Department of Physics and Astronomy
University of Heidelberg

Master's Thesis
in Physics
submitted by
Vitali Karasenko
born in Kryvyi Rih, Ukraine

A communication infrastructure for a neuromorphic system

This Master's Thesis has been carried out by Vitali Karasenko at the
Kirchhoff Institute for Physics
Ruprecht-Karls-Universität Heidelberg
under the supervision of
Prof. Dr. Karlheinz Meier

A communication infrastructure for a neuromorphic system

This thesis presents the integration and testing of a generic ARQ protocol into two distinct network architectures which are part of the BrainScaleS HMF. The achieved net bandwidths of up to 117 MB/s for the host link and up to 83 MB/s for the FPGA-HICANN connection are a significant improvement over the existing implementation. Long-term evaluation proved the necessary protocol stability to allow production usage in the HMF. Several discovered shortcomings in the system are discussed and suggestions for their improvements are provided.

Eine Kommunikationsinfrastruktur für neuromorphe Systeme

Diese Arbeit stellt die Integration und Evaluation eines generischen ARQ Protokolls in zwei verschiedene Netzwerkarchitekturen innerhalb des BrainScaleS HMF Systems vor. Die erreichten Nettobandbreiten von bis zu 117 MB/s in der host-Anbindung und bis zu 83 MB/s in der FPGA-HICANN Verbindung stellen eine signifikante Verbesserung gegenüber der vorhandenen Lösung dar. Langzeittests bestätigen die Protokollstabilität und ermöglichen dadurch die aktive Benutzung im HMF System. Mehrere gefundene Mangel werden diskutiert und Optimierungsmöglichkeiten dafür vorgestellt.

"A delayed game is eventually good, but a rushed game is forever bad"

Shigeru Miyamoto

Contents

1 Introduction

As a computational device, the mammalian brain vastly outperforms any artificial construct that humankind was able to come up with until today - not necessarily in terms of raw FLOPS, but rather in the ability to process multi-sensory, multi-modal information (i.e real world data) in real time. Every new result leading towards a systematic understanding of these capabilities will likely yield valuable applications for the modern information-driven society. Apart from the impressive energy efficiency compared to our engineered devices, other intriguing features are the ability to learn, and the capability of compensating for faults or defects during runtime. In mammals, the most salient anatomical structure responsible for the processing and integration of information (colloquially summed up by the rather ambiguous concept of "intelligence") is the neocortex. Biological experiments indicate that the neocortex is a rather homogeneously connected network of some tens of billions of neurons with each neuron having an average of several thousand presynaptic partners.

The neocortex is not a product of systematic engineering: its structure and function are the result of hundreds of millions of years of evolution. While random genetic modifications coupled with natural selection effectively guarantee improvement over many generations, they are the exact opposite of structured design. The fact that nature does not select for what programmers would call "beautiful code" is essentially the obstacle we need to overcome before understanding the complicated structure of the brain in general and the neocortex in particular. Continuous advances in neurophysiological experimental techniques have provided us with a plethora of valuable data over the last couple of decades. However, our current tools do not give us the required resolution and precision to analyze the structure of a large brain volume to a degree that would allow direct replication. Therefore, in addition to advancing the experimental side of neuroscience, we also need tools that allow us the modeling of large neural networks in order to verify hypotheses about how high-level brain functions emerge from low-level network dynamics.

Up until now, the preferred method to model neural networks has been simulation software such as NEST (*Diesmann and Gewaltig*, 2002), executed by various types of von-Neumann machines. The main reasons are the convenient programming interfaces provided for most of these "conventional" architectures and, even more importantly, their rapid and stable performance increase over many decades, as epitomized by Moore's law. Furthermore, the software abstraction from the underlying substrate enables portability and repeatability: the same software can be compiled for various architectures and will reproduce the exact same network behavior[1], regardless of the machine it runs on.

However, these advantages come at a certain cost. Virtualization requires computational overhead, and the general-purpose structure of current CPUs sacrifices power efficiency and execution speed for flexibility. Furthermore, the only reasonable way to parallelize software simulations is to partition the network and run the parts on separate cores while exchanging spiking information over a network. The execution speed of this I/O intensive system is quickly limited – not by the speed of the individual cores but rather by the synchronous, fixed-bandwidth

[1] With the exception of varying implementations of random number generation.

network structure between them that is used in every High Performance Computing (HPC) environment. In general, while simulating large networks on single core machines results in prohibitively large execution times, simulations on HPC-clusters suffer from power efficiency and communication issues.

An alternative approach, first described in the 1980s by *Mead* (1989), is sometimes referred to as *physical modeling*. It is part of what is called today the *neuromorphic computing*[2] subfield in neuroscience. The fundamental change in paradigm is to construct physical representations of neuron models so that their dynamics can be observed instead of numerically calculated. At the moment, the most suitable substrate for this emulation are microelectronic circuits built using Complementary Metal–Oxide–Semiconductor (CMOS) technology. Embedded in a switch matrix on a silicon microchip, they physically represent parts of neurons such as synapses and membranes. The transition from simulation to emulation bears new challenges, but promises benefits that could potentially allow to push beyond the limitations on experiment time and network size imposed by software. Most importantly, assuming that a given network can be mapped onto the neuromorphic hardware, the experiment runtimes are independent from the network size because of the direct representation of each neuron in hardware.

1.1 The BrainScaleS Hybrid Multiscale Facility

The Electronic Visions group at the University of Heidelberg developed several generations of neuromorphic hardware since 2001 (*Schemmel et al.*, 2001). The current installation is called the Hybrid Multiscale Facility (HMF) and was built during the BrainScaleS project (*BrainScaleS*, 2012). A comprehensive overview of the HMF can be found in *Brüderle et al.* (2011), a brief summary is presented in the following. *Brüderle et al.* (2011) also serves as primary citation for the information in this section. The HMF combines neuromorphic components with a cluster of conventional von-Neumann machines that serve as command and control nodes. The neuromorphic part consists of uncut wafers that contain 384 Application Specific Integrated Circuits (ASICs) which are called the High Input Count Analog Neural Network (HICANN). All of the HICANNs on a wafer can be interconnected via a dedicated asynchronous bus structure that is extended beyond the reticle boundaries[3] during a post processing step. A HICANN implements 512 analog circuits called *denmems* that act as building blocks for realising point neurons with a variable number of inputs. These analog neurons can be connected using a digital synapse array with an 8-bit weight resolution. Additional digital circuitry is used for configuration and communication to the outside world. The whole wafer is placed on a Printed Circuit Board (PCB) that provides the contact pins for power, clock and communication signals. On the other side of the PCB, there are 12 smaller boards with each an FPGA and the so-called Digital Network Chip (DNC) on them. The DNC bundles the communication links of 8 HICANNs and routes them to the FPGA which can be accessed over switched Gigabit Ethernet by the host.

[2] Meade et al. did not subdivide the neuromorphic computing field. However, recent developments such as the Manchester SpiNNaker system (*Furber et al.*, 2012), which follows a different philosophy, motivated the introduction of physical modeling as a distinct category

[3] A reticle is the largest unique area that can be lithographically produced with a given technology and is replicated on the wafer. In the 180nm technology used by BrainScaleS, a single reticle contains 8 HICANNs

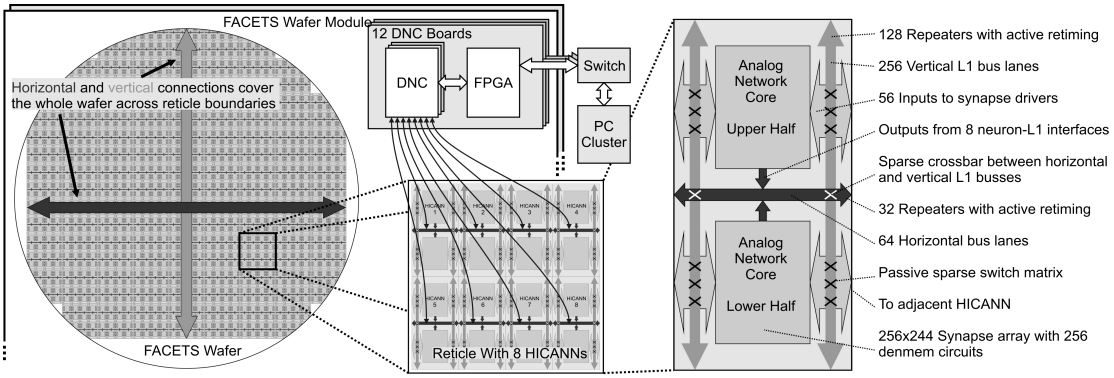

Figure 1.1: Structure of the HMF with the individual subsystems. The PC cluster controls HICANNs on a wafer indirectly by communicating with FPGAs over a switched Ethernet network. The FPGA then routes configuration and spike data to up to 4 DNC microchips which can access up to 8 HICANNs each. Figure taken from *Brüderle et al.* (2011)

One major advantage of the neuromorphic part of the HMF is the very high acceleration factor compared to biology of between 10^3 and 10^5. Since biological neuron activity is typically in the 10 Hz range, this acceleration poses high demands on the I/O capabilities of the HMF. Therefore, the communication infrastructure plays a key role in the performance and usability of the system.

1.1.1 Communicating with a HICANN

Figure 1.1 shows that there is no direct line of digital communication between the host and a HICANN. The FPGA, together with the DNC, serves as an intermediate stage to buffer spike and configuration data. From a communication point of view, both the host Ethernet link as well as the FPGA-DNC-HICANN connection are unreliable because they both require signals to travel relatively long distances between chips and are thus susceptible to signal noise and other disturbances. However, it is necessary to provide a reliable communication path between host and HICANNs so that data is guaranteed to be eventually exchanged without loss in the correct order. The challenge to provide a lossless communication link between clients over a lossy medium is very common in all of computer science and many solutions have been developed. The most popular idea is a protocol that buffers data exchanged between clients and maintains state using additional meta data that tracks whether data has been successfully exchanged. If data loss is detected, a resend from the buffer is initiated. Perhaps the best known protocol suite that implements this is TCP/IP (*Forouzan*, 2003), used virtually everywhere on the internet. The strong suit of TCP/IP is that it is a proven and established technology with native support in every modern operating system. However, hardware implementations of TCP/IP are prohibitively resource intensive and were not used in the HICANN and FPGA chips. Instead, a generic and light-weight custom ARQ-style protocol with sliding window was developed in-house to be used for lossless communication in the HMF.

1.1.2 The ARQ

The Automatic Repeat reQuest (ARQ) protocol was designed by Stefan Philipp in 2009 (*Philipp*, 2008a) and is a parametrizable go-back-N protocol with selective repeat. It defines a ring buffer segmented in N packets that can be of arbitrary size and format. The ARQ requires meta data that is sent together with the payload and modifies state between the two clients.

Field	Description
SEQ	Sequence number of the current packet.
ACK	Sequence number of the last received packet.
seqv	Flag that distinguishes whether packet carries payload

Table 1.1: The ARQ meta data fields and their meaning for the protocol. The size and format of the fields is configurable and not directly relevant since they are converted to integers and treated as such by the ARQ

With the fields described in Table 1.1 and the ring buffer the ARQ implements a sliding window strategy for maintaining throughput while ensuring link integrity. The ARQ sends and resends a packet in its window until it receives a response packet with an ACK number equal or higher. The ARQ will then assume that the packet has been successfully transmitted and moves its window to accept new data in its place. The selective-repeat functionality refers to the ability of the ARQ to move the window by more than one at a time. For example, if packet number 0 to 7 have been sent by the master but packet 5 was lost in the network, only this single packet needs to be resent and the target will move its window to packet number 7 since this is now a contiguous data block.

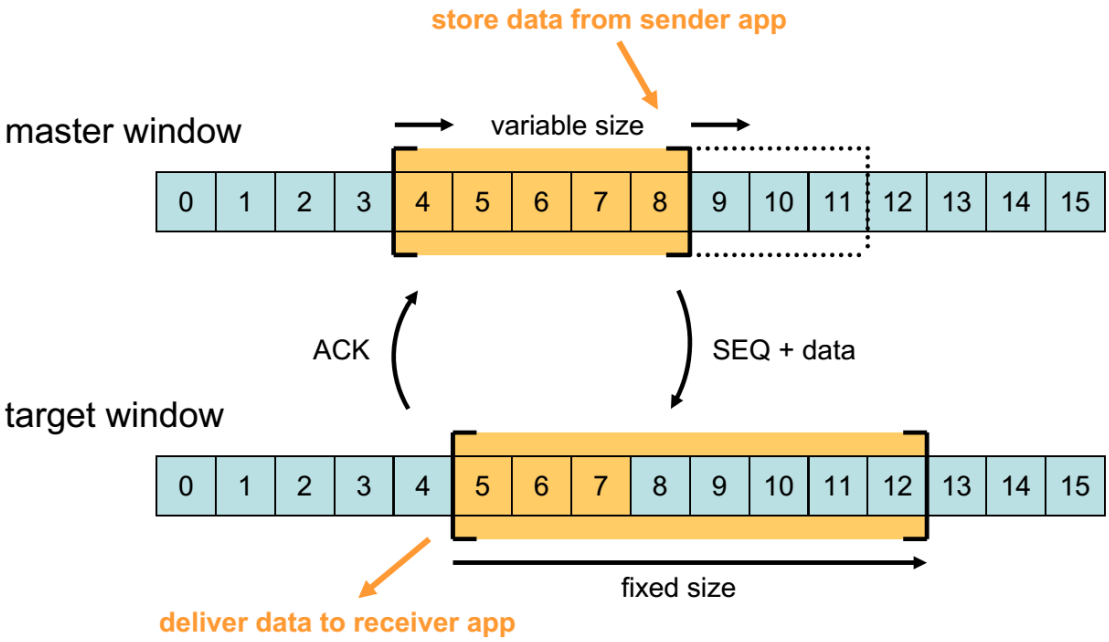

Figure 1.2: ARQ sliding window algorithm. The master accepts packets until the ring buffer is filled and tries to send them to the target over the unreliable network. The target always accepts a fixed amount of packets starting with the sequence number of the last packet that has been read out to the application + 1. The master is updated with rising ACK numbers about which packets have been received so that it can move its window accordingly. Figure taken from *Philipp* (2008b)

As long as the provided buffers are large enough, the ARQ is capable of maintaining wire speed even with packet losses. This can be expressed in the so-called Bandwidth-delay product which is discussed in more detail in subsection 2.8.2. Furthermore, the generic implementation makes the ARQ completely agnostic of the exact nature of the network which makes it an ideal candidate to use in the HMF system which employs two very different link architectures.

1.2 Outline

The main aspect of the thesis was to design and implement HDL code in the FPGA that allows ARQ-secured communication to the host as well as to the HICANNs. Because of the different network structures, the ARQ is integrated into two completely different modules which are tested and evaluated separately. Chapter 2 contains the documentation about the host ARQ implementation in the FPGA, chapter 3 documents the HICANN-ARQ-link side in the FPGA. While the testing and evaluation of the two modules is described in their respective chapters, chapter 4 provides discussion and outlook for them both. Finally, chapter 5 compares the two modules with the legacy system and provides estimates over the overall achieved improvements.

2 The host ARQ

This chapter describes the implementation of the ARQ protocol over a switched Ethernet link between the FPGA and the host. Similar work has been done in *Karasenko* (2011). While the two projects share no code except for the ARQ modules, they share many basic concepts such as encapsulating the receiving and sending circuitry in submodules, using Finite State Machines (FSMs) for parsing and building protocol headers and taking special consideration into introducing as little delay as possible into the communication while also maximising the throughput.

2.1 Packet structure

Ethernet is a packet-based communication technology, this means that a series of data words with start and end signals are treated as belonging logically together. In the Open Systems Interconnection (OSI)-model (ISO/IEC 7498-1:1994), the payload data is prepended with additional header data that is processed and stripped while it travels through the layers which is called framing.

Of course, the term "payload data" is relative, since any header data that belongs to a higher OSI layer looks like payload to a deeper one. Keeping that in mind, we define payload data as any words that are in the packet after the ARQ header fields. The ARQ is acting as Transport Layer according to the OSI model connecting to UDP/IP as the deeper Network Layer. Figure 2.1 shows the structure of a host-ARQ packet. For an overview of the Ethernet/IP/UDP headers see *Braden* (1989).

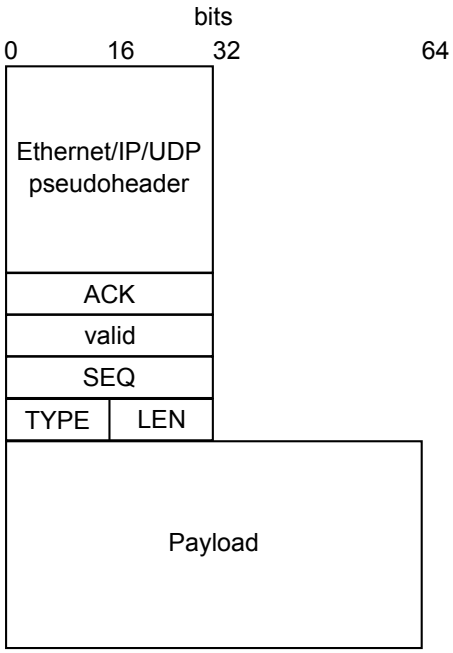

Figure 2.1: Structure and alignment of the data packets as seen by the host-ARQ.

The host-ARQ will only deal with the IP and PORT numbers from the UDP/IP header with the other fields being either ignored or processed by deeper layers. For the meaning of the SEQ, ACK and valid fields see subsection 1.1.2. Although the valid field actually represents only a single bit it was more convenient to reserve 32 bits for it to preserve data alignment. The TYPE/LEN field was introduced in *Ehrlich et al.* (2013) for carrying information about the amount and type of the payload words in the packet and will be treated as part of the ARQ header. Note that while the ARQ header words are aligned to 32 bits, it was convenient to align the payload to 64 bits since this is the word size the Application Layer (AL) in the FPGA operates on. Consequently, the LEN field counts in 64 bit words. Due to buffer limitations in the UDP module the maximum total packet size is a standard 1500 Bytes long Ethernet frame, subtracting from that 14 Bytes of Ethernet header, 40 Bytes IP/UDP header and 16 Bytes ARQ header yields the maximum supported payload size as 178 64-bit words.

2.2 Module overview

The top level module *host_arq_top* implements the desired Transport Layer functionality. Figure 2.2 shows a block schematic with the neighboring modules, an overview of the top level ports is given in section 2.7 . The module serves as a bridge between the AL and the UDP-layer providing full duplex data transfers which look as fifo-like as possible to the AL. It runs at a single clock frequency of 125MHz synchronously to the UDP and AL modules. For the detailed interfaces and timing diagrams see sections 2.4.4 and 2.5.4. Sending and receiving data is handled separately in two submodules called rx_link and tx_link with minimal non-blocking communication between them to ensure independent functionality and thus full-duplex capability.

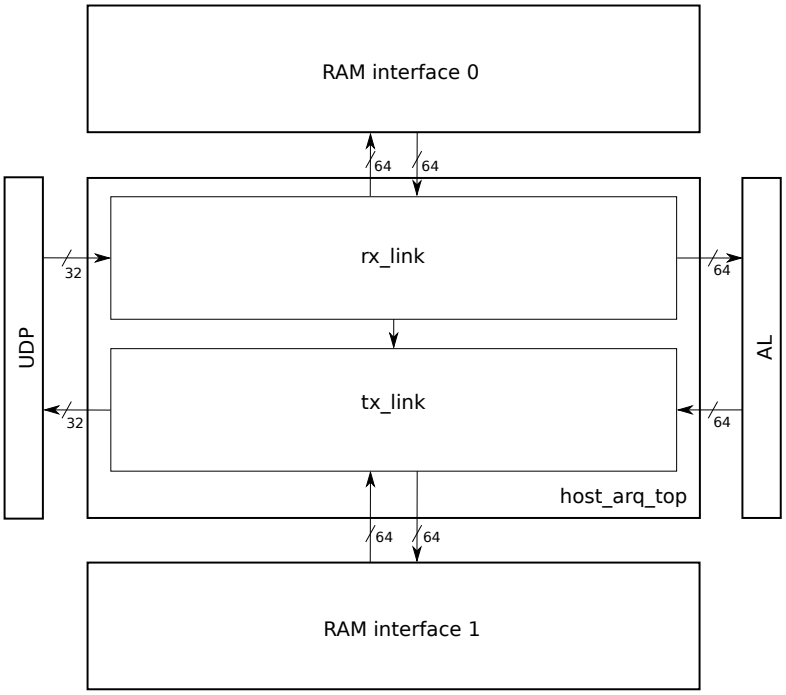

Figure 2.2: Top level view of the *host_arq_top* module with the immediately connecting layers. The OSI-layer data flow is from left to right while the RAM interfaces provide access to the ARQ frame buffers which are stored in DDR2 RAM. The port widths apply to the data only, control signals to the UDP/AL/RAM modules are not shown

2.3 DDR2 DRAM as ARQ packet buffer

As is discussed in subsection 1.1.2, the ARQ manages some memory which contains the data that has yet to be sent to the host or the AL. The minimum size of these buffers is determined by the bandwidth-delay product which is discussed in subsection 2.8.2. Larger buffer sizes are beneficial for the performance because they relax the scheduling requirements of the software, i.e how often it has to wake up to send or receive data without impacting the throughput. While it is possible to use the BlockRAM resources of the FPGA (xil, 2009) to implement the packet buffers as has been done for example in *Karasenko* (2011), it proved to be an overall inferior solution. The obvious benefit of this approach is the simplicity of the implementation since at the Register Transfer Level (RTL) a BlockRAM looks like a dual ported array of configurable width and depth that can be accessed within a single clock cycle. However, the total BlockRAM memory on a typical FPGA is only a few megabytes in size, and concatenating large amounts of BlockRAM primitives to form a larger array poses high strain on the Place And Route (PAR) step of the design synthesis. Measurements like shown in Figure 2.10 suggest desirable buffer sizes in the low megabyte range which would be very expensive to realize in BlockRAMs. Furthermore, because the sending and receiving side need their own window the total buffer requirements are doubled. Since the entire FPGA design often makes uses of BlockRAMs for different purposes it is therefore impractical to use them for large ARQ buffers.

However, the FPGA board also features 256MB of on-board DDR2 RAM which is capable of being used as packet buffers. On the FPGA, the access to the memory is facilitated by the Xilinx Multi Port Memory Core (MPMC) (xil, 2011) which provides up to six independent read/write ports with internal arbitration between them. To further simplify the interface, a small adapter module was written which presents three FIFOs to host_arq_top as a single access point to the RAM that are called *read*, *write* and *command*.

Writing data to memory consists of pushing some number of words into the *write* FIFO and pushing a command into the *command* FIFO afterwards. To read from memory a corresponding command has to be pushed into the *command* FIFO. Monitoring the read_fifo_empty flag gives notice when the data has arrived and can be read by popping the FIFO. The command word specifies whether the request is to read or write, how many words are requested and what the start address of the first word is. The formatting of the command words is given in subsection 2.7.2. The rx_link and tx_link modules will try to access the memory using bursts of data as much as possible because this maximises the performance of the DRAM. Because of this they need to know the largest amount of words that can be accessed with a single command, this RAM_MAX_BURSTSIZE parameter can change depending on the actual configuration of the memory. See subsection 2.7.2 for more information.

2.4 rx_link

The rx_link submodule encapsulates the receiving side of the protocol. Figure 2.3 shows a block schematic that sketches the flow of the payload, control and protocol data. The main paradigm followed in the design is to ensure quick protocol handling between reception of data and passing it to the AL. The total delay introduced by the rx_link between the UDP and AL is completely dominated by the access delays of the DRAM. This could only be made possible by storing the TYPE/LEN field in a small but fast dedicated BlockRAM array which makes reading data to the AL much faster than it would be if the TYPE/LEN field would also be stored in DRAM. This pre fetching is further elaborated on in subsection 2.4.2. The timing diagrams for the UDP and AL interfaces can be found in subsection 2.4.4.

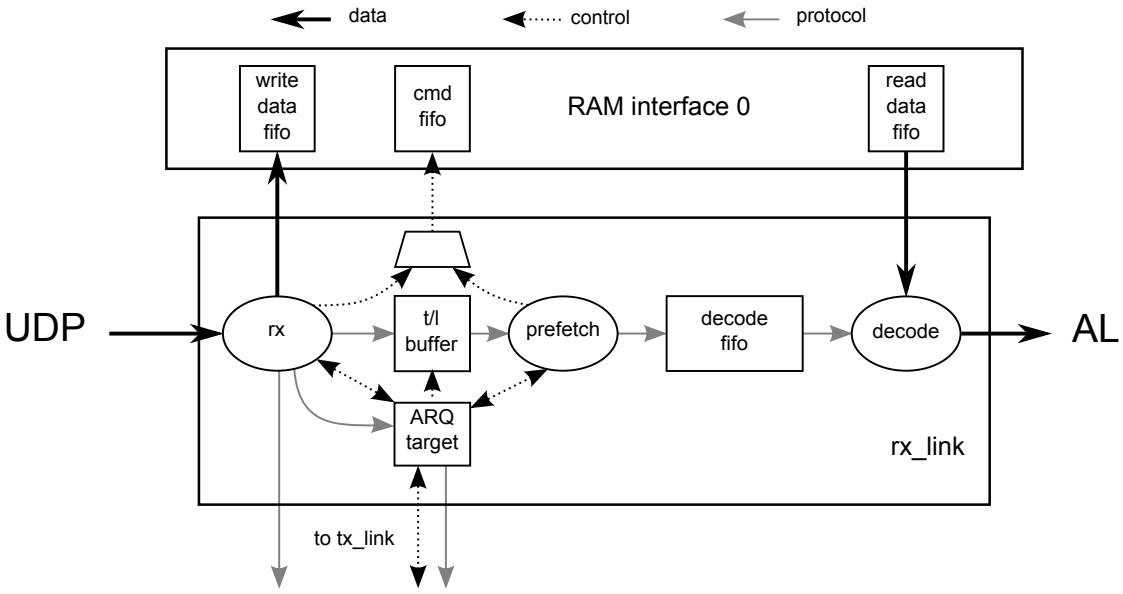

Figure 2.3: Block schematic of the rx_link submodule.

2.4.1 The rx FSM

The rx FSM filters the incoming UDP packets, communicates with the ARQ target and writes valid packets to memory. It also delivers received ACKs to the ARQ master module residing in the tx_link submodule

Receiving UDP packets Incoming UDP frames are filtered for a valid port/IP combination that can be set at runtime. After passing this check the rx FSM will assume that the frame is a valid ARQ frame and parse the next four 32 bit words according to their meaning in the protocol as defined in Figure 2.1.

Parsing the ARQ header The ACK field is sent to the ARQ master residing in the tx_link submodule by raising the *master_rx_valid* signal. The SEQ/valid pair is used to make a request to the ARQ target whether to write the payload to memory. If the frame is to be dropped the rx FSM goes into a sleep state and waits for the end of frame signal after which it resets into the ready state. Otherwise, the ARQ target generates an address for the packet within the window and raises the *target_rx_write* flag to announce that the frame is to be written to memory.

The TYPE/LEN buffer The TYPE/LEN field is written to a BlockRAM array using the *target_rx_bufpos* address. This TYPE/LEN - buffer is configured as a dual ported array which holds WINDOWSIZE 32 bit words. Since a single dual port BlockRAM primitive is 36kbit in size it can hold up to 1125 such words. If larger windows are needed, several BlockRAMs can be concatenated to provide the necessary memory space. One reason for using a dedicated buffer for the TYPE/LEN field instead of writing it to the DDR2 RAM is to preserve memory alignment since the payload is aligned to 64 bits. A much more important reason however is that the TYPE/LEN - buffer now allows to pre fetch data from memory. This is explained in more detail in subsection 2.4.2.

Writing the payload to memory Now that the payload is to be written to memory, each two 32 bit words are grouped into a single 64 bit word using temporary registers and pushed into the *write_data_fifo*. After the number of pushed words has reached RAM_MAX_BURSTSIZE a write command is issued for that amount of words with an address that is determined using the base address given by the ARQ target and an offset generated by the amount of words already written to memory during the current packet that are not part of the current burst. When the end-of-frame signal is raised the rx FSM issues one last write command for the remaining words in the current burst if necessary, notifies the ARQ target that the packet has been successfully written to memory and goes back into the ready state.

This behaviour has some implications for the software. For one thing, no sanity check is performed on whether the LEN field actually agrees with the number of payload words in the packet, and only the amount specified there will be read out from memory to the AL. This means that one could get corrupted or old data at the AL if the LEN field is larger than the payload in that frame. The host has also to make sure that the payload is correctly aligned to 64 bit since otherwise the last word will not be written to memory. The total payload size per packet can also be no larger that the PAYLOAD_SIZE parameter, violating this constraint will mean either overflows into neighboring buffer slots or wrap-arounds which corrupt the currently receiving payload.

Handling abort conditions Assuming a correct packet formatting there is still the possibility that the DRAM might stall during frame reception. The two conditions are:

1. *write_fifo* is full when a new word needs to be pushed.
2. *command_fifo* is full while trying to issue a command for the current burst.

If at least one of these conditions is met the reception of the frame is aborted and it has to be resent by the host. The rx FSM then goes into a flushing state to ensure that every word that has already been pushed into the *write_fifo* has been accounted for with a corresponding command. The occurrence of the stalls is dependent on the DRAM bandwidth which is estimated in xil (2011) to be much higher than the UDP bandwidth which makes such events unlikely. Nonetheless, it is important to ensure the proper handling of such states to ensure compatibility with many different setups.

2.4.2 The pre-fetch FSM

When there is valid data to be read to the AL, the ARQ target notifies the pre-fetch FSM and generates the corresponding base address. The pre-fetch FSM then first uses this address to read the corresponding TYPE/LEN field from the TYPE/LEN - buffer and saves the LEN field in a working register while also pushing the current TYPE/LEN field into the *decode_fifo*. After these preparations the main objective is to keep the *read_data_fifo* as full as possible. The pre-fetch FSM knows the depth of the *read_data_fifo* and how many words are in it at most[1] since it monitors how much data it requested and how much data has been popped out of the *read_data_fifo* by the AL. Whenever there is at least RAM_MAX_BURSTSIZE space left in the *read_data_fifo* the pre-fetch FSM issues a read command for that many words. The arbitration between read and write commands has been set to a fixed priority towards write commands because it is more important to write the packet quickly to memory than to get the data to the AL.

[1] The uncertainty stems from the fact that there might be some data that is being fetched from the DRAM but is not pushed into the fifo yet

2.4.3 The decoder FSM

The main purpose of this state machine is to translate the packet format shown in figure 2.1 into the individual TYPE - PAYLOAD WORD pairs for the AL as shown in timing diagram 2.5. This has the advantage of abstracting protocol details like packet sizes away from the AL which simplifies the interface. Since the *decoder_fifo* contains the ordered TYPE/LEN fields of the data stream the decoder FSM knows that the next LEN words that appear in the *read_ data_fifo* have the type TYPE and need to be announced as such to the AL.

2.4.4 Interfaces

UDP rx side The start and end of a packet is announced via separate flags with data being marked as valid by raising a valid flag. Note that in the current implementation of the UDP layer the frame will be read out as a 32 bit word every two clock cycles which yields an effective bandwidth of 2Gbit/s.

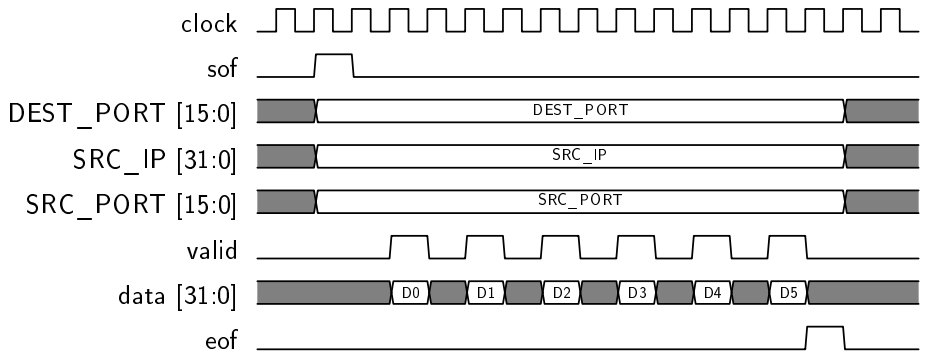

Figure 2.4: Timing diagram depicting the rx interface of the UDP module. The shown transmission is consistent with an ARQ packet that carries a single 64 - bit payload word

Application Layer read interface The AL-read interface has been designed to look as fifo-like as possible. The ARQ implements full flow control which means that the AL can stall for arbitrary periods of time between popping payload without loss of data. The decoder FSM as described in 2.4.3 takes care of pairing the payload words with their corresponding type. Figure 2.5 shows some examples for reading data to the AL. Signals with the suffix *_i denote input signals from the rx_link's point of view, signals marked with *_o are output signals driven by rx_link.

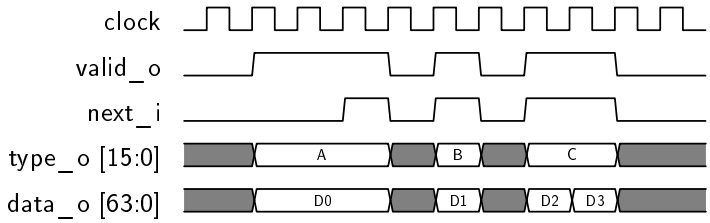

Figure 2.5: Timing diagram showing several examples of reading data to the AL from rx_link's point of view. First, a single word transaction with type A where the AL responds after a delay. Second, a single word transaction with type B where the AL acknowledges the data in the same clock period. Lastly, a two-word back-to-back transfer with type C

2.5 tx_link

The tx_link submodule encapsulates the sending side of the ARQ protocol. It stores the data passed to it by the AL, handles the framing and implements the transport layer functionality to prevent data loss or corruption at the receiving host. A block diagram is shown in figure 2.6, the timing diagrams for the AL and UDP interfaces can be found in subsection 2.5.4. A certain symmetry to the rx_link is apparent, in fact the module behaves largely the same with the direction of the data flow now going from the AL to the UDP module.

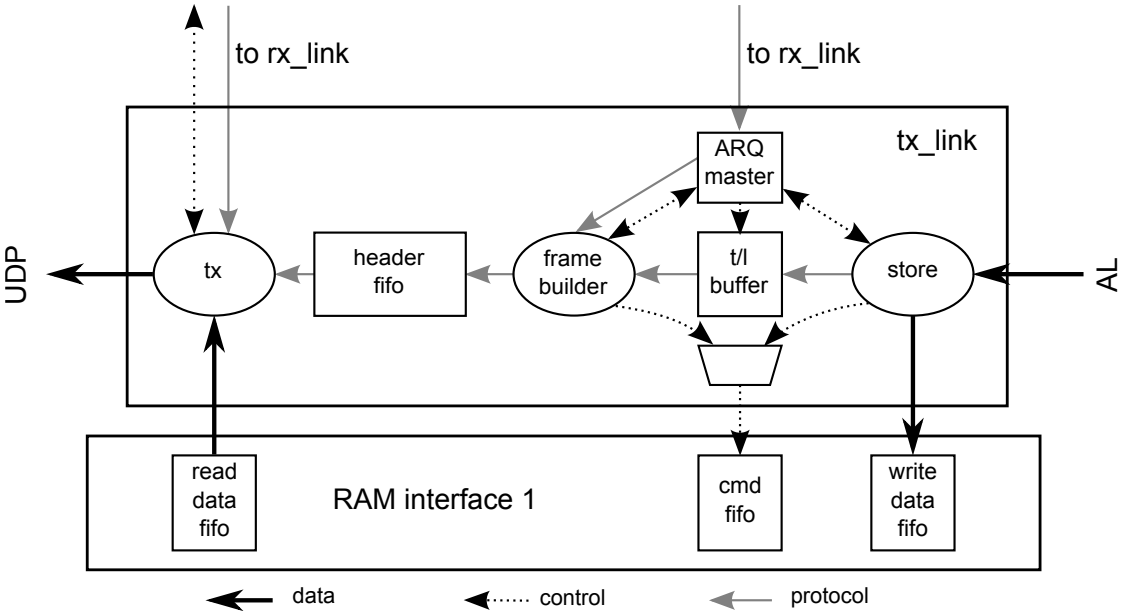

Figure 2.6: Block diagram of the tx_link submodule

2.5.1 The store FSM

When the AL attempts to send data to the host it interacts with the store FSM via the fifo-like interface shown in Figure 2.7. The store FSM has three main objectives:

1. Interfacing the ARQ master module to determine whether there is space left in window for pushing new data.
2. Managing the access to DRAM, i.e keeping track of the words pushed into the *write_data_fifo* and issuing write commands when necessary. A new ARQ frame has to be requested when the current one holds PAYLOAD_SIZE words
3. Making sure that each ARQ frame contains only data that belongs to the same type to adhere to the packet specification described in section 2.1. If the type changes a new ARQ frame has to be requested.

The AL flush timeout From the AL's point of view, once it has pushed data into the tx_link it can safely assume that this data will reach the host in finite time during normal operation. To maximise the available UDP packet sizes the tx_link will normally wait until either the number of pushed words has reached the maximum frame size or a changing of the type before it closes the current frame which is then sent away. This however means that at the end of an experiment there might be data stuck in the ARQ that never gets sent because the corresponding buffer never gets closed. This behaviour is solved by introducing a configurable timeout that is internal to the tx_link. The timer starts counting down whenever a new frame is opened and is reset every time a new word has been written to it. Once the timer reaches zero the tx_link will close the ARQ frame even if it has not reached the maximum frame size. It makes sense setting the flush timeout to a rather large value since otherwise some unnecessary fragmentation might occur when there are some pauses between large bursts of data.

Generating the TYPE/LEN header When an ARQ frame is ready to be closed for whatever reason the store FSM will write the active type and number of words for it in a BlockRAM based *t/l buffer* to be used as the TYPE/LEN field of the frame later. The reason to do so is the same as for the *t/l buffer* residing in the rx_link: It allows pre-fetching the data and also improves alignment and thus access performance of the DRAM.

2.5.2 The frame builder FSM

When the ARQ master wants to send a frame it activates the frame builder FSM which will push the corresponding SEQ number and TYPE/LEN field in the header fifo and then request the payload data from DRAM. As in the pre-fetch stage of the rx_link the frame builder FSM keeps track of the fill status of the *read_data_fifo* and avoids requesting too much data that would potentially not be able to fit in it. Read commands from the frame builder are prioritized over write commands from the store FSM to ensure a steady data flow to the UDP layer.

2.5.3 The tx FSM

This state machine interfaces the UDP modules for sending out payload and ACK-only frames to the host. The timing diagram for sending a frame to the UDP is shown in Figure 2.8. Sending payload frames is triggered when there is data in the *header_fifo*, ack-only frames can be triggered by the ARQ target residing in the rx_link. The ARQ target is notified every time a frame is sent out so that it can reset its ACK Timeout.

2.5.4 Interfaces

The AL write interface Writing data to the tx_link is very similar to the read interface as discussed earlier. A data word is presented to the tx_link by the AL together with its type by raising the *al_write_valid* flag. The tx_link will acknowledge the reception of that word by raising the *al_write_next* flag which completes the handshake. This handshake is necessary because it depends on many factors whether a data word can be stored in memory at a particular time as is described in subsection 2.5.1. Figure 2.7 shows some example writes.

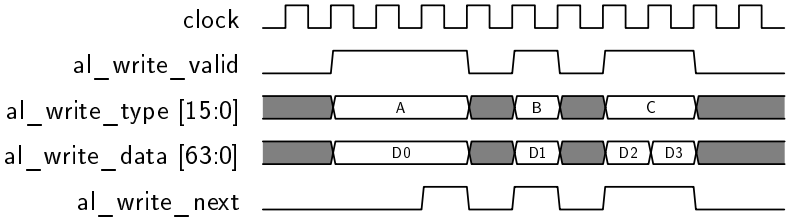

Figure 2.7: Timing diagram showing several examples of writing data to the tx_link. Depending on the situation data can be written within a single clock, back to back or after a certain delay.

The UDP tx interface Transmitting a frame through the UDP requires two handshakes where the UDP acknowledges both with the *ready* flag. The first handshake represents the request to send a frame which is indicated through raising the *sof* flag. After the UDP responded by raising the *ready* flag the frame data can now be pushed into the UDP with the *push* flag. To end a frame the tx_link raises the *sof* flag after the last data word has been pushed. Note that the current implementation of the UDP module buffers the frame entirely before it is sent to the Ethernet MAC, thus there are no timing requirements on the length of the pause between two pushes of data. This is useful because while the ARQ header data is quickly available to be pushed into the UDP there is a significant delay before the payload data is returned from memory. However, the tx FSM can be modified to adhere to more strict timings which would require a maximum delay between two pushes if necessary. An example timing diagram is shown in Figure 2.8.

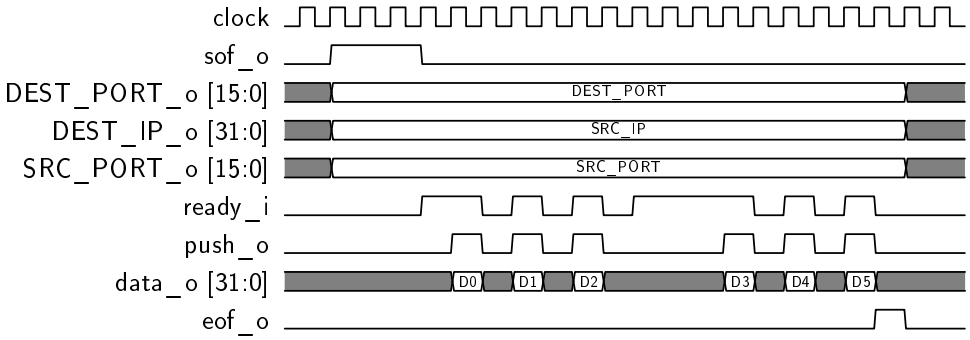

Figure 2.8: Timing diagram demonstrating the interface between the tx_link and the UDP module.

2.6 The resetter module

Also part of the host_arq_top module is a small state machine that listens for a certain UDP port and a magic payload word. If it finds such a word the resetter will use the rest of the frame to set the protocol timings and also the host IP for the ARQ. This enables the configuration of the ARQ at runtime to a certain degree. The resetter can also optionally reset the ARQ as the name implies, however resetting the ARQ is not required to change the protocol timings. Figure 2.9 shows the payload structure for a resetter packet. The host IP is extracted from the UPD/IP header which implies that each ARQ host is responsible for resetting its own FPGA counterpart during operation.

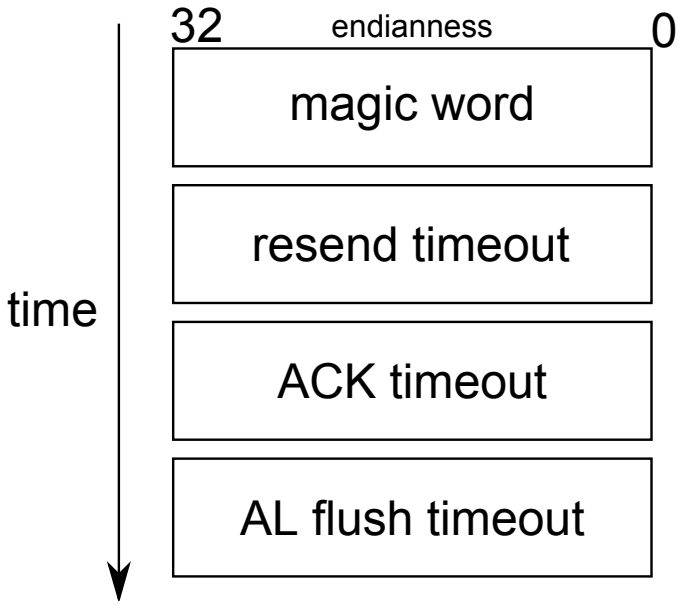

Figure 2.9: Payload structure for a resetter packet. The values are used to set their respective registers as described in section 2.7

2.7 Design parameters and port descriptions

The tables in this section describe the parameters and signals of the host_arq_top module. The signals have been all typed to be either std_logic or std_logic_vector except the design parameters to make integration with verilog modules easier although some of them are internally cast to ranged integer types.

2.7.1 Design Parameters

Table 2.1: The design parameters of the host_arq_top module

Name	Default Value	Allowed Values	Description
DEBUG	false	true, false	Enables the synthesis of debug registers. It is recommended to only use this option when explicitly needed to keep the FPGA resource utilization down.
USE_OLD_UDP_INTERFACE	true	true, false	Switches between the slightly different interfaces of the UDP modules found in the Virtex5 and Kintex7 designs
RESETTER_PORT	"AFFE"	16 bit value	The resetter module described in section 2.6 listens for UDP packets that match this value.
ADDR_WIDTH	25	INT	Used to determine the width of ram_command_fifo_data
SIZE_WIDTH	6	INT	Used to determine the width of ram_command_fifo_data
RAM_MAX_BURSTSIZE	32	INT	Largest number of 64 bit words that fit in a single read/write burst of the DRAM.
TX_FIFO_DEPTH	512	INT	This value is used by the frame builder FSM in the tx_link to gauge the fill status of the *tx_read_fifo* If different fifos are used this value has to be adjusted to avoid possible data loss.

Table 2.1 – *Continued from previous page*

Name	Default Value	Allowed Values	Description
RX_FIFO_-DEPTH	512	INT	This value is used by the pre-fetcher FSM in the rx_link to gauge the fill status of the *rx_read_-fifo* If different fifos are used this value has to be adjusted to avoid possible data loss.
FPGA_ARQ_-PORT	"04D2"	16 bit value	The rx_link will only accept data from this port
AL_FLUSH_-TIMEOUT_MAX	2^{16}	INT	Determines the width of the register that holds the AL flush timeout value that is set by the resetter. Note that since the value received by the resetter is 32 bits wide overflows are possible.
MASTER_TIME-OUT_CYCLES_-MAX	2^{16}	INT	Determines the width of the resend timeout counter in the ARQ master.
TARGET_TIME-OUT_CYCLES_-MAX	2^{16}	INT	Determines the width of the **ACK** timeout counter in the ARQ target.
PAYLOAD_SIZE	176	INT	Maximum amount of 64 bit words that fit in a single ARQ frame
WINDOW_SIZE	512	2^n	ARQ window size
SEQ_SIZE	2^{16}	INT	ARQ sequence size. Should be at least double of the WINDOW_SIZE parameter

2.7.2 DRAM I/O signals

This interface is present twice in the host_arq_top module, each one for the rx- and tx side. The signals have a corresponding prefix in the name to signify where they belong to in the module declaration.

Table 2.2: DRAM I/O signals

Signal Name	Direction	Width	Description
Command FIFO Signals			
command_fifo_data	Output	32	Highest bit is **read-not-write**, followed by size and address. The width of the subfields is specified by **SIZE_WIDTH** and **ADDR_WIDTH** respectively.
command_fifo_full	Input	1	Fifo will not accept data when high
command_fifo_push	Output	1	Active high
Write FIFO Signals			
write_fifo_data	Output	64	
write_fifo_full	Input	1	Fifo will not accept data when high
write_fifo_push	Output	1	Active high
Read FIFO Signals			
read_fifo_data	Input	64	Data is valid in the same clock cycle as read_fifo_empty going low.
read_fifo_empty	Input	1	read_fifo_empty is valid when low
write_fifo_pop	Output	1	Active high

2.7.3 UDP interface I/O signals

Table 2.3: Description of ports that interface the UDP module

Signal Name	Direction	Width	Description
rx interface signals			
rx_udp_source_port	Input	16	UDP source port, valid during the whole frame
rx_udp_source_ip	Input	32	UDP source IP, valid during the whole frame
rx_udp_sof	Input	1	start-of-frame signal, high for one clock cycle
rx_udp_eof	Input	1	end-of-frame signal, high for one clock cycle
rx_udp_din	Input	32	UDP payload data, valid when rx_udp_din_valid is high
rx_udp_din_valid	Input	1	Active high
tx interface signals			

Table 2.3 – *Continued from previous page*

Signal Name	Direction	Width	Description
tx_udp_source_port	Output	16	UDP source port for outgoing packet, should be kept stable during frame
tx_udp_dest_port	Output	16	UDP destination port for outgoing packet, should be kept stable during frame
tx_udp_dest_IP	Output	32	UDP destination IP for outgoing packet, should be kept stable during frame
tx_udp_sof	Output	1	start-of-frame signal, requests a new transmission to the UDP. Has to be kept high until sof_ack or rdy is raised
tx_udp_sof_ack	Input	1	Only used when **USE_OLD_UDP_INTERFACE** is set, indicates that frame is ready for receiving payload
tx_udp_eof	Output	1	end-of-frame signal, high for one clock cycle, closes the frame
tx_udp_rdy	Input	1	UDP accepts new payload word when high. Is not used when **USE_OLD_UDP_INTERFACE** is set, internal logic has to emulate that signal for the tx_link
tx_udp_din_valid	Output	1	Pushes new payload word in UDP when high
tx_udp_din	Output	32	UDP payload data

2.7.4 AL interface I/O signals

Table 2.4: Description of ports that interface the AL

Signal Name	Direction	Width	Description
read interface signals			
al_read_data	Output	64	Application Layer payload word
al_read_type	Output	16	Type is always valid together with the payload word
al_read_valid	Output	1	Data and type is valid in the same clock cycle that this signal is high
al_read_next	Input	1	Acknowledges the reception of active word and type when high
write interface signals			
al_write_data	Input	64	Application Layer payload word, must be kept valid as long as al_write_valid is high
al_write_type	Input	16	Type must be always valid together with the payload word
al_write_valid	Input	1	Active high
al_write_next	Output	1	Completes the handshake, data word has been committed to ARQ.

2.7.5 Debug ports

These ports only carry data when the **DEBUG** parameter is set. All of them are 64 bit wide counters that count certain events which were deemed valuable for debugging the design.

Name	Description
rx_cnt	Counts the total amount of received ARQ frames since reset.
rx_abort_cnt	Counts the amount of times a receiving frame had to be aborted due to memory stalls since reset.
rx_drop_count	Counts the amount of times a frame was dropped by the ARQ since reset.

Table 2.5: Debug registers in host_arq_top

2.8 Testing and evaluation

Apart from verification in simulation, extensive testing has been done in hardware between the FPGA and the host. The used software ARQ implementation in the host was a slightly modified

version of the work done in *Schilling* (2010). The FPGA design was a stripped down version of the BrainScaleS design that contained only the modules necessary for testing. In particular, the used BrainScaleS modules were the Ethernet UDP/IP core and the DDR2 on-board memory interface that the host_arq module connects to. The measurements and plots in this section were done in collaboration with Eric Müller who provided assistance with the software side of the experiments.

2.8.1 The test_al module

To replace the BrainScaleS AL a small test_al module was written that served as termination point for the ARQ connection in the FPGA. The test_al has several features:

- Data with a certain type is looped back by feeding it into an internal loopback FIFO that connects to the write interface of the host ARQ.
- The test_al is able to generate dummy data with either sequentially rising or pseudo random content. This data has a different type than looped back data.
- Additionally, stats can be generated and periodically sent under a special type back to the host. The stats contain the debug registers listed in Table 2.5 and also internal registers that count the amount of transmitted words to and from the host ARQ.
- The test_al recognizes received words with a reserved type as a configuration packet that enables and configures the dummy data as well as stats transmissions.
- Incoming words that are neither of the loopback nor of the configuration type are popped from the host ARQ but not processed further.

2.8.2 Measuring the uni-directional throughput

The inherent parameters of the ARQ protocol that describe its behaviour are the timings and the buffer size. To gauge the influence of these parameters on the protocol performance it is best to first measure the half duplex throughput, i.e there is only payload transmission on one side of the link while the back channel carries only the ACKs. Since Ethernet is a very reliable protocol it can be assumed that the performance impact due to packet loss should be minimal under normal circumstances. This has been experimentally verified and thus the resend timer has been set to some large value such as to not affect normal operation but still make the recovery of lost data possible. The main measurements for the half duplex performance were done with the direction host $\rightarrow$ FPGA for the sake of convenience.

Buffer size and ACK timing In data communications, the bandwidth-delay product is commonly used to relate the two network properties with each other. The notion is that if at a certain link speed a certain delay is to be expected, there has to be a total buffer capacity of at least

$$\text{bits-in-flight} \geq \text{bandwidth} \cdot \text{delay} \tag{2.1}$$

Since the ARQ is a packet based protocol the total bits-in-flight capacity can be expressed as

$$\text{bits-in-flight} = \text{PAYLOAD\_SIZE} \cdot \text{WINDOW\_SIZE} \tag{2.2}$$

Note that the definition of the bits-in-flight parameter depends on the definition of what constitutes a single packet. Here, the bits-in-flight mean the total amount of payload words for the AL that the ARQ needs to have in its buffer. This ignores the protocol overhead that is part of a single frame but is for the moment irrelevant for this discussion.

Protocol delay The delay is the protocol round-trip-time, i.e the time it takes for the host to receive an ACK for a packet it sent earlier. This parameter has two main contributions:

- The ACK timeout set in the hardware determines how long the ARQ target will wait after the reception of a packet before it initiates an ACK-only response. In a full-duplex configuration most of the ACKs will be carried piggyback by payload, thus the ACK timeout should be carefully tuned to not waste bandwidth.
- The host implements the ARQ in software which will be inherently slower than real time. Thus, part of the effective delay is the time it takes for the software to wake up and notice that a new ACK has arrived.

Sweeping the WINDOW_SIZE Figure 2.10 shows a selection of sweeps over a range of different windows and ACK timeouts. Note that it takes a total buffer size of at least $16 \cdot 176 \cdot 8 = 22.528$ Bytes for the protocol to reliably achieve the maximum performance but only at a rather small ACK-timeout range. It also makes sense that whenever the ACK-timeout gets larger than the maximum value dictated by the bandwidth-delay product the throughput quickly breaks down because the network capacity is now smaller than the rate at which the host sees new ACKs. It might however be surprising that the throughput also goes down at very small ACK-delays. This is explained by the fact that the host-CPU is interrupt starved in that range because every new ACK packet causes a context switch which introduces an additional delay and takes time away from actual protocol processing. This is a special property of software-based ARQ and has to be kept in mind during operation. From everything discussed above it seems reasonable to always set the ACK-delay as large as possible without going over the bandwidth-delay threshold. Also, a WINDOW_SIZE of 512 seems to produce satisfactory performance with the best ACK delays ranging in the milliseconds without consuming too many FPGA resources.[2]

[2] The current FPGA ARQ implementation requires a WINDOW_SIZE-bit large array that is very expensive to synthesize. For large windows this array dominates the total resource consumption for the whole host ARQ module

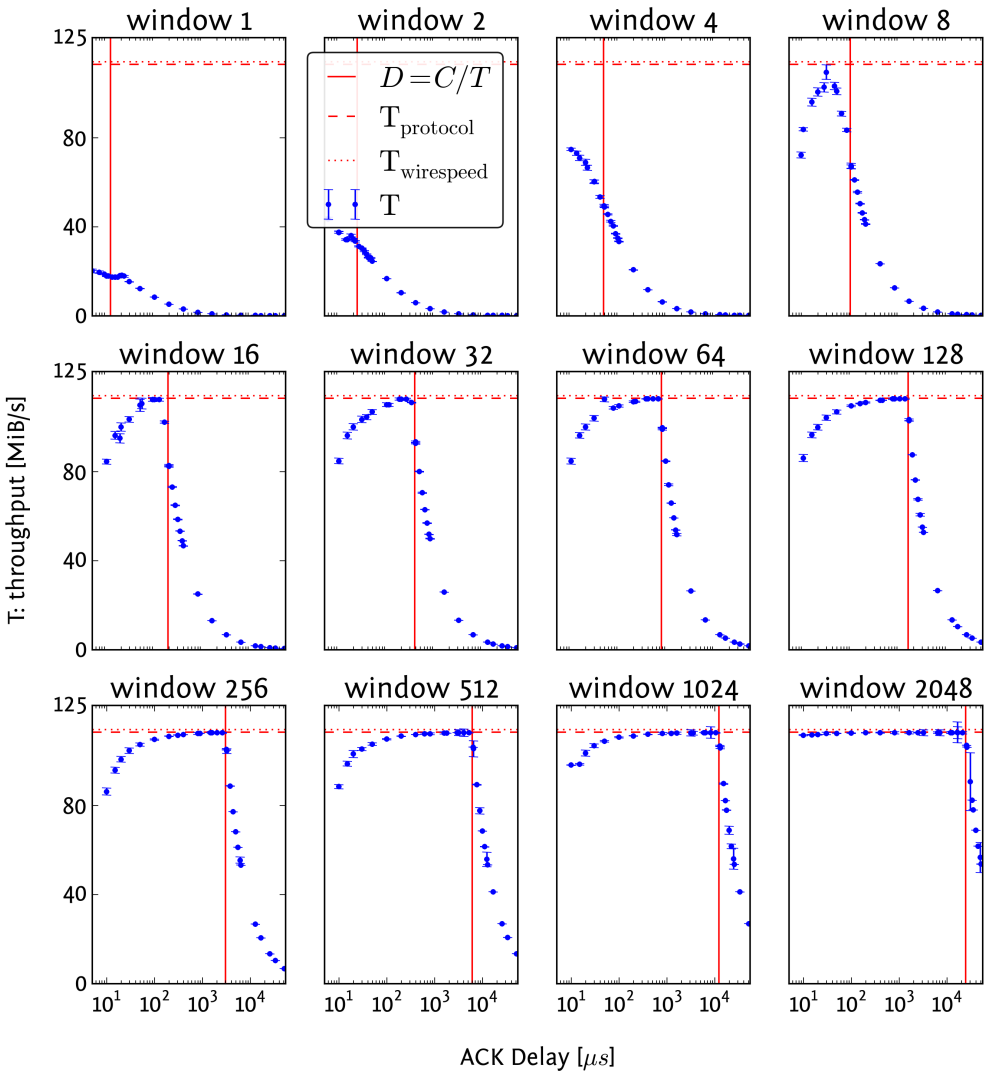

Figure 2.10: Measuring the half duplex throughput from host to FPGA with varying WINDOW_SIZEs and ACK-timeouts at constant PAYLOAD_SIZE = 176*8 Bytes. The vertical line marks the delay $\frac{\text{bits-in-flight}}{1\text{Gbit}}$

PAYLOAD_SIZE vs. WINDOW_SIZE The bandwidth-delay product only takes into account the total buffer size to gauge the protocol performance with no regard of how fragmentation into packets and window slots might affect it. Figure 2.11 shows the performance of the ARQ at a constant buffer size but varying PAYLOAD_SIZE/WINDOW_SIZE combinations. For each data point several runs with different ACK timings have been done choosing the ones with the best performance to factor this influence out of the measurement.

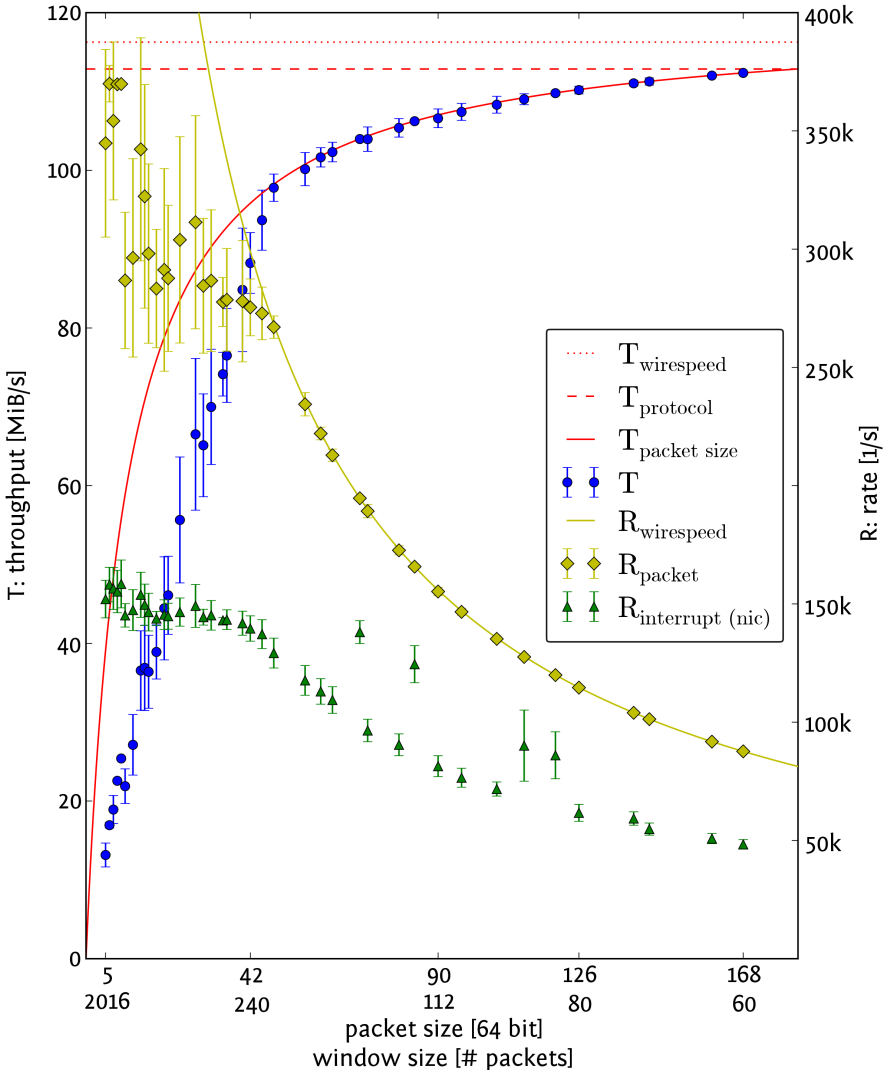

Figure 2.11: Host-to-FPGA half-duplex ARQ throughput for different {PAYLOAD_SIZE, WINDOW_SIZE} pairs with the product kept constant. Also shown are the measured packet rates as well as the software interrupts by the NIC

There are two main observations to be made from Figure 2.11:

1. The packet rate massively impacts the throughput. This is obviously because at very low PAYLOAD_SIZEs the protocol overhead[3] is no longer negligible and massively drives down the performance.

2. The host CPU will reach an interrupt threshold for very high packet rates which also negatively impacts the throughput because the software is unable to send packets at wire

[3] such as Ethernet Inter-Frame-Gaps, Headers and CRC

speed anymore.[4]

2.8.3 Protocol symmetry and full-duplex performance

It is reasonable to expect that it should be possible to achieve the same maximum throughput between the host and the FPGA when using symmetric timings and buffer sizes. However, measurements such as shown in Figure 2.12 show that the throughput from the FPGA to the host is limited to about 80 MB/s. Investigations revealed an inefficient processing of the outgoing ARQ packets in the UDP module which makes it impossible to send packets back to back to the host thus wasting bandwidth. Barring this phenomenon the ARQ achieves symmetric performance and also full-duplex capability since the total throughput is the sum of the throughputs over the individual links[5]. Figure 2.13 shows how the host ARQ connection is able to sustain the highest possible throughput on each side in full duplex mode over many windows.

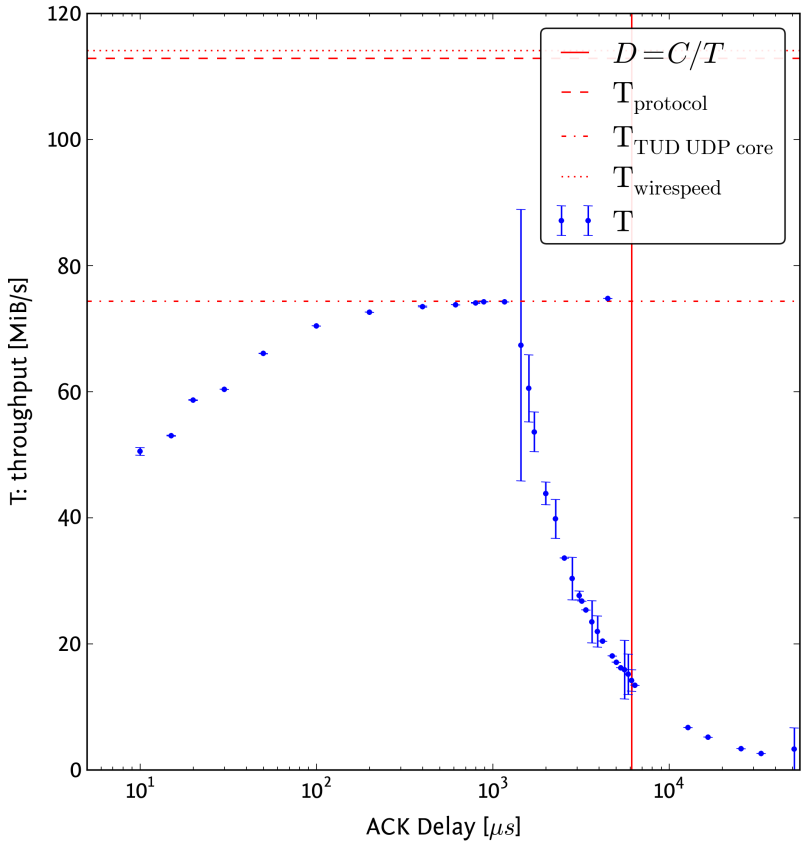

Figure 2.12: Half-duplex ARQ throughput from FPGA to host as a function of software ACK delays. The horizontal line drawn at ≈ 78 MiB/s is the limit at which the UDP core in the FPGA is capable of transmitting

[4] An interesting side note is that apparently the maximum interrupt rate of the tested system is about 150 kHz

[5] Obviously, to achieve optimal performance in full-duplex mode the rate of ACK-only packets has to be kept to a minimum, which is another reason why the ACK-timeout should be rather high

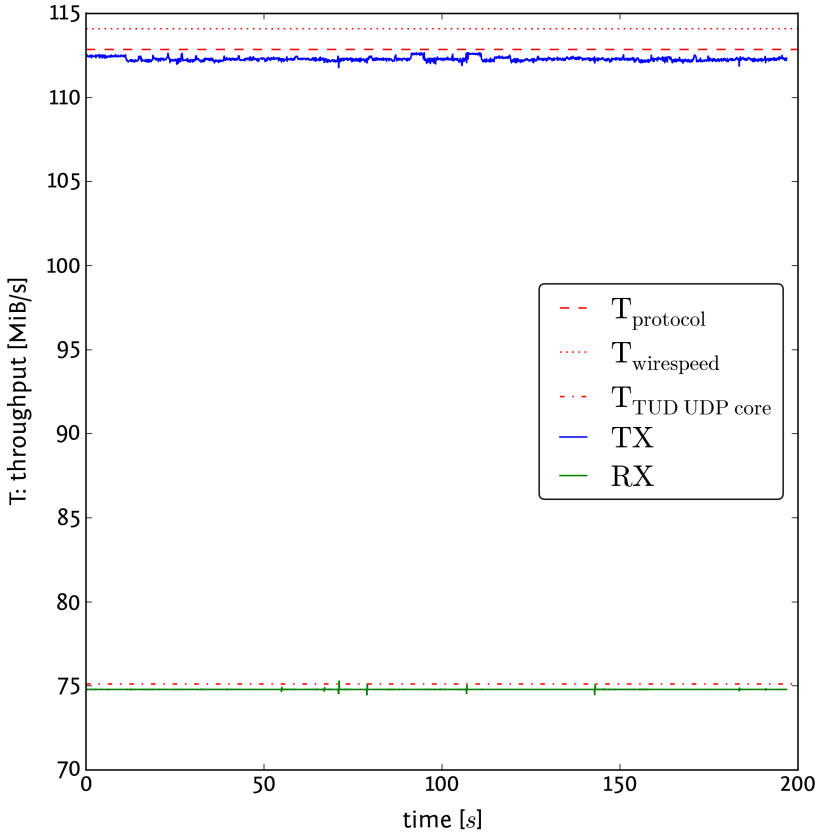

Figure 2.13: Full duplex ARQ throughput between host and FPGA.

2.8.4 Payload integrity and reliability

The host ARQ has undergone extensive stress testing looping back several Terabytes of random payload and comparing it with the original data. The tests showed neither corruption nor unexplained protocol breakdowns which suggests that the system is ready for integration in the BrainScaleS FPGA project and production usage.

3 The HICANN ARQ

This chapter describes how a HICANN connects to the FPGA and introduces the HICANN ARQ module that serves as an end point for HICANN configuration data in the FPGA.

3.1 The HICANN - FPGA connection

3.1.1 The HICANN high-speed link

The HICANN has two main layers of I/O to the outside world. The Layer-1 link connects neighboring HICANNs on a wafer and carries only spike event data. The high-speed link uses a differential 1-bit-plus-clock signal running at 1 Ghz to connect the HICANN to the so-called DNC which is an ASIC built at the Dresden University of Technology. A fast serial link instead of a slower parallel link was chosen because it reduces the pin count and also because it reduces the link complexity since a parallel link needs to make sure that the individual bits align properly which is difficult to achieve over large distances. Directly connecting a HICANN to the FPGA was not possible in the original design because at the time the FPGAs were not able to drive an I/O pin at Gigahertz speeds which made the development of the DNC necessary. The DNC connects to the FPGA via a 16 bits wide parallel link serving as a hub for up to 8 HICANNs.[1].

The high-speed link carries both spike events and HICANN configuration data with a static prioritisation towards spike data. This is done because the spike data has to preserve the inter-spike-intervals as much as possible since they are an important property of neuromorphic computation[2]. While all high-speed link data carries CRC bits to detect corruption, the spike data does not use the ARQ for reliability because the buffering as well as resending will skew the timings between spikes. The downside is of course that the spike trains will experience irrecoverable data loss which poses high demands on the high-speed link stability. On the other hand, HICANN configuration data has to be secured against data loss via the ARQ because an incomplete HICANN configuration will put the chip in an undefined state making reliable experiments impossible. Furthermore, since the sender will aggressively drop configuration data in favor of spike data, the probability of losing configuration data is independent of the actual link stability which makes the necessity of some recovery mechanism even more evident.

3.1.2 HICANN configuration data

The HICANN has a variety of values that can be programmed during runtime. While these values are stored in different ways on the chip[3], they can all be accessed using a unified programming model. Configuration values on the HICANN can be accessed with a {read/write, address, value} command that is encapsulated in a 64 bit word together with the ARQ header. See Table 3.1

[1] Note that the DNC handles each HICANN link independently and that it is not possible for two HICANNs to communicate directly over the DNC.
[2] The sender will even go as far as drop configuration data in favor of spikes if necessary
[3] E.g in a floating-gate transistor, entries in a SRAM or flip-flops

for the configuration packet structure, further information about configuring a HICANN can be found in *Schemmel et al.* (2010)

name	bit no.	size	description
Tag	63:62	2	tag-id
Seq Valid	61	1	**valid**
Seq	60:55	6	**SEQ**
Ack	54:49	6	**ACK**
Read/Write	48:47	2	"10" when write, "01" when read
Address	46:32	15	address
Data	31:0	32	value

Table 3.1: configuration interface for the HICANN

3.1.3 ARQ tag id

As mentioned before, the HICANN stores configuration parameters in different kinds of memory which have different access latencies. The most important difference is that the access time for a synapse weight is a single clock cycle while setting an analog neuron parameter that is stored in floating gate memory can take hundreds of cycles. Since these two types of accesses make up the most of the HICANN configuration it was decided to handle them using two separate ARQ links that share the same network. The benefit is that writing the slow floating gate parameters does not interfere with setting the synapse weights because they have their own buffer. The tag-id field in the command word is used to determine which ARQ stream it belongs to. Figure 3.1 shows a block schematic of the data paths in the HICANN.

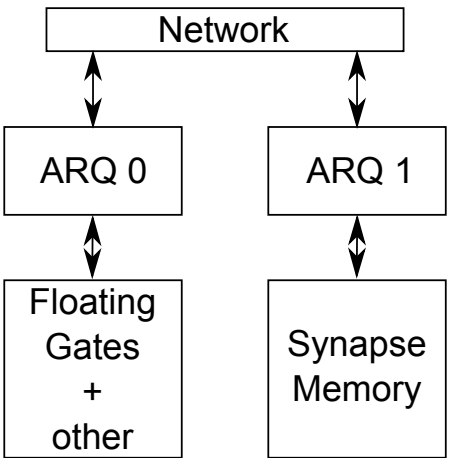

Figure 3.1: Block schematic showing how different types of configuration data are handled using two ARQ links that share the high speed HICANN network link. Note that while ARQ 1 only services the synapse memory, ARQ 0 also handles the rest of the configurable parameters in the chip.

3.2 The DNC_ARQ module

3.2.1 Overview

The FPGA connects via a single DNC to up to 8 HICANNs. Thus it makes sense to bundle all the ARQ links that communicate via a single DNC link in a module called the *dnc_arq*. It resides between the Application Layer and the DNC interface. Figure 3.2 shows the top level view and the word formats on both sides. The dnc_arq is capable of receiving and transmitting one configuration word per clock cycle in both directions, the additional header information at the DNC interface is added and processed internally which explains the difference in the interface widths.

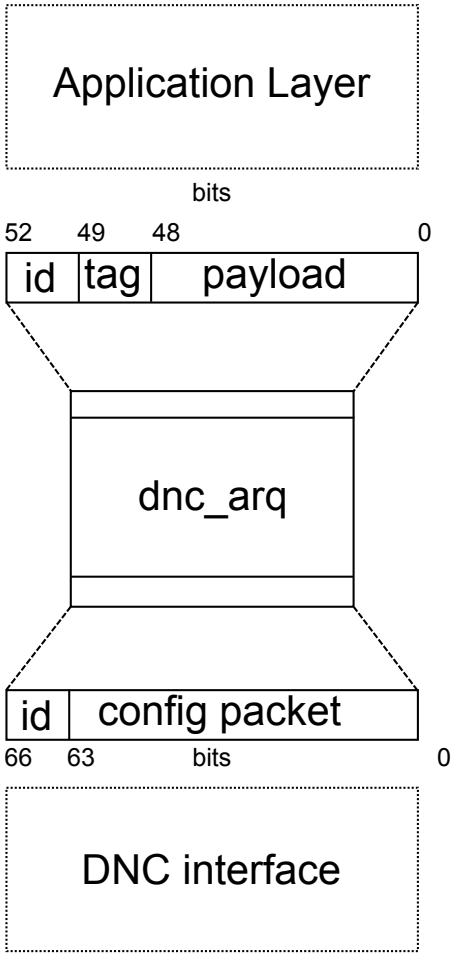

Figure 3.2: Top level view of the dnc_arq module acting as a buffer between the AL and DNC interfaces in the FPGA. Also shown are the word formats on either side. Table 3.1 describes the exact formatting of a config packet at the DNC interface, its lower 49 bits are the payload at the AL side. The id field identifies which of the up to 8 HICANNs connecting to a single DNC the word belongs to.

3.2.2 ARQ configuration

While the host ARQ connection is very configurable in terms of buffer- and sequence sizes, this is not possible for the HICANN ARQ links because the FPGA modules need to be symmetric to the implementation in the HICANN. Thus, a single ARQ link has the following fixed parameters:

Name	Value
SEQ_SIZE	64
WINDOW_SIZE	16

The buffer requirements are also fixed and can be calculated to be

$$\text{buffer size} = \text{WINDOW\_SIZE} \cdot 49 \text{ bit} = 784 \text{ bit} \tag{3.1}$$

per link and direction.

3.2.3 Payload buffer

The dnc_arq module uses a single dual-port BlockRAM primitive per direction to store the windows of all 16 links that it manages. There are several reasons why this choice of implementation makes sense:

- All 16 windows fit comfortably in a single BlockRAM primitive.
- Low access latencies are important because each ARQ packet is only a single word in size which makes bursting impossible.
- Each cycle at most two ARQ clients will need access to memory which is satisfied by the two independent ports of the BlockRAM

Figure 3.3 shows how the 16 ARQ windows are aligned in the BlockRAM.

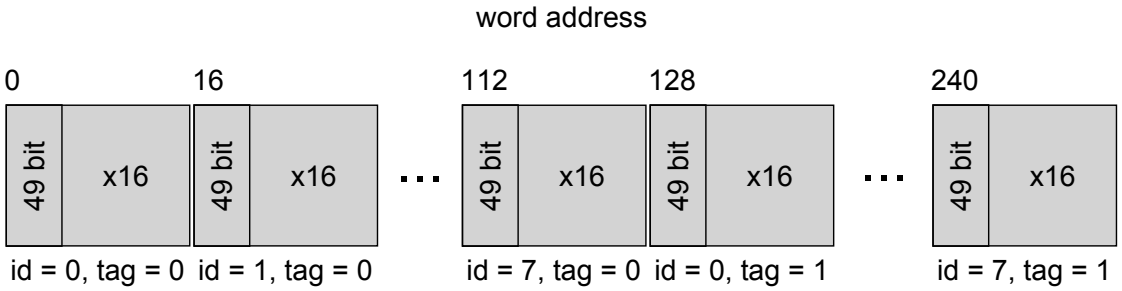

Figure 3.3: Arrangement of the 16 ARQ windows that belong to a single DNC interface in the BlockRAM buffer. Word address 0 marks the start of the window from HICANN #0 with tag 0, followed by HICANN #1, tag 0. The tag 1 windows are aligned in the same way starting at word address 128, i.e after HICANN #7, tag 0

3.2.4 Arbitration

There are two cases where arbitration in the dnc_arq module is necessary:

- Several ARQ master clients might want access to the DNC interface for transmission at the same time.
- Several ARQ target clients might have data for the AL to read at the same time.

There are many different arbiter implementations with varying degree of complexity. The *rrarbiter* module presented by Benjamin Krill in *Krill* implements a light-weight round-robin arbiter with a configurable number of inputs. This module was used and slightly modified to be used within the BrainScaleS system, for the original implementation see *Krill*. The rrarbiter used in the dnc_arq module implements the following features:

Fair arbitration The arbiter implements the round-robin scheme, this means that every client has an equal opportunity for accessing the bus.

Fast arbitration The module does not waste cycles traversing clients which do not request access, the arbitration decision always takes a single cycle.

Optional delay In some situations it is advantageous to have a delay that makes sure each client only gets access at most every N cycles[4] without losing the equal prioritization of the individual clients. The usage of this delay is explained in more detail in subsection 3.2.5

Figure 3.4 shows the arbitration structure in the dnc_arq module. Note that instead of allowing each of the 16 individual clients make a request to the arbiter, the arbitration is implemented HICANN-wise instead. This saves some resources in the arbiter itself, the more important reason however is that this arbitration makes more sense in the direction to the DNC because clients that belong to the same HICANN will share the same physical connection. See subsection 3.2.5 for details. After the arbiter chooses the winning HICANN a priority toggle decides which tag ID gets bus access which makes sure that both tags get equal access opportunity.

[4] where N is a configurable parameter

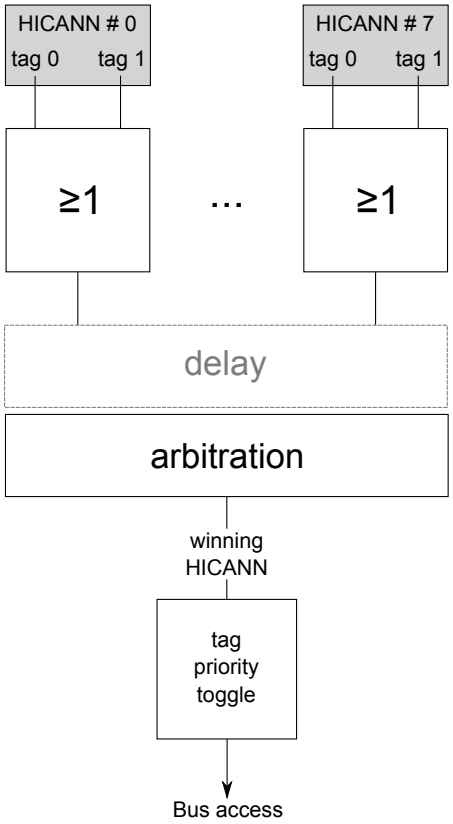

Figure 3.4: Arbiter structure in the dnc_arq module. Arbitration is done HICANN-wise, the priority for tag 0 and tag 1 is toggled to make sure that the tags get equal access opportunity. When activated, the delay stage implements individual counters for each HICANN which prevent requests from reaching the arbiter unless enough time has passed.

3.2.5 Sending data to the HICANNs

Although the dnc_arq module is not divided into rx and tx submodules there is still dedicated and independent circuitry in it that allows simultaneous sending and receiving data from and to a HICANN. Figure 3.5 shows the involved signal paths for the sending of configuration data to the HICANN.

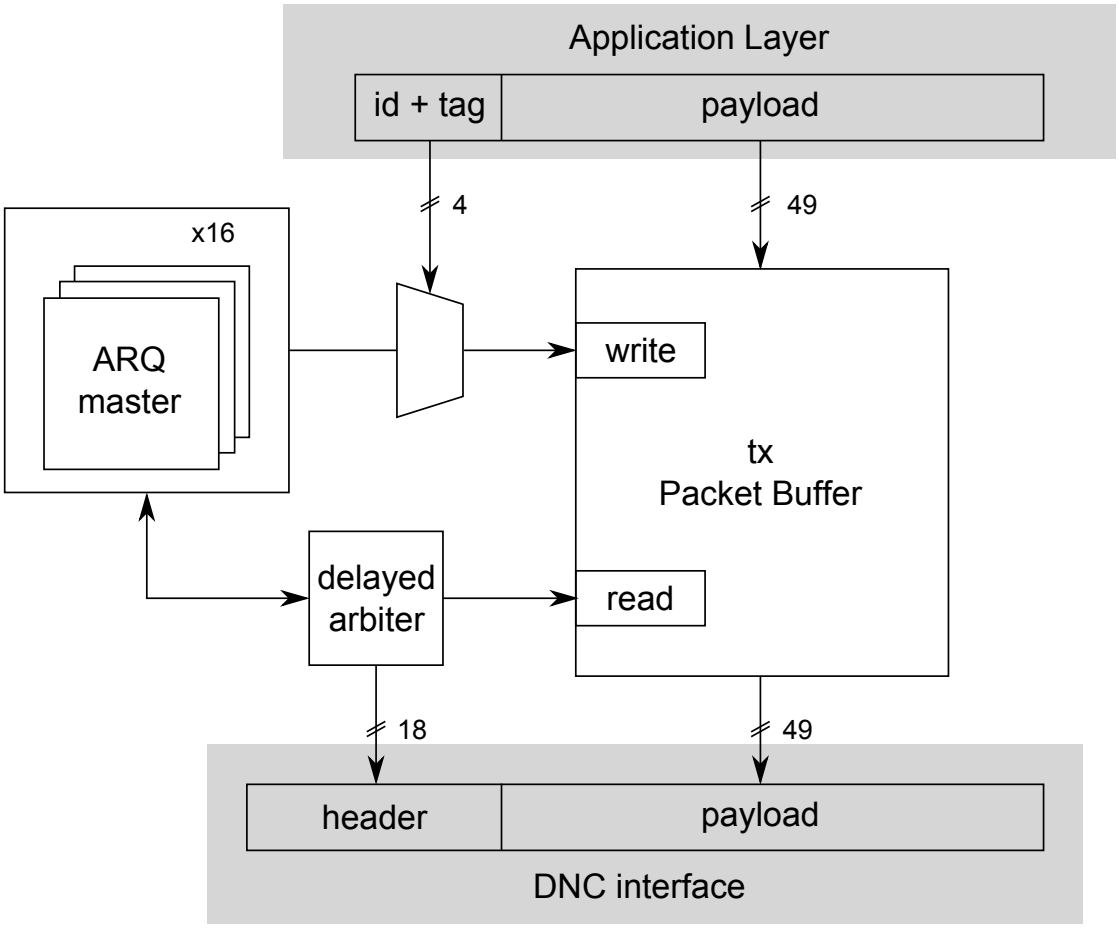

Figure 3.5: Block schematic showing the involved circuitry in the dnc_arq module for transmitting configuration words from the FPGA to a HICANN over the DNC interface.

When writing into the dnc_arq module from the AL a simple multiplexer selects the correct ARQ client based on the tag and HICANN-ID information of the configuration word and writes it in the buffer if possible. However, several ARQ clients might want to transmit data at the same time over the DNC interface which requires some arbitration scheme. The arbitration submodule is discussed in more detail in subsection 3.2.4, the module shown in Figure 3.5 has the delay feature enabled additionally to the pure arbitration stage.

Since there is no flow control between the FPGA and the DNC in the current system and the connection from the FPGA to the DNC has a much higher bandwidth than a single DNC - HICANN link, it is possible to overflow the DNC with configuration data to a single HICANN by allowing the corresponding ARQ clients to send their packets too frequently. The delay stage in the arbiter prevents this by making sure that ARQ clients which belong to the same HICANN have to wait at least some configurable number of clock cycles before they get to send data again.

3.2.6 Receiving data from HICANN

Figure 3.6 shows a schematic of the data flow towards the AL from the DNC interface.

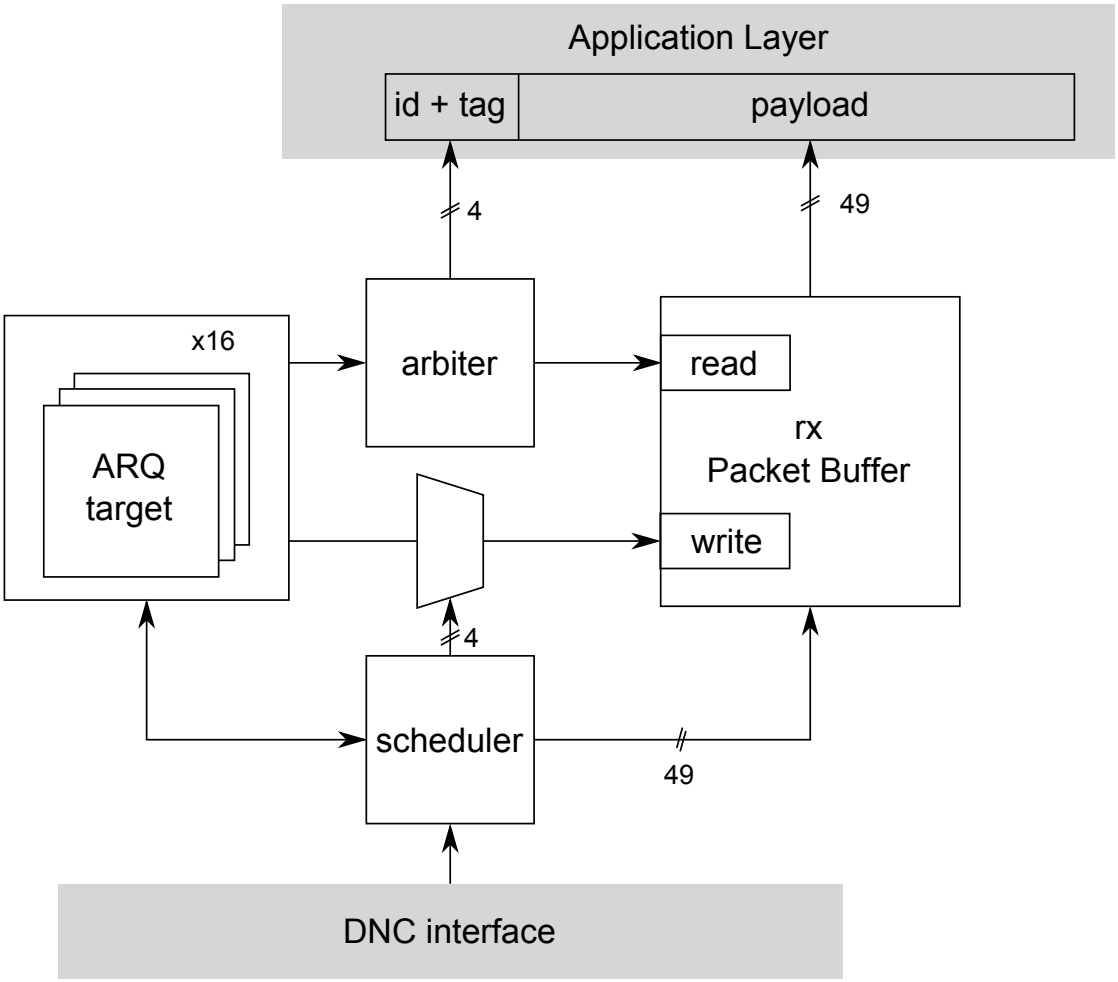

Figure 3.6: Block schematic showing the signal paths between circuitry involved with receiving configuration data from a HICANN over the DNC interface.

The scheduler The ARQ target module has a one clock cycle delay when receiving a packet which means that a packet reception takes two cycles per link. However, this effectively halves the bandwidth of the DNC interface if configuration packets can only be processed every two cycles. The implemented solution was to write a scheduler that has two temporary registers and is capable of processing two ARQ requests simultaneously as long as they target different clients. Since the bandwidth to a single HICANN is 1 Gbit/s, it takes at least eight 125 MHz clock cycles[5] for the DNC to receive a new 64 bit configuration packet from a HICANN and direct it to the FPGA which means that this is also the shortest interval between two configuration

[5] Actually at least 10 clock cycles because there is additional CRC information that is stripped in the DNC

packets that can target the same ARQ link.

Writing data to the AL is done by using an arbiter that decides which ARQ client gets to transmit data from the packet buffer to the AL. Since there is flow control implemented between the dnc_arq and the AL there is no need for delaying the arbitration requests to artificially limit the upstream bandwidth.

3.3 The DNC bug

3.3.1 Overview

When transmitting data to a HICANN, the DNC needs to do two things among others: Multiplexing between pulses and configuration data and perform a clock domain crossing between the 125 MHz FPGA - DNC interface and the 1 Ghz DNC - HICANN link. The designing team at University of Technology Dresden reported that this functionality is faulty and might produce corrupted configuration data.

In particular, it seems that the synchronisation stage is not working properly as the multiplexer is sometimes not holding the right output for long enough and instead of selecting a configuration packet the pulse event input is chosen because this is the default input. Since a pulse event is only 49 bits long the missing bits are padded with zeroes from the top. This corruption is not caught by the CRC because it happens in the DNC before the CRC is calculated and applied. There seems to be no reliable way to predict how likely this phenomenon is to occur because it highly depends on the phase relation between the two clocks whose dynamics is a priori unknown during operation. Thus it is to be assumed that sometimes the HICANN will receive corrupted ARQ packets which carry random payload.

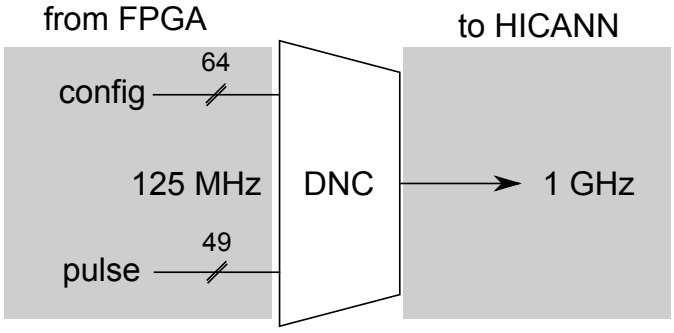

Figure 3.7: When transmitting data to a HICANN, the DNC multiplexes between configuration and pulse data and performs clock domain crossing among other things. This particular functionality was reported to be faulty to some degree and constitutes the DNC bug as it is possible that the DNC selects a pulse event and wrongly transmits it as a configuration packet.

3.3.2 Impact on the protocol

Taking the description above at face value one might assume that this behaviour breaks the protocol because the ARQ is not designed to recognize corrupted packets since it assumes that

every packet that passes the CRC carries uncorrupted information. Fortunately, the corruption of the ARQ protocol fields is static because they are in the high bit positions of the packet that get overwritten to zero when the DNC bug happens. Figure 3.8 shows how a configuration packet looks after it was affected by the DNC bug.

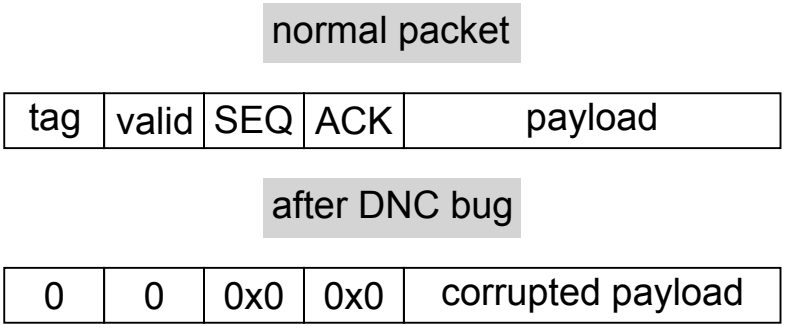

Figure 3.8: When the DNC bug occurs it will replace the payload of the packet with the last content of the pulse event register but always overwrite the ARQ protocol fields to zero. This can potentially break the protocol.

Fortunately, no corrupted payload is ever written to the ARQ buffers because the **valid** field of the broken packet is zero. However, the packet can be interpreted by the HICANN ARQ target as the acknowledgement for the reception of a packet with sequence number 0 sent by the HICANN. This is irrelevant when the HICANN never did send such a packet, but will lead to a desync of the ARQ windows if the HICANN ARQ master sends a packet with sequence number 0 that is never delivered to the FPGA ARQ target because of CRC errors or because the HICANN dropped this packet in favor of an outgoing spike event. If the HICANN ARQ master receives a packet affected by the DNC bug before it has the chance to resend the dropped packet it will wrongly assume that the packet has been delivered because it has seen an **ACK** for 0. This leads to link loss and no more data from the HICANN to the FPGA can be transmitted. Figure 3.9 shows a protocol diagram that describes the necessary conditions that would lead to such protocol break down. Note that the channel from the FPGA to the HICANN is unaffected by this behaviour.

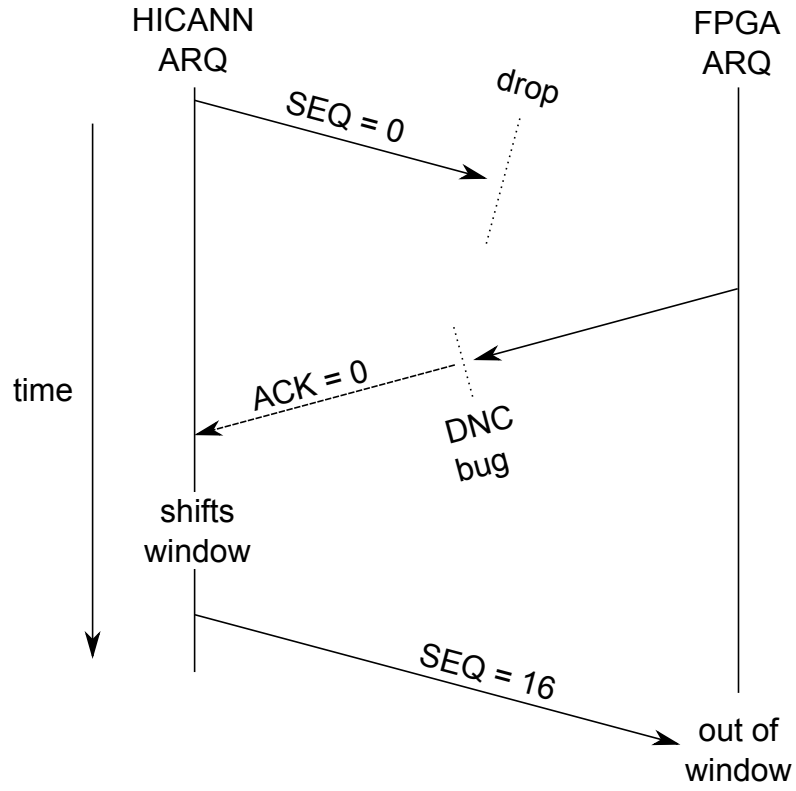

Figure 3.9: The conditions shown in this protocol diagram can desynchronize the ARQ windows between the HICANN and the FPGA. The HICANN sends a packet with the sequence number 0 that is lost in the link before it reaches the FPGA. The FPGA sends some other packet that is randomly affected by the DNC bug and turned into an ACK for 0 which falsely indicates that the FPGA ARQ has received the previous packet from the HICANN. The HICANN ARQ will then move its window since it now assumes that SEQ 0 has been successfully received and starts sending more data. However, the FPGA ARQ will never move its window again because it will wait for packet with sequence number 0 and only accept packets with sequence numbers up to 15. Thus, no more data from the HICANN to the FPGA can be received.

3.3.3 The Workaround

Surprisingly, it is possible to prevent this link loss using only additional logic in the FPGA and without modifying the ARQ protocol itself. The main reason is that the HICANN will never send data on its own, but only when polled by read commands. Since each read command that is sent by the FPGA returns exactly one word from the HICANN it is possible to track the window state of the HICANN ARQ and make sure that the collision described in subsection 3.3.2 never happens.

The basic idea is that by counting the amount of read commands pushed into the ARQ by the AL and keeping track of how many words have been written to the AL by the ARQ it is possible to know beforehand when the HICANN will be sending the crucial packet with SEQ 0.

Dummy read packets are injected into the ARQ at the right time to make sure that the payload of the SEQ 0 packet sent by the HICANN is not actually needed. Before the dummy read packet is sent the workaround makes sure that every packet sent until then has been correctly acknowledged by the HICANN ARQ. Only then will the FPGA send the dummy read packet that triggers the SEQ 0 response by the HICANN while simultaneously injecting a dummy response into the ARQ target in the FPGA which makes it shift its window. This way, even if the SEQ 0 packet from the HICANN is lost the FPGA ARQ target will still believe that it was correctly received due to the dummy response injection. If the SEQ 0 packet is not dropped in the network it will be dropped by the ARQ because it has already moved its window. The dummy response injection is then popped out of the ARQ at the upper layer but is not pushed into the AL because the AL does not know anything about the workaround.

This workaround needs to be replicated eight times to service all HICANNs. It can be enabled at compile time which makes sure that the heavy logic overhead in the FPGA does not need to be synthesized when the DNC bug is fixed in a future revision of the system. Its best feature is possibly that the workaround is completely transparent to the rest of the system with all of the additional functionality encapsulated in the dnc_arq module. The only perceived change for the AL would be a slightly lower average throughput when the workaround is enabled.

3.4 Design Parameters and port description

The only visible generic parameter of the dnc_arq module is the **DNC_BUG** boolean that enables the synthesis of the DNC bug workaround described in section 3.3. The dnc_arq interfaces are designed to be as FIFO-like as possible. While there is full flow control at the AL side, the DNC interface side always needs to be ready to transmit or receive data risking packet loss otherwise.

3.4.1 AL interface signals

Tables 3.2 and 3.3 list the ports for the AL interface.

Name	Width	Direction	Description
ul_fifo_empty	1	Input	Indicates that there is data to be pushed into the dnc_arq when low
ul_write_data [52:50]	3	Input	HICANN ID of the current word
ul_write_data [49]	1	Input	tag ID of the current word
ul_write_data [48:0]	49	Input	Payload
ul_fifo_pop	1	Output	pops current word from the AL

Table 3.2: AL write interface from the dnc_arq's point of view

Name	Width	Direction	Description
ul_read	1	Output	Indicates that there is read data available when high
ul_read_data [52:50]	3	Output	HICANN ID of the current word
ul_read_data [49]	1	Output	tag ID of the current word
ul_read_data [48:0]	49	Output	Payload
ul_read_next	1	Input	pops current word from the dnc_arq

Table 3.3: AL read interface from the dnc_arq's point of view

3.4.2 DNC interface signals

Tables 3.4 and 3.5 list the ports for the DNC interface.

Name	Width	Direction	Description
tx_hicann_config_wr_en	1	Output	Pushes configuration word to the DNC
tx_hicann_fifo_full	1	Input	Is statically set to 1 in the current system, can provide flow control in the future
tx_hicann_config_data_in [66:64]	3	Output	HICANN ID of the current configuration packet
tx_hicann_config_data_in [63:0]	64	Output	Configuration packet, see subsection 3.1.2 for more information

Table 3.4: DNC interface for writing configuration data to a HICANN

Name	Width	Direction	Description
rx_config_valid	1	Input	DNC interface has data when high
rx_config_packet [66:64]	3	Input	HICANN ID of current incoming packet
rx_config_packet [63:0]	64	Input	Incoming configuration packet, see subsection 3.1.2 for more information

Table 3.5: DNC interface for receiving configuration data to a HICANN

3.4.3 Status ports

Table 3.6 lists the available status ports for the dnc_arq module as well as their meaning.

Name	Width	Description
ul_packet_cnt_w [31]	1	AL fifo empty flag, high when the AL doesn't have data to send to the dnc_arq
ul_packet_cnt_w [30:0]	31	Counts the total amount of words pushed into the dnc_arq by the AL since reset
ul_packet_cnt_r [31]	1	dnc_arq has data for the AL that has not been popped yet when high
ul_packet_cnt_r [30:0]	31	Counts the total amount of words popped from the dnc_arq to the AL since reset
network_debug_reg[63:32]	32	Counts the total amount of configuration words sent by the dnc_arq to the DNC interface since reset
network_debug_reg[31:0]	32	Counts the total amount of configuration words received by the dnc_arq from the DNC interface since reset

Table 3.6: Available status ports for the dnc_arq module

3.5 Evaluation and testing

It is always difficult to gauge the stand-alone performance of a single part of a system as large as the HMF, especially when it is as deeply nested in the system as the dnc_arq module. The dnc_arq module is no exception and the best solution is to come up with a test mode that tries to demonstrate the best possible performance of the module regardless of how realistic that usage might be for the experiment. That way a benchmark can be set and the experimental set up can be optimized to try and reach that benchmark if the current performance is deemed too low.

The test mode The dnc_arq is evaluated based on the time it takes to push a certain number of read commands into it from the AL side and the time it takes until the same amount of response words can be read out from the dnc_arq by the AL. The particular read commands were chosen to be fast register accesses in the HICANN to try and minimize their influence on the raw ARQ performance. Among other things, this performance is dependent on the set of timings both in the FPGA and HICANN ARQ clients as well as the total amount of used HICANNs and tags. Note that it is not possible to vary the buffer sizes and explore their influence on the throughput like it was done during the evaluation of the host_arq module since the buffer size in the HICANN is fixed. The plots and measurements done in this section were done in collaboration with Eric Müller who provided assistance at the software side of the experiments.

3.5.1 The perftest module

The test mode used in the performance evaluation of the dnc_arq was implemented in the so called *perftest* module that replaced the normal BrainScaleS Application Layer during testing.

This made the measurements more precise because the perftest module can time the amount of sent and received packets directly and is not affected by the host connection. A single experiment is comprised of the following steps:

1. Program the resend and ack timings in both the FPGA and HICANN ARQ modules, as well as the arbiter delay of the dnc_arq module.
2. Specify in the perftest module via configuration registers that are accessible by the host the total amount of words and which tags and HICANNs are to be used.
3. Start the experiment via a programmable flag in the perftest module.

The perftest module will then begin to try and push configuration packets into the dnc_arq that would trigger the fastest possible response by the HICANN for that particular tag. A timer is also started that counts the amount of clock cycles since the beginning of the experiment. This will continue until the specified amount of words is reached at which point the current timer value is stored in the *test_time_push* output register. The perftest will also pop any data that is presented to it by the dnc_arq module. When the amount of popped words has reached the pre-set number the perftest module will stop the timer, raise the *test_done* flag and output the timer value in another output register called *test_time_pop*.

The ratio $\frac{num\_packets}{test\_time\_push}$ gives the downwards throughput of the dnc_arq module in terms of bits per second. Conversely, it might be interesting to think of the throughput as the inverse of that number, i.e how many cycles it takes at average until a new word can be pushed into the dnc_arq.

3.5.2 Sweeping the ARQ timings

Using the perftest module about 300 thousand experiment runs were carried out sweeping the ARQ timings in both the HICANN and FPGA as well as the arbiter delay. During a single run some thousand packets were sent to tag 0 of a single HICANN.

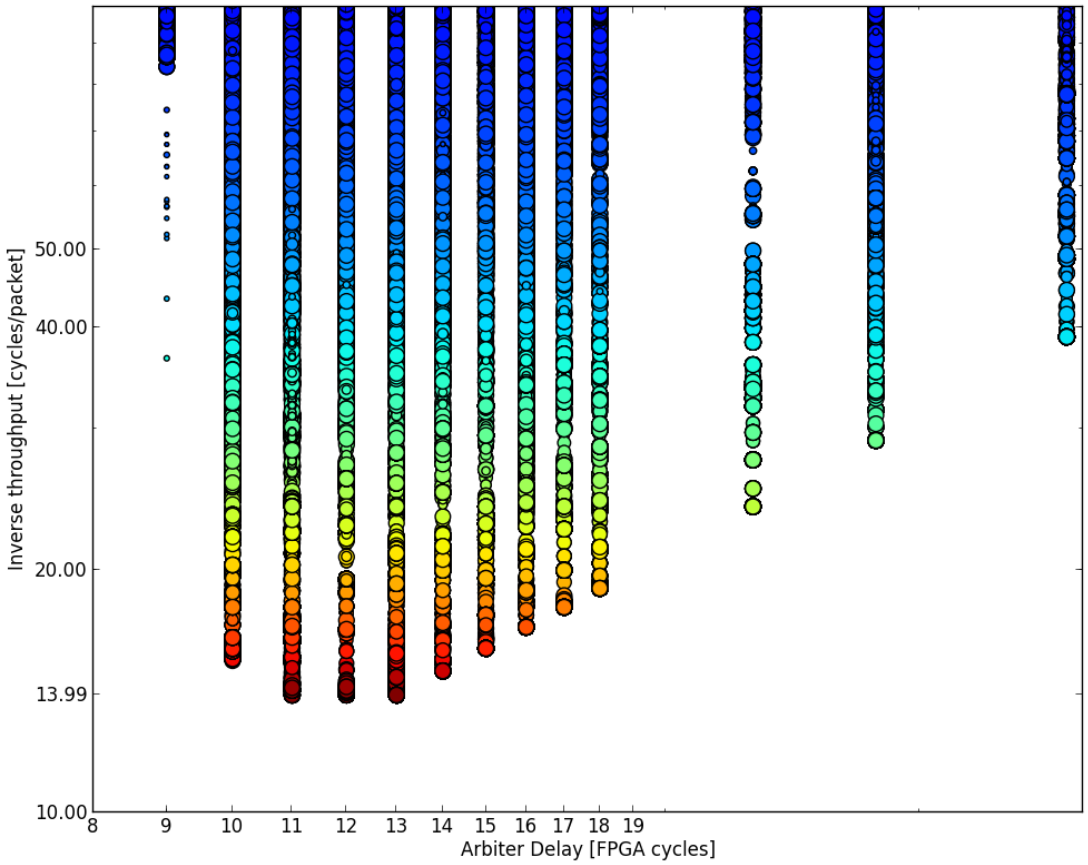

Figure 3.10: Projection of the data set on the arbiter delay setting. Vertically aligned points share the same arbiter delay value but may differ in other settings. Color encoding is from red (high throughput) to blue (low throughput), the size of the points is inversely proportional to the variance of the measurement of that particular configuration. The performance reaches a very prominent optimum at 11 to 13 clock cycles with every other setting yielding worse results regardless of the other timings. Higher values perform worse because this is equivalent to an artificial throttling of the link since the arbiter delay dictates the lowest period in which two packets to the same HICANN can be sent. Lower values than the optimum perform worse because they result in a higher bandwidth than the HICANN serial link can handle which forces the DNC to drop packets

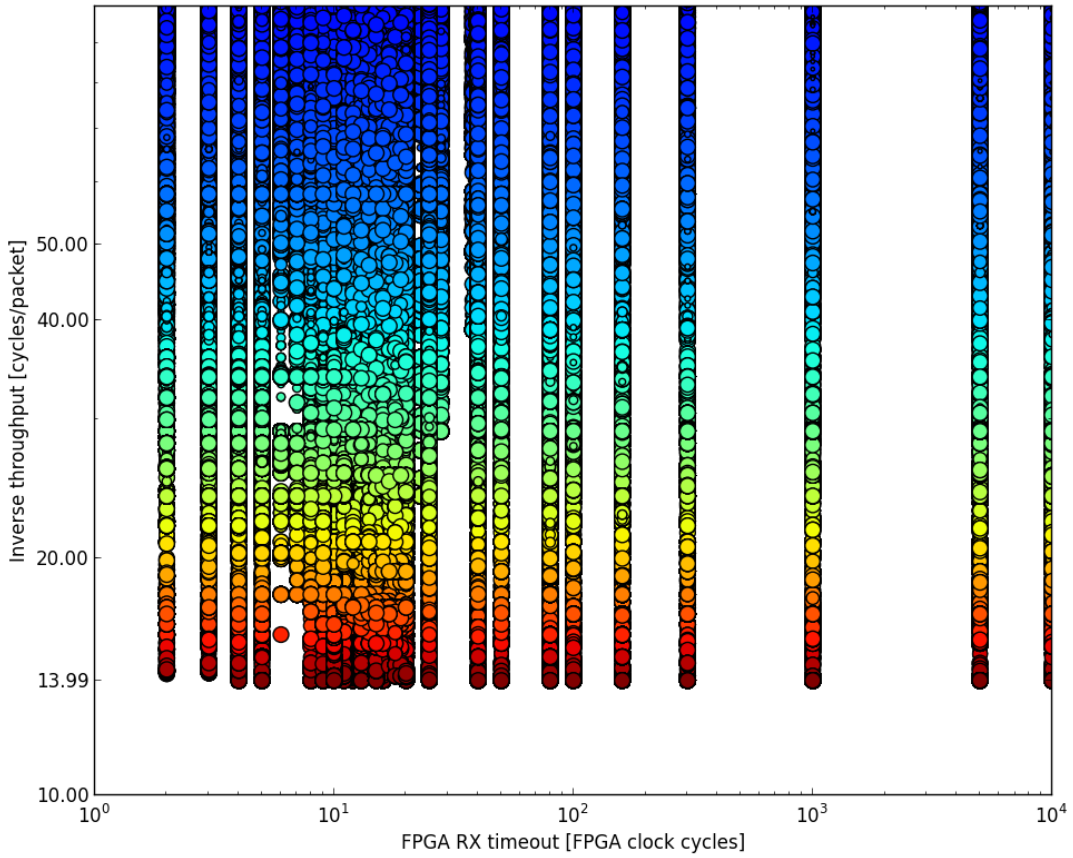

Figure 3.11: Projection of the data set on the FPGA RX timeout axis. Vertically aligned points share the same FPGA RX timeout value but may differ in other settings. Color encoding is from red (high throughput) to blue (low throughput). Evidently, the FPGA rx timeout doesn't seem to influence the throughput very much in this configuration because for every value the same maximal performance can be reached. This is due to the fact that there is no need for ACK only packets because data is continuously being sent to the HICANN carrying the ACKs piggyback and updating the HICANN ARQ in time. Note that there is a performance drop when the timeout is set to be $\approx$7 clock cycles. This can be explained by a race condition between payload and ACK only packets where the latter get sent unnecessarily which wastes bandwidth.

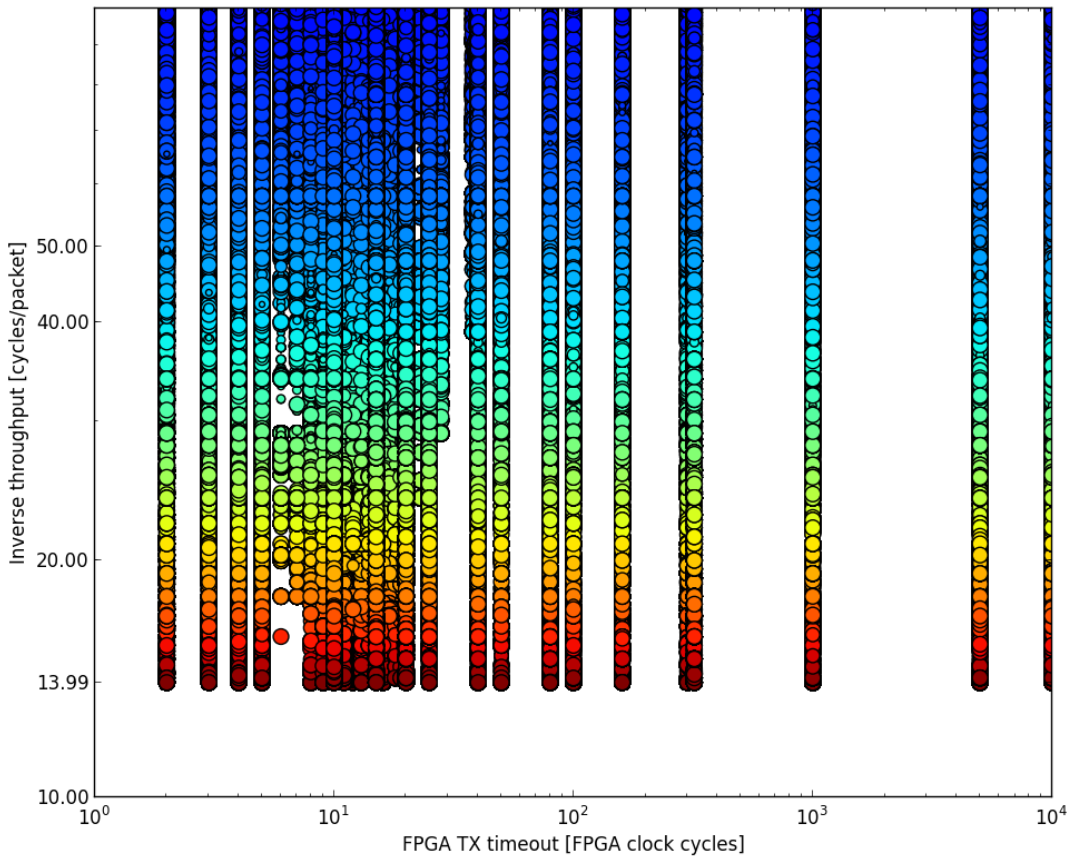

Figure 3.12: Projection of the data set on the FPGA resend timeout axis. Vertically aligned points share the same FPGA TX timeout value but may differ in other settings. Color encoding is from red (high throughput) to blue (low throughput). The resend timeout value is also rather unimportant for this experiment which indicates that the connection is very stable and packets are dropped very rarely due to CRC errors. The only drop in performance is again when the resend value is set to be at about 7 clock cycles which has the same explanation as in the RX timeout case: Data is unnecessarily resent before the ACK from the HICANN can reach the FPGA which wastes bandwidth

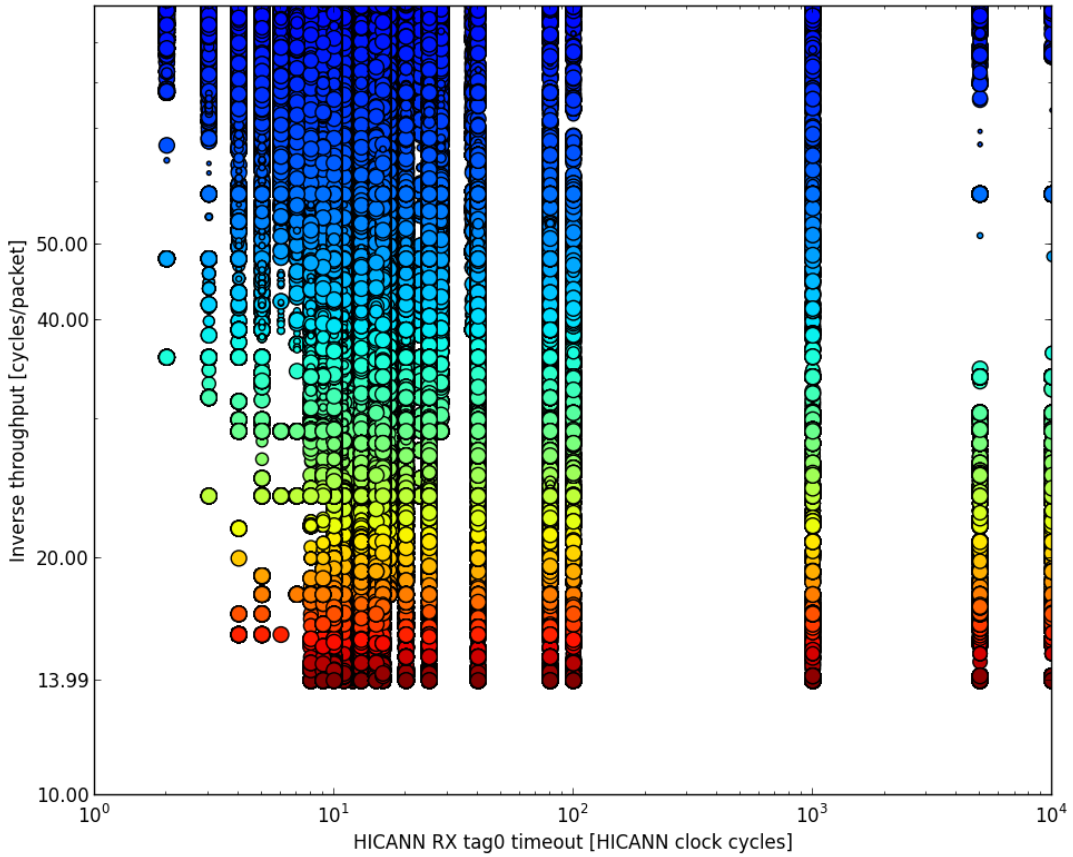

Figure 3.13: Projection of the data set on the HICANN RX timeout axis. Vertically aligned points share the same HICANN RX timeout value but may differ in other settings. Color encoding is from red (high throughput) to blue (low throughput). Low values negatively impact the performance because the HICANN needs a few cycles to return the requested payload for sending back. If an ACK only packet is sent during this time the HICANN needs to wait until the link is free again to send the actual data. Higher RX timeout values do not impact the performance because there is always traffic back to the FPGA that carries the ACKs piggyback in this setup

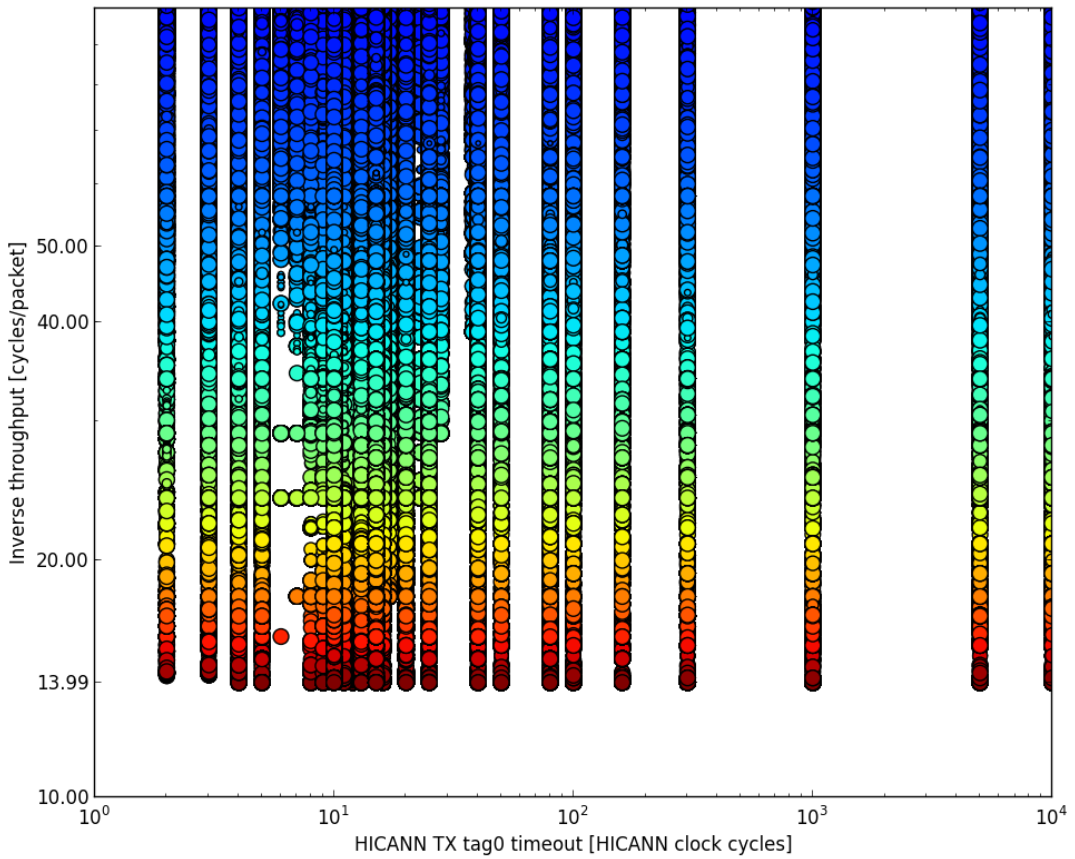

Figure 3.14: Projection of the data set on the HICANN resend timeout axis. Vertically aligned points share the same HICANN RX timeout value but may differ in other settings. Color encoding is from red (high throughput) to blue (low throughput). As with the FPGA resend timings, this setting seems to be irrelevant because of the stability of the serial link. The drop in performance between 7 and 8 cycles is again due to unnecessary resends, lower values are irrelevant because the HICANN is not able to resend data that quickly, higher values don't change the performance because ACKs from the FPGA arrive quickly enough so that the resend is not carried out

3.5.3 Using multiple tags and HICANNs

The previous measurements indicated that the best case throughput for sending configuration data to a single HICANN at tag 0 is one word every 14 clock cycles on average. Although the configuration data is only 49 bits wide at the AL it is aligned to 64 bits in memory and also during transmission from the host, thus it is fair to calculate the effective AL bandwidth as

$$T = \frac{64 \text{ bit}}{14 \cdot 8 \text{ ns}} = 0.571 \frac{\text{Gbit}}{\text{s}} \tag{3.2}$$

This number is some 43 per cent smaller than the raw network bandwidth of 1 Gbit. Taking the protocol overhead as well as the CRC field into account the total throughput becomes

$$T = \frac{80 \text{ bit}}{14 \cdot 8 \text{ ns}} = 0.714 \frac{\text{Gbit}}{\text{s}} \tag{3.3}$$

The only explanation for the missing bandwidth is that the ARQ window is not large enough to buffer the latency between FPGA and HICANN to reach full performance. One contribution to that latency is the speed of the upper layer in the HICANN that is on top of the ARQ. In fact, using tag 1 instead of tag 0 the best case throughput improves to a packet every twelve clock cycles as can be seen in Figure 3.15. This is because tag 1 pipelines the interface to the ARQ which allows it to free the window quicker at the receiving side.

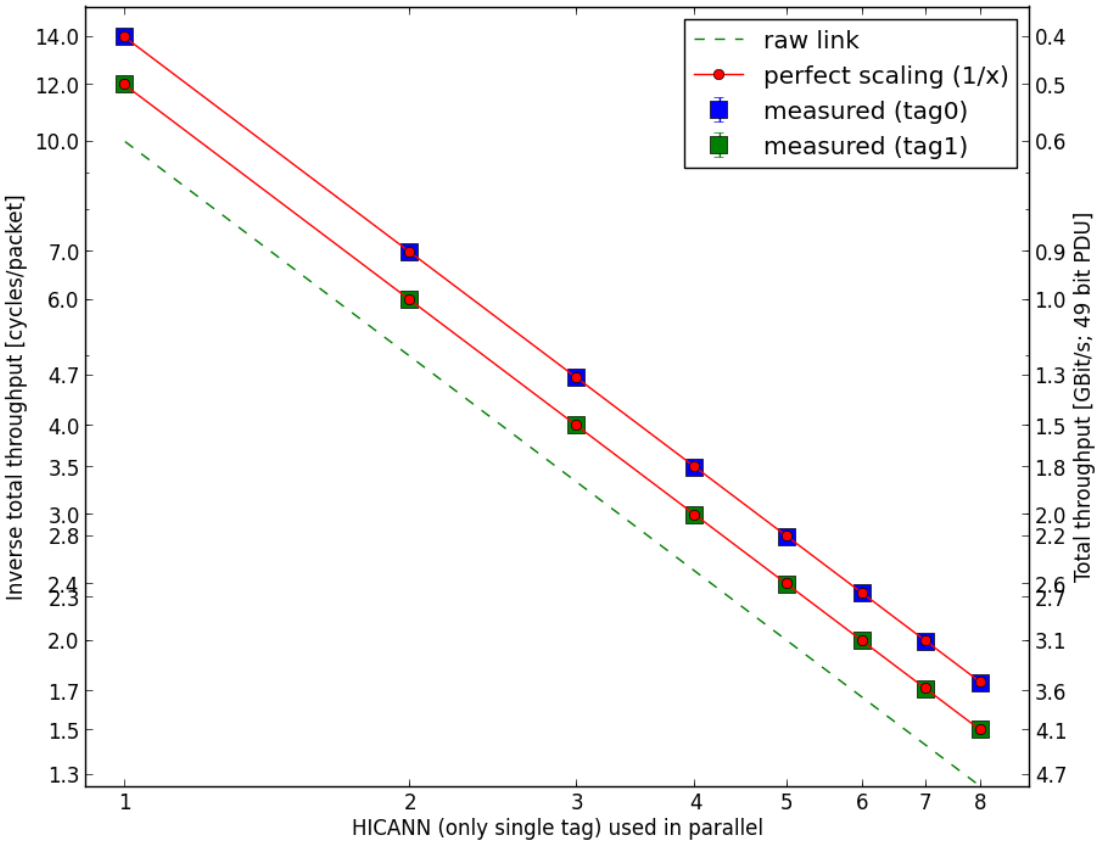

Figure 3.15: Performance of the dnc_arq at the AL side when using different tags and multiple HICANNs

Theoretically, using both tags at the same time should yield a higher total throughput because the effective window for the link is now doubled. However, the two ARQ clients in the HICANN unfortunately interfere with each other when sending data to the FPGA which results in packet loss and gravely impacts the performance. Thus, it is advised to not use both tags to communicate with a HICANN at the same time.

Figure 3.15 also demonstrates that using multiple HICANNs linearly improves the throughput at the AL as is to be expected which indicates that the arbitration in the dnc_arq module

works very efficiently without noticeable overhead. In fact, configuring two or three HICANNs in parallel already saturates the 1 Gbit/s Ethernet host connection when using host ARQ with current generation hardware.

3.5.4 Introducing network noise

During the configuration phase of a HICANN, where the biggest usage of the ARQ links is to be expected, there are no pulse events in the network. However, it might be useful to read out configuration data from the HICANN during an experiment where pulse events are also present. The existing test setup can be extended to introduce network noise using Background Event Generators (BEGs) in the FPGA that send Poisson-distributed spike events with a configurable rate to the HICANN. Figure 3.16 shows the AL throughput for a single HICANN tag 0 for various spike event rates.

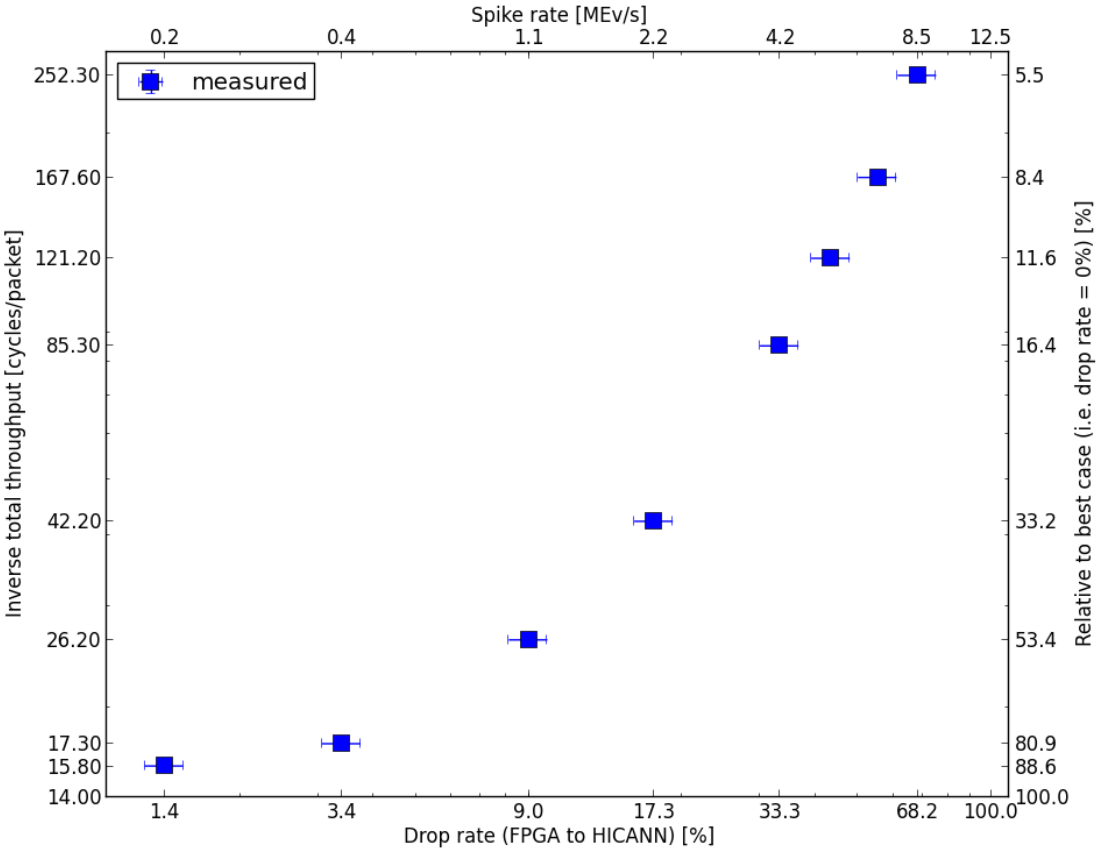

Figure 3.16: Change in throughput at the AL for tag 0 configuration data to a single HICANN when additional pulse events are present in the network. The x-axis shows the spike event rate in the network which is equivalent to a drop rate for ARQ packets because spike events are always prioritized

Evidently, even small noise rates cause a very sharp decrease in the ARQ throughput. This

is again because of the very small ARQ windows that can not buffer enough data to cope with packet loss. Additionally, because of the static prioritisation of pulse events packets it is possible to completely starve the ARQ so that no configuration data can be transmitted over the high-speed link when the pulse event rate is high enough to saturate the available bandwidth. It is therefore not recommended to use the HICANN ARQ links in parallel with pulse events in the current system except when low throughput is acceptable.

3.5.5 Investigating the DNC bug

During testing it was not possible to gather any substantial evidence that would confirm the existence of the DNC bug. In particular, during long running experiments in the Terabyte range the window de-synchronisation that can happen because of the DNC bug was never observed. Furthermore, experiments showed that it is possible to transmit large amounts of configuration packets to the HICANN without any resends which suggests that the probability for the DNC bug to occur at all seems to be very low. The only found phenomenon where data arrives corrupted in the HICANN is when the packet rate from the FPGA exceeds the HICANN link capacity. However, as this behaviour is deterministic and is completely fixed by the arbiter delay without requiring the very complex workaround presented in subsection 3.3.3, it does not fit with the original description of the problem as was described by the DNC designing team during meetings.

Still, it would be too rash to conclude that the DNC bug does not exist at all, as there is no hard evidence supporting that claim either. That the DNC bug was not clearly observed could also mean that some other yet unknown effect prevents it from happening. As long as this is not completely understood it would be foolish to stop investigations because the HICANN ARQ links can not be declared as completely reliable otherwise and the workaround should be kept enabled for the time being. Fortunately, the upgraded HMF will not have the DNC ASIC anymore because its functionality will be moved into a bigger FPGA and merged with the current FPGA design, which offers the opportunity to fix the original bug in the first place.

4 Discussion and Outlook

This chapter discusses the measurements and observations made during the evaluation of the host ARQ and the HICANN ARQ connections and proposes some future improvements to the system.

4.1 The host link

4.1.1 Summary of evaluation

The measurements done for testing the host ARQ connection as described in section 2.8 yielded generally very satisfactory results. The host_arq module performs to specification, reaching wire speed for a wide range of parameters. Implementing the packet buffer in DRAM to save BlockRAM usage was also successful and the pre-fetching scheme made sure that the protocol latency between the network and the application layers is dominated by the DRAM latency, since the actual protocol processing time is negligible compared to the time it takes to move the payload through the network stack. Still, there are some issues addressing which would further improve the link performance.

4.1.2 Packet size vs window size

Experiments clearly showed that while large buffer sizes generally improve the throughput, the segmentation in packets makes a significant difference. There are several reasons why large packet sizes are better than large window sizes:

- Less resource consumption in the FPGA because the ARQ needs a very resource-intensive bit array for maintaining the receiving window. Window sizes larger than 512 packets are economically impractical, while the DRAM has Gigabytes of space that can be extensively utilized with sufficiently large packet sizes.
- Better throughput because of the smaller protocol overhead relative to the total packet size.
- Expensive software interrupts can be decreased in frequency when using large packet sizes. This is important because the host needs to manage 48 ARQ links per wafer, which means that for example a single 8 core CPU needs to be able to run 6 software ARQ instances per core at full speed to fully saturate the available bandwidth. Assuming a peak interrupt rate of 150 kHz for a single CPU, a single ARQ link is allowed 12,5 kHz interrupts per direction at most. This is an extreme case however, because in that scenario the host would have to sacrifice enormous computing resources just for maintaining throughput.

Small packets are only needed when low latency communication between the host and FPGA is required, for example in the so-called closed-loop experiments[1]. However, these experiments

[1] In this context, a closed loop refers to a bidirectional connection, e.g between a neural network emulated on neuromorphic hardware and an environment simulated on the host PC at an equivalent speedup

will likely not use the ARQ at all because the buffering of spike events introduces too much jitter in the inter-spike-intervals. The normal operation mode uses large playback memories in the FPGA that are preloaded with spike events by the host and are capable of controlling the timing between spike events very finely. Thus, only the total throughput to the ARQ link matters when writing to the playback memory.

While the Ethernet MAC in the FPGA as well as the host Network Interface Cards (NICs) and the switches support framesizes of up to 9kB[2], the current limitation to 1500 Bytes is set by the UDP layer implementation in the FPGA. Although not a top priority, upgrading the UDP to support large frames will improve the system due to the reasons listed above.

4.1.3 Improvement of the ARQ implementation

It is possible to reimplement the already mentioned resource-intensive bit array that represents the rx window so that it can be synthesized using BlockRAMs instead of flip-flops. This would make very large window sizes possible because a single BlockRAM is already 36 kbits in size. However, as larger packet sizes are more important than large windows, implementing this feature is less important than enabling jumbo framing in the UDP core.

4.1.4 Improving throughput to the host

As described in subsection 2.8.3, the UDP layer seems to not be able to achieve wire speed when transmitting data to the host. This bottleneck obviously limits the ARQ throughput to the host to about 80 MB/s which is about 30 % lower than the performance achieved from the host to the FPGA. The wasted bandwidth is quite significant, especially in a I/O-intensive system like the HMF, and thus needs immediate addressing and improvement of the current UDP layer.

4.1.5 Enabling Ethernet flow control

Clause 31 of the IEEE Std 802.3-2005 standard specifies Ethernet flow control frames that can throttle throughput between two nodes at the MAC level. This feature is supported by the Xilinx MAC that is used in the HMF FPGAs and can be used to throttle traffic between host and FPGA when ARQ buffers are about to overrun. Of course, the ARQ resend functionality will eventually make sure that any dropped packets are restored regardless of cause. However, frequent resends come at a rather large throughput penalty and should be avoided whenever possible. As a general rule, the ARQ resend functionality should only deal with data loss associated with link corruption and not be used as a failsafe against congestion, where Ethernet flow control is much more suited.

4.2 HICANN configuration

Improving the HICANN ARQ connection is more difficult because it necessarily involves changes in the DNC and/or the HICANN ASICs. Fortunately, almost all of the discovered problems can be attributed to technological limitations in the current system and not so much to poor design. The proposed changes should be taken into consideration for the next generation HICANN design which is entering the implementation phase at the time of writing of this thesis.

[2] This feature is often called Jumbo Framing

As a general note, it is important to emphasize that the existing implementation fulfills its purpose, namely to provide a secure configuration link to a HICANN, perfectly fine. Since a complete HICANN configuration is only a few kilobytes in size and needs to be done once before the experiment with no concurrent spike events, the current state of the HICANN ARQ link is very much capable to transfer it efficiently. The only lacking operation mode is when the ARQ has to share the link with pulse events, which might be of interest when reading out synapse weights from the HICANN during an experiment or the like. Several changes can be made to improve the current design in that regard.

4.2.1 Improving the bandwidth arbitration

The current system employs a strong prioritisation scheme towards spike events in the DNC which can lead to drops of ARQ packets. This rather aggressive behaviour makes sure that spike events never get delayed or dropped in the network in favor of configuration packets, but can also dramatically decrease the ARQ throughput as was demonstrated in subsection 3.5.4. Furthermore, the fact that the ARQ can be completely starved on bandwidth at high spike event rates means that it is rather dangerous to try and use it during an experiment at all. To address these problems two changes are proposed.

Enabling flow control between spikes and configuration The ARQ should be made aware of the pulse events and only allowed to send packets to the network when it is free. This is not possible in the current system because the prioritization happens in the DNC which has no flow control ports to the FPGA. However, such a feature can be easily implemented in the next generation FPGA since the DNC functionality will be moved into it without requiring the ASIC anymore. Similarly, the next generation HICANN should also employ some sort of flow control to stall the ARQ when the link is occupied. Note that this scheme will not affect the timings of the pulses because they are still statically prioritised. The difference to the current behaviour is that instead of dropping frames in the network which the ARQ does not know about, the ARQ itself waits until it can send data. This would dramatically improve the ARQ throughput when pulse events are present.

Dynamic prioritisation Softening the prioritisation towards the pulse events, effectively allocating a minimum guaranteed bandwidth for the ARQ connection, would make sure that the HICANN is accessible for configuration even at high pulse event rates. In practice, this might be realised via a timeout counter that counts the number of cycles where the ARQ is requesting access to the link but is not allowed to send because of event packets. When that timeout is reached the priority is reversed and the ARQ gets to send data even if it means that events are dropped. This timeout can be configurable to allow individual settings for the experiments.

4.2.2 The tag structure

It has been observed that using both HICANN tag clients simultaneously will hinder the throughput instead of improving it because the buffer size is effectively doubled. The cause was found to be in the HICANN tag arbitration circuitry which induces drops when both tags request access to the network simultaneously. This can be easily fixed and should be implemented in the next HICANN generation.

This minor problem notwithstanding, an argument is to be made against the usage of two tags altogether. The only reason why two ARQ clients were used is so that different types of

configuration words that have a different processing time deeper in the HICANN have their own buffer and are transported independently from each other. However, since the dnc_arq module only has a single access port to the AL for eight HICANNs and two tags, the distribution of targets in the stream can affect the throughput. For example, if the dnc_arq is not able to pop data from the AL because the corresponding ARQ buffer is full it will stall data for all other clients as well, even if their buffers might be free[3]. Thus, to maximise the throughput to the dnc_arq module the host should already order the configuration stream in a way that fills all of the ARQ buffers evenly.

If the ARQ tag structure is deprecated and only a single ARQ client manages the transport of configuration data over the high speed link instead, the host could also make sure to scatter slow commands in the configuration stream to avoid congestion. In the HICANN, two additional FIFOs separate the slow and fast commands from the ARQ. The proposed structure in the HICANN is sketched in Figure 4.1

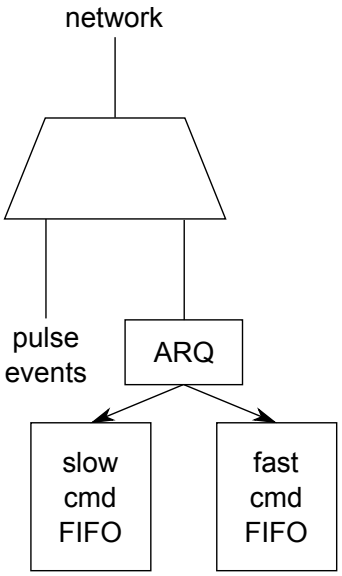

Figure 4.1: Proposed changes for the next generation HICANN communication bus. Instead of using two ARQ clients that share the network together with pulses there is only one ARQ client and the split in slow and fast commands is done afterwards

[3]It is possible to mitigate this problem by providing the AL with 16 FIFOs that each feed in their own ARQ buffer, but since all of the configuration data comes in a single contiguous stream from the host ARQ this does not provide a general solution.

5 Conclusion

The modules designed and implemented during the presented thesis replace analogous functionality in the FPGA and host systems designed by a team at the University of Technology Dresden. The at the time available host and HICANN links were always meant to be only an temporary solution until the final system is completed. This chapter compares the two solutions and estimates the performance increases for high level tasks to the user.

5.1 HICANN configuration performance

Before starting an experiment, the HICANN needs to be configured by the host. To avoid undefined behaviour, all configuration values in the HICANN are set even if they are not needed for the particular experiment. A complete configuration of a HICANN requires 26090 configuration packets sent over the HICANN ARQ. Since the configuration packets are aligned to 64 bits at the FPGA Application Layer, the total amount of data that has to be sent from the host is 208720 Bytes per HICANN.

Description	**Number**
SwitchRAM	$4 \cdot 112$
Crossbar switches	$2 \cdot 64$
Floating Gate parameters	$2 \cdot 129 \cdot 21$
Synapse values	$8960 \cdot 2$
Synapse Driver	$2 \cdot 3 \cdot 224$
Repeater	$4 \cdot 64 \cdot 32$
Neuron builder	512
Sum	26090

Table 5.1: Various types of configuration values that can be configured by the HICANN ARQ configuration packets

5.1.1 Comparing configuration times

Configuration via JTAG In the original FPGA system there was no equivalent circuitry to the dnc_arq module, i.e the FPGA was not able to exchange configuration packets with the HICANN via the high speed serial link. Instead, the HICANN had to be configured using a JTAG debug port which has access to the ARQ module in the HICANN, bypassing the high speed link. In this scheme, the FPGA ARQ module facing the HICANN has to be emulated in software at the host, and the individual configuration packets are sent to the FPGA via Ethernet and from there to the HICANN via JTAG.

HostAL + dnc_arq When the dnc_arq module was finished it was accessible by the host via the HostAL software layer that is capable of packing several configuration values in a single Ethernet frame and send it to the FPGA where they are fed to the dnc_arq module. The

HostAL realizes transport layer functionality by using the stop-and-go scheme, i.e it waits for a response from the FPGA after each individual frame before sending the next.

Since there are large discrepancies between processing times of various configuration values in the HICANN it is rather difficult to gauge the difference the communication makes. A rather artificial experiment is to compare the time it takes to write all of the 26090 configuration packets to the HICANN but only use the fastest values available. This way, a fair comparison can be made between the different communication links without the influence of the HICANN itself. Using the two different methods, this experiment had the following results:

Method	**Time**
JTAG	39.72 s (measured)
HostAL + dnc_arq	0.521 s (measured)
host ARQ + dnc_arq	≈5 ms (projected)

Table 5.2: Measured and estimated times for transmitting configuration files to a HICANN from the host using different methods

Currently, no direct measurements exist on how long this experiment will take on the finished system because the integration of the host ARQ module into the FPGA will be completed after the submission of this thesis. However, using available data it can be estimated that transmitting 26090 configuration words to the HICANN should take of the order of milliseconds from the host over the host ARQ and the HICANN ARQ[1].

Subsection 3.5.3 demonstrated that the dnc_arq module takes 12 clock cycles to send a configuration packet to the HICANN in best case. Consequently, a full HICANN configuration file will take about 2.5 ms to transmit at the dnc_arq AL side, assuming that the AL is fast enough. Furthermore, performance measurements described in section 2.8 reported bandwidths of up to 117 MB/s from host to the FPGA. This bandwidth is enough to transmit the 208720 Byte large HICANN configuration file in under 2 milliseconds. The total configuration delay from host to FPGA and from FPGA to HICANN is thus smaller than 5 milliseconds. The only remaining unknown factor is the delay of the AL in the FPGA, which should however be negligible compared to network delays – assuming efficient implementation – because FPGAs are easily capable of moving Gigabytes per second in the chip.

5.2 Spike data throughput

The HICANN configuration data size is negligible compared to the Gigabytes of spike events that need to be transmitted from the host to the HICANN during an experiment. The large amount of spike data is simply explained by the high acceleration factor of the hardware neurons and the fact that neurons are typically stimulated with spike frequencies of about 10 Hz over seconds or even minutes in biological time. The spikes are preloaded in the FPGA from the host and are sent under precise timings to the HICANNs after starting the experiment. The relevant bottleneck is thus the throughput between the host and the FPGA.

[1] It should be again noted that the current HICANN generation takes a very long time to write analog floating gate values which limits the ARQ bandwidth in itself. However, the next generation will have only fast SRAM or register memory which allows single clock cycle accesses

5.2.1 HostAL vs Host ARQ

The main limitation in the current Host AL connection is that the host needs to wait for a response between individual Ethernet frames because the protocol does not implement a window. Measurements show that the achieved maximum packet rate from host to FPGA is about 4kHz. Since the maximum packet size is fixed to 1500 Bytes, the maximum HostAL throughput is in the low Megabytes per second range even when neglecting protocol overhead. Conversely, the evaluation of the host ARQ connection demonstrated stable net bandwidths from host to FPGA of up to 117 MB/s which improves the connection by a factor of at least 20 and dramatically decreases experiment setup times.

List of acronyms

AL Application Layer

ARP Address Resolution Protocol

ARQ Automatic Repeat reQuest

ASIC Application Specific Integrated Circuit

BEG Background Event Generator

CMOS Complementary Metal–Oxide–Semiconductor

DNC Digital Network Chip

FPGA Field Programmable Gate Array

FSM Finite State Machine

HDL Hardware Description Language

HICANN High Input Count Analog Neural Network

HMF Hybrid Multiscale Facility

HPC High Performance Computing

MPMC Multi Port Memory Core

NIC Network Interface Card

OSI Open Systems Interconnection

PAR Place And Route

PCB Printed Circuit Board

RGMII Reduced Gigabit Media Independent Interface

RTL Register Transfer Level

SO-DIMM Small Outline Dual In-line Memory Module

Bibliography

Virtex-5 FPGA User Guide, Xilinx, Inc., 2009.

LogiCORE IP Multi-Port Memory Controller (MPMC), Xilinx, Inc., 2011.

Braden, R. T., RFC 1122: Requirements for Internet hosts — communication layers, 1989.

BrainScaleS, Research, `http://brainscales.kip.uni-heidelberg.de/public/index.html`, 2012.

Brüderle, D., et al., A comprehensive workflow for general-purpose neural modeling with highly configurable neuromorphic hardware systems, *Biological Cybernetics*, *104*, 263–296, 2011.

Diesmann, M., and M.-O. Gewaltig, NEST: An environment for neural systems simulations, in *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001*, *GWDG-Bericht*, vol. 58, edited by T. Plesser and V. Macho, pp. 43–70, Ges. für Wiss. Datenverarbeitung, Göttingen, 2002.

Ehrlich, M., A. Grübl, S. Hartmann, E. Müller, J. Partzsch, S. Scholze, and V. Thanasoulis, Specification of facets/brainscales stage2 host <-> fpga communication, FACETS/BrainScaleS project internal documentation, 2013.

Forouzan, B. A., *TCP/IP Protocol Suite (2nd ed.)*, McGraw-Hill, 2003.

Furber, S. B., D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown, Overview of the SpiNNaker system architecture, *IEEE Transactions on Computers*, *99*(PrePrints), doi:http://doi.ieeecomputersociety.org/10.1109/TC.2012.142, 2012.

ISO/IEC 7498-1:1994, Information Technology — Open Systems Interconnection — Basic Reference Model: The Basic Model, *ISO/IEC 7498-1:1994*, ISO, Geneva, Switzerland, 1994.

Karasenko, V., Design, implementation and testing of a high speed reliable link over an unreliable medium between a host computer and a xilinx virtex5 fpga, Bachelor's thesis (English), University of Heidelberg, 2011.

Krill, B., Round-robin arbiter, http://www.krll.de/en/portfolio/round-robin-arbiter/.

Mead, C. A., *Analog VLSI and Neural Systems*, Addison Wesley, Reading, MA, 1989.

Philipp, S., Generic arq protocol in vhdl, *Internal FACETS documentation.*, 2008a.

Philipp, S., Generic arq protocol in vhdl, Internal Visions presentation, 2008b.

Schemmel, J., K. Meier, and F. Schürmann, A VLSI implementation of an analog neural network suited for genetic algorithms, in *Proceedings of the International Conference on Evolvable Systems ICES 2001*, pp. 50–61, Springer Verlag, 2001.

Schemmel, J., A. Grübl, and S. Millner, Specification of the HICANN microchip, FACETS project internal documentation, 2010.

Schilling, M., A highly efficient transport layer for the connection of neuromorphic hardware systems, Diploma thesis, University of Heidelberg, HD-KIP-10-09, http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=1999, 2010.

The author would like to thank the following people for their support in writing this thesis:

- Prof. Dr. Karlheinz Meier for his amazing organizational and supervising skills which are the corner stone for the success of our work.
- Dr Johannes Schemmel for the technical leadership and his way of harshly criticising poor work but always praising good work which lets us strive to be better at what we do.
- Dr. Andreas Grübl for always trying to make time to listen for my daily struggles despite his overly full schedule. Also for teaching me that while it's better to not make mistakes in the first place, an ugly workaround is sometimes better than pretty, but non-functioning code. Were he not to organize our weekly development meetings with TUD we would have never made the progress we have achieved now.
- Eric Müller for never complaining when I pestered him to fix something software-related that would have taken me a lot longer than him but still also stole some of his time too. He did as much work on the software side of the project as I did on the FPGA side and his immense knowledge of communication protocols and all things IT in general as well as his strive for maximum performance helped me to improve the system as much as we finally did.
- Mihai Petrovici for being a constant annoyance in my life which drives me to be better at everything I do so that he may be quiet for just a minute. His impeccable judgement on the quality of academic writing made me rewrite quite a few pages to try and adhere to his impossible standards. Thank you.

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, February 28, 2014

.....
(signature)