Department of Physics and Astronomy
University of Heidelberg

Bachelor Thesis in Physics
submitted by

Vitali Karasenko

born in Kriwoi Rog (Ukraine)

2011

Design, implementation and testing of a high speed reliable link over an unreliable medium between a host computer and a Xilinx Virtex5 FPGA

This Bachelor Thesis has been carried out by Vitali Karasenko at the

Kirchhoff Institute for Physics in Heidelberg

under the supervision of

Prof. Dr. Karlheinz Meier

Design, implementation and testing of a high speed reliable link over an unreliable medium between a host computer and a Xilinx Virtex5 FPGA

The presented thesis describes a method of communication between a host computer and a Virtex5 FPGA over a packet network. The link features speeds of up to 1 Gbit/s in full duplex mode and provides error correction and retransmit functionality to ensure lossless communication over a lossy medium. The protocol was implemented using VHDL. Testing the protocol yielded net bandwidths of over 100 MB/s in both directions for realistic use cases. The throughput is unstable at large timescales with as of yet unknown reasons. The only bug that was found during testing is described with a workaround which does not affect the usability of the system.

Design, Inbetriebnahme und Test einer sicheren Hochgeschwindigkeitsverbindung über ein unsicheres Medium zwischen einem Computer und einem Xilinx Virtex5 FPGA

Die vorliegende Arbeit beschreibt eine Kommunikationsmethode zwischen einem Computer und einem Virtex5 FPGA über ein Paketnetzwerk. Die Verbindung unterstützt Bandbreiten von bis zu 1 Gbit/s im Duplexmodus und verfügt über automatische Fehlererkennung und Neuübertragung von korrumpierten oder verlorenen Daten um eine verlustfreie Verbindung über ein verlustbehaftetes Netzwerk sicherzustellen. Das Protokoll wurde implementiert mittels der Hardwarebeschreibungssprache VHDL. Der Test des Protokolls lieferte eine Bandbreite von über 100 MB/s in beide Richtungen unter günstigen Bedingungen, die einem wahrscheinlichen Anwendungsfall entsprechen können. Die Bandbreite des Protokolls ist instabil auf langen Zeitskalen aus bislang unbekannten Gründen. Der einzige im Test gefundene Fehler sowie eine Methode zur Umgehung wird beschrieben und es wird festgestellt, dass die Anwendungsmöglichkeiten des Protokolls trotz des Fehlers nicht beeinträchtigt werden.

Contents

1 Introduction

The field of neuroscience studies the behaviour of neurons and neuronal networks with the ultimate goal to understand how the assembly of 10^{11} neurons in a mammalian brain gives rise to complex behaviour or even intelligence. The Electronic Vision(s) group participates in this research in various ways: Models of neuronal networks are studied and simulated in software to understand their complex behaviour. Additionally, artificial neuronal network chips were developed and built to be able to emulate large networks directly in hardware which is much faster than a numeric simulation in software.

Several such chips were developed, first the Spikey [**?**], and later the HICANN [**?**]. These chips do not communicate directly with a host computer but are first interfaced via a FPGA which is then accessed by the host. An often used configuration is when a single FPGA connects to several chips and is in turn accessed by a single host. The FPGA thus acts as a switch between the host and the chips, routing the configuration and spike data between them. Because of this topology and the fact that spike data is usually very large the link between the FPGA and the host has to be as fast as possible. Since the communication medium is unreliable, a protocol needs to be implemented that secures the link against corrupted or missing data. Moreover, the availability of modern FPGAs like the Virtex5 [**?**] by Xilinx pose a chance to use the built in ASIC modules to implement parts of the protocol thus greatly reducing the usage of custom logic.

The goal of this thesis was to design, implement and test a reliable high speed connection between a host PC and the Virtex5 FPGA using already available digital circuits and custom built logic. The system should also use up as little of the FPGA's resources as possible so that there is enough space left for the implementation of additional modules.

Outline

The thesis begins by giving an overview over the available hardware and the to-be-implemented protocol. Then, the implementation of the system and the design of the custom modules is described. In the next chapter the methods of testing the performance and functionality of the system are explained and results are presented and interpreted. The thesis then proceeds to discuss the obtained results and suggesting in the outlook how to further improve the design and closes with a conclusion summarising whether the posed goals were achieved.

2 Description

The presented custom protocol is related to the Ethernet technology standardized in IEEE 802.3. The communication is packet based and also uses layers with similar functionality as described in the OSI model [**?**]. However, the protocol only implements the bottom four layers. The protocol is capable of receiving and transmitting data simultaneously, this is usually called the full duplex mode. Both sides are separated from each other because they operate with different clocks without a fixed phase relation. Thus, as little communication between the receiving and the transmitting side should be done without precaution to avoid corruption.

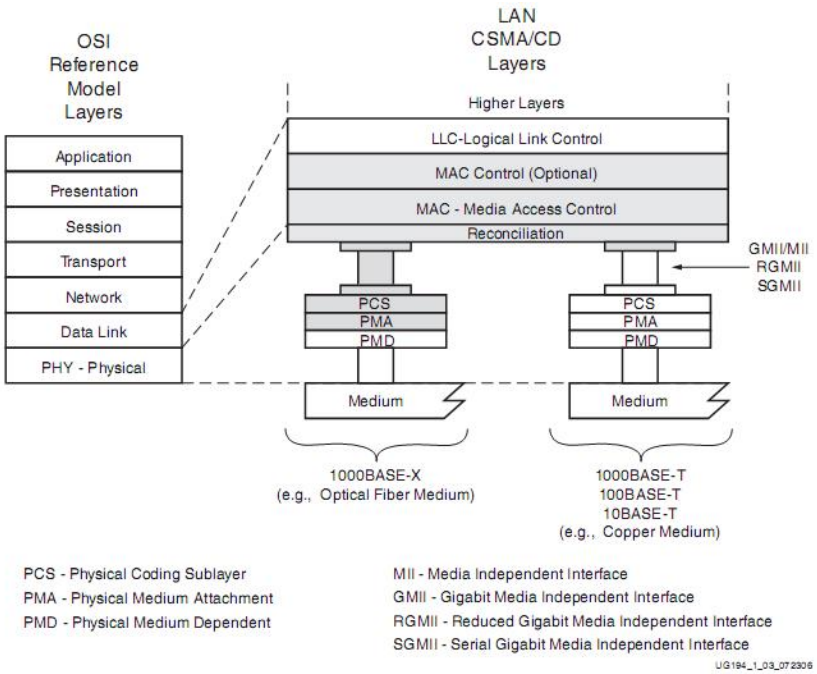

Figure 2.1: The different layers in the OSI model. Pictured on the left are the reference OSI layers and on the right the breakdown of the data link and physical layers in sublayers in the used system. The presented protocol implements the bottom four OSI layers. Figure taken from [**?**]

Each of these individual layers can be seen as a separate logic entity, the physical layer is a

separately packaged ASIC, the data link layer functionality is provided by an ASIC embedded in the FPGA, and the network and transport layers are represented by a design implemented in custom logic in the FPGA. The focus and task of this thesis was to encorporate all of these layers into one working design such as that it is possible to establish a line of communication between a host PC and the ml505 Evaluation Board which is comparable to the TCP/IP protocol [?] at the Transport Layer.

2.1 Description of the layers

The protocol can be divided into four layers, each performing a specific task. The communication over the network is conducted through packets which are a series of bytes. The actual payload is encapsuled in an ethernet frame which carries additional information to ensure that the packet finds it's way through the network. Figure 2.2 shows a schematic of an ethernet frame. The packet begins with the preamble and the start-of-frame delimiter which indicates an incoming packet to the physical layer on the receiving side. The physical layer then indicates to the data link layer an incoming packet via a control signal and passes the destination and source addresses[1] of the frame encoded in the first 12 bytes. Following them is a 2 Byte long Type/Length field which typically indicates the packet lenghth if it's value is 0x0600 or lower and carries information about the used protocol in the upper layer if it is higher. The presented protocol uses the Type/Length field to indicate that the packet is intended for the transport layer via an arbitrarily chosen value of 0x8899. After the Type-/Lenght field begins the payload and after the payload comes a padding field which assures that the whole frame is always at least 64 Bytes long as required by the Ethernet specification. No padding is done if the framesize is already 64 bytes or longer. Following the pad is the 4 Byte long FCS field which carries the checksum for the frame. Note that the first few bytes of the payload are actually fields reserved for the transport layer, their meaning will be explained in the following sections.

Figure 2.2: Schematic of an ethernet frame

2.1.1 The physical layer

The physical layer (PHY) encodes the packets received from the data link layer into the physical representation on the medium and decodes them on the receiving side into binary bytesignals. This layer therefore abstracts what kind of medium is chosen for communication, as several possibilities like electric or optic cables are possible. The chosen setup for

[1] often called the MAC addresses

this thesis was an electric wire using the Gigabit Media Independent Interface [**?**] (GMII) to connect to the data link layer.

2.1.2 The data link layer

The data link layer is represented via the Media Access Control [**?**] (MAC) module in the FPGA. It's function is to strip incoming packets off the pad and the FCS field and perform the CRC check to test whether the packet was corrupted. Likewise, data passed to the MAC from the network layer is appended with a possible pad field and the MAC will also automatically calculate the checksum and transmit it with the data.

2.1.3 The network layer

The network layer module was designed during this thesis and mostly acts like an adapter between the the data link and the transport layer. However, as the module also filters the incoming traffic for special MAC addresses and also decides to which MAC address outgoing packets should be sent, it also performs some network layer functionality. The exact functionality of the network layer is described in chapter 3

2.1.4 The transport layer

The Automatic Repeat reQuest (ARQ) [**?**] Protocol was written in VHDL and implemented in custom logic. It performs tasks similar to the Data-Link/Transport-layer functionality in the conventional Ethernet protocol using a sliding window algorithm. The purpose of the ARQ is to provide a reliable communication between clients over an unreliable network. It was not written during this thesis. The ARQ manages pointers to memoryslots in each one memorybuffer for receiving and transmitting direction based on sequence- and acknowledgement numbers that are part of the packetheader. In receiving direction, the ARQ target module reads the sequence number of the packet and assignes a corresponding slot in the packet buffer. If the sequence number is out of the window or is already acknowledged the target instead initiates a retransmit request or drops the frame respectively. Once a valid packet is written into the buffer the ARQ signals the upper layer that there is new data available by raising a flag and selecting the corresponding slot in the buffer. The upper layer then reads the data and raises a flag when it's done which causes the ARQ to change the pointer to the next packet. The buffer is implemented as a dual port RAM, that means that while the ARQ controls the writing side of the receiver-side buffer, the reading side is directed at the upper layer which can read the stored data independently, hence no arbiter is needed to control the access to the RAM.

The transmitting side works by the same principle: The upper layer waits for the ARQ to raise a flag when it's allowed to write to the buffer. Once this is done the upper layer starts writing at the position marked by a pointer given by the ARQ and notifies the ARQ when it is done. The ARQ then calculates the sequence number for that packet and passes it to the network which fills the rest of the header and transmits the frame over the network. The master then waits a fixed amount of cycles to receive an acknowledgement for that packet. If that time is up the master initiates a retransmit, otherwise it starts sending the next packet after the correct ACK is received. To speed up the protocol the master does not require to receive an ACK for each packet individually, instead it assumes that upon receiving an ACK for a packet the packets with lower sequence numbers are correctly received as well.

As is discussed in [**?**], the range of the sequence and ACK numbers needs to be at least 2*(window size) + 1 to avoid window desynchronizations. The largest tested window size was 16, thus a sequence size of 256 was sufficient, which is represented with a single Byte. Thus, the SEQ and ACK fields were only a single Byte wide. This, however does not need to be the case and it is possible to make these fields larger if necessary. The seqv field however is always only one Byte long. It denotes whether the packet carries actual payload or was only send to update the ACK numbers on the receiving side. These packets are often called ACK packets.

A block diagram of the ARQ modules in duplex mode is showed in figure 2.3

Figure 2.3: Blockdiagram of the ARQ modules. Figure taken from [**?**], for a detailed description of the protocol see also [**?**]

2.1.4.1 Packet Buffers

The ARQ does not store the packet by itself but only provides the pointers to write in and read from a suitable packet buffer. Such a packet buffer was implemented using the Block-RAM [**?**] components of the Virtex5 FPGA. A reference design for the packet buffer was included with the ARQ modules which was then slightly modified to enable different read-/write speeds at the buffer. The protocol is designed to operate on a twodimensional memory array of the size $M = \text{packetsize} \times \text{windowsize}$. When accessing the buffer, the actual adress in the memory is calculated using two parameters. The first integer is supplied by the ARQ denoting the position of the packet within the window. The second integer is supplied by the network layer and the upper layer denoting the position of the byte within a single packet. This has the immediate advantage of abstracting the window from the network and the upper layer, because neither need to have any knowledge of how large the window is and where the current position within it is. During operation, the W coordinate is supplied by the ARQ according to its sliding window algorithm, and the P coordinate is determined

by the Network which will store a packet in that buffer or read from it to pass it on to the MAC. The buffers are in a dual port configuration, that means that read and write operations can be performed on them simultaneously at different speeds and datawidths. The buffer mark the connection between the ARQ and the upper layer, because the upper layer reads from and writes into the outward facing side of the buffer. There are two buffers each one for the receiving and transmitting side, thus they are called the rx_buffer and tx_buffer respectively.

2.2 The Hardware

The design was implemented on a ml505 Evaluation Board by Xilinx [**?**], which features a Virtex 5 FPGA and various other useful components such as buttons and switches, communication ports or LED screens which can be accessed via custom logic loaded onto the FPGA. The hardware communicates via a standard CAT-5 Ethernet cable with a host PC.

Figure 2.4: Layout of the hardware and the positions of the separate modules featured in the protocol. The PHY is a separately packaged ASIC on the board and represents the physical layer. The MAC represents the data link layer and is an ASIC embedded into the FPGA. It is connected with the PHY via the GMII interface. The ARQ module performs data link and transport layer functionality. The adapter connects the MAC with the ARQ and performs some network layer functionality. After the ARQ resides the upper layer which is implemented according to the used scenario

2.3 The Software

The software used to communicate with the FPGA is presented in [**?**]. It implements the transport layer and uses standard interfaces provided by the operating system to access the lower layers. The writing of the software was not part of this thesis instead already existing slightly modified code was used. The software provided several binaries that were useful for testing. It was possible to send an infinite amount of random data continuously to the hardware to achieve maximum throughput over an extended period of time. Furthermore, another binary could take a file as argument and send it to the hardware. This was used for a loopback test.

3 Design and Implementation

Both the MAC and ARQ modules were already available as tested and implemented modules. The primary task during this thesis was to design and implement an adapter module that would handle the communication between the MAC and the ARQ. The purpose of the network layer is to interface both the MAC and the ARQ to allow the flow of data between both layers. As previously noted, the transmitting and receiving side of ethernet operate in two separate clock domains independently in duplex mode. This means that the network layer also consists of two submodules each dealing only with the receiving or transmitting side, respectively. They are built to operate within their own clockdomain with as little crossdomain communication as possible. From here on, both modules together will be referred to as "the network layer", the receiving module is called *rx_link* and the transmitting module is called *tx_link*.

3.1 rx_link

The rx_link handles incoming packets, receiving them from the MAC and writing them into a buffer guided by the ARQ. A filter is implemented so that only packets from a certain MAC-Address and with a special Type/Length field will be passed on to the ARQ. Furthermore, the rx_link makes use of the CRC check provided by the MAC and drops packets which have been marked as corrupted without disturbing the ARQ. The rx_link is designed to not buffer or delay the incoming packets but to negotiate with the ARQ how to handle a packet while it is being received.

3.1.1 Interfaces

Since the rx_link resides between two modules which have each a fixed interface, the ports of the module are dictated by the signals defined in these interfaces. This section briefly describes the receiving side interfaces of the MAC and the ARQ target, for more information see [?] and [?].

3.1.1.1 MAC RX interface

Figure 3.1 shows the interface provided by the MAC for incoming packets.

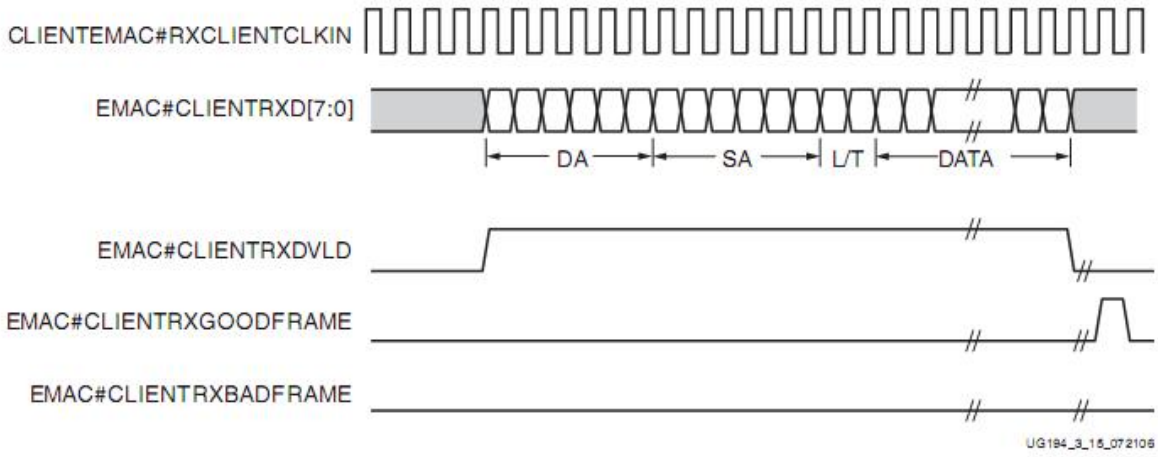

Figure 3.1: Waveform demonstrating the interface between the MAC and the rx_link for a packet that has passed the CRC check. The EMAC#CLIENTRXD signal is a 8 bit wide bus that outputs the data, the EMAC#CLIENTRXDVLD is high when there is a packet on wire and the signals EMAC#CLIENTRXGOODFRAME and EMAC#CLIENTRXBADFRAME mark whether the packet passed the crc check. Figure taken from [**?**]

The important thing to note is that the interface is completely driven by the MAC. This means that the MAC will expect that the rx_link is always ready to receive a packet. Furthermore, the CRC check is performed after the reception of the packet, thus the rx_link needs to go into a waiting state after finishing writing a packet before it can close the connection to the ARQ.

3.1.1.2 ARQ-target RX interface

Figure 3.2 shows the interface specified for the ARQ-target module.

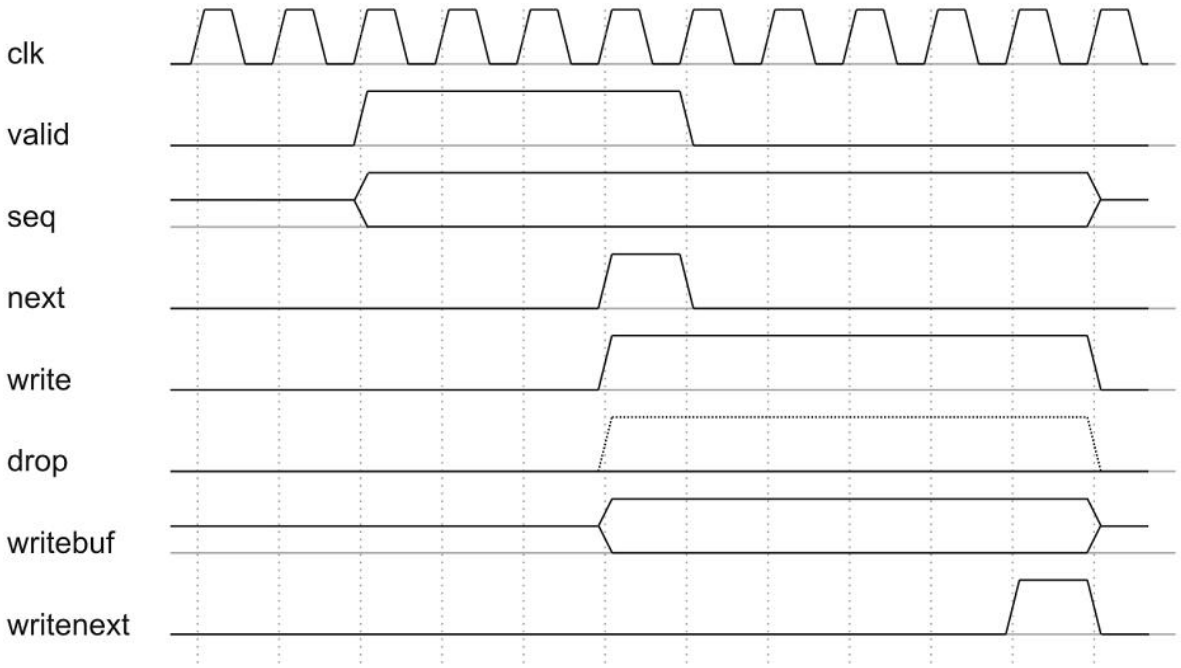

Figure 3.2: Waveform demonstrating a packet request to the ARQ-target module. The rx_link signals an incoming packet with the *valid* signal also presenting the extracted SEQ number as an integer. The target will then assert the *next* flag indicating whether the packet should be written to the buffer or dropped via the *write* and *drop* signals. The *writebuf* integer signal is connected to the rx_buffer selecting the corresponding slot for the rx_link to write into. Once the rx_link is done writing into the rx_buffer it asserts *writenext* signaling that the packet has been successfully stored. Figure taken from [**?**]

The protocol technically allows for the ARQ-target to take several cycles after a request to respond with a write/drop instruction. However, the rx_link will decide whether to send a request to the ARQ based on the seqv field which is the last byte of the packet header. Thus, if for some reason there is no response from the ARQ in the next cycle the rx_link will drop the packet to avoid packet corruption. This is currently more of a theoretical scenario though, as the current ARQ implementation always returns an instruction the next cycle after a request but still requires a fail-safe in the implementation.

3.1.2 The Implementation

The main part of the rx_link is a Finite-State Machine that triggers the sampling processes for the headerdata and also negotiates communication with the ARQ-target. Since the packets all have a fixed length that is known at compile-time, the most obvious solution was to

implement a counter which denotes the position of the data within the packet and is started once an incoming packet is detected. This way, the FSM only has to listen for the counter to decide when to perform a certain operation. The FSM is not directly responsible for the sampling of the header, rather activating helper processes which do the actual sampling and filtering. After notifying the ARQ that there is a valid packet and receiving permission to store it, the rx_FSM writes the packet into the buffer using the counter to calculate the address and offsetting it by 17 Bytes thus ensuring that the first payload byte is written into address 0 of the bufferslot.

3.1.3 Description of the rx_FSM

The state diagram for the FSM is depicted in figure 3.3. The rx_FSM is reset into the idle state. When the MAC asserts the data_valid signal indicating an incoming packet, the rx_fsm activates the counter and transitions into the sample_header state where it activates the helper process that samples the first 14 Bytes of the packet. The FSM idly remains in that state until the 14th Byte of the packet [1] and then transitions into sampling the SEQ number by again activating a helper process. Although the SEQ numbers do not exceed 256 this does not necessarily have to be tha case since the SEQ and ACK fields of the packet can be more than one byte wide. The transition into sampling the ACK nubmers is done in the same fashion as the SEQ numbers. While sampling the ACK numbers the helper process sampling the ethernet header has decided whether to drop the packet because it didn't match the filter. Hence, the packet might be dropped by the Network and the rx_FSM goes into a sleep state where it waits until the packet reception is over to return into the idle state. Note that if this happens the ARQ does not get any notice that there was an incoming packet at all. However, if the ethernet header matched the filter, the rx_FSM transitions into the *wait_for_write* state where it decides whether to notify the target that there is incoming data or simply pass the received ACK number to the ARQ-master on the transmitting side because this was a ACK-only packet without carrying actual payload based on the content of the seqv field of the packet.

Figure 3.3: The state diagram for the Finite State Machine of the rx_link. Pictured are the names of the states, their output and the transition conditions. Note that some states have a continuous output like the writing state, others only do something once upon transition and then stay idle until transitioning into the next state. This is the case for the sampling states which sample the header into several flipflop arrays to perform filtering and presenting the SEQ and ACK numbers to the ARQ.

[1] rx_counter = 13 counting from 0

3.2 tx_link

The transmitting side of the Network compiles packets and sends them via request from the ARQ to the MAC. Essentially, the tx_link works as a multiplexer that passes data from different sources to the input databus of the MAC guided by the instructions from the ARQ thus building a packet that is transmitted to the host.

3.2.1 tx interface of the ARQ

Fig 3.4 shows the waveforms of the signals that perform a request for sending a packet.

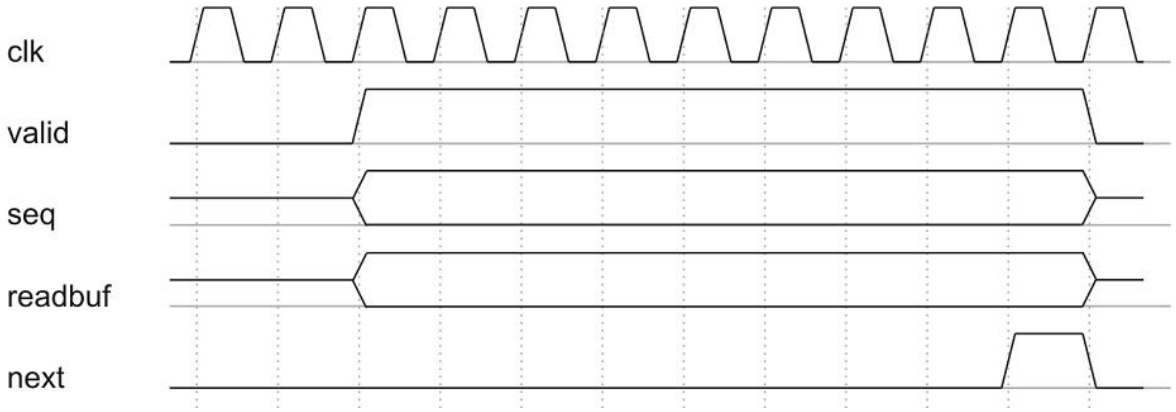

Figure 3.4: Waveform showing a sending request from the ARQ to the Network. The ARQ asserts the valid signal indicating that there is a packet to send and also providing the corresponding SEQ number and selecting the correct packet in the buffer with the *readbuf* signal. The ARQ then waits until the tx_link sends the packet and closes the session by asserting *next* for a clock cycle. Picture taken from [**?**]

Conveniently, the ARQ does not set any timeframe in which the packet must be transmitted, but simply waits until the sending acknowledgement before continuing with the next packet. This is important because the MAC is not always ready to accept a packet to send since the wire might be busy.

3.2.2 tx interface of the MAC

Figure 3.5 depicts the interface for the MAC to send a packet to the host.

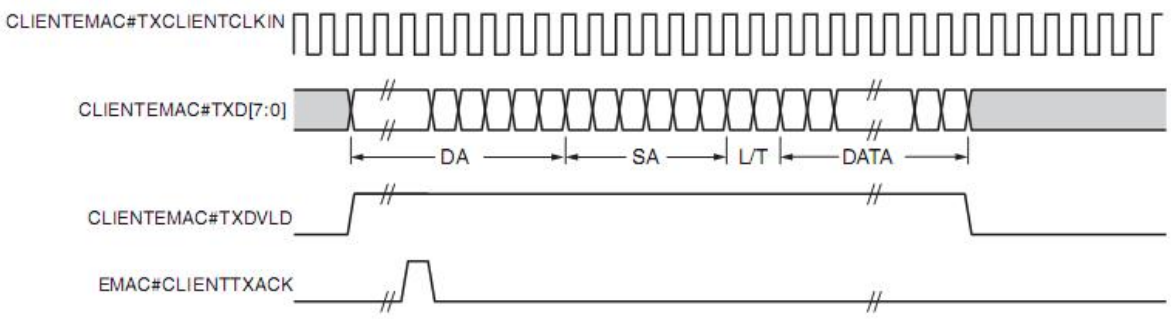

Figure 3.5: Waveform showing a sending request to the MAC. The Network asserts the TXDVLD signal while presenting the first Byte of the packet at the data input and waits for the MAC to raise the CLIENTTXACK signal indicating that the wire is not busy. After that the Network transmits the packet bytewise into the data input of the MAC, deasserting the TXDVLD when it's done.

3.2.3 Implementation

The tx_link mainly consists of two FSMs: The transmitter FSM interfaces between the ARQ and the MAC managing the control signals. The framebuilder FSM handles the building of the packet, i.e deciding which data to put on the bus to the MAC next based on the information provided by the ARQ modules. Figures 3.6 and 3.7 show the state diagrams for both state machines.

Figure 3.6: The state diagram of the transmitter FSM. The FSM consists mostly of waiting states only performing specific actions once upon transitioning The FSM is reset into the *ready* state, waiting for a request from the ARQ. If a request is made the state machine requests sending a packet to the MAC and transitions into the *wait_for_ack* state where it waits until the MAC acknowledges the request. After that the transmitter FSM will notify the framebuilder FSM that a packet needs to be sent while also activating a counter that denotes the position of data in the packet and transitions into the *write* state holding the connection open until the framebuilder notifies the transmitter that the packet has been sent. The transmitter than transitions through a one cycle delay state to close the session and reset the counter into the *ready* state again.

Figure 3.7: The state diagram of the framebuilder FSM. This state machine acts like a multiplexer that routes data from different sources into a single bus based on the counter that is started by the transmitter FSM. The framebuilder manages several flipflop arrays in which the ethernet header is held as well as the arrays which store the SEQ and ACK numbers provided by the ARQ which have to be transmitted with the packet. The seqv field, which decides whether a packet carries actual data or is only an ACK packet is filled by the framebuilder based on where the request to send a packet came from. The ARQ-master will only attempt to send data packets, while the target module only sends ACK packets. If a simultaneous request from both modules is detected the framebuilder will send a data packet because it also contains the ACK number thus satisfying the request of the target as well

3.3 Additional Features

Besides the implementation of the actual protocol stack additional features were devised and implemented. These features adress the following problems and desirable functionality which arise in a typical use case environment.

Reset
: It is often useful to be able to reset not only the whole system but also parts of it during operation. Even more useful is the ability to do so remotely, i.e without physical access to the device.

packet filtering
: The network layer needs to have some sort of filtering mechanism to discern between the packets meant for the ARQ and other traffic. This is especially of concern if the communication is routed through a LAN, as the Network sees packets from other devices which do not intend to communicate with it. Furthermore, the host typically will use its ethernet connection for all sorts of communication, so even if the Network recognizes that a packet is coming from the correct host, it still needs to make sure that this packet is intended to communicate with the ARQ. Thus, some sort of filtering mechanism needs to be implemented so that the Network can decide which packets are passed to the ARQ because the MAC layer does not do any kind of filtering, instead simply passing all traffic to the Network.

Recognizing valid hosts
: For the filter to work the Network has to know which MAC addresses represent valid hosts for communication. This can be done for example by providing a hardcoded list of allowed MAC addresses but this of course has the downside that a new building/-booting process has to be done to modify valid hosts. Hence, it would be very useful

to provide a means of authentification so that the Network knows which MAC address to accept packets from.

3.3.1 The magic Packet

All these concerns have been solved by introducing a “magic packet” additionally to the normal communication flow. The network layer is set up to listen for a particular Type-/Length field that is different from the one used for normal communication. Also, to provide additional security to make it less likely to incorrectly recognize a magic packet, the first few bytes of the payload are sampled into a special memory array which is checked against a preset codeword. For example, the protocol was tested with the magic packet having the Type/Length field of 0x0F0F (as opposed to 0x8899 for normal communication) and the codeword being the ASCII encoding of the string “RESET” : 0x52_45_53_45_54[2] . Once the Network recognizes such a packet several operations are performed: The ARQ layer is reset for the duration of the packet and the source address from the magic packet is set as the filtering adress for incoming traffic thus ensuring that only packets from that adress will be passed on to the ARQ[3]. Furthermore, the Network sets the source address of the magic packet as the destination address for outgoing packets so that all outgoing traffic is now directed to the sender of the magic packet.

[2] both values are arbitrary, but there is a tradeoff to make for the payload codeword: longer strings will require additional memory which is inefficient because it is rarely needed. On the other hand, short codewords might be considered not reliable enough to ensure that a random packet will be falsely recognized as a reset packet

[3] The filter does not prevent the network layer from receiving and reacting to new magic packets, it only blocks traffic to the ARQ

4 Operation and Testing

After the design was written, extensive RTL simulation was performed to ensure that the interfaces were correctly implemented and the protocol could handle arbitrary packetsizes and windowsizes as intended. Several scenarios were tested that may occur during operation like the reception of a corrupted packet, packets out of window, missing ACK packets etc. to test the stability of the protocol. The simulated design consisted of the MAC, Network and ARQ modules in full duplex mode with some kind of lightweight upper layer to emulate communication between devices. The host was emulated within the testbench stimulating the design with predefined packets and sampling the returned packets into an array for examination. After the simulations were satisfying the design was synhesized and loaded onto the ml_505 board.

4.1 Testing

In this section the methods testing of the protocol are discussed and the measured results are presented. The discussion of the results is done in chapter 5.

When testing the communication, the main goal was trying to achieve maximum bandwidth between the the hardware and the host without taking the delay of the upper layer into account. Ideally, it should be possible to achieve full Gigabit speeds at the network layer, although the net bandwidth at the upper layers will be smaller due to resends and timeouts. The upper layer was thus made "arbitrarily fast", i.e it reads and writes data within 5 cycles which is a negligible delay compared to the several thousand cycles that a typical packet takes to be sent or received. The test scenario was to continuously send random data between the host and the hardware monitoring the achieved bandwidth on the host side.

4.1.1 Changing Packet- and Windowsizes

The protocol was written to support arbitrary packet- and windowsizes, thus part of the testing was to pick several combinations of these and see how the performance changed. An implicit parameter that needs to be adjusted are the two timeoutcounters of the ARQ:

Master timeout

After the ARQ master sends a packet it starts a timeout counter and attempts a resend operation if the counter reaches zero and the master has not yet received the corresponding ACK number. A resend operation should only happen because the packet got lost in on the medium, i.e the target on the other side did not receive it. Thus, the timeout needs to be set to a value where one can be certain that the packet has been lost and needs to be resend. Because the software has response times at the order of milliseconds the timeout can also be set in this time range. The Master timeout does not play a very significant role for the performance as the physical medium proved to be very reliable, i.e the occurence corrupted or lost packets were negligible for the network performancee. This is of course because the medium was simply a point to point connection, packets traveling through a wide network over several switches tend to get lost more often.

Target timeout

The ARQ-target has its own countdown that starts when a packet is received and triggers the transmitting of an ACK packet without payload when it runs out. This timeout is crucial to the achievable throughput because the master will only move its window when it has received an ACK either piggybacked on payload packets or via dedicated ACK packets. Either way, the master should receive the ACK for a sent packet as soon as possible. This can be achieved by setting the target timeout to a low value. The drawback, though, is that ACK packets use up bandwidth which cannot be used to transport data.

The calibration of the timings is non-trivial and highly dependent on upper layers. If the upper layers mostly communicate in a way where data is sent and a response is received some time later, it is advisable to set the target timeout to a relatively low value because the high traffic of acknowledgement packets does not matter since the receiving device probably doesn't send payload anyway. This also matters when small windows are used because in that case the buffers tend to run full which lowers the throughput. Hence, the timings must be calibrated so that almost every packet gets acknowledged as soon as possible to quickly free the buffer.

The testing of the modules was conducted in a way that both the host and the FPGA continuously transmitted data as quickly as possible. Thus, in most cases the target timeout can be relatively large i.e about the time it takes to fill a window, because one can assume that there will always be traffic on the wire carrying acknowledge numbers. However, a timeout that is larger than several thousand clockcycles is bad even for large packet and window sizes because it takes too long to acknowledge packets this way. ACK packets are only 60 Bytes long, so it is feasible to wait only a few hundred cycles before sending an ACK packet because this delays data packets only by a small margin compared to the packet length.

4.2 Results

Figure 4.1 shows the measured total throughput from the host side. Wireshark[1] was used to monitor the traffic on the host side.

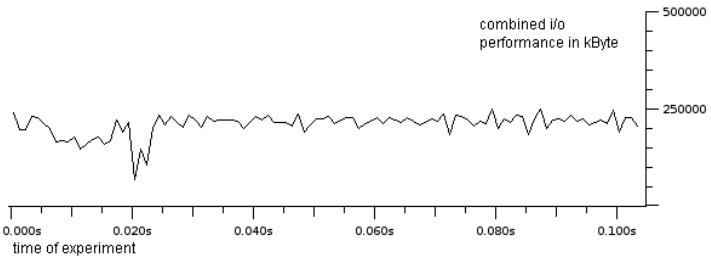

Figure 4.1: Plot of the combined i/o traffic as seen by the host measured via Wireshark. Window size 16 and a packet size of 1500 Bytes were used in this run to test the performance for a likely used case.

On this scale the protocol seems to be stable at about 200 MB/s sometimes peaking at the theoretical limit at 250 MB/s. However, figure 4.2 shows the same experiment at a wider zoom angle.

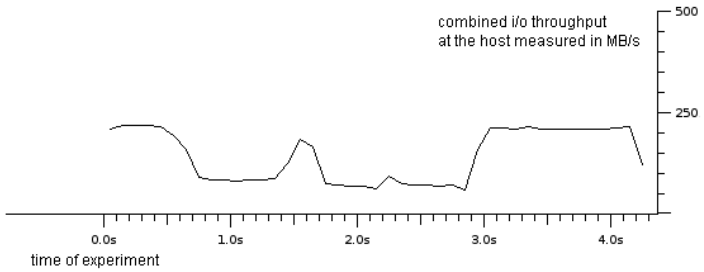

Figure 4.2: Large scale plot of the same experiment as in 4.1.

Notice how the performance drops significantly for a long period of time compared to the timescale of the protocol. The timescale is approximately the time it takes to transmit a window and can be quantified using the following formula:

$$\tau \approx \text{window size} \times \text{packet size} \times 8 \text{ ns} \tag{4.1}$$

[1] a free network protocol analyzer, for more information visit www.wireshark.org

It is interesting to know how the protocol behaves for different window and packet sizes, thus the same experiment was conducted for the following set of parameters:

$$\text{window size} \in \{1;\ 2;\ 8;\ 16;\ 32\} \tag{4.2}$$

$$\text{packet size} \in \{64;\ 1500;\ 3000;\ 5000;\ 7200\}\ [\text{Byte}] \tag{4.3}$$

The resulting in- and output performances are shown in figures 4.3 and 4.4[2]. Note that the measured performance is mainly to provide a frame of reference of what to expect when chosing a set of parameters. The throughput heavily depends on the upper layer and the timings of the protocol. It is therefore best to take these results into account when using the protocol for a specific implementation but it is still highly advisable to fine-tune the timings to achieve maximum performance.

Figure 4.3: Host input performance plotted for several window and packet sizes

Figure 4.4: Host output performance plotted for several window and packet sizes

4.2.1 Interpretation

To achieve maximum performance the ACK timeouts have to be of the order of τ as previously noted. This can always be achieved in the hardware because the packet size, the window size and the timeouts can all be chosen freely and are independent from each other. This is unfortunately not the case in the software implementation. The packet- and the window sizes can be freely chosen but the timeouts cannot be lower than a millisecond. This was a design decision because otherwise the cpu usage would be too heavy and slow the rest of the system down. But that also means that on the software side the performance is best when τ is as large as possible preferably also about a millisecond. However, this requires very large window and packet sizes at Gigabit speeds, i.e packet size of 7200 Bytes and a window size of 16 were the largest feasible parameters because of hardware limitations which correspond to $\tau \approx 7200 \times 16 \times 8ns = 0\ 92ms$.

With that in mind it is not surprising that small window sizes always yield poor performance because τ is small. Additionally, there is the innate problem with small windows because the upper layer is not able to constantly read from or write into the buffer. The transmitting

[2]Plotting of both figures done by Oliver Breitwieser

buffer is almost always full because the ARQ master is waiting for acknowledgements for them and thus the upper layer cannot write the next data. The receiving buffers on the other hand are almost always empty because one can safely assume that the upper layer reads the data faster than it takes for the ARQ to send back ACKs so that more data can be sent.

Larger window sizes will diminish that effect because now more packets are buffered and ready to be sent. Hence, the ARQ only has to wait for ACKs and does not need to additionally wait for the upper layer to first fill the buffer. Still, the problem of the small τ remains, lowering the performance of the protocol for small packetsizes even using large windows. A window size of 16 seems to be sufficient to achieve performance of about 100 MB/s in one direction if the packet size is big enough to produce a large τ. Larger windows do not improve the performance in any significant way, implying a saturation effect.

On the other hand, larger packet sizes should always yield better performances because they raise the value of τ and additionally make it less expensive to send ACK only packets because these are so much smaller than payload packets [3] The measurements show instead declining performance when using packets larger than 3000 Bytes in length. The origin of this unexpected behaviour is not clear although several hypotheses will be presented in chapter 5

4.3 Testing of special configurations

4.3.1 Unidirectionoal communication

Speeds of over 120 MB/s were achieved in half duplex mode, i.e the host continuously sent data and the FPGA only responded with ACK packets (or vice versa). For this, the target timeouts were set to a very low value of approximately the duration of one packet or lower. This lead to a high traffic of ACK packets which does not matter in a half duplex configuration.

4.3.2 Loopback

To check that the ARQ actually supplies the upper layer only with uncorrupted data in the right order a simple loopback module was implemented to act as the upper layer. The module reads a single packet into a memory array and writes it to the ARQ layer afterwards.

[3] consider this: when the target wants to send an ACK packet the master has to wait up to 60 cycles before it can send payload which takes several thousand cycles to do. Thus, the penalty of sending an unnecessary ACK packet is that it takes for example not 3000 but 3060 cycles to send payload. The benefit however, is that ACK packets are sent much quicker if the master doesn't send data on it's own thus greatly improving the throughput.

This means that it takes a single packet four times it's length to be looped back under ideal conditions in this setup. The experiment consisted of sending a large textfile from the host to the FPGA and compare it to the data that is returned. The experiment was successful in the sense that the received data was identical to the sent copy. However, this setup only achieved very low speeds of 10-15 MB/s depending on the timeout settings illustrating the high dependence of the throughput on the design of the upper layer.

5 Discussion

In this chapter the results presented in chapter 4 are discussed, known bugs reported and future work is proposed.

5.1 Discussion of the system

Generally speaking, the system performed very satisfactory. Although it was not possible to achieve full Gigabit speed for every possible configuration, it is possible to achieve total i/o throughput of more than 200 MB/s in full duplex mode. This is a very good result and should prove to be very useful for integrating the protocol in future systems. The adapter module that was written during this thesis to provide a link between the MAC and ARQ layers performed well and the goal to keep the module as generic as possible to easily allow changing parameters was achieved. Because very high throughput over the network was measured it is fair to assume that the adapter does not serve as a bottleneck. The adapter also proved to be stable as it was never observed that the state machine is stuck in a state forcing a reset. Still, there are several issues that need to be addressed.

5.1.1 Instability

As shown in section 4.2, the protocol seems to be unstable at large timescales, exhibiting periods of time where the throughput breaks down to as low as 10% of the maximum speeds and recovering again to almost full Gigabit speed. This is unexpected behaviour and needs to be investigated further because the total bandwidth obviously suffers from such instabilities. It is possible that these effects arise because of secondary effects in the host. For instance, if the software is not running in high-priority mode it is possible that the operating system won't wake the process up for a long time because other tasks need to get CPU time as well.

5.1.2 performance using large packets

As previously noted in section 4.2.1 it is surprising that the performance declines for very large packet sizes. Again, this could be a non-trivial effect stemming from the interaction

between the timings of the host and the hardware. A further contributing factor may be the fact that the time to receive an ACK rises with the packet size in a scenario where few ACK-only packets are transmitted and the sent data packets mostly are acknowledged by received payload packets. It is crucial for the performance to receive acknowledges as soon as possible, hence one should rather choose a smaller packet size and a larger window size because this leads to the same buffer size but improves the overall performance.

5.2 Known bugs

Several problems appeared during implementation and testing and most of them were solved. Sometimes though, the situation arises where the host and the client window desynchronize effectively cutting off transmission on one side. The windows desynchronize when the ARQ master on one side disagrees with the ARQ target on the other side about latest sent ACK. One side will then initiate retransmits waiting for an ACK number that the other side already thought it had sent and already moved its own window to the next position. Hence, the communication on this side ends, but incoming data will still be acknowledged correctly.

This bug can happen at all packet sizes and window sizes and seems to depend upon the timings of the hardware. It happens in the first few seconds after the beginning of an experiment and still occurs after a reload of the configuration file or a reset of the hardware. However, if one changes the timeouts at the hardware side[1] and builds a new configuration file which is loaded onto the FPGA, it is likely for the bug to vanish.

5.2.1 Solutions and workarounds

As previously stated, if one encounters the bug, a new configuration file with slightly different timings needs to be generated [2]. If the bugs appears once than it will always appear for this configuration file. However, if the bug does not appear within the first minute of communication then it will never appear and the used configuration settings can be deemed as “clean”. During testing, experiments of more than 14 hours duration have been conducted with normal operation. The origin of the bug is unclear as of yet and requires further investigation.

[1] larger timeouts tend to rarely produce the bug but do not always eliminate it.
[2] usually changes of a few percent suffice

6 Conclusion and Outlook

The main goal, namely the establishing of a reliable high-speed bi-directional communication link between a host and a FPGA was achieved. Speeds closely approaching the theoretical maximum were reached under certain practically feasible circumstances. The design was heavily tested and performed in an overall very satisfying manner. Furthermore, the design uses up very little of the logic resources of the FPGA[1] which is a huge advantage because it leaves most of the space for the rest of the system. Few problems were encountered during testing and the only serious bug has a fairly easy workaround although it is still important to know why the bug occurs. The custom written modules were designed to be clearly separated modules with easy to understand functions. The modules are very flexible and can be easily modified. Additional useful features were implemented with the "magic packet" mechanism which allows for extensive reset functionality and packet filtering.

6.1 Outlook

6.1.1 Addressing the memory shortage

The most important feature that needs to be yet implemented is the possibility to use other memory than the Virtex5 BlockRAM as the packet buffers. Using the BlockRAM modules was a very convenient way to implement the packet buffers because the synthesis tool uses them automatically to generate the necessary arrays. However, it is a scarce resource because the total memory provided by these modules is 5.3MB. Furthermore, because the size of the individual modules is only 36 kB and the way the buffer is accessed, they have to be inefficiently concantenated to produce a buffer with the desired size. Trials showed that an implementation of the design with a loopback upper layer using packet sizes of 7200 Byte and a window size of 16 packets takes up 44% (2.3MB) of the available BlockRAM modules although the required memory was only $M = (2 \times 7200 \times 16 + 7200) = 237.6$ kB

This inefficient memory usage is unacceptable and needs to be addressed for future implementations. A viable solution would be using instead of the internal BlockRAM modules the external DDR2 RAM on the ml_505 Board since it gives access to 256 MBytes of memory and should be able to meet the requirements to be used as packet buffers.

[1] 2% used slices in the Virtex5 FPGA

6.1.2 Improving the software implementation

Furthermore, the software implementation needs to be modified to adapt to the low timings that the FPGA is capable of. In particular, a way needs to be found to lower the latency of the software significantly below the millisecond timescale because this is one of the suspected reasons which lead to the observed instability in the bandwidth.

A Bibliography

[1] FACETS M7-5. Verify that the layer-2 communication reaches the bandwidth requirements for a multi-wafer system, including the host communication via GBit-Ethernet. FACETS Milestone M7-5, UHEI and TUD, 2009.

[2] Behrouz Forouzan. *TCP/IP Protocol Suite (2nd ed.).* McGraw-Hill., 2003.

[3] Andreas Grübl. *VLSI Implementation of a Spiking Neural Network.* PhD thesis, Ruprecht-Karls-University, Heidelberg, 2007. Document No. HD-KIP 07-10.

[4] Stefan Philipp. *Generic ARQ Protocol in VHDL.* Electronic Vision(s), 2009.

[5] Johannes Schemmel, Andreas Grübl, and Sebastian Millner. Specification of the HICANN microchip. FACETS project internal documentation, 2010.

[6] Gerd Sigmund. *Grundlagen der Vermittlungstechnik.* R. v. Decker, Heidelberg, 1992.

[7] Xilinx. *ML505/ML506/ML507 Evaluation Platform User Guide*, 2009.

[8] Xilinx. *Virtex-5 FPGA Embedded Tri-Mode Ethernet MAC User Guide*, 2009.

[9] Xilinx. *Virtex-5 FPGA User Guide*, 2010.

B Acronyms and Technical Terms

ARQ Automatic Repeat reQuest

MAC Media Access Control

FPGA Field Programmable Gate Array

VHDL VHSIC Hardware Description Language

VHSIC Very-High-Speed Integrated Circuits

SEQ Sequence

ACK Acknowledge

seqv Sequencenumber valid

ASIC Application-Specific Integrated Circuit

OSI Open Systems Interconnection

RAM Random Access Memory

DDR Double Data Rate

Statement of Originality (Erklärung):

I certify that this thesis, and the research to which it refers, are the product of my own work. Any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

Ich versichere, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, 27th February, 2014

.....
(signature)