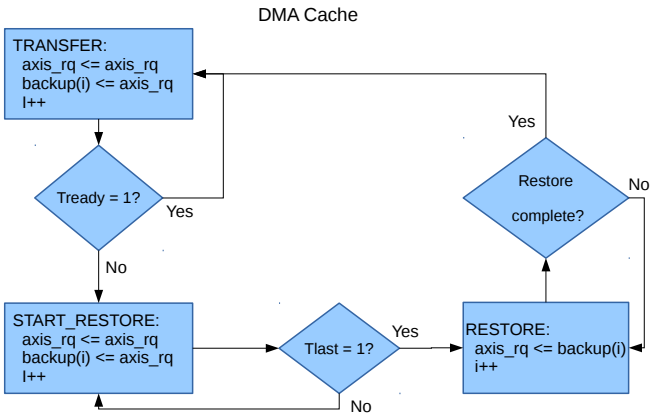

Figure 5: Flow of the Backup cache process

In case the PC (or PCIe master) sends back-pressure by deasserting the AXI4 $tready$ signal in the $axis\_rq$ bus during a Transaction Layer Packet (TLP) transfer, it has been observed that the content of the TLP may get lost, even though the TLP will apparently complete successfully. Therefore the entity DMA_Cache has been introduced, it will record all TLPs and play it back whenever back pressure occurs.

3 Xilinx PCIe Core

The PCIe Engine is based on the interface of the Virtex-7 FPGA Gen3 Integrated Block for PCI Express v3.0 [5]. This core is using a PCIe hard block in Virtex-7 FPGAs. The hard block is equipped with an AXI4-Stream interface.

3.1 Xilinx AXI4-Stream interface

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

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

Table 2: AXI4-Stream streams

3.2 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 enabled. All other options are set to default, the reference clock frequency is 100MHz and the only option for the AXI4-Stream interface is 256 bit at 250MHz, see Figure 6 to 16.

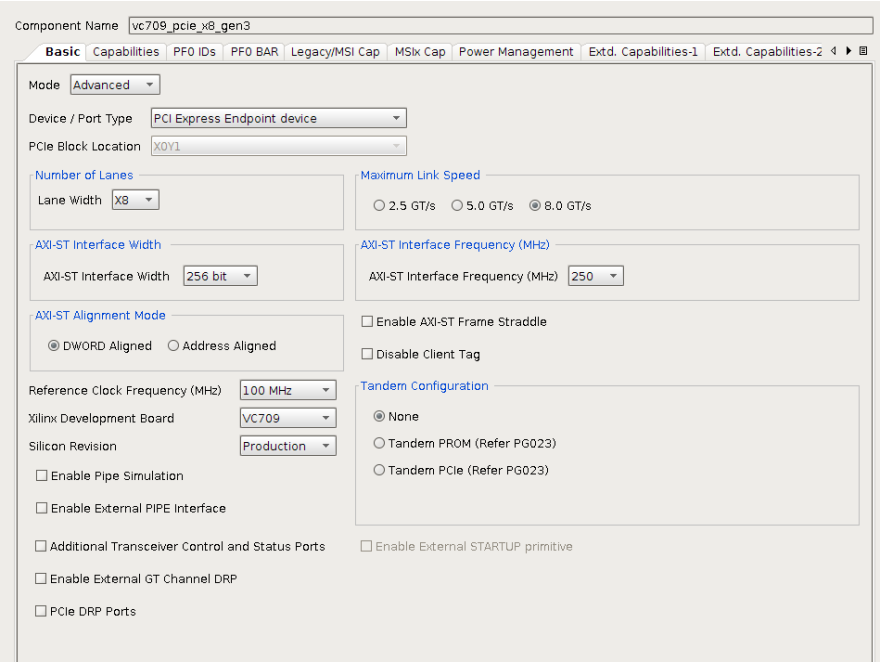

Figure 6: PCIe core configuration in Vivado [Basic]

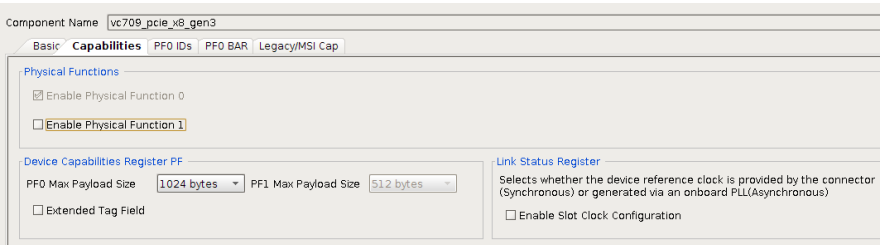

Figure 7: PCIe core configuration in Vivado [Capabilities]

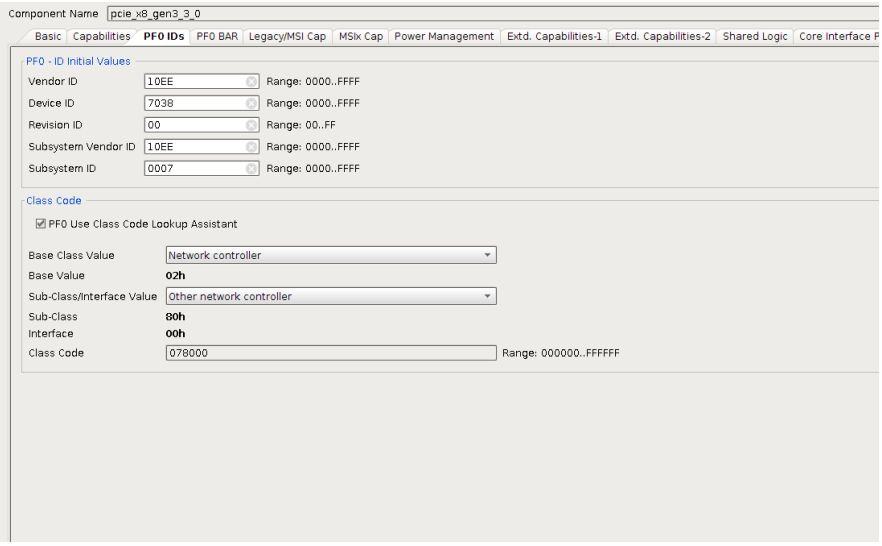

Figure 8: PCIe core configuration in Vivado [PF0 IDs]

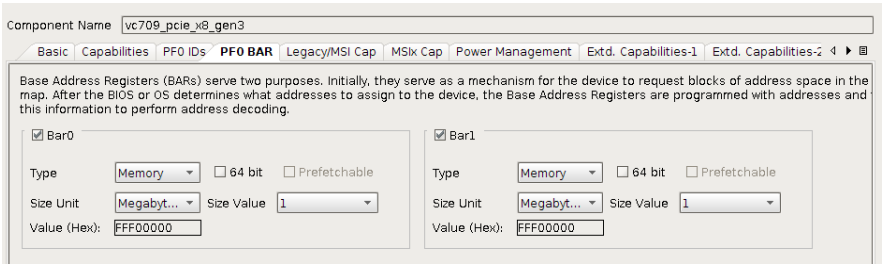

Figure 9: PCIe core configuration in Vivado [PF0 BAR]

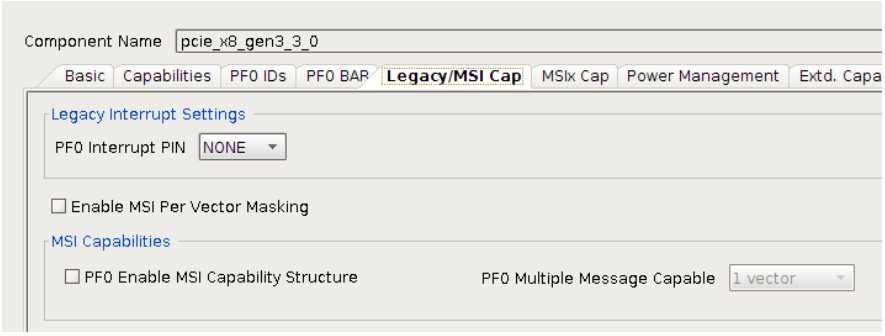

Figure 10: PCIe core configuration in Vivado [Legacy/MSI Cap]

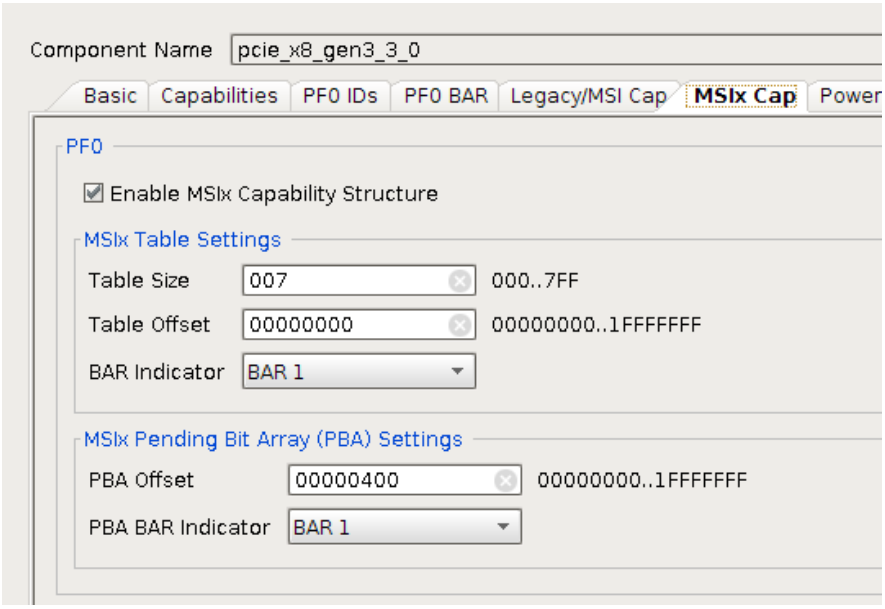

Figure 11: PCIe core configuration in Vivado [MSIx]

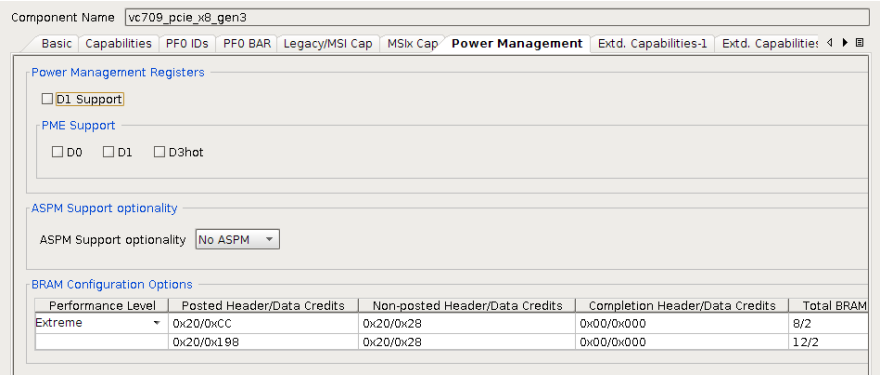

Figure 12: PCIe core configuration in Vivado [Power Management]

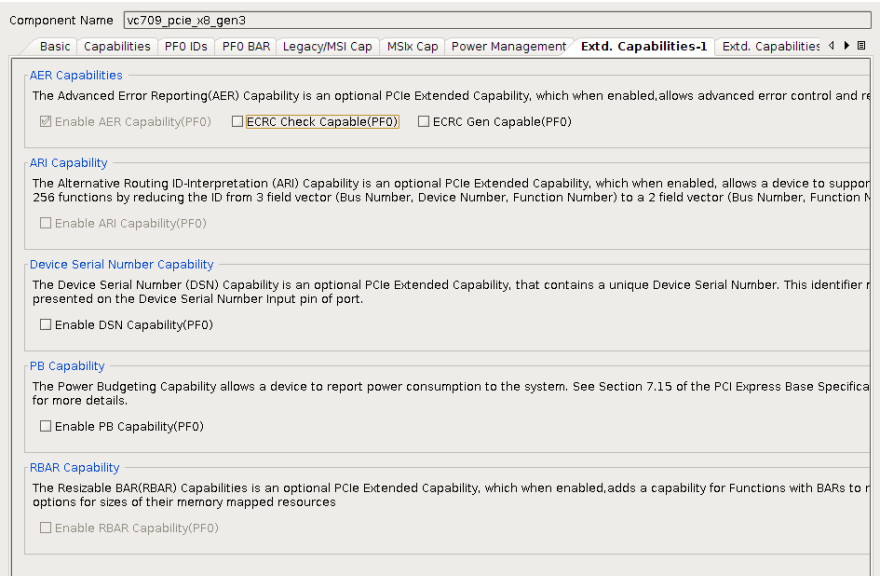

Figure 13: PCIe core configuration in Vivado [Extd. Capabilities 1]

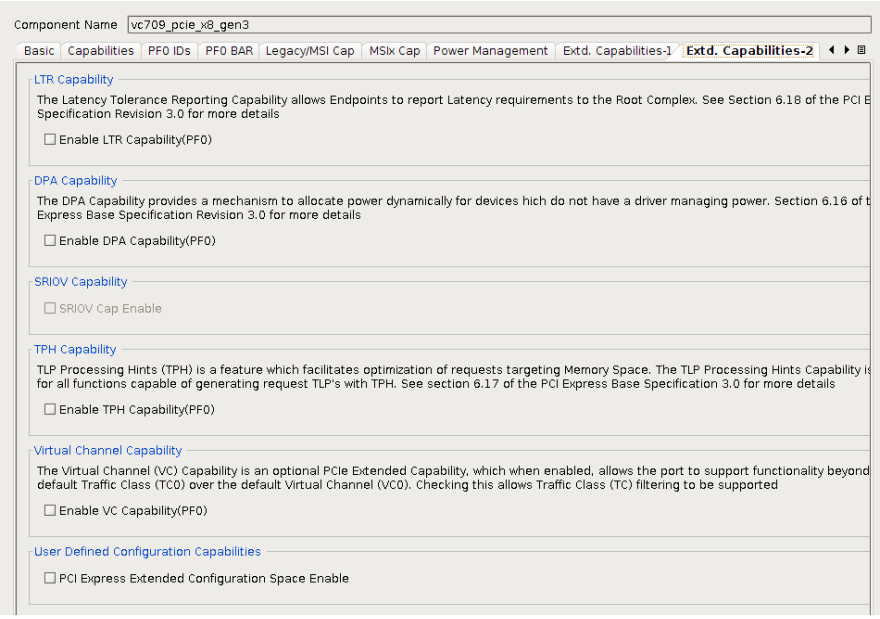

Figure 14: PCIe core configuration in Vivado [Extd. Capabilities 2]

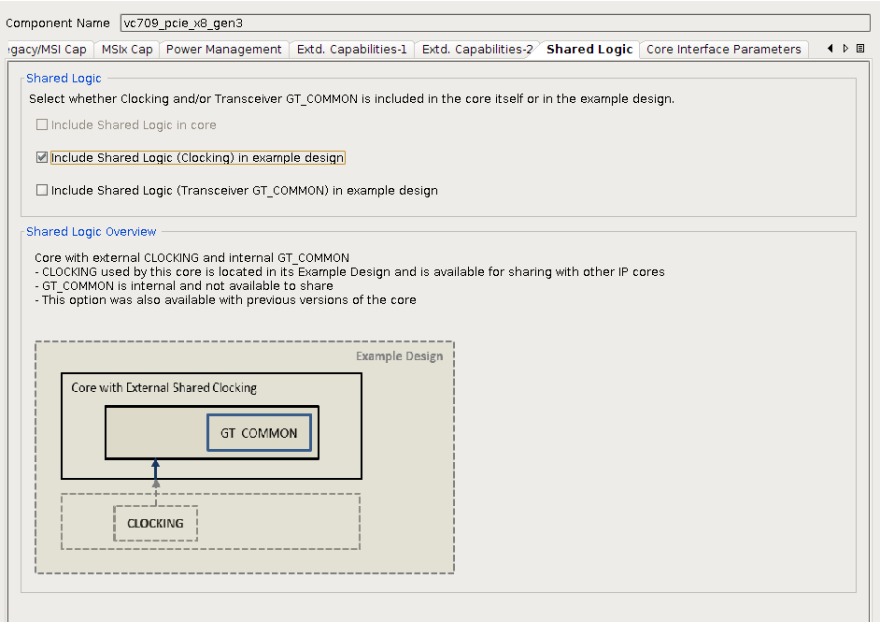

Figure 15: PCIe core configuration in Vivado [Shared LogicMSIx]

Figure 16: PCIe core configuration in Vivado [Core Interface Parameters]

4 Obtaining and building the PCIe Engine

The repository is divided in several directories:

directory	**contents**
firmware/constraints	Contains an XDC file with Vivado constraints including Chipscope ILA definitions, may differ over different commits
firmware/output	Empty placeholder where bit files will be generated
firmware/Projects	Empty placeholder where the Vivado projects will be generated
firmware/scripts/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 3: 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*.

4.1 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
  Should contain this document as well as a doxygen script to document the firmware structure.

- driver
  Still empty, we would like to ask the community to help us writing an open source driver.

4.2 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/pcie_dma_top/
source ./vivado_import.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".

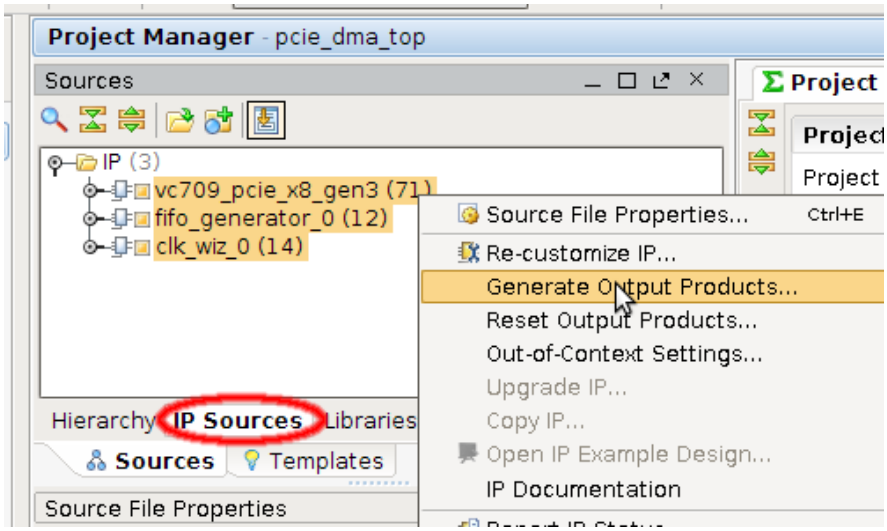

Figure 17: Generate IP Cores output products

4.3 Running synthesis and implementation

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

Listing 3: start synthesis / implementation

```
cd /path/to/svn/checkout/firmware/scripts/pcie_dma_top/
source ./do_synt.tcl
```

5 Simulation

The directory *firmware/simulation/pcie_dma_top* contains all necessary files to run the simulation in Mentor Graphics Modelsim or Questasim [7].

5.1 Prerequisites

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

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

5.2 Creating the project and running the simulation.

Like the Vivado project, also the Questasim project is generated and operated using .tcl scripts. To create and run the project execute the following commands from the Questasim console:

Listing 4: Run the simulation

```
cd firmware/simulation/pcie_dma_top/
#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 also 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.

6 Operating the PCIe Engine

No driver nor software, which can be released under an open source license, has been yet developed. For that reason we kindly ask people in the community to develop a driver which can handle the PCIe Engine according to the specifications in this chapter.

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. [8]

6.1 Loading the driver

TODO: describe the driver stuff

The driver can be loaded as follows:

Listing 5: Loading the driver

```
# load the driver
sudo insmod ../path_to_driver/driver_name.ko
# unload the driver
sudo rmmod driver_name
```

6.2 Setting the DMA descriptors

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.

6.2.1 The register map

Starting from the offset address of BAR0, BAR1 and BAR2, the register map for BAR0 expands from 0x0000 to 0x0420 for the PCIe control registers. BAR0 only contains registers associated with DMA. The offset for BAR0 is usually 0xFBB00000.

Offset	Description	Bits	Direction	Fields
0x0000	DMA Descriptor 0	[127:0] [63:0]	R/W R/W	End Address Start address
0x0010	DMA Descriptor 0a	[11] [10:0]	R/W R/W	$Read/\overline{Write}$ Number of 32 bit words
0x0020	DMA Descriptor 1	[127:0] [63:0]	R/W R/W	End Address Start address
0x0030	DMA Descriptor 1a	[11] [10:0]	R/W R/W	$Read/\overline{Write}$ Number of 32 bit words
...				
0x01e0	DMA Descriptor 15	[127:0] [63:0]	R/W R/W	End Address Start address
0x01f0	DMA Descriptor 15a	[11] [10:0]	R/W R/W	$Read/\overline{Write}$ Number of 32 bit words
0x0200	Status of descriptor 0	[64] [63:0]	R R	Descriptor done Current Address
0x0210	Status of descriptor 1	[64] [63:0]	R R	Descriptor done Current Address
...				
0x02F0	Status of descriptor 15	[64] [63:0]	R R	Descriptor done Current Address
0x0300	BAR0 value	[31:0]	R	Copy of BAR0 offset register
0x0310	BAR1 value	[31:0]	R	Copy of BAR1 offset register
0x0320	BAR2 value	[31:0]	R	Copy of BAR2 offset register
0x0400	Descriptor enable register Will be cleared when Descriptor is handled	[0] [31:1]	R/W R/W	Enable descriptor 0 Enable descriptor 31:1
0x0410	FIFO Flush, any write to this register will flush (reset) the DMA Fifo	any	W	Flush
0x0420	DMA Reset, any write to this register will reset the PCIe Engine, the backup cache and the state machines.	any	W	Reset
0x0430	Soft Reset, any write to this register issue a soft reset for the application.	any	W	Reset

Table 4: PCIe Engine register map BAR0

BAR1 holds registers associated with the Interrupt vector, the offset for BAR1 is usually 0xFBA00000

Offset	Description	Bits	Direction	Fields
0x000	Interrupt vector 0	[63:0]	R/W	Interrupt Address
		[95:64]	R/W	Interrupt Data
		[127:96]	R/W	Interrupt Control
0x010		[63:0]	R/W	Interrupt Address
..	...	[95:64]	R/W	Interrupt Data
0x060		[127:96]	R/W	Interrupt Control
0x070	Interrupt vector 7	[63:0]	R/W	Interrupt Address
		[95:64]	R/W	Interrupt Data
		[127:96]	R/W	Interrupt Control
0x100	Interrupt Table enable	[0]	R/W	Enable interrupts

Table 5: PCIe Engine register map BAR1

BAR2 currently holds some registers for testing and debugging the Engine, it will eventually hold application specific registers. the offset for BAR2 is usually 0xFB900000

Offset	Description	Bits	Direction	Fields
0x0000	REG_BOARD_ID	[63:0]	R/W	Board ID Value
0x0010	REG_STATUS_LEDS	[7:0]	R/W	Board GPIO Leds
0x0020	GENERIC_CONSTANTS	[7:0]	R	NUMBER_OF_DESCRIPTORS
		[15:8]	R	NUMBER_OF_INTERRUPTS
0x0300	REG_PLL_LOCK	[0]	R	PLL Locked status
0x1060	REG_INT_TEST_2	any	W	Fire a test MSIx interrupt #2
0x1070	REG_INT_TEST_3	any	W	Fire a test MSIx interrupt #3

Table 6: PCIe Engine register map BAR2

6.2.2 Driver functionality

Before any DMA actions can be performed, one or more memory buffers have to be allocated.

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
```

7 Customizing the application

7.1 connection of the DMA FIFOs

The Virtex-7 PCIe Engine comes with a very simple user application, described in application.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.

In this example the FIFO that holds data from the PC direction PCIe, has been omitted and the port is left unconnected.

The FIFO that holds data which will be sent towards the PC, is connected to a simple process, this process has a 32 bit counter and a constant value, the 256 bit data which is sent into the PC memory has the following format:

Listing 8: Data format of example application

```
s_fifo_din <= x"DEADBEEF"&cnt&x"00001111"&cnt&x"22223333"&cnt&x"44445555"
    &cnt;
```

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 9: custom register types

```
type register_map_control_type is record
  BOARD_ID          : std_logic_vector(63 downto 0);
  STATUS_LEDS       : std_logic_vector(7 downto 0);
end record;

type register_map_monitor_type is record
  READ_ONLY              : std_logic_vector(0 downto 0);
  PLL_LOCK               : std_logic_vector(0 downto 0);
end record;
```

When adding registers to this map, there are two files (besides the application implementation) that need to be modified:

- pcie_package.vhd containing the address, the default value and the record type specification
- dma_control.vhd containing the constant initialization, the read state machine and the write state machine

sections that should be modified can be recognized by searching for comments like this:

Listing 10: customizable sections

```
-------------------------------------------------
---- Application specific registers BEGIN   ----
-------------------------------------------------
...
-------------------------------------------------
---- Application specific registers END     ----
-------------------------------------------------
```

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 PCI Express core
http://www.xilinx.com/support/documentation/ip_documentation/pcie3_7x/v3_0/pg023_v7_pcie_gen3.pdf

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

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

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

List of Figures

List of Tables