



CERN Summer Student Report

Digital Design of Tracking Processor Unit

Wendo Beuker

Supervised by Riccardo Fantechi and Niko Neufeld

UNIVERSITY OF TWENTE.

Contents

1	Preamble	3
I	CERN Summer Student project	4
2	Introduction to CERN, LHCb and the High Luminosity upgrade	5
3	LHCb Data Acquisition for HL-LHC	6
4	Co-processor for EB nodes and project goal	8
4.1	Co-processor for EB nodes	8
4.2	Project goal	9
5	Porting current design to Arria V GZ	10
5.1	Porting to different device in QSys	10
5.2	PCIe/DMA interface	10
5.3	ROMs and RAMs	10
5.4	PLL	10
5.5	Top level entity and other files	11
5.6	Compilation scripts	11
5.7	Git commits related to this chapter	11
6	Simulations	12
6.1	Root port	12
6.2	/scripts/mentor/common.tcl	12
6.2.1	Qsys generate from tcl	12
6.2.2	Library order	12
6.3	General remarks	13
6.4	Results	13
6.5	Git commits related to this chapter	13
II	Future work	14
7	DMA controller	15
7.1	General overview	15

7.2	Stream	15
7.3	Scheduler	16
7.4	Hard IP for PCIe	17
7.5	Base Address Registers	17
8	Drivers and PCIe40 Emulator	18
8.1	Introduction into the drivers, general concepts and (shared) structs	18
8.2	Start-up	19
8.3	Consuming data	19
8.4	PCIe40 Emulator	20
9	Continuation of this work	21
9.1	(Possible) Demo setup	21
9.2	Expected changes	21
9.2.1	Drivers and Emulator changes	21
9.2.2	Firmware changes	22
9.2.3	Validation of Emulator behaviour	22
III	Conclusions	23
10	Reflection, conclusions and recommendations	24
	Bibliography	25
	Appendix A Firmware git folder structure and purpose	26
A.1	Naming conventions	26
A.2	Git content	26
	Appendix B Content and changes of firmware commit c165fe1	28
	Appendix C Content and changes of simulation commit fa8238f	29
	Appendix D Generating project and simulation environment	30
D.1	Before you start	30
D.2	Setup Quartus project	30
D.3	Run simulation in QuestaSim	30

Chapter 1: Preamble

In front of you, you find the report of the Cern Summer Student project performed by Wendo Beuker in the Experimental Physics - LHCb Computing Group at CERN. This report is divided in three parts:

- The first part of this research will give an introduction into CERN, LHCb and the project. It will provide information about the actual assignment as well as what has been done to try to reach this assignment.
- Because the goal of the project was not reached, the second part of this report focuses on continuation of this project. It will provide more in-depth knowledge of the existing hardware and software, how these work and operate, and what should be altered and added to eventually reach the goal of the project.
- The third part will consist out of a general conclusion and reflection on the project and my time at CERN.

In this report jargon of the Experimental Physics - LHCb Computing Group is used. The phrase "firmware" refers to the digital hardware design, designed for an FPGA. With the phrase "software", mainly the Linux drivers to communicate with the digital hardware loaded on the FPGA (firmware) are addressed.

For this project two git repositories are available at the CERN Gitlab server. One for the firmware (<https://gitlab.cern.ch/lhcb-daq40/lhcb-daq40-firmware.git>) and one for the software (<https://gitlab.cern.ch/lhcb-daq40/lhcb-daq40-software.git>). When referring to files in the firmware repository, the file location will start with `firmware-repo`, while when referring to files in the software repository, the file location will start with `software-repo`. During this project a couple of commits have been pushed to the firmware repository into the *devel-av* branch. When applicable commit hashes will be stated in text such that the changes regarding a certain topic can be easily found. Note that Appendix A elaborates on the folder structure and important files of the firmware-repo.

Last but not least, special thanks would be rewarded Niko Neufeld and Riccardo Fantechi for providing this opportunity and to Paolo Durante for all his help during this CERN Summer Student Project.

Part I

CERN Summer Student project

Chapter 2: Introduction to CERN, LHCb and the High Luminosity upgrade

CERN, the European Organisation for Nuclear Research, is a scientific institute at the border of Switzerland and France. It houses the largest particle accelerator complex in the world. Via different stages of acceleration, protons are accelerated up to 6.5 TeV in the Large Hadron Collider (LHC). The LHC is basically a very large microscope to probe physics at the level of elementary particles. This is done by colliding two beams of protons rotating in opposite direction which results in interaction between the (partons of the) protons at up to 13 TeV.

Along the ring of the LHC, 4 large and a couple of smaller experiments can be found. The four large experiments are: ATLAS, CMS, LHCb and ALICE. ATLAS and CMS are general purpose detectors. LHCb is specially set-up to perform measurements on B-quarks to look for CP-violation and ALICE for quark-gluon-physics with heavy ion collisions. Where ALICE, CMS and ATLAS are symmetrical detectors along the beam pipe, LHCb is asymmetric and can only detect particles leaving within a certain angle (however with a very high precision). See Figure 2.1 for an overview image of the (asymmetric) LHCb experiment.

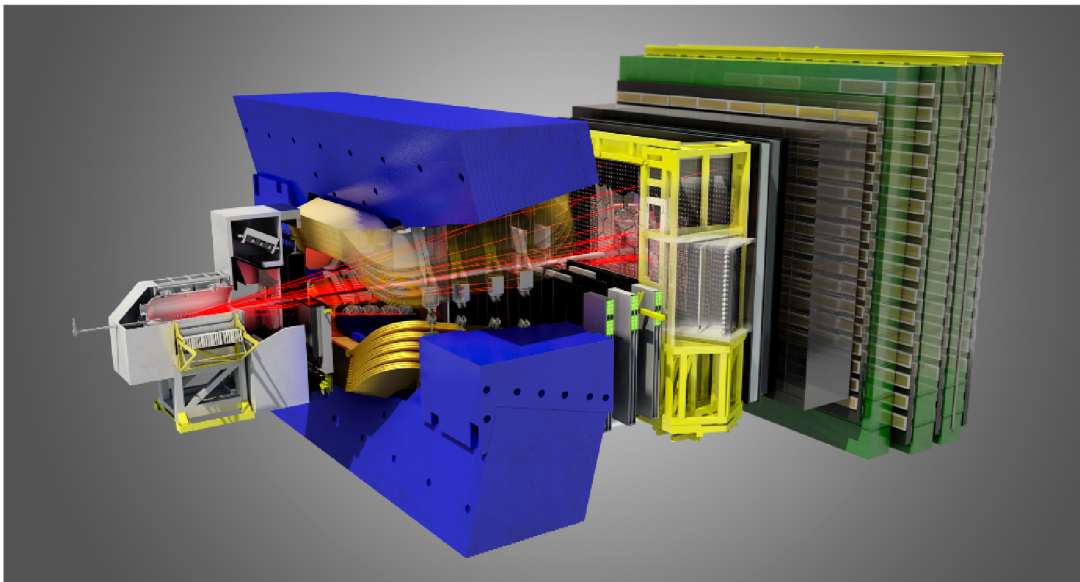


Figure 2.1: Overview of the asymmetric LHCb detector.

At the end of 2018, the LHC will be stopped for the next upgrade: High Luminosity LHC (HL-LHC). During this stop of about two years, the LHC will be upgraded to increase its luminosity¹. During this stop also the experiments have the possibility to upgrade their detectors and systems. For the LHCb this means that both the detector as the Data Acquisition system will be upgraded.

¹The luminosity is the amount of particles that will pass a certain surface per second. A high luminosity does result in a higher probability of interaction. See also <https://home.cern/cern-people/opinion/2011/03/luminosity-why-dont-we-just-say-collision-rate> for more information on this topic.

Chapter 3: LHCb Data Acquisition for HL-LHC

This section will give an introduction into the proposed Data Acquisition (DAQ) topology for the LHCb upgrade. For a schematic overview of the DAQ system of LHCb, see Figure 3.1.

In the LHC, proton-proton interaction/collision (these interactions are called "events") happen at a rate of 40 MHz. The data of the events is sent to a cluster of 500 computers; the Event Builders (EB). Every EB receives a data-stream from a different detector. The task of the EB is to package all data of a set of event into one data package, such that all further calculations (for example track reconstruction in the Event Filter Farm) can be performed on per event data instead of per detector data.

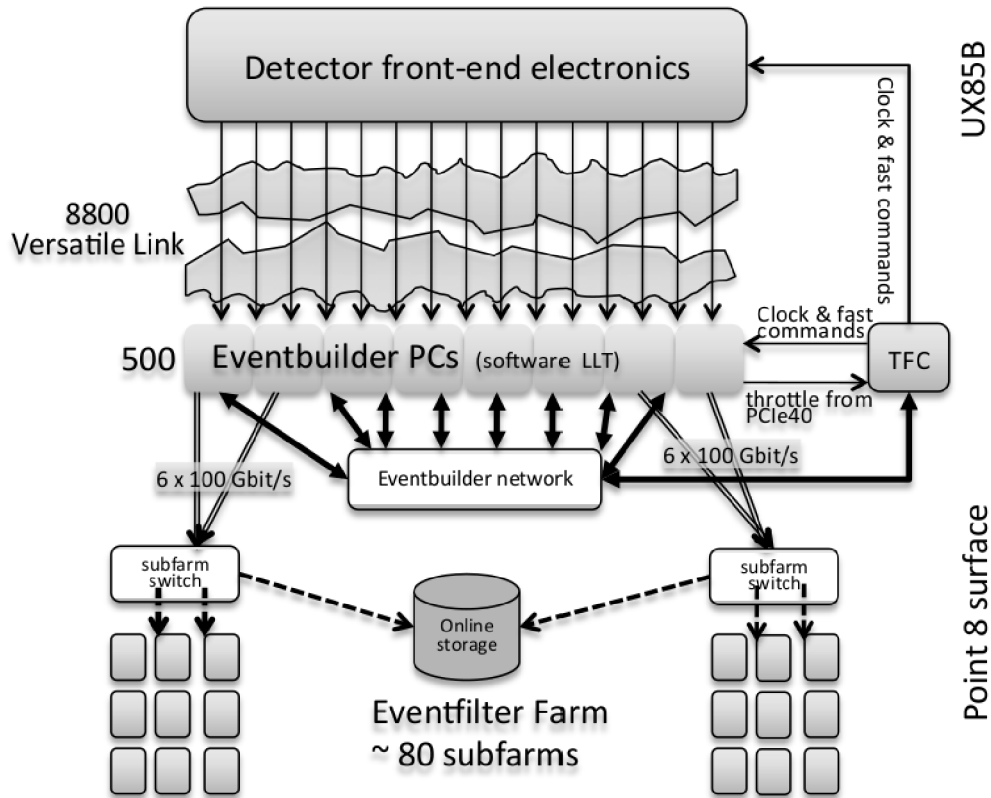


Figure 3.1: Overview of the LHCb DAQ system.

An EB consist out of a two processor system with separate memories. Figure 3.2 shows an overview of an EB as well as the data-streams within an EB. The data of the different detectors enters an EB via optical interfaces at the PCIe40 card. This PCI-express expansion card will dump the data at the optical interfaces directly into Memory 1 via a DMA transaction (bold blue arrow). In the meanwhile the EBs elect one of the 500 EBs which will be responsible for building a certain set of events. All EBs will send the data they have related to this set to the elected EB via a LAN network (bold green arrow). The elected EB will receive this data via its EB Network Interface and store this data in Memory 0 (thin orange arrow) and moves the data it already knew about this set even from Memory 1 to Memory 0 (thin green arrow). If all data is available, the data is combined (the event is built) and the data package is forwarded to the Event Filter Farm via a LAN (thin black arrows).

The main challenge of this process are the large amounts of data transferred. For example the incoming detector data (bold blue arrow) can be up to 100 Gb/s such that the memory load in CPU1 is quite heavy while the computational workload of both CPUs is moderate.

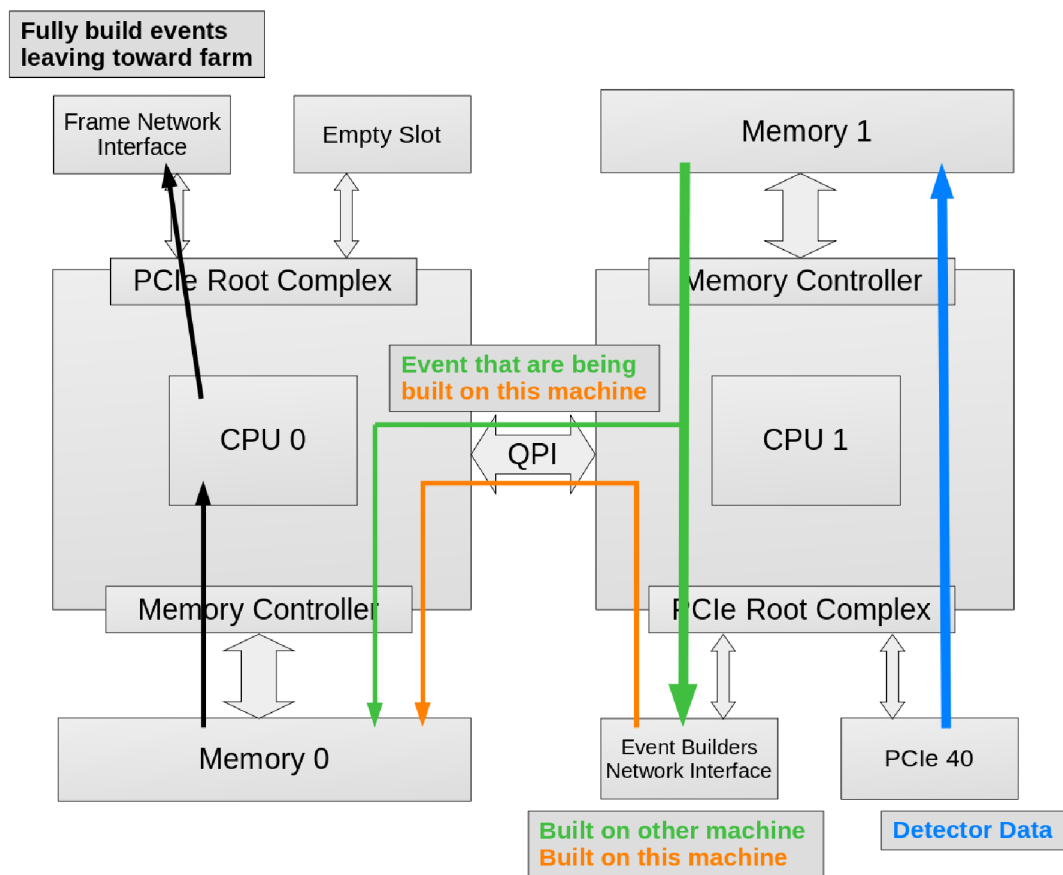


Figure 3.2: Overview of one Event Builder and the data streams within the system.

Chapter 4: Co-processor for EB nodes and project goal

The LHCb collaboration is investigating the possibility to perform partial data processing in the EB nodes. This section will elaborate on this idea and explain how it is related to the CERN Summer Student project.

4.1 Co-processor for EB nodes

One option currently under investigation would be to add a Tracking Processor Unit (TPU) to the empty PCIe slot of CPU0¹. This TPU could be implemented by an FPGA running the Artificial Retina algorithm², but also other kind of accelerators can be used. This accelerator should get the detector data out of Memory 1 and performs its calculations on it. Because every TPU only constructs a subset of tracks for each event, the tracking data is not complete and should be added to the detector data to be distributed to the elected EB node³. The results are therefore injected back into the outgoing data-streams (probably by injecting the results into Memory 1). For performance reasons both data transfers from and to memory should be implemented by DMA transactions. A schematic overview of the new layout of the event builders is shown in Figure 4.1. In this figure, the TPU is shown in pink, and the related data-streams (due to TPU's DMA transactions) are also shown in pink.

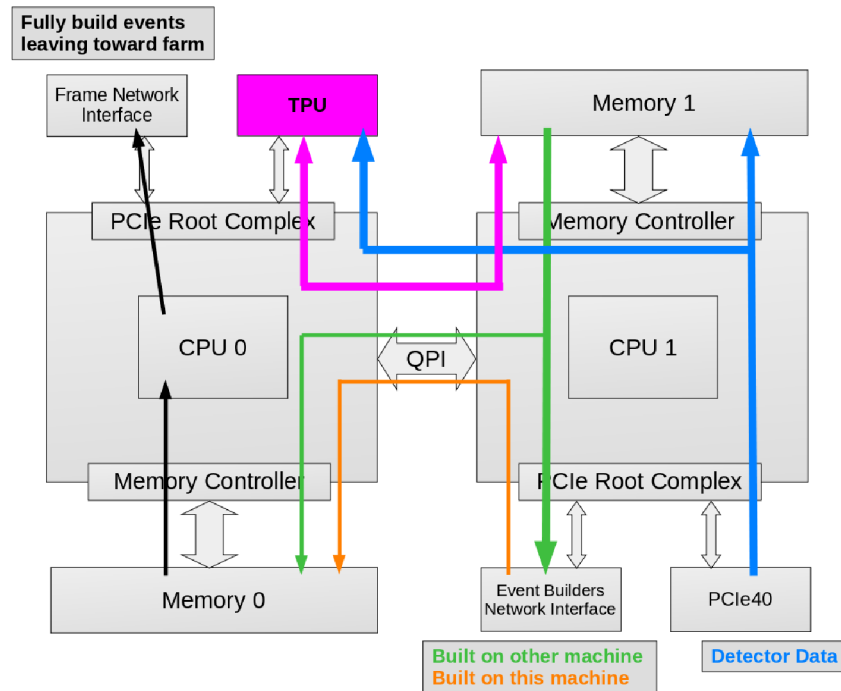


Figure 4.1: An overview of an upgraded Event Builder (with the TPU). Data streams are visualized by arrows.

¹The main reason to add this TPU to the EB nodes is that, for some detectors, partial track reconstruction is a local problem. Tracks who are close in parameter space, are also close in detector space (thus each TPU will only construct a subset of tracks for each event). So only local data is necessary to solve this problem, which is available at the EB nodes. This makes it a logical choice to add the TPUs to the EB nodes.

²The Artificial Retina algorithm is an algorithm that can perform partial track reconstruction on the raw data of the detector. For more information see for example [2] or [3].

³The EB node that will build the set of events, will thus also receive all partial track reconstructions and add it to the data set.

4.2 Project goal

The goal of this internship is to build a demo of a two-way DMA transfer system on an FPGA mounted on a Bittware A5PL⁴ development board, taking into account that it should be based on the existing firmware of the PCIe40 cards and it should use the same drivers. This implies that

- the current firmware (developed for the Intel Arria 10 FPGA) should be ported to the Intel Arria V FPGA
- a simulation environment should be setup for the Intel Arria V GZ device.
- the drivers and firmware should be altered such that they support two-way DMA transfers instead of one-way.

This all should be performed while keeping in mind the available resources of both the TPU/FPGA and the EB node. Finding out the resources needed to accommodate for two-way DMA transfers compared to one-way DMA transfers can help in deciding what model of FPGA is suitable for the implementation of the TPU. Besides the resources in the FPGA, the operations of the TPU should not affect/interfere the main operations of the EB nodes.

⁴The product page of the Bittware A5PL development page is <https://www.bittware.com/fpga/intel/boards/a5pl/>.

Chapter 5: Porting current design to Arria V GZ

The first objective was to port the existing PCIe40 (Arria 10 based) design to the Bittware A5PL (Arria V GZ ME7H2F35C3 based). This chapter will provide some details what has been done, which problems were encountered and how these problems have been tackled. Note that Appendix D provides some info on how to setup and compile the Quartus project.

5.1 Porting to different device in QSys

Porting IPs from one device to another is easily done in QSys. Make sure that the "Device Family"-panel is shown (select it under the drop down menu "View"). Select an IP in the "System Contents" view and in the "Device Family"-panel, just select the FPGA you would like to port the IP to, as shown in Figure 5.1. Not all IPs are available for all devices. In that case QSys shows an error in the "Messages"-panel. If this is the case, you should come up with another solution to port the IP or use a similar IP which is available for the target device family.

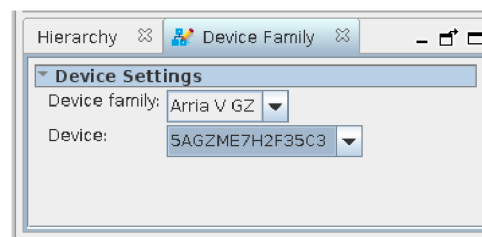


Figure 5.1: The "Device Family"-panel to port IPs from one device to another.

5.2 PCIe/DMA interface

The PCIe40 design for the Arria 10 cannot be ported to the Arria V GZ easily because the Arria 10 and Arria V GZ PCIe IP differ. Luckily there was also an existing design based on the Stratix V. The Arria V GZ and Stratix V PCIe IP are similar and therefore this design has been used as basis for the Bittware firmware.

5.3 ROMs and RAMs

The PCIe40 design contains some ROMs and RAMs for keeping track of which build version is at which board and chip (called Chip-ID, Board-ID and Build-ID). It turned out that the used ROMs and RAMs are not anymore available in the QSys IP Library (Quartus 16.1). However the MegaWizard does provide in these IP blocks. Therefore custom IPs have been added to the QSys IP Library, using the ROMs and RAMs created by the MegaWizard as basis. These custom IPs are basically just wrappers to make the IPs created by the MegaWizard available in QSys.

5.4 PLL

The IOPLL of the Arria 10 is not available for the Arria V GZ device family. Therefore the Intel PLL IP has been utilised, using similar settings as the original IOPLL.

5.5 Top level entity and other files

There have been no other significant changes to QSys systems or VHDL files (i.e. top level entity, PCIe wrapper). Also the DMA controller is not changed.

5.6 Compilation scripts

The compilation script for the firmware is not changed significantly (only minor changes related to file references and possibly library references). One important note is that one has to be sure that the right device is used in the assignment scripts. For some reason the `firmware-repo/assignments/bittware_a5pl.tcl` assignment script set the device to "5AGZME1H2F35C3" instead of "5AGZME7H2F35C3". Although the "5AGZME1H2F35C3" is different from what is selected in the QSys files/systems, the system compiles without any problems, errors or critical warnings (according to Paolo Durante, QSys just overrides its default to "5AGZME1H2F35C3" without any warning). Since programming the FPGA with the programming files for the wrong device is not possible, it took significant time to figure out what was the problem in this case¹.

5.7 Git commits related to this chapter

This short section will state which git commits are related to the work discussed in this chapter.

- **c165fe155149e54513a88050b38cab6c2638b794**: Initial commit, adds all folders structures needed to port hardware (components) to Arria V GZ and compilation scripts. See Appendix B for the content and changes made in this commit.
- **58b2f9cb1577f2840a8ce902411ae5b446d47c27**: Fixes bad reference to `top.sdc` in `firmware-repo/configs/other/a5pl/pcie/dma_ecs/quartus.tcl`. Furthermore it includes some code style changes.
- **5a2632c1f936349c25eced7991ddf179b08a3128**: Adds the right DEVICE to the `firmware-repo/assignments/bittware_a5pl.tcl` assignment file. This is the commit related to the problem described in section 5.6.
- **4dc421071ea0b31b9dec3cafd5e6cfae8f773622**: Adds some clock constraining for the Arria V GZ. Such that the firmware compiles without timing violations and the firmware actually worked on the Bittware A5PL cards.

¹A problem occurred when it was tried to program the FPGA via the Bittware configuration tool. Although the tool did not signal a warning or error, the FPGA seemed not to be programmed as well. It took more than a week to figure out that this was caused by the problem described in the text. It turned out that the Quartus Programmer does signal a warning, but it was not used because it experienced some other problems related to the jtagd process.

Chapter 6: Simulations

For validation purposes, it has been tried to setup a simulation environment based on the already existing one. This section will elaborate on that. Note that Appendix D provides some info on how to run the simulations within the QuestaSim environment.

6.1 Root port

The root complex/port is so to say the PCIe controller on a CPU. To validate the PCIe transaction of the FPGA in simulation, a virtual root port is needed. Altera offers a virtual root complex ip, which is called "tbed". However that is about it. There is no documentation and you cannot find anything about it on the World Wide Web. One would expect that a behavioural model of a root complex is independent of the target device. However this is not the case. Different target devices use different flavours of the tbed IP. It seems to have something to do with certain control signals which are only used for simulation purposes. However the details are not clear due to the lack of documentation of the tbed IP. It is therefore not easily possible to find for which devices such a virtual root complex is available and for which it isn't. Based on the already existing simulation environments there is a specific root complex available for the Arria 10 device family. The Stratix V device family seems to use a generic root port. There seems to be no specific root complex available for the Arria V GZ device family. Because the Qsys IP used for the Stratix V PCIe interface and the Arria V GZ PCIe interface are similar, it was tried to use the same tbed implementation as for the Arria V GZ as for the Stratix V device. This seems to be working; the systems compiles and the simulation runs.

6.2 /scripts/mentor/common.tcl

In this common TCL script, some important aspects have been changed. This subsection will elaborate on that.

6.2.1 Qsys generate from tcl

The process *qsys.generate()* experiences some problems. When the process sends a command to the Linux terminal via *exec*, an error is returned. Searching the World Wide Web for this error does not provide any useful results. However if the exact same command is run in the Linux or QuestaSim terminal by hand, it executes without any problems. Therefore the total command is printed to the QuestaSim terminal, such that it can be copy pasted to a working terminal. Note that this operation should be performed twice: One time for the generation of the PLL Qsys system (`firmware-repo/modules/pcie/emc_pcie/arriav/pll_pll_av.qsys`) and one for the generation of the PCIe interface (`firmware-repo/modules/pcie/emc_pcie/arriav/qsys_pcie_dma_ecs_x8.qsys`).

6.2.2 Library order

The process *do.elab()* signaled some errors because the libraries are loaded in the wrong order¹. Therefore the order of loading libraries has been changed such that they are loaded in the right order. Note that also the *work* library is added to this loading process. It was not yet part of the loaded libraries, but the simulations for the Arria V GZ device crashes when it is not included.

¹See <https://www.altera.com/support/support-resources/knowledge-base/solutions/rd01212014.488.html> for the description and solution of the problem.

6.3 General remarks

A lot of different pieces of code has been altered or tailored to get the simulation starting and running. These minor changes will not be discussed.

6.4 Results

The results at the moment regarding the simulation is that the simulation loads and starts running. Unfortunately there seems to be something wrong during the setup of the PCIe connection. In this case it means that the device does gets into the LTSSM "POLLING.CONFIG" state, but it seems not to be able to setup a stable connection. It is unknown why this is the case although it seems likely that the tbed virtual root port IP is part of the problem. Since there is no documentation which makes debugging difficult, it has been chosen (in consultation with Paolo Durante) to not investigate this problem for now.

6.5 Git commits related to this chapter

- **fa8238f7be5bd05c0eb0116fce83255aede1f50e**: All important changes for getting the simulations up and running up to the point described at this point related to the simulation scripts. The important files and changes are discussed in Appendix C.
- **3791cc290e6da15bf63022cf02955fff252a450a**: It turned out that a couple of pins and wires were wrongly connected such that the simulation would not run. Also a clock signal was missing. All hardware changes done to get the simulations up to the point described in this chapter are in this commit.

Part II

Future work

Chapter 7: DMA controller

This section provides a textual overview of how the DMA controller within the FPGA works and what is expected to be altered to provide 2-way DMA. Note that some documentation of the firmware could be found at <http://lbyum.cern.ch/daq40/guide/firmware/devel/>.

7.1 General overview

For a schematic overview of the DMA controller, see Figure 7.1. The system can be divided in four different parts: The streams, scheduler, Hard IP for PCIe (abbreviated as HIP) and Host PC. The HIP is the physical PCIe interface which communicates with the Host system's root complex. The different streams generate DMA descriptors¹. These descriptors are then scheduled by the scheduler before sending to the HIP. The HIP will then perform the actual DMA transaction.

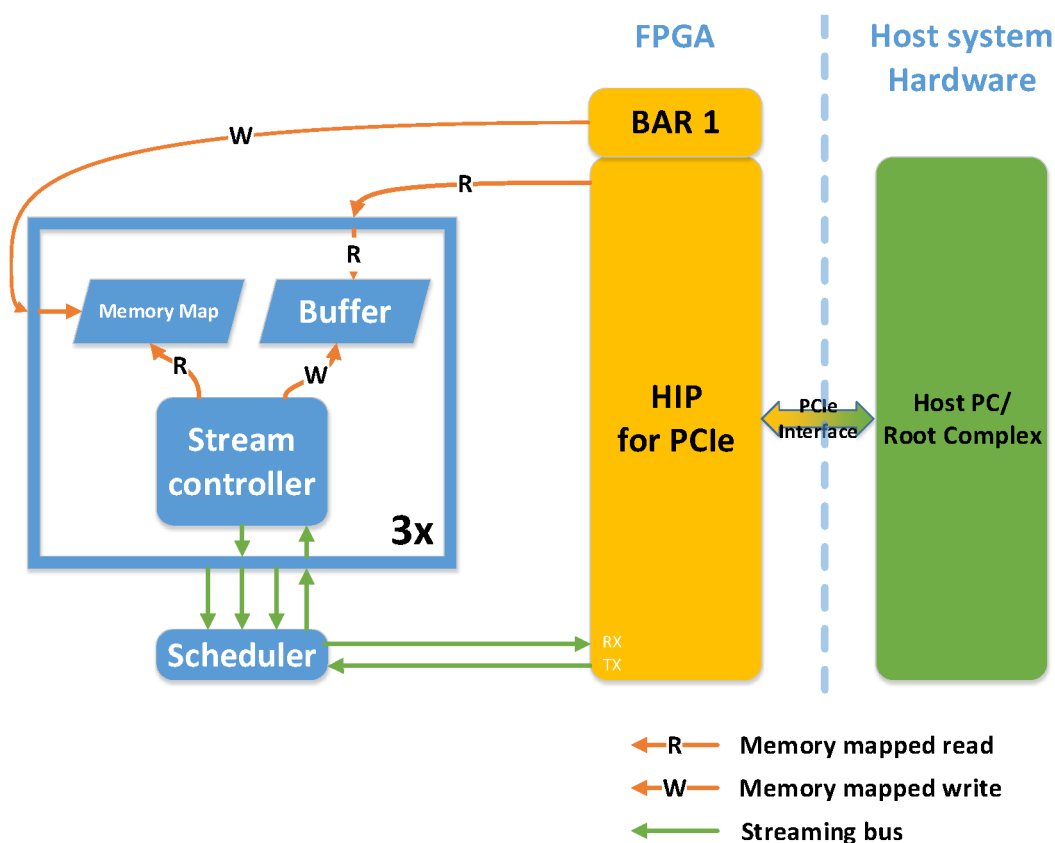


Figure 7.1: A schematic overview of the DMA controller within the FPGA.

7.2 Stream

The DMA transfers start at the streams. This is a set of digital components contains the Stream controller, a circular Buffer-memory (called Buffer) and a Memory Map-memory (called Memory Map). Let's start with the Memory Map. This is a two-port memory which is written via BAR1 via the driver and read by the Stream controller. Both ports are memory mapped interfaces.

¹A DMA descriptor contains information about the requested DMA transfer. It basically states x bytes starting at address y should be transferred to memroy location z of the Host PC.

The Buffer is a two port memory. This memory is used, as the name suggest, as a circular buffer. The Stream controller writes data that should be transferred to main memory in this buffer. Writes to this memory are performed in a memory mapped fashion. The Stream controller keeps track of a read and write pointer for the circular buffer. The write pointer indicates up to where the circular buffer is filled while the read pointer points indicated up to where the data in the circular buffer is send to the HIP/host PC ²

The Memory Map is also a two port memory which keeps track of the memory allocated by the drivers. The drivers allocate (currently) up to 4 GBs of memory, in chunks of 4 MB³, for a stream in main memory. This memory is used to by the stream controller to write data out of the Buffer info the main memory of the Host PC. In this report, this allocated memory at the Host PC is called memory mapped Buffer. The size of the memory mapped Buffer is far larger then the size of the Buffer, such that the Host PC has sufficient time to process the data of the PCIe40 card. This allocated memory might not be adjacent in main memory. Therefore the driver writes a memory map to the Memory Map via BAR1. This memory contains both the locations in physical memory as the size of all chunks allocated by the driver.

If the data is written to the Buffer, the Stream controller will generate a DMA descriptor. The structure of this descriptor is shown in Figure 7.2⁴. The information regarding the DMA descriptors is retrieved from [1] at page 4-15.

159 Immediate Write	158-154 Reserved	153-146 DMA Descriptor ID	145-128 DMA Length in dwords	127-96 Destination HI	95-64 Destination LO	63-31 Source HI	31-0 Source LO
---------------------------	---------------------	---------------------------------	------------------------------------	-----------------------------	----------------------------	--------------------	-------------------

Figure 7.2: The structure of an Altera DMA descriptor.

The first 32 bits of the descriptor are the lower 32 bits of the source address (Source LO) of the DMA transfer, where as the bit 63-32 are the high part of the source address (Source HI). The source address is thus a 64 bit address. This address refers to a location in the circular Buffer. Bits 95-64 are the lower part of the destination address (Destination LO) for the DMA transfer, whereas bit 127-96 are the high part of the destination address (Destination HI). The destination address is a 64 bit address that refers to a physical location in main memory. The next field (bits 145-128) defines the length of the data in data word. In this case a data word is 32 bits (4 bytes) of data. Maximum data size transferred in 1 transaction is thus 1 MB - 4 bytes. Bytes 146-153 define the DMA Descriptor ID. So a total of 128 descriptors can be in the system/queue at the same time. Bits 154-158 are reserved. Bit 159 is the Immediate Write bit.

Because there are multiple streams, every Stream Controller has been allocated a certain range of the DMA Descriptor IDs.

The descriptors are send to the Scheduler via a streaming interface (one per Stream Controller).

7.3 Scheduler

The Scheduler is a piece of firmware that schedules the DMA Descriptors of the Stream Controllers. Since the HIP has only one descriptor streaming interface, the Scheduler is necessary to handle the multiple scheduler streams. The scheduler has (at the moment) three input streams and one output stream. It schedules the descriptor streams based on priority; some streams have a higher priority than others and these descriptor will be send to the HIP earlier.

²Note that the Stream controller actually keeps track of two sets of read pointers and write pointers: One set is to keep track of the state of the circular Buffer and set to keep track of the state of the memory mapped Buffer. The Stream controller need to know the read pointer of the memory mapped Buffer of the drivers to keep track of which part of the memory mapped Buffer could be (re)used. The write pointer of the drivers is also of interest because the memory mapped Buffer is larger than the circular buffer. Read the next paragraph for more information.

³Data in main memory is allocated in chunks of 4 MB due to limitations of the Linux kernel. Linux cannot allocated consecutive blocks larger then 4 MB.

⁴Although shown as little-endian, the thing might actually be bigendian, see also https://www.intel.com/content/www/us/en/programmable/support/support-resources/knowledge-base/solutions/rd09182008_586.html.

When a transaction is completed successfully, the HIP will send a status descriptor via a streaming bus to the Scheduler. This is basically an acknowledgement which states that the DMA transaction with descriptor ID x was successful. Because the Scheduler does not know which stream has send this descriptor, it sends this status message back to all streams such that they can bookkeep which of their descriptors has been acknowledged and which haven't.

The structure of this status descriptor is shown in Figure 7.3. When bit 8 is asserted, the DMA transaction of the DMA descriptor was completed successfully. Bits 7-0 provide the ID of the DMA descriptor about which this status descriptor is providing information. The rest of the bits (31-9) are reserved. The info regarding the status descriptor is retrieved from [1] at page 4-15 and page 4-16.

31-9 Reserved	8 Done	7-0 Descriptor ID
------------------	-----------	-------------------------

Figure 7.3: The structure of an Altera status descriptor.

7.4 Hard IP for PCIe

The Hard IP for PCIe (or in short HIP) is the Altera IP within the FPGA which handles the PCIe transactions. For this project, only the descriptor input and output streaming bus and the data memory mapped bus are important. The descriptors, arriving at the input streaming bus, contain a memory source address. This address will be in one of the streams buffers. All buffers have a different part of the memory mapped, such that they are uniquely addressable. The HIP will read the data stated in the descriptor out of the Buffer and send it over the PCIe bus to the destination memory address.

7.5 Base Address Registers

The current configuration of the HIP provides three Base Address Registers (BAR). These register can be used to send data to the components in the FPGA as well. BAR1 is used to program Memory Map memories and to read-out some settings out of the Stream Controllers. BAR0 is for the LHCb control system to communicate with the PCIe40 cards. BAR2 is for another group of developers to access their hardware within the FPGA.

Chapter 8: Drivers and PCIe40 Emulator

This chapter gives some insight into the PCIe40 drivers and Emulator and their working principles. Note that:

- the drivers are a complex piece of software and therefore this description does not cover all aspects related to the driver.
- the most interesting scripts could be found in the `software-repo/pcie40_driver` folder.
- There is also some documentation available in two fashions: A User Guide (<http://lbyum.cern.ch/daq40/guide/software/devel/ug.html>) and a Developer Guide (<http://lbyum.cern.ch/daq40/guide/software/latest-unstable/dg.html>).

8.1 Introduction into the drivers, general concepts and (shared) structs

The drivers and PCIe40 Emulator (called Emulator in advance) share as much code as possible. The shared code can be find in the file `software-repo/pcie40_driver/daq.h`. Beside this shared code, there is driver specific code which is located in the file `software-repo/pcie40_driver/daq.c` and there is emulator specific code which is located in the file `software-repo/pcie40_driver/daq_emu.c`.

There are a couple of structs defined in `software-repo/pcie40_driver/daq.h` on which the whole driver and Emulator structure is based. These are:

- *pcie40_daq_state*: This is a struct keeping track of the state of a PCIe used for DAQ purposes. The most important members of this struct are the streams: *mean_stream*, *meta_stream* and *odin_stream*. They keep track of the streams related to this interface
- *pcie40_dma_stream*: This is a struct keeping track of info related to a stream. The *pcie40_dma_stream* has the following important members:
 - *cdev_name*, *cdev_minor*, *cdev*: These members administer and keep track of the chracter device associated with this stream.
 - *pcie40_dma_map*: This struct keeps track of the mapping of the Buffer of the stream into the main memory of the Host PC.
 - *read_off*: This is the read pointer for keeping track up to which level the memory mapped Buffer is read. The Stream controller also keeps a copy of this pointer such that it knows which part of the memory mapped Buffer can be (re)used for writing data to. The function *dma_stream_set_read_off* is used to update this member and push its new value toward the Stream controller via BAR1.
 - *write_off*: This is the write pointer. It keeps track of up to which level the memory mapped Buffer is filled with data. This pointer is administered by the stream Controller and retrieved to the driver via the function *dma_stream_get_write_off*.
 - *lock_pid* and *off_lock*: These members keep track of locking: They make sure the memory mapped Buffer of the stream is not accessed or written when another process is performing operations on this data.
 - *map*: This member of the type *pcie40_dma_map* keeps track of the memory mapped Buffer for this PCIe40 DAQ stream.
- *pcie40_dma_map*: This struct keeps track of the mapping of the Buffer of a stream in main memory. The member *entries* is a array of *pcie40_dam_buffer*-structs. The length maximum length of this array is defined by the *max_entries* member. The member *num_entries* contains how many entries are actually in use (the size of the memory map can be smaller dan hte maximum value). The *size* member contains the total amount of memory that is mapped, in bytes.

- *pcie40_dma_buffer*: This struct contains the start address and size of a single Buffer element.

Furthermore there are some important functions for the proceeding of this text. These are:

- *dma_stream_get_write_off*: This function reads the current value of the write pointer of the memory mapped Buffer administered by the stream controller via a PCIe transaction via BAR1. The resulting write pointer is returned by this function.
- *dma_stream_get_read_off*: This function read the current value of the read pointer of the memory mapped Buffer of known by the stream controller via a PCIe transaction via BAR1.
- *dma_stream_set_read_off*: This functions updates the current value of the read pointer of the memory mapped Buffer administered by the stream controller via a PCIe transaction via BAR1.

The structs and functions in this file are shared between both the drivers and Emulator. In the proceeding of this chapter, first the drivers are discussed and afterwards the Emulator.

8.2 Start-up

At start-up, the most important function call will be that of the *pcie40_probe()* (`software-repo/pcie40_driver/main.c`). It scans the PCIe busses for PCIe40 cards. For every found interface, a *pcie40_state* struct (`software-repo/pcie40_driver/common.h`) is created. This struct keeps track of the state of the found interface. The structs have a unique identifier per device. Certain cards (like the PCIe40 card) have two PCIex8 links. It therefore keeps also track of which of those two link this instance of the *pcie40_state* keeps track of (either 0 or 1). The Bittware A5PL has only one PCIex8 interface so only one struct will be initialized. The structs keeping track of the 3 streams of this interface. This is done by the members *ecs_state*, *daq_state* and *cvp_state*. These are respectively of the type *pcie40_ecs_state*, *pcie40_daq_state* and *pcie40_cvp_state*. In our case, only the struct related to the DAQ interface is of interest. The struct keeping track of the DAQ stream is initialised within the *pcie40_probe()* function via a call to *pcie40_daq_probe()* function.

After the cards are initialised, memory should be allocated for the streams. This is done by the function *dma_map_alloc()* (`software-repo/pcie40_driver/daq.c`). This function saves the data in the struct of type *pcie40_dma_map* (`software-repo/pcie40_driver/daq.h`). This struct contains pointers/references to the allocated space for the stream in kernel space. There are also function available to map this kernel space to user space in a consecutive manner.

The configurations regarding the amount of mapped memory for every stream is located in the file `software-repo/pcie40_driver/pcie40_dma_regmap.h`. This file contains a copy of the memory addresses of the different memories and Buffers as defined in QSys as well as how much memory should be reserved per stream. This file is generated from `software-repo/common/pcie40_dma_remap.cfg`. Both the memory mapping for the VHDL package as the drivers are generated from this file such that both the firmware as the drivers use the same memory map. The .cfg config file is made by hand and should be altered by hand. So when extra streams are added, this file should be altered accordingly.

8.3 Consuming data

Now the card is initialised and memory is mapped for the circular Buffer, the data of the card can be consumed. This should be done by reading the read pointer and write pointer of the memory mapped Buffer are used to determine which data is not yet consumed such that this data can be read out of it using these pointers¹.

¹Keep in mind that the memory mapped Buffer is implemented in a "double buffer" fashion in user space. This means that when a certain set of data starts at the end of the circular Buffer and end at the begin (because it overlaps with the end of the circular Buffer), from the application side of the story it looks like that the begin of the circular Buffer is repeated after the last entry of the buffer. In this manner any application should not take into account any chunk of data is wrapped around the circular Buffer.

The second approach might be to read data out of the character device file. The drivers keep track of if there is data available to be consumed and keep track of the status of the read and write pointer. However this approach is not (yet) implemented because an extra copy of the data is generated which should be avoided on the fast datapath as much as possible. Thus high performance applications this is not desirable, for testing/debugging purposes of the two-way DMA this might be of interest to implement and use (mainly due to the ease of use).

8.4 PCIe40 Emulator

Besides the PCIe40 drivers, there also exist a PCIe40_emu driver. This driver emulates a PCIe40 card such that testing and development of the user space software can be performed without having the access to the actual hardware card. The drivers and Emulator share common code in **software-repo/pcie40_driver/daq.h**. The files related specifically to the Emulator can be recognized at the "_emu" label in their filename and are found in the **software-repo/pcie40_driver** folder. Define *PCIE40_EMU* in **software-repo/pcie40_driver/daq.h** to compile the drivers with the emulator as endpoint instead of a PCIe40 card.

The Emulator emulates a data generator. It starts a kernel thread that loops over the memory mapped locations of the circular Buffer and writes data in locations that are free. It determines free locations based on the value of the read pointer and write pointer. It also updates the write pointer after filling an empty location. In this manner it mimics the behaviour of the current PCIe40 cards.

Chapter 9: Continuation of this work

Now it is known how the PCIe40 systems works and operate, this short chapter will give a brief view on how we (mainly Paolo Durante) believe this work should be continued to make a demonstration of two-way DMA transfers.

9.1 (Possible) Demo setup

The most elegant demo would be a system which the Host PC sends data to the PCIe40 card via DMA and gets this same data looped back via another stream. Using such a demonstration setup, things like data validity could be checked easily. This requires the addition of one extra stream: one which receives data from the Host PC. Lets call this stream "retina" for now. The received data should be consumed by a consumer and written to the circular buffer of an already existing outward stream. In this way the data received from the Host PC is looped back to the FPGA board.

9.2 Expected changes

The advice from Paolo Durante is to first alter the Emulator and driver. If this combinations seems to work fine, also the firmware could be altered in such a way that the desired behaviour is accomplished.

9.2.1 Drivers and Emulator changes

The changes to the drivers and Emulator will start by adding a function to write the write pointer to the stream controller, thus by adding the function *dma_stream_set_write_off* to the file `software-repo/pcie40_driver/daq.h`¹. Also the extra stream should be added to the *pcie40_daq_state* struct and the stream should be initialised in the function *pcie40_daq_emu_probe()* of `software-repo/pcie40_driver/daq_emu.c`. Furthermore an extra kernel thread should be added to the `software-repo/pcie40_driver/daq_emu.c` file. This thread should loop over the memory mapped buffer of the retina stream and should consume any new data (and update the read pointer afterwards²). Thereafter this thread should make sure the consumed data is written to the memory mapped buffer of stream that will return the data to the Host PC.

If the streams and Emulator are implemented correctly, a way to send data to the retina stream should be implemented. Similar to reading data from a stream (as discussed in section 8.3), this could be implemented in two ways³.

The first way would be by using the known read and write pointers to directly write data to the right memory location within the memory mapped Buffer (after which the write pointer is updated).

The second way would be to implement a system write call for the character device file of the retina stream. The data written to the character device file, should be written to the right location in the memory mapped Buffer and write pointer should be updated accordingly.

In the second way, data will first be copied to the character file after which they are copied to the memory mapped Buffer. Therefore it is expected that this implementation is slower than the first way. It might be of interest to implement both ways and investigate the difference in performance between the two ways.

¹This might include some changes to the Stream controller to make the buffer/memory location `P40_DMA_DAQ_STREAM_OFF_HOST_BUF_WRITE_OFF` writable via BAR1.

²Note that the read pointer and write pointer are administered by the stream controller and therefore are from the perspective of the stream controller. In this case this means that the writer to the memory mapped buffer should update the write pointer while the consumer (stream controller) updates the read pointer. This is the other way around compared to the original version in which the stream controller updated the write pointer and the driver updated the read pointer after data consumption.

³Their might also be other way to implement this, but the two ways discussed are the most obvious ones.

9.2.2 Firmware changes

If the drivers and Emulator works properly, the retina stream should also be added to the DMA controller as well. The retina stream controller should handle the received data in a bit different way than the other streams. The read and write pointer should work opposite; it should start reading data out of the circular buffer when the write pointer is larger than the read pointer. Also the scheduler should have an extra set of inputs and outputs to handle the descriptor for the retina stream.

Note that the retina stream controller still is the piece of hardware which initiates the DMA transfer from the host PC to the FPGA. After data is written to the memory mapped circular Buffer, the write pointer is updated via the BAR1 register. This makes that the retina stream controller knows data is available to be transferred to the FPGA. Compared to the original descriptors the destination and source address should be changed, such that data is retrieved from memory to the circular buffer. Because the descriptors for data transfer from FPGA to Host PC use the same pins as descriptors for data transfer from Host PC to FPGA, it is expect that the same range of IDs for the descriptors is used (please verify this via the HIP documentation). Make sure that this retina stream uses unused descriptor IDs.

The last part is adding hardware to consume the data for the retina stream. This hardware should read data from the circular Buffer of the retina stream (and make sure the read pointer is updated afterwards) and dump this data into the circular Buffer of the returning stream. Such that data received from the Host PC is looped back to the Host PC.

Don't forget to update the register map in `software-repo/common/pcie40.dma.regmap.cfg` with the locations and size of the new Buffer and stream.

9.2.3 Validation of Emulator behaviour

An important question for driver developers now is if the behaviour of the Emulator is equal to that of the hardware. A way to test if at least the same data is received is to feed both systems (Emulator and real firmware) with the same (large) set of data and check if the check-sum of the returning data is equal for both systems. Although this does not ensure that timing behaviour is similar for both systems (think of when are the pointers updated in relation to the data becoming available) but it validates that both systems have the a similar functional behaviour.

Part III

Conclusions

Chapter 10: Reflection, conclusions and recommendations

The end of my 9 week stay at CERN is closing in on me. 9 Weeks in which I made new friends (possibly for life), visited all kind of places near Geneva, learned a lot about High Energy Physics, and also tried to improve my Embedded Systems/Digital Design skills. 9 Weeks is not a lot of time to work on a project, especially if also 20 hours a week of lectures is offered for 5 weeks. The amount of achieved results are less than I expected upfront. Looking back I think there are three main reasons for the limit amount of results. First of all the existing project is large and complex. It consists out of thousands and thousands of line of code and therefore it is hard to get familiar with. Secondly, I encountered a lot of small problems related to the tools but also to faults made by myself, which just consumed a lot of my time (see for example section 5.6). This, let call it tool fighting, consumed a lot of time while adding virtually no value. Lastly it was in the middle of the summer holidays which made that supervisors and other experts also had their own holidays. Even though they were still reachable by email, this makes communication to retrieve extra information or to discuss problems way harder. Although this sounds like a bad thing, this touch me that I should be more aware when important people are in and out and what information I expect to need of them within their out-of-office period such that I could retrieve this info before they leave.

Looking back at the goals stated in section 4.2, the 9 weeks were not without any results. It was managed to get the PCIe40 firmware ported to the Arria V GZ FPGA and it seems to be run fine. This is a major achievement because it would cost considerable amounts of time if hardware had to be adjusted problems arising in this specific FPGA family/model. Also a good start has been made to port/setup the simulation environment for this device. Although it is not yet working correctly, it might be very close to be working correctly. Unfortunately it did not succeed to change the firmware and drivers in such a manner that two way DMA is possible. However a great deal of the information needed to get this up and running is included in this report such that someone who will continue with this project should have a head-start.

As a recommendation for performing a similar project (Summer Student project) within a large project, it would be great if some high level documentation would be available. The PCIe40 project is an extensive one and it takes time to get familiar with it. Having some high level documentation available stating the basic concepts, relations and where to find what, would make it far easier and foremost less time consuming to get familiar with the project and the code base. I hope this project could make a (little) difference for a next student which will continue this work. Besides this tip, I would like to state that I was impressed by the way the PCIe40 code was structured and the automation of how Quartus and QuestaSim projects are setup. All files were well and logically ordered within folder structures and only files expected within a repository were within the Git repository. The necessary Quartus and QuestaSim project could just be initialised via Tcl scripts instead of manually. I had never thought of performing version management of VHDL code within Git, but it seems to work perfectly. I'll definitely going to adapt this within my own projects to prevent file names like "FIFO_v1_v2_final_improved_evenBetter.vhdl".

Bibliography

- [1] V-Series Avalon-MM DMA Interface for PCIe Solutions User Guide, retrieved at August 10th 2018, Oct 2016.
- [2] A. Abba, G. Punzi, F. Spinella, P. Marino, D. Tonelli, S. Stracka, F. Lionetto, D. Ninci, M. Petruzzo, A. Cusimano, et al. A specialized track processor for the LHCb upgrade. Technical report, 2014.
- [3] L. Ristori. An artificial retina for fast track finding. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 453(1-2):425–429, 2000.

Appendix A: Firmware git folder structure and purpose

This appendix will describe the folder structure of the *devel-av* branch of the *lhbc-daq40-firmware* git repository as well as the purpose of the folders and the specific files. Only the most important folders and file will be discussed. Note that with a `intrinsic(*)` it is meant all files/folders except the ones already discussed in the current folder.

A.1 Naming conventions

Some naming conventions of files and folders are discussed in this section

- In a lot of folder you will find folders called "arriax", "arriav" and "stratixv", which mean those folder contain files related to respectively the Arria 10, Arria V GZ and Stratix V FPGA.
- Files containing a qsys system contain start with preamble *qsys_*.
- "ecs" means "Experiment Control System". Folder or files with this in their name probably have an interface with the ECS in a certain manner.
- "x8": x8 stand for PCIe x8 interface. Some folders contain "x8x8". This has to do that the PCIe40 card has 16 PCIe lanes which are divided in two PCIe x8 interface. So a folder named "x8x8" will contain a part of the firmware that initialises two PCIe x8 interface while a folder "x8" will contain parts of the firmware that will only initialise one PCIe x8 interface.

A.2 Git content

- **assignments:** Contains the assignment files. These contain for example Device assignment, pin assignments and alike.
 - **Bittware_a5pl.tcl:** This is the assignment file related Bittware A5PL card (the card used in this project).
 - **nallatech.385a.tcl:** It is assumed that this is the PCIe40 card assignment files. Content might be of interest as comparison.
 - *****: It is assumed that the other files are assignment files of other boards for the PCIe40 firmware.
- **configs:** This directory contains TCL files for generating, configuring and compiling the Quartus project of the firmware for different different boards
 - **minidaq:** Purpose of this folder is unknown
 - **other/a5pl/pcie/dma.ecs:** Contains the `quartus.tcl` file which generates, configures and compiles the quartus project of the firmware for the Bittware A5PL board.
 - **others/*:** Folders for the different boards for which a firmware version have been made in the past.
- **docs:** Probably related to documentation or the automatic generation of it. However this directory has never been entered during this project so these are just assumptions.
- **modules:** This folder contains all files for the different modules ("components") used in the firmware of the PCIe40 card.

- **board_id**: It is assumed that this folder contains files related to a Board.ID memory (to uniquely identify every different board within the EB cluster).
- **build_id**: This folder contains files and Qsys systems for the build_id ROM. This ROM is used to store the firmware build number on the FPGA.
- **chip_id**: This folder contains files and Qsys systems for the chip_id RAM. It is assumed that this RAM is used to store an ID to uniquely identify every FPGA within a certain cluster of FPGAs.
- **pcie**: These files are related to the PCIe interface.
 - * **dma_ecs**: This folder contains all files related to the PCIe interface setup to perform DMA transfers with an interface to ECS. Most of the files that should be changed, added or altered are in this folder.
 - **arriav**: Files related to the PCIe interface and some memories.
 - **pcie_pll_av.qsys**: PLL for Arria V GZ. This PLL is used to generate clock signals for the HIP/PCIe interface.
 - **qsys_pcie_dma_ecs_x8.qsys**: This Qsys system contains the PCIe interface as well as a couple the buffers and memory map memories for the different streams.
 - **qsys_pcie_dma_ecs_x8_wrapper.vhd**: Wrapper for the **pcie_pll_av.qsys** and **qsys_pcie_dma_ecs_x8** such that the output ports are always equal even when different IPs for different devices are used.
 - **sim/tb/x8**: Files related to the simulation of the HIP.
 - * **mentor.tcl**: This script file is the starting point of a simulation run for the Arria V GZ FPGA.
 - * **sim.top.sv**: Top level entity to connect A5PL top level entity to tbed virtual root complex. This is the top-level entity of the simulations.
 - * **wave.do**: File containing the wave setup during simulations.
 - **arriax**: Similar to **arriav** but for Arria 10 FPGA.
 - **stratixv**: Similar to **arriav** but for the Stratix V.
 - **sim**: It is unknown what the use of this directory is.
 - *****: These seem to be all files related to DMA with are not related to the PCIe interface, thus the DMA controller. This includes (but the list is not exhaustive) the streams, scheduler and top level entity of the PCIe system.
 - * **sim**: This folder contains files regarding the virtual root complex needed for the simulation environment of for the firmware. The `$$device$$/mentor.tcl` file is the one in which the tbed IP is initialized.
 - * *****: It is unknown where all other files and folders are used for.
- **profiles**: This directory contains top level entities of the designs as well as timing constrain files.
 - **others/a5pl/pcie/dma_ecs/**: This directory contains the top level entity for the firmware made for the Bittware A5PL as well as the timing constraint file.
 - **others/***: Same as above but for different boards and possibly different versions of the firmware.
 - *****: It is unknown where these files are used for.
- **scripts**: This folder contains commonly (used by firmwares for different devices) or generally used scripts.
 - **mentor/common.tcl**: Common functions used for setting up and running simulations
 - **quartus***: It is assumed that these are the function used by the script that generates and compiles the firmwares.
 - **vsim***: It is assumed that these are the functions used by the script that performs the simulations.
 - **gitlab/ci-retry.pi**: config file for the CI GitLab pipelines.
 - *****: The purpose of these files and folders is unknown.

Appendix B: Content and changes of firmware commit c165fe1

This Appendix contains some info regarding the content of commit c165fe155149e54513a88050b38cab6c2638b794. It states some specific info about files, where they are based on and what their purpose is.

- `.gitignore`: To keep project specific files out of the git repo.
- `assignments/bittware_a5pl.tcl`: Assignment file which assigns the device for the project (see also section 5.6 for related experienced problems) and assignment of pins and clocks. It is not known anymore which file is used as basis. Pin assignments are exported from Quartus (after a successful compilation) and copied in this file. All assignments for pins that are not present have been commented and some changes have been made based on info provided by the Assignment Editor of Quartus.
- `configs/other/a5pl/pcie/dma_ecs/quartus.tcl`: Based on `configs/other/svpe/pcie/dma_ecs/quartus.tcl`. Mainly paths have been changed to refer to files and Qsys systems that are made for for the Arria V GZ.
- `modules/board_id/arriav/board_id_issp_x12.qsys`: Board ID Qsys system. Copy of `modules/board_id/board_id_issp_x12.qsys` ported to Arria V.
- `modules/build_id/arriav/build_id_rom.qsys`: Build-ID Qsys system. Basically a copy of the Build ID Qsys file for the Arria 10, however it uses the custom Build-ID ROM IP, see also section 5.3
- `modules/build_id/arriav/build_id_rom.tcl`: TCL file for adding Build-ID ROM IP to IP library.
- `modules/build_id/arriav/build_id_rom_rom.vhd`: VHDL file of Build-ID ROM, generated by MegaWizzard. See also section 5.3.
- `modules/chip_id/arriav/chip_id.qsys`: QSys System for the Altera Unique Chip-ID IP. Ported from `modules/chip_id/arriax/chip_id.qsys`
- `modules/chip_id/arriav/chip_id_ram.qsys`: QSys system for Chip-ID RAM. Basically a copy of the Chip-ID Qsys file for the Arria 10, however it uses the custom Chip-ID RAM IP, see also section 5.3
- `modules/chip_id/arriav/chip_id_ram_ram.vhd`: VHDL file of Chip-ID RAM, generated by MegaWizzard. See also section 5.3.
- `modules/chip_id/arriav/chip_id_ram_ram_hw.tcl`: TCL file for adding Chip-ID RAM IP to IP library.
- `modules/pcie/dma_ecs/arriav/es/*`: It is unknown know where this directory is used for.
- `modules/pcie/dma_ecs/arriav/pcie_pll_ax.qsys`: Qsys system for PLL, see section 5.4.
- `modules/pcie/dma_ecs/arriav/qsys_pcie_dma_ecs_x8.qsys`: Qsys system for the PCIe interface and stream memories, see also section 5.2.
- `modules/pcie/dma_ecs/arriav/qsys_pcie_dma_ecs_x8_wrapper.vhd`: Wrapper for PLL and PCIe interface. Makes sure that, although using different FPGAs and PLLs, the PCIe interface has the same ports such that the same DMA Controller can be used. Based on wrapper Stratix V. FPGA pin names have been changed.
- `modules/pcie/dma_ecs/arriav/sim/*`: Copied from `modules/pcie/dma_ecs/arriax/sim/*`. Has not been changed in this commit.
- `profiles/other/a5pl/common.sdc`: Added because it was also there (and empty) for `profiles/other/svpl/common.sdc`
- `profiles/other/a5pl/pcie/dma_ecs/top.sdc`: Copied from `profiles/other/s5pl/pcie/dma_ecs/top.sdc`. File has not been altered.

Appendix C: Content and changes of simulation commit fa8238f

This Appendix contains some info regarding the content of commit fa8238f7be5bd05c0eb0116fce83255aede1f50e. It states some specific info about files, where they are based on and what their purpose is.

- `modules/pcie/dma_ecs/arriav/sim/tb/x8/mentor.tcl`: This file is the basis of the simulation. It start the process of setting up, starting, and running the simulation. The changes made to this file made it compatible with the Arria V GZ. For example in line 31-35 the order of loading some libraries has been changed (see also subsection 6.2.2). Furthermore most changes are just changes of references to Arria V GZ files instead of Arria 10 files.
- `modules/pcie/dma_ecs/arriav/sim/tb/x8/sim_top.sv`: This is the top level entity of the simulation. It is based on `modules/pcie/dma_ecs/arriax/sim/tb/x8/sim_top.sv`. Because a different tbed version is used, the name of the instance of it should be changed.
- `modules/pcie/dma_ecs/arriav/sim/tb/x8/wave.do`: Setups the wave panel in QuestaSim. Some waves have been added.
- `modules/pcie/sim/arriav/altpcierd_tl_cfg_sample.v`: This file is probably added because it was not there and it had to be there. It is unknown what the purpose of this file is. It is expected that it is a copy of `modules/pcie/sim/arriax/altpcierd_tl_cfg_sample.v`
- `modules/pcie/sim/arriav/altpcietb_bfm_rp_gen3_x8.sv`: It looks like that this file is related to the tbed virtual root complex. However it is not known what the exact function of this file is.
- `modules/pcie/sim/arriav/mentor.tcl`: This script defines and generates the virtual root complex. It is a copy of `modules/pcie/sim/arriav/mentor.tcl`, but with some updates names.
- `scripts/mentor/common.tcl`: This file contains the common functions used for setting up and running simulations. A function for setting up/loading the Arria V GZ libraries has been added as well as some changes in library order (see subsection 6.2.2) and some command related to the issue discussed in subsection 6.2.1.

Appendix D: Generating project and simulation environment

This Appendix gives some info in how to setup a Quartus project and how to start the simulations. Note that Quartus version 16.1 is used (and is supposed to be used) as well as QuestaSim version 10.1d.

D.1 Before you start

Before you begin, make sure that you have set the environment variable *QUARTUS_ROOTDIR* by adding *export QUARTUS_ROOTDIR=<path/to/quartus/rootdir>* to your **bash.rc** (or alike).

D.2 Setup Quartus project

Navigate in a terminal to the folder you want to create the project directory. Start Quartus from this folder (or start Quartus and navigate in the Tcl Console to the right directory). Start the script by using the command *source /path/to/firmware/repo/configs/other/a5pl/pcie/dma_ecs/quartus.tcl*. You will see some options, which are self explaining.

D.3 Run simulation in QuestaSim

At cern, the *vsim* command to run QuestaSim is located at */afs/cern.ch/project/parc/questa101d/questa_sim/linux_x86_64* (so you could add this path to the PATH environment variable). Go to the directory you would like to create the simulation directory. Run *vsim -do /path/to/firmware/repo/modules/pcie/dma_ecs/arriav/sim/tb/x8/mentor.tcl* (or start QuestaSim, navigate to the directory you would like to create the simulation directory and *source /path/to/firmware/repo/modules/pcie/dma_ecs/arriav/sim/tb/x8/mentor.tcl*).

When QuestaSim has started, type *do_all* in the console to start the setup and running of the simulation.

If you changed something and want to start a new simulation, break the current simulation (menu: Simulate→Break), enter the command *reload* in the QuestaSim terminal ("Transcript") and execute the *do_all* command.

A possibly handy reference for QuestaSim/ModelSim Tcl command could be found at https://users.ece.cmu.edu/~kbiswas/se_cmds.pdf.