Fakultät für Physik und Astronomie

Ruprecht-Karls-Universität Heidelberg

Masterarbeit

Im Studiengang Technische Informatik

vorgelegt von

Tobias Thommes

geboren in Heidelberg

2018

Design and Implementation of an EXTOLL Network-Interface for the Communication FPGA in the BrainScaleS Neuromorphic Computing System

Die Masterarbeit wurde von Tobias Thommes

ausgeführt am

Institut für Technische Informatik (ZITI)

unter der Betreuung von

Herrn Prof. Ulrich Brüning

Design and Implementation of an EXTOLL Network-Interface for the Communication FPGA in the BrainScaleS Neuromorphic Computing System:

Modern computer chips are constantly becoming more and more efficient, providing more Floating Point Operations per second (FLOPs) by consuming less or equal power than their predecessors. In spite of this trend, the power consumption of large super-computers is still enormous: The number one on the *Top 500 List*, *Sunway TaihuLight* provides up to 125 PFLOP/s and consumes power of around 15 MW. The most efficient Super Computing System on the *Green 500 List* is *Shoubu System B* which provides 842 TFLOP/s while consuming 50 kW.
In contrast to that, the human brain can cope with intelligent operations and thoughts and additionally controls the human body, by only using an amount of about 20 W. The Human Brain Project (HBP) aims to understand by means of Synthesis Biology how this inconceivably efficient system works.
The BrainScaleS system at the Kirchhof-Institute for Physics (KIP) in Heidelberg is part of the HBP and pursues this goal by developing a neuromorphic analog hardware system in combination with a conventional computing cluster.
Up to now the BrainScaleS system is connected via Ethernet over USB 3.0 cables, which not only negatively affects latency and bandwidth, but also results in inefficient cabling density and effort.
This work at hand describes the development of a new network interface for the FPGAs controlling the data communication between the neuromorphic hardware chips and the conventional digital system. The new interface will enable the BrainScaleS system to use the benefits of the EXTOLL network, a high-performance interconnection network, optimised for low latency and high message rates.
This will significantly increase the network-performance of the BrainScaleS system in terms of latency, message-rate and cabling effort.

Entwurf und Implementierung einer EXTOLL Netzwerkschnittstelle für das Kommunikations FPGA im neuromorphen BrainScaleS Computer-System:

Moderne Computer-Chips werden immer effizienter, stellen immer mehr FLOPs zur Verfügung und benötigen dabei weniger oder genau so viel Energie wie ihre Vorgänger. Trotz dieses Trends ist der Energieverbrauch großer Supercomputer immer noch enorm: Die Nummer eins auf der *Top 500 Liste*, *Sunway TaihuLight* leistet 125 PFLOP/s und benötigt dafür eine elektrische Leistung von ca. 15 MW. Das effizienteste Supercomputer-System auf der *Green 500 Liste* ist das *Shoubu System B*. Es leistet 842 TFLOP/s und verbraucht eine Leistung von 50 kW.
Im Gegensatz dazu kann das menschliche Gehirn intelligente Operationen und Gedanken verarbeiten, während es nebenbei den menschlichen Körper steuert. Dazu verbraucht es lediglich etwa 20 W. Das Human Brain Project (HBP) hat sich zum Ziel gesetzt, durch Methoden der synthetischen Biologie zu verstehen, wie dieses unglaublich effiziente System funktioniert.
Das BrainScaleS System am Kirchhof-Institut für Physik an der Universität Heidelberg ist Teil des HBP und verfolgt dieses Ziel durch die Entwicklung eines neuromorphen analogen Hardwaresystems in Verbindung mit einem konventionellen Computer Cluster.
Bisher ist das BrainScaleS System durch Ethernet über USB 3.0 Kabel verbunden. Dies ist nicht nur ungünstig für Latenz und Bandbreite, sondern resultiert auch in einer ineffi-

zienten Kabeldichte und hohem Verkabelungsaufwand.
Die vorliegende Masterarbeit beschreibt die Entwicklung einer neuen Netzwerkschnittstelle für die FPGAs, welche die Daten-Kommunikation zwischen den neuromorphen Hardware-Chips und dem konventionellen digitalen System kontrollieren. Die neue Schnittstelle wird das BrainScaleS System in die Lage versetzten, die Vorteile des EXTOLL Netzwerkes zu nutzen. Dabei handelt es sich um ein hoch-performantes Verbindungsnetzwerk, welches für niedrige Latenz und hohe Nachrichtenraten optimiert wurde. Dadurch wird die Netzwerk-Performanz des BrainScaleS Systems im Bereich der Latenz, der Nachrichten-Rate und des Verkabelungsaufwands signifikant verbessert werden.

Contents

1 Introduction

Modern computer chips are constantly becoming more and more efficient, providing more FLOPs by consuming less or equal power than their predecessors. However, the power consumption per Floating Point Operation (FLOP) is still enormous: The number one on the *Top 500 List*, *Sunway TaihuLight* provides up to 125 PFLOP/s and consumes power of around 15 MW [1]. This is an energy amount of 0.1 nJ FLOP^{-1}. The most efficient Super Computing System on the *Green 500 List* is *Shoubu System B* which provides 842 TFLOP/s while consuming 50 kW [2], which is 0.06 nJ FLOP^{-1}.
These numbers already sound incredibly low for one Floating Point Operation, but in contrast to that, the human brain can cope with intelligent operations and thoughts and additionally controls the human body, by only using an amount of about 20 W. It is difficult to numeralise the human intelligence in terms of FLOPs. However, just the training of an artificial neuronal network on a conventional computing system, for the task of recognising objects in images needs enormous computing power and time compared with a human.
The Human Brain Project (HBP) aims to understand how this inconceivably efficient system of the human brain works. For this purpose, it uses the method of Synthesis Biology [3]. This means that it tries to understand the biological system from the bottom-up direction instead of using the conventional analytic top-down methodology.
With this work we develop a new network interface for the BrainScaleS neuromorphic computing system, which is part of the HBP. Up to now, the communication FPGAs are connected through an Ethernet network using USB 3.0 cables. With the new network interface, developed in this thesis work, the system will be able to use the benefits of EXTOLL, a new high-performance interconnection network, optimised for low latency and high message rates. Using EXTOLL will not only significantly improve latency and bandwidth of the network, but also significantly decrease the cabling density and effort.

1.1 The Human Brain Project

The HBP is beside *Graphene* one of two European FET Flagship projects in the *Horizon 2020* EU Research and Innovation programme. The HBP aims to bring ahead the research fields of computing, neuroscience and medicine related to the human brain. It also includes social and ethical aspects and is therefore a truly interdisciplinary field of research. [4, 5, 6]
The HBP is composed of six scientific platforms and fields of research:

- Neuroinformatics Platform
- Brain Simulation Platform
- High Performance Analytics & Computing (HPAC) Platform

- Medical Informatics Platform
- Neuromorphic Computing Platform
- Neurorobotics Platform

These platforms work together to find answers to the underlying theoretical questions of cognition and ethical reflections. A seventh platform, the *HBP Joint Platform* provides a modern infrastructure and gathers tools and services, developed by the different scientific fields of the HBP. It provides access to this infrastructure for scientists, engineers, developers and students all over the world, having an interest in neuroscience and its applications. [6]
This work at hand refers to the Neuromorphic Computing Platform (NCP). This part of the HBP implements models of biological neuronal networks on digital or analogue hardware-systems. This approach gives the possibility to investigate learning processes and to implement cognitive computing algorithms in an energy-efficient way with fast execution speeds and robustness against local hardware-failures.
There are two different approaches towards these NCP goals: The SpiNNaker system, located in Manchester (UK) and comprising around 500 000 ARM-cores, aims to implement learning algorithms on a massively parallel system which is designed to work at biological real-time. In contrast, the BrainScaleS system, located in Heidelberg (DE), currently implements around 4 Million neurons and 1 Billion synapses on 20 Wafer Modules. The biological structures are modelled as analogue electrical circuits. Communication of these analogue neuromorphic circuits is done via a digital network. [7] The BrainScaleS system, to which this thesis is addressed, is to be described in more detail in the following section.

1.2 The BrainScaleS Neuromorphic Computing System

The BrainScaleS Neuromorphic Computing System (NCS) (shown in Figure 1.1), often also referred to as Neuromorphic Physical Model (NM-PM), mainly consists of two parts: a custom hardware system and a conventional compute cluster. These two parts work together synchronously through digital network communication. The main component of the custom hardware system is the so called Wafer Module. It houses a 20 cm silicon wafer on a large Printed Circuit Board (PCB). The wafer contains 384 identical Application Specific Integrated Circuits (ASICs), manufactured in 180 nm CMOS UMC technology. These ASICs are called *High-Input Count Analog Neuronal Network Chip (HICANN)*. The HICANNs are manufactured in groups of eight, the so called reticles. Each HICANN implements physical models of up to 512 neurons and around 115 000 synapses. A Wafer Module contains 48 reticles with 8 HICANNs each. The current version of the NM-PM system contains 20 of these Wafer Modules and therefore implements up to 4 million neurons and 0.88 billion synapses.
An exploded view of the Wafer Module can be seen in Figure 1.2. The *Wafer* is placed in the middle on a Positioning Mask and connects to the *MainPCB* on top of it. The FPGA Communication PCBs (PCBs) are placed between the *MainPCB* and the *Wafer I/O PCB*s around the *Wafer*.

Figure 1.1: Rendered View of the NM-PM1 system. 1: Wafer Module, 2: Wafer Module network switch, 3: analog readout subsystem, 4: Top-of-Rack (ToR) 40 Gbit network switch, 5: storage server node, 6: computer server node, 7: Wafer Module power supply, 8: top and bottom fan units for Wafer Module. [8, p. 31]

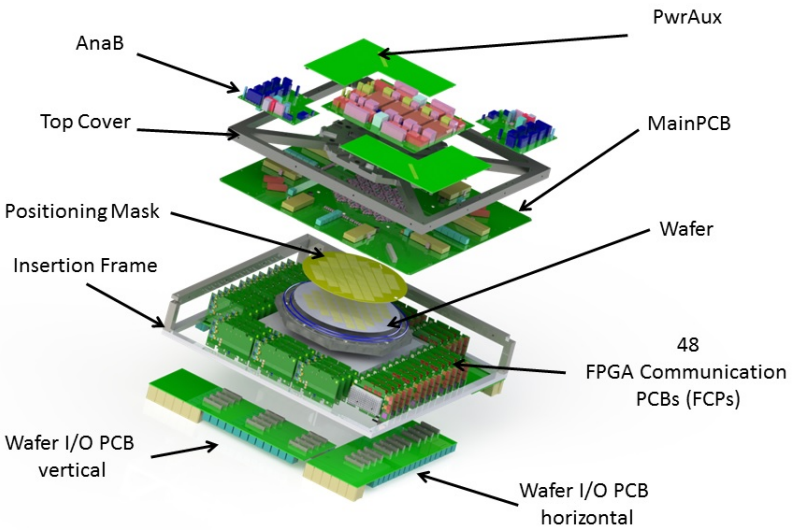


Figure 1.2: Exploded view of the design drawing of the Wafer Module. [8, p. 108]

The system can simulate neural activities with an acceleration-factor of typically 10^4 compared with biological real-time and each modelled neuron can cope with up to 14 336 input synapses. Because the system can emulate neural activities on various scales and in almost arbitrary configurations of neurons and synapses, it is often referred to as *Hybrid Multiscale Facility (HMF)*. Thereby the hybridity refers to the combination of the analog neuromorphic hardware and the conventional compute cluster in one system.
The communication of synaptic spike-events between the HICANN-chips is partly carried out through an on-wafer-network. The communication between different Wafer Modules and with the compute-cluster is managed by 48 FCPs per Wafer Module, i.e. one FCP per reticle. These FCPs are each equipped with a Kintex7 FPGA from Xilinx. The layout of these FPGAs is described in detail in Chapter 2. The FCPs are connected via a 10 Gbit

Ethernet network with the compute-cluster. Besides the FCPs, the Wafer Modules are also connected to Analog Readout Modules (AnaRMs) which digitise the analog membrane voltages in the modelled neurons.
The power supply of the system is controlled by a Raspberry Pi Computer, which is connected to the FCPs via an I2C bus.
The user can access and program the NM-PM hardware using the PyNN API for python. The API hides the mapping and calibration steps from the user. The PyNN API also provides access to software simulators as for example NEST and NEURON. [8, I-1 - I-2]

1.3 The EXTOLL Network

EXTOLL is a new interconnection network for High-Performance Computing (HPC) systems. It is optimised for ultra low latency for message-based communication and provides extremely high hardware message rates. It is designed without network-switching and is highly scalable. The design is implemented in a single chip, which integrates host-interface, Network Interface Controller (NIC) and network functions.[9]

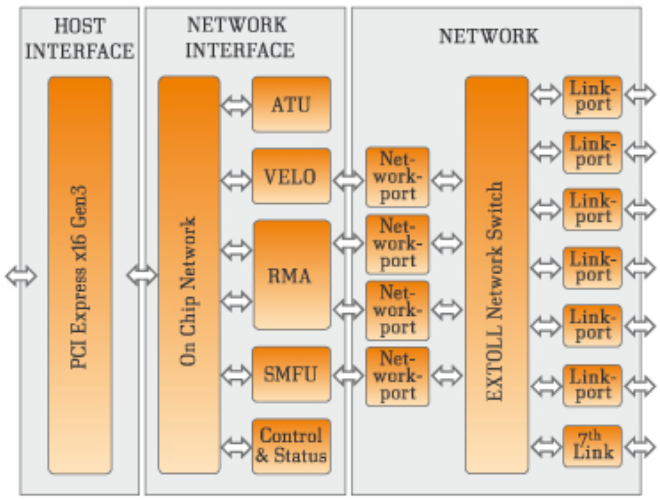


Figure 1.3: Block-Diagram of the EXTOLL Hardware-Architecture [10]

A block-diagram of the EXTOLL Hardware-Architecture is shown in Figure 1.3. It shows the design-structure of an EXTOLL network-chip. On the right side the network-interface is shown with its seven ports. With these ports it is possible to implement arbitrary direct network topologies as for example a 3D-Torus. As a 3D-Torus only uses six ports per node, the seventh Link can be used for an additional connection. The Link-Ports are connected via the *EXTOLL Network Switch* to the internal network-ports. The Architecture includes an RMA-engine for Remote Memory Access as well as the VELO-engine, which provides low latency for small messages between processes. There is also a hardware address translation unit that translates between physical and virtual addresses.
The EXTOLL network supports up to 64 000 nodes with hundreds of processes each. It also supports multicast routing with up to 64 multicast-groups. Packets are always delivered in-order and routing can be adaptive as well as deterministic.[9]
The Remote Memory Access (RMA) unit, which is mainly used in this Thesis, can handle packets with sizes up to 512 B which is referred to as Maximum Transmission Unit

(MTU). All packets are secured by an automatically generate CRC code transmitted with the message. Details about the protocol are presented in Chapter 3.
The EXTOLL network-chip is available on the Tourmalet PCB (see Figure 1.4). The implementation reaches bidirectional bandwidths of up to 8 GB s^{-1} for each link and has hop-latencies of under 60 ns. The board can be integrated into servers and PCs through a standard PCIe x16 Gen3 connector.

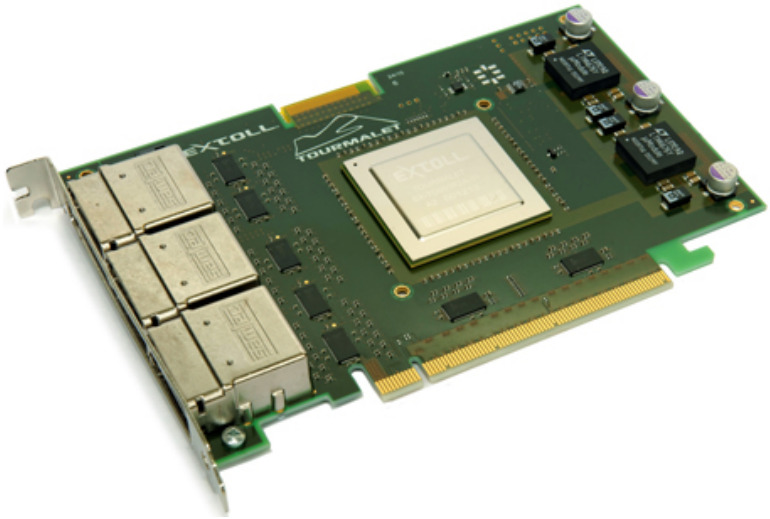


Figure 1.4: Picture of a Tourmalet PCB.[11]

1.4 Using Extoll for BrainScaleS

To use the EXTOLL network for the BrainScaleS system, the FPGAs and also the FCPs and Wafer I/O PCBs have to be made compatible with the EXTOLL Network interface. For this purpose the FPGA design is changed from Ethernet- to EXTOLL-support. This is described in more detail in Chapter 2. The Network-Interface part of the EXTOLL-Architecture (Fig. 1.3) is strictly simplified and included into the FPGA. Only one Networkport and one Linkport are kept and connected directly. The EXTOLL Network-Switch is obsolete in this case, as the FPGAs are network-endpoints in the BrainScaleS system and do not conduct any packet-switching. The Networkport is then interfaced directly to the remaining FPGA-logic by using a newly designed interface module. This module is described in detail in Chapter 4 and is the main subject of this work at hand.
The FCP boards are currently connected to the existing network through USB 3.0 and standard Ethernet plugs and cables. To make them compatible with the EXTOLL network, an adapter cable has to be built. This cable would have to interface several USB plugs to one EXTOLL-cable. As one USB 3.0 cable only contains two Super-Speed differential pairs, forming one communication lane, four USB-plugs are needed to interface the four lanes of one FPGA. In contrast, an EXTOLL cable contains 12 independent lanes. Thus, the four USB 3.0 plugs have to be interfaced to one EXTOLL-cable (see Section 8.1.1).
Figure 1.5 shows the planned network topology. Six Kintex FPGAs are connected together with one concentrator-node which interfaces them to a 3D-Torus of EXTOLL-nodes. Different configurations of concentrator-nodes have been discussed in [13]. It has been concluded that the 6:1 configuration offers the best cost - performance relationship.

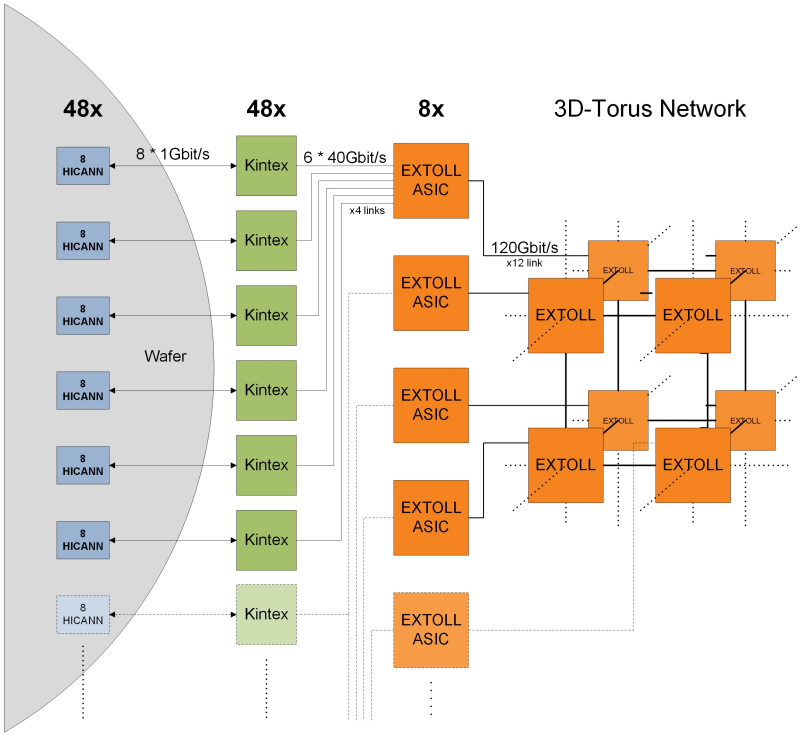


Figure 1.5: Network topology for the BrainScaleS EXTOLL network.[12]

With this topology an amount of 320 EXTOLL ASICs is needed to build the network. At the moment and with the 6:1 topology, the FPGAs have a bandwidth of

$$6[\text{FPGAs}] \times 4[\text{Lanes}] \times 4.2\,\text{Gbps} = 100.8\,\text{Gbps}$$

which exactly matches the EXTOLL bandwidth of

$$12[\text{Lanes}] \times 8.4\,\text{Gbps} = 100.8\,\text{Gbps}$$

In a future hardware setup of the BrainScaleS system it is planned to redesign the FCP boards. On this occasion it is envisioned to directly place the EXTOLL concentrator nodes on the Wafer I/O PCBs. This has some significant advantages. First, this approach avoids the cabling interface between the I/O boards and the EXTOLL network chips. That is advantageous because impedance discontinuities between connectors, cables and PCBs are completely avoided. This significantly improves the signal integrity. Second, this approach is more cost-efficient, because there is no need for Tourmalet cards to host the concentrator nodes. Also fewer EXTOLL cables are needed with this approach. Another advantage is, that the vast number of USB cables (192 per Wafer Module) can be replaced by much less EXTOLL cables (only 8 per Wafer Module).
With these improvements, it will be possible to double the FPGA bandwidth from 4.2 Gbps to 8.4 Gbps. In the current implementation this is not possible due to bad signal integrity over the USB-EXTOLL adapter cables. Unfortunately, then the concentrator nodes can only use half of the FPGA bandwidth, but this might not be a great problem as the FPGAs are not expected to work at full capacity all the time. Short bursts can be buffered by FIFOs in the FPGA-design.

1.5 Outline

After this brief introduction, Chapter 2 will discuss the overall FPGA structure and the changes made in the design to support the EXTOLL network. In Chapter 3, the communication protocol, used by the EXTOLL network is presented and explained in detail. This chapter will also go into details about the implemented communication patterns for registerfile access and Host-FPGA communication. Then, Chapter 4 will explain the implementation and architectural details of the developed network-interface module. Some JTAG controller and chain optimisations are described in Chapter 5. The second half of this thesis starts with the FPGA-Implementation (Chapter 6), followed by the verification (Chapter 7) and testing (Chapter 8) of the developed interface module. Finally a conceptual chapter about the future implementation of pulse-routing is given (Chapter 9) before the conclusion and future work outlook (Chapter 10).

2 FPGA Structure

This chapter presents the current FPGA design (Section 2.1) and the changes introduced during this work (Section 2.2). The purpose of the individual modules is shortly outlined to convey an overview of the BrainScaleS FPGA structure.

2.1 Ethernet Communication Structure

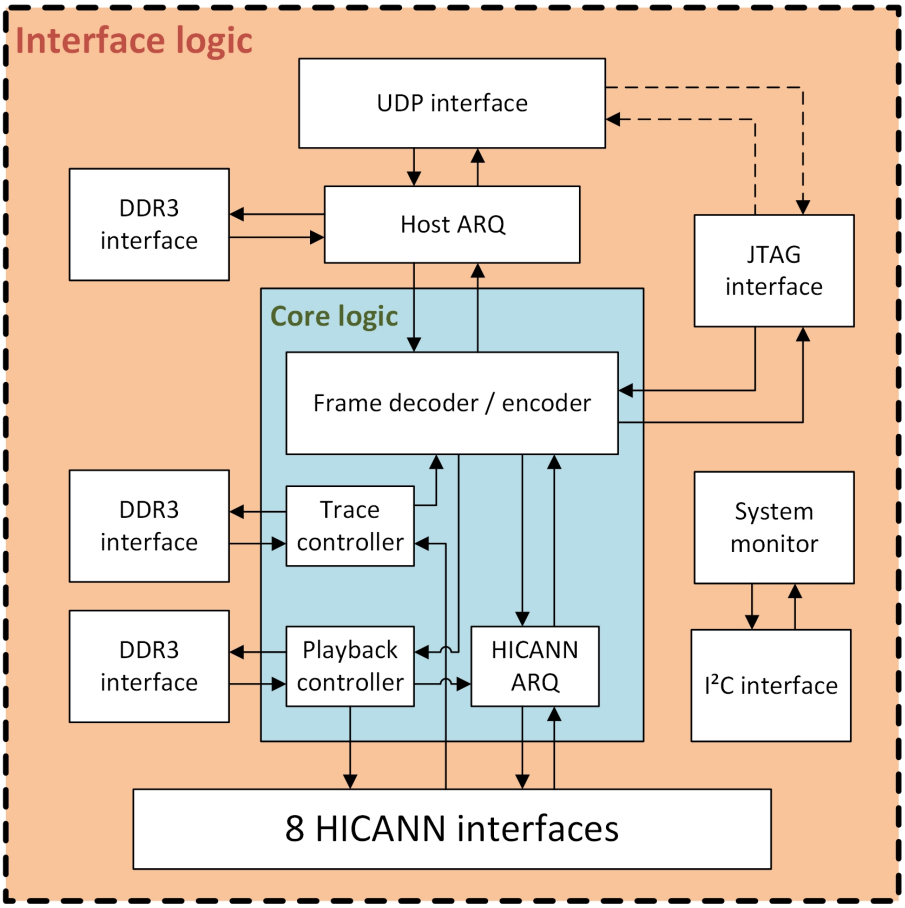


Figure 2.1: Blockdiagram of the communication-FPGA design as given for Ethernet-communication.

The given structure of the BrainScaleS communication-FPGA was designed for Ethernet network communication and is shown in Figure 2.1. This section describes the current design, before EXTOLL is included. The eight HICANN-interfaces are mainly driven by the *Playback controller* for pulse-event communication. The *Playback controller* reads data upon experiment start from a 512 MB DDR3 memory and feeds the HICANN interfaces with the obtained data. Configuration data is communicated via the *HICANN*

ARQ (Automatic Repeat reQuest) module and can be obtained either from the *Playback controller* or directly from the *Frame decoder / encoder*. The ARQ protocol ensures the correct transmission of the configuration packets by requesting a retransmission when a transmission error is detected.
All pulse-events that are received on the HICANN interfaces during an experiment are logged by the *Trace controller* to a second 512 MB DDR3 memory. The Playback memory is filled by the *Playback controller* with data from the network-host and through the *Frame decoder* at the beginning of each experiment. The Trace memory is then read out after the experiments end by the *Trace controller* and sent to the network through the *Frame encoder*.
The *Frame decoder / encoder* module is connected through a FIFO-like interface called the *Application-Layer (AL)-interface* over the *Host ARQ* module to the *UDP interface*, which controls the Ethernet communication. The AL-interface provides bidirectional communication of data-Quad Words (QWs) (64 bit) and an associated type-word (16 bit). At a time the next QW is requested with a *next*-signal and validated through a *valid*-signal in either direction. The *Host ARQ* module is also connected to another DDR3 memory which is of 256 MB size. This memory module serves as frame buffer for the retransmission of corrupted Ethernet packets.
The *UDP interface* also connects a JTAG controller which provides backdoor access to some FPGA- and HICANN registers. The JTAG chain connects one FPGA and eight HICANNs within a reticle. [8, p. 232] It is planned to move the JTAG controller from the UDP interface behind the AL-interface, where the JTAG-packets are as well under control of the Host ARQ module. This will also simplify the structure of the UDP interface, as it is then only connected to the ARQ-module.
Besides all these modules there is also a *System monitor* which is connected via I2C to the outside. The *System monitor* reads the systems temperature and power-supply parameters. The I2C bus connects all FPGAs to a Raspberry Pi computer. [8, p. 125, 142]

2.2 EXTOLL Communication Structure

The EXTOLL communication structure has some changes in comparison with the given Ethernet based design. A block diagram of the design is shown in Figure 2.2.
The basic difference is that the *UDP interface* and the *Host ARQ* module are replaced by the *EXTOLL Link-Port (LP) / Network-Port (NP)* and a special *HBP - EXTOLL interface*, which will be developed in this work at hand. This interface will be called Network HMF Transaction Layer (NHTL) in the further course of this work. All network traffic between host and FPGA will be directed to the core-logic block through the AL-interface between *NHTL* and *Frame decoder / encoder*.
The design is made configurable through a registerfile. This module collects configuration- and status-registers in a dedicated design entity. Hardware designed by the Computer Architecture Group and also the EXTOLL designs use automatically generated registerfiles. These are defined through a simple TCL script and provide two interfaces, one for hardware-access and one for software-access. The HW-interface gives access to the fundamental functions of each register, whereas the SW-interface grants access to the registers through an address. Access permissions for read- and / or write-access can be defined in the TCL-description for both interfaces. The registerfile can also contain other useful features as for example counter-registers with automatic reset or RAM-like addressing

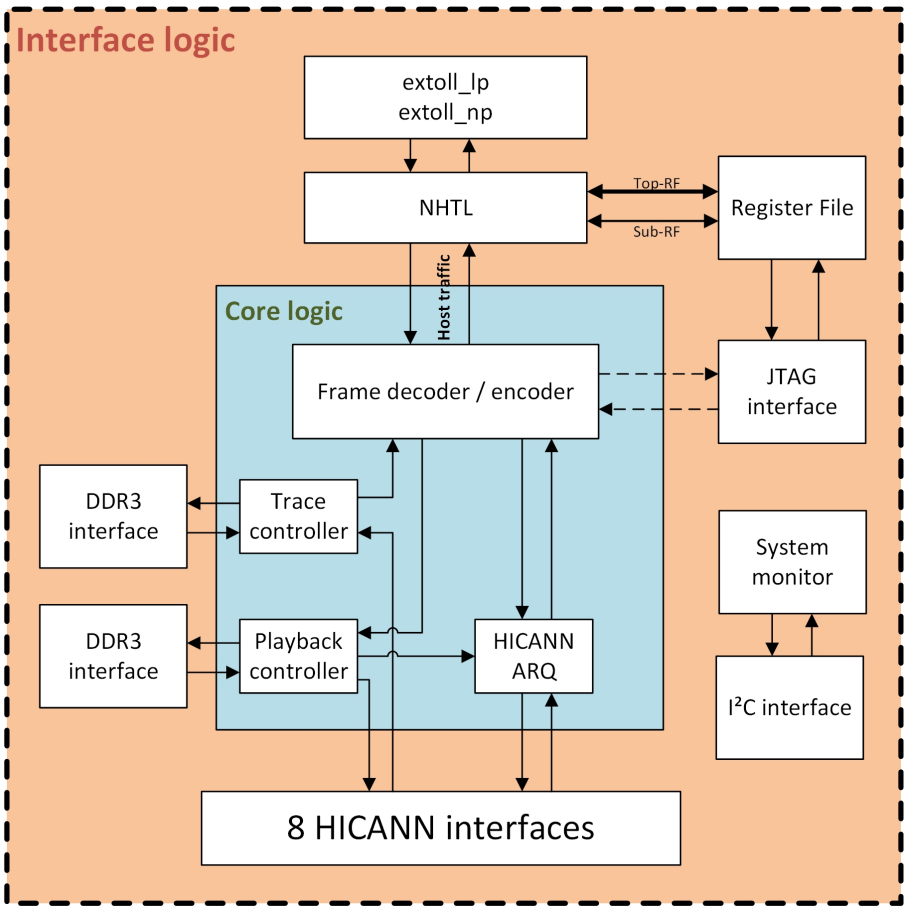


Figure 2.2: Blockdiagram of the communication-FPGA design as designed for EXTOLL-communication.

regions in the registerfile. Several registerfiles can be organized in a hierarchical structure including one top-Register File (RF) and several sub-RFs. The registerfile generator software can be referenced under [14].
The *JTAG controller* can either be placed behind the AL-interface (dashed connections in Figure 2.2) or connected to the EXTOLL registerfile (solid connections in Figure 2.2). Controlling the JTAG module through an EXTOLL registerfile provides significant performance advantages over the current practice of sending JTAG nibbles through the AL-interface. These advantages are described in more detail in Section 5.1.

3 The Communication Protocol

This chapter describes the communication protocol that is used for communicating data and configuration commands between host-computers and the FPGAs. With some modifications this protocol can also be used for the communication among different FPGAs. EXTOLL Packets are always separated into cells of 64 bit size (1 QW). The first cell of each EXTOLL packet is always the Start Of Packet (SOP) header. The bit-structure of this cell is depicted in Figure 3.1. For the meaning of the colour-code see Table 3.1

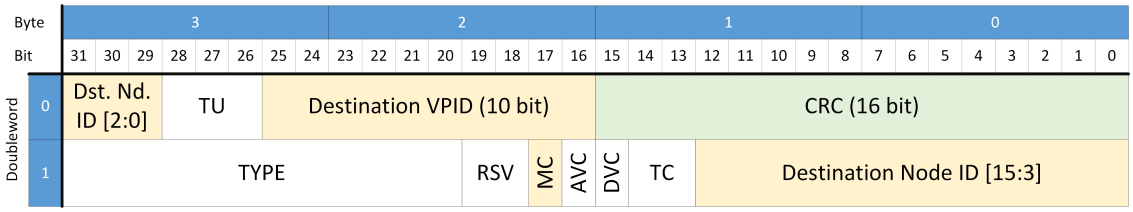


Figure 3.1: Format of an EXTOLL SOP cell

Colour	Meaning
Yellow	The field is used in the NHTL Communication Protocol and has to be filled with correct data.
Grey	The field is used in the NHTL Communication Protocol and contains type-specific information.
White	The field is not used in the NHTL Communication Protocol. The value is either handled *don't care* or is fixed to a standard-value.
Green	The field is automatically filled by the EXTOLL-Network-Port. The value shall not be set by custom-hardware (as e.g. the NHTL).

Table 3.1: Colour-code used for the packet-format figures.

The SOP cell contains the *Destination Node ID* (16 bit) as well as a *Destination Virtual Process ID (VPID)*, which is normally used to identify one out of multiple processes using the RMA network recourse concurrently. The NHTL interface does not use the VPID when receiving packets, but has to set the correct VPID when sending responses to the host. The TU-field (3 bit) identifies the *Target Unit* (Completer, Responder or Excellerate-Unit) which is also not used when receiving packets at the NHTL. TU identifies the host's completer-unit (`3'b001`) when sending responses. The TYPE-field is always set to `12'h004` for RMA packets, identifying the SOP-cell. The setting of this TYPE-field is important as there are also other management-cells, that are internally used in the EXTOLL network. The MC-bit determines whether to perform a multicast or not. In case

of multicast, the Destination Node-ID-field changes its meaning to a multicast-group-ID, identifying a group of nodes instead of a single node. AVC and DVC together determine the use of one out of three *virtual channels* in the EXTOLL network. For the use in the NHTL, they are fixed to `2'b00`. The Traffic Class (TC) identifies independent data streams in the EXTOLL network, preserving the ordering. It is not used in the context of the NHTL and therefore fixed to `2'b00`. Finally the EXTOLL NP will calculate and insert a Cyclic Redundancy Checksum (CRC) (16 bit) to ensure correct transmission of the SOP-cell. RSV denotes reserved bits, that should always be kept `2'b00`. [15, p. 24 ff.]
Behind the SOP-cell, the EXTOLL-packets generally transport the payload-data-cells, followed by a closing End Of Packet (EOP) cell. This cell contains an automatically generated packet-CRC for error-correction. Generally the number of payload-data-cells is limited by the MTU, which has a size of 512 B.

3.1 RMA Protocol

The RMA protocol was originally defined for Remote Memory Access (RMA) transactions. These are normally used for communication between several equitable hosts. In this context it is used for the communication between a master-host and several slave-FPGAs and later also for the neighbour traffic between FPGAs (Chapter 9).

3.1.1 RMA Packet-Types

The RMA-Protocol defines ten different packet-types [15, p. 11]. In the context of this work only six of them are used. These six commands are listed in Table 3.2 together with their specific use. The overall packet-format for each of these types is shown in Figure 3.2.
As Figure 3.2 shows, the RMA_PUT_QW command is limited by the MTU. By taking into account the 16 B of header information, this results in a maximum payload size of 496 B (62 QW). Registerfile access packets and notifications are not limited by the MTU because they are of constant size.

3.1.2 RMA Protocol Format

For the RMA protocol, a network-descriptor is sent after the SOP-cell (for the general form see Figure 3.3). Depending on the Packet-Type, the network-descriptor consists of one up to three 64 bit cells and can also contain send- and write- memory-addresses following the general header-information.
The header contains several fields referring to the source of the packet. These are the *Source Node-ID* (16 bit) and the *Source VPID* (8 bit). The *Protection Domain ID (PDID)* (16 bit) is, like the VPID, not used when decoding incoming packets at the NHTL. The two *Error* bits indicate whether an error occurred in the network hierarchy with this packet. The *Command*-field determines the Packet-Type and is one of the most important fields in the header. The two *Notification* bits tell the Completer and / or Responder unit to create a notification-response packet (see Section 3.2.1). There are also six *Modifier* bits: the Remote Registerfile Access (RRA), the Interrupt (INT), Translate Enable (TE) to enable the Address-Translation-Unit (ATU), the Excellerate Write Access (EWA), the

Name	Encoding	Description
RMA_PUT_QW	4'b0011	Used for data communication between Host and FPGA (both directions) and for neighbour-traffic between FPGAs.
RMA_PUT_BYTE	4'b0010	Used for Registerfile access write from Host to FPGA.
RMA_PUT_IMM	4'b0110	Used for short (1 QW) data communication from host to FPGA. This packet type avoids the invocation of the DMA-engine when a packet is sent.
RMA_PUT_NOTI	4'b0101	Used for sending notifications between FPGA and Host (both directions).
RMA_GET_BYTE	4'b0000	Used for Registerfile access read request from Host to FPGA.
RMA_GET_BYTE_RSP	4'b1010	Used for Registerfile access read responses from FPGA to Host.

Table 3.2: RMA-Commands and their usage.

Notification Replicate (NTR) and the Excellerate Read Access (ERA). From these six bits only the RRA and TE bits are used in the context of this work. [15, p. 27 ff.]

3.2 Registerfile Access

When the registerfile is to be accessed via RMA_PUT_BYTE or RMA_GET_BYTE commands, the RRA modifier-bit has to be set in the RMA-descriptor.

3.2.1 Write Access

A registerfile entry can be written, using the RMA_PUT_BYTE command. The format of its network descriptor is shown in Figure 3.4. The *Source Node ID* defines the network-address of the requesting host. *PDID*, *Error* and *MODE[5:1]* are ignored as well as the responder-notification request *NOTI[0]*. The RRA bit has to be set as for all registerfile accesses. The *Payload Size* generally represents the number of actually attached bytes of payload minus one. In this case the payload-size has to be exactly 8 bytes (*Payload Size = 3'b111*) because the registerfile has a width of 64 bit. The next QW following the network-descriptor-header defines the registerfile-address to which the payload shall be written.
If the completer-notification request bit *NOTI[1]* is set, the FPGA-Logic will generate a Notification PUT request in response (see Figure 3.5). In this descriptor the *Source Node ID* represents the network-address of the sending HMF-FPGA. In the corresponding SOP-cell, the *Destination Node ID* as well as the *Destination VPID* are copied from the previously received header from which this notification has been requested. The *NOTI[1]*

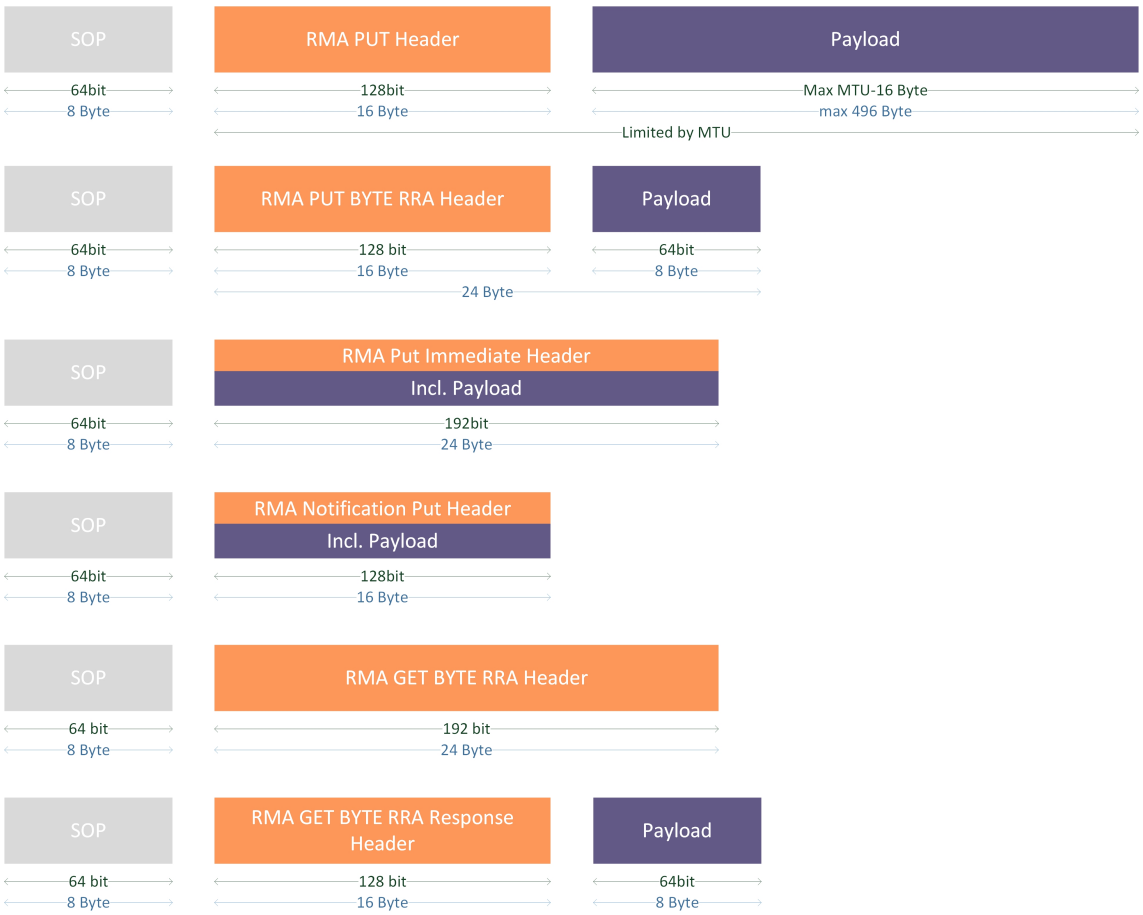


Figure 3.2: Overall packet format of the used RMA types.

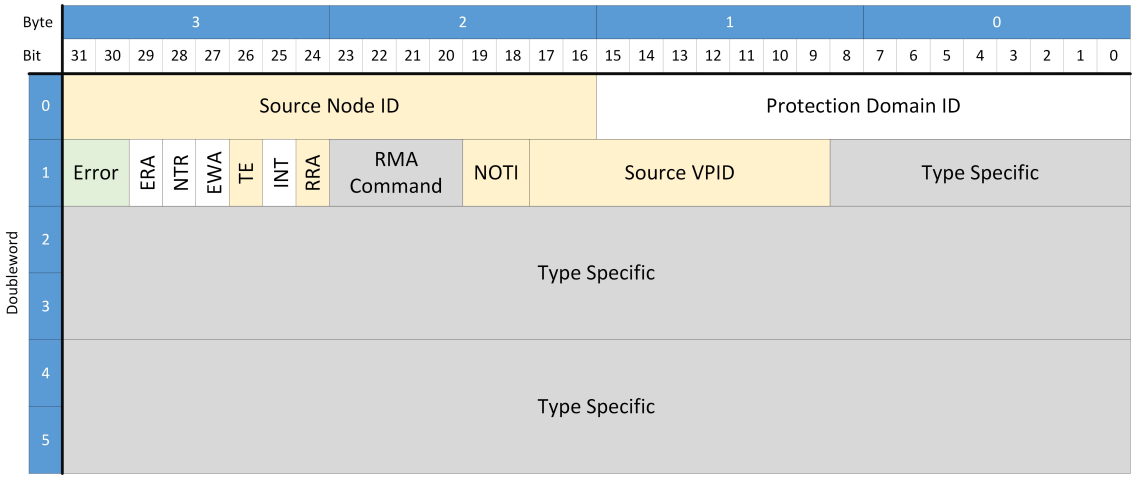


Figure 3.3: General format of a network descriptor header

bit in the NOTI_PUT descriptor is also copied as well as the *Source VPID*, which is copied to the *Destination VPID*.

The *Payload* of the Notification is not used here.

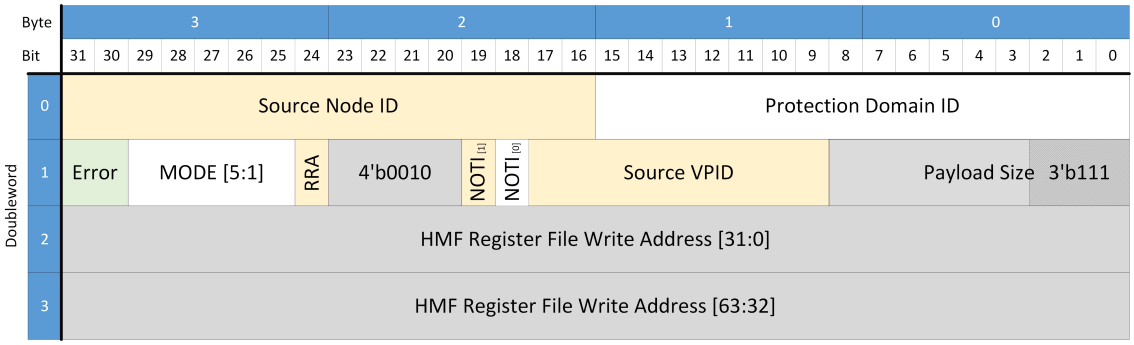


Figure 3.4: Registerfile PUT request network descriptor format

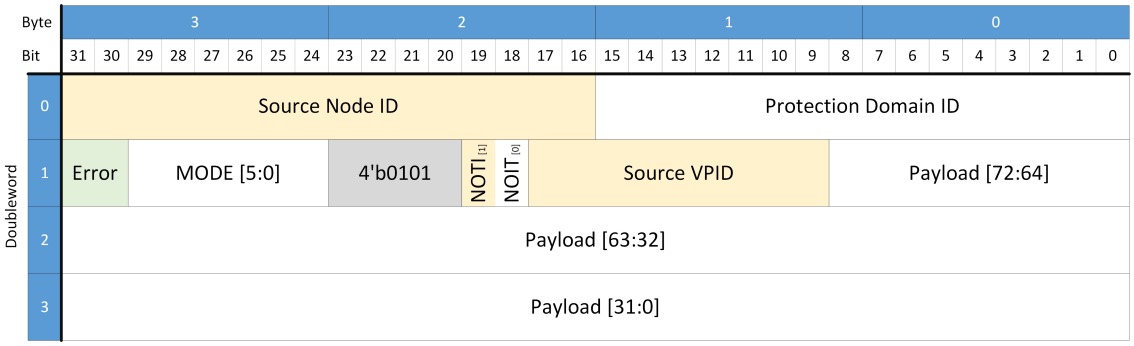


Figure 3.5: Notification PUT request network descriptor format

3.2.2 Read Access

For Read-Access on the registerfile, the host sends an RMA_GET_BYTE request with the RRA-bit set and the payload-size to be requested exactly equals 8 bytes (*Payload Size = 3'b111*) (see Figure 3.6). If the responder-notification request bit is set, a NOTI_PUT packet is triggered as response in the FPGA-logic as described in Section 3.2.1.

Upon reception of the RMA_GET_BYTE request packet with RRA-bit set, the FPGA-logic reads out the requested registerfile address and generates a corresponding RMA_-GET_BYTE_RESP packet containing the answer for the host (see Figure 3.7). The destination fields in the SOP-cell are copied from the previously received RMA_GET_BYTE request as well as the *PDID*, the *Translate Enable* bit and the *NOTI[1]* bit. If the *NOTI[1]* bit was set, the RMA_GET_BYTE__RESP packet triggers a notification in the host. The *HMF Write Address*, which is the destination address in the host memory, is also copied from the respective field in the RMA_GET_BYTE request descriptor.

3.3 Host - FPGA Communication

The communication between the host and the FPGA core-logic has to be driven across the application-layer interface at the *Frame decoder / encoder*. This interface is very similar to a FIFO-interface and communicates 64 bit of payload-data along with 16 bit of type-information (*payload-type*). (See Section 2.1)

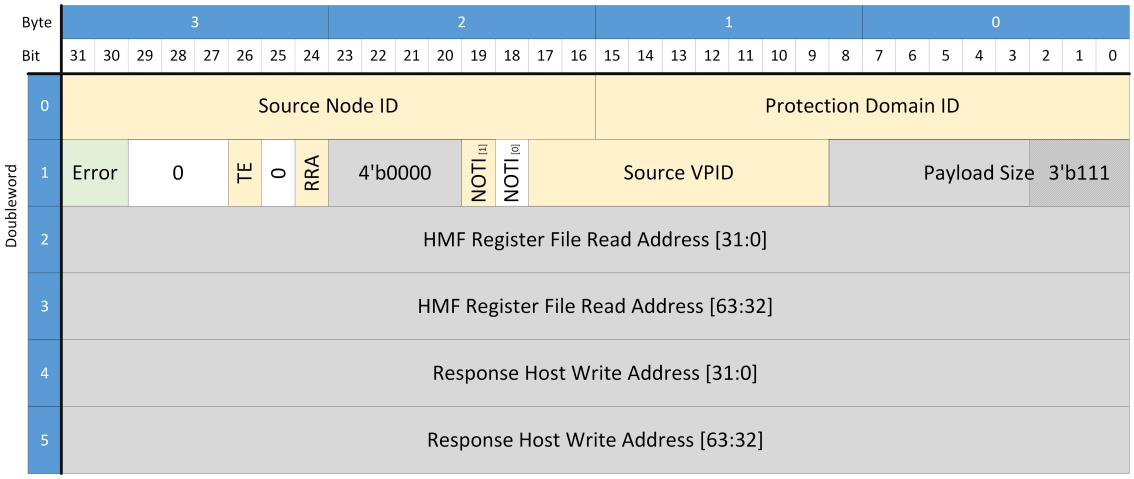


Figure 3.6: Registerfile GET request network descriptor format

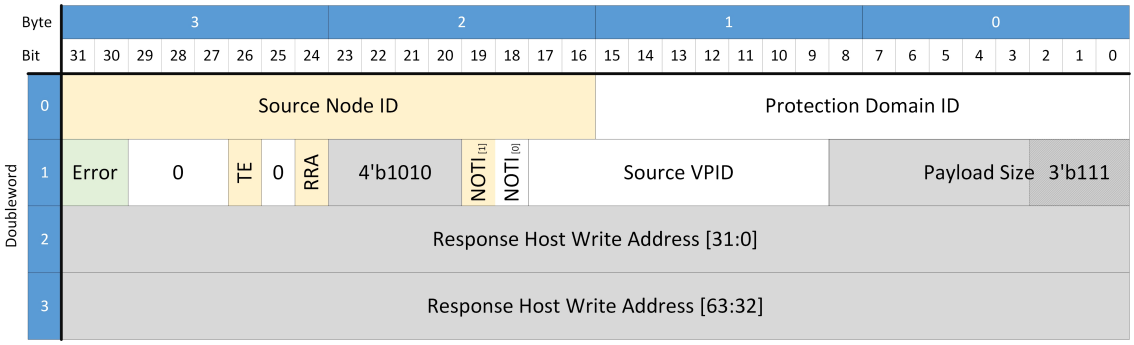


Figure 3.7: Registerfile GET response network descriptor format

3.3.1 Payload Types

There are five (four if JTAG Access is omitted) different payload-types which are listed in Table 3.3.

Hex.-Code	Type	Direction
0x0C5A	FPGA Playback Data	Host → FPGA
0x0CA5	FPGA Trace/Pulse	Host ← FPGA
0x0C1B	FPGA Configuration	Host ↔ FPGA
0x2A1B	HICANN Configuration	Host ↔ FPGA
0x06A4	[JTAG Access]	Host ↔ FPGA

Table 3.3: Application Layer Payload Types

FPGA Playback Data describes initialisation data containing spike-events for certain synapses on the underlying HICANN-chips and configuration commands for the HICANNs. These data are forwarded to the DDR3 Playback-Memory on the FCP where they are kept until the experiment starts. *FPGA Playback Data* are only communicated in the direction from host to FPGA.
FPGA Trace/Pulse describes the experiment-results data. It contains tracked spike-events and configuration responses from the HICANNs. Currently these data are gathered in the

DDR3 Trace-Memory on the FCP until the experiment ends. When the experiment is finished, the Trace-Memory is read out and sent altogether to the host. In future versions of the HMF-FPGA design these data will directly be sent to the host as they occur at the HICANN interfaces. *FPGA Trace/Pulse* data is only communicated in the direction from FPGA to host.
FPGA Configuration packets carry, as the name already tells, configuration commands for the HMF-FPGA core-logic. In the newest version of the HMF-FPGA design these commands always generate a response packet, delivering acknowledgement information back to the host-software, that the configuration has been successful. Accordingly, this packet type can occur in both directions between the host and the NHTL interface logic.
HICANN Configuration packets carry, in analogy to the *FPGA Configuration* packets, configuration commands for accessing the internal registerfile of the underlying HICANN-ASICs. They can be distinguished into read- and write-requests and also produce answers back to the host-software. Hence this packet type can also occur in both directions between the host and the NHTL logic.
JTAG Access packets deliver up to now JTAG nibble data for controlling the Joint Test Action Group controller module, that was formerly directly connected to the Ethernet interface. The responses back to the host contain the monitored TDI-bits from the JTAG chain. The current HMF-FPGA design implements the JTAG-controller directly behind the Frame-Decoder/Encoder, which is why this packet type was defined. In the EXTOLL implementation, which is under development with this work, the JTAG controller will directly be connected to an EXTOLL registerfile and can be accessed through RRA-commands over the EXTOLL network. Consequently this packet type will become obsolete in the future and is therefore considered as deprecated throughout this work.

3.3.2 Direction Host to FPGA

By sending RMA_PUT_QW or RMA_PUT_IMM packets to the FPGA, the AL-interface can be fed with all supported types of data, shown in Table 3.3. These packets are recognised by the nhtl_completer module and directly forwarded to the application-layer interface through an asynchronous FIFO (to cross the clk-domain border between EXTOLL and HMF).
Playback-data packets are shifted into the playback memory by the FPGA core-logic and kept there to feed the experiment with initialisation data or to feed the HICANNs with configuration requests. FPGA-config and HICANN-config packets, consisting of 1 QW each produce an answer also of 64 bit length. These answers are generated by the HMF-FPGA-core logic or the HICANN-chips themselves respectively. By sending special FPGA-config packets, the neural experiment can be started or stopped. One of these FPGA-config packets can request the core-logic to read out all the traced data from the trace-memory and push the traced events through the application-layer interface towards the network.

Playback Data

For communication of *FPGA Playback Data* from the host to the FPGA, the host sends RMA_PUT_QW packets. The header format of this packet-type is shown in Figure 3.8. The *Payload Size* has to be a multiple of 8 B and is always the maximum payload size of 62 QW until the last packet is transferred, which is usually less than 62 QW.

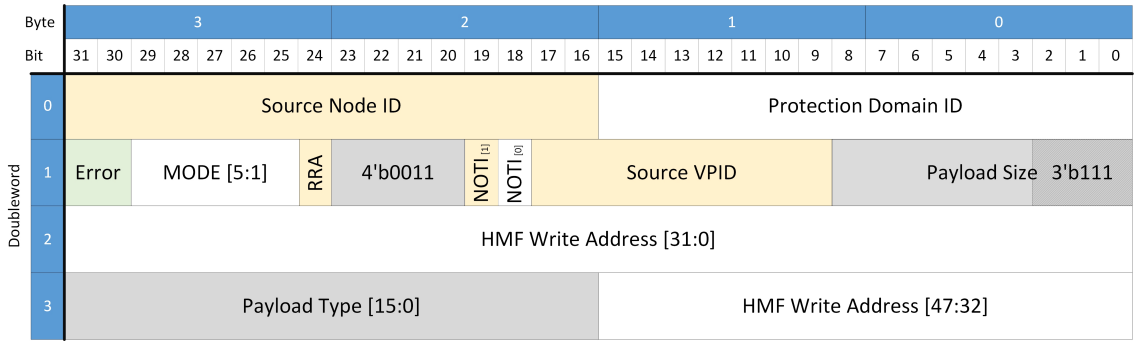


Figure 3.8: Host to FPGA PUT-QW network descriptor format

Configuration Data

Shorter payload types like configuration commands (FPGA and HICANN) are sent using RMA_PUT_IMM packets. These provide the advantage of avoiding the DMA-access by sending the 64 bit of payload data directly with the packet-header. Thereby the network performance can be increased. The header format of RMA_PUT_IMM packets is shown in Figure 3.9.

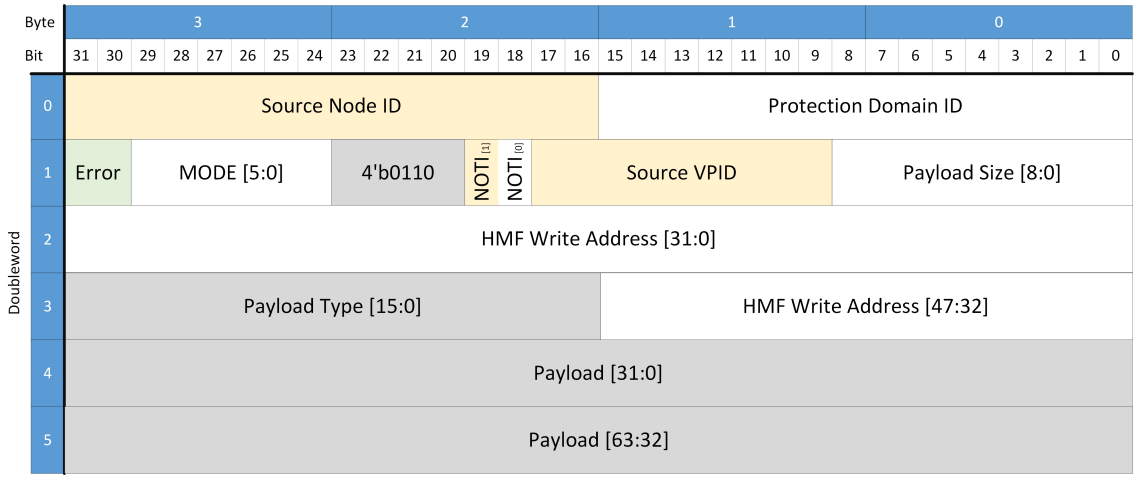


Figure 3.9: Host to FPGA PUT-IMM network descriptor format

For both, playback- and configuration-data the *Payload Type* is encoded in the 16 uppermost bits of the *HMF Write Address* which is not used otherwise when communicating towards the FPGA.
If the completer-notification bit (*NOTI[1]*) is set, the NHTL logic triggers the generation of an RMA_PUT_NOTI packet in the direction back to the host. The payload-field of the notification packet is left empty. The *Source Node ID* and the *Source VPID* as well as the *PDID* of the requesting RMA_PUT_QW header are copied to the respective fields of the notification SOP-cell and network descriptor (see Figure 3.5 in Section 3.2.1).

3.3.3 Direction FPGA to Host

For sending back data from the FPGA to the host, a very similar network descriptor as in Section 3.3.2 is used. The header format of these packets is shown in Figure 3.10. The

Payload Size again is a multiple of 8 B and is set to maximum size until the last packet is transferred. The last payload-QW which marks the end of traced pulses always contains the word `0xE11D`. This is called the End Of Trace (EOT) marker.
When the host is used as destination node, the *Payload Type* cannot be directly encoded into the address field. This is because the host-software strictly interprets the 64 bit field as memory address (either as physical if TE is set or as logical otherwise).

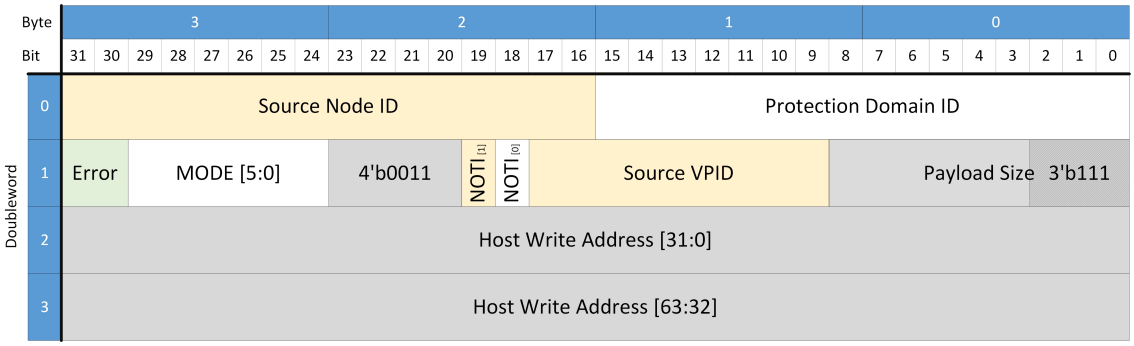


Figure 3.10: FPGA to Host PUT-QW network descriptor format

To solve this problem, there are two different possibilities which shall be evaluated.
One solution would be to request the transmission of a given payload type explicitly by sending a GET-QW command from the host to the FPGA. In this case, the GET-QW header would have to contain the exact number of data-bytes to request from the FPGA. However this information cannot easily be derived at the host, as it is a priori unknown to software, how much data is being contained in the trace-memory. The software cannot know how many packets to request and how many bytes the last packet shall contain.
The second and preferred solution to the problem is to configure the header information including a host memory address for each payload type into the FPGA register-file. Of the five possible payload types only the Trace-Data (`0x0CA5`) and FPGA-/HICANN-Configurations (`0x0C1B` / `0x2A1B`) (and also JTAG (`0x06A4`)) appear on the Application-Layer-Interface directing to the host. So we configure the hosts *Node ID*, *PDID*, *VPID* and one write-address for each occurring type into the registerfile. The JTAG-Responses can be omitted because the JTAG-controller will be reimplemented to be controlled directly through a registerfile (see Section 5.1).
The AL-interface does not deliver information about the exact number of payload QWs that are to be sent to the host. Nonetheless this is not a big problem to the solution, because the EOT-marker signals when the transfer is complete. This can be used to determine the *Payload Size*, as described in Section 4.1.
Trace-Data are always sent back to the host with maximum payload-size of 62 QW as opposed to configuration responses, which are sent back separately as single QW payload. This handling saves the logic, required to aggregate the configuration responses to bigger packets as for trace-data responses. Usually configuration packets are not sent in high numbers, so this simplification does not have a big impact on performance. In contrary, it would be inefficient to wait until enough configuration responses have been generated to fill a whole packet. Furthermore this would create the possibility of deadlocks, when the logic waits for configuration responses to fill a packet, but blocks the AL-interface because the current data-item is for example of trace-data type. This would have to be solved by introducing a timeout which would significantly impact on latency.

Trace-Data and HICANN-Configuration responses are sent directly to the host. They are being notified from time to time as described in Section 3.3.4 while FPGA-Configuration responses are sent to the host with the completer-notification-bit set to true, telling the host to insert the packet header into its notification queue. This is because FPGA-Configurations are mostly sent as single commands whereas HICANN-Configurations are also sent in bulk (around 100 commands) before an experiment-start. Notifying every single response would then put excessive load on the notification queue in the host.

3.3.4 Payload Notifications

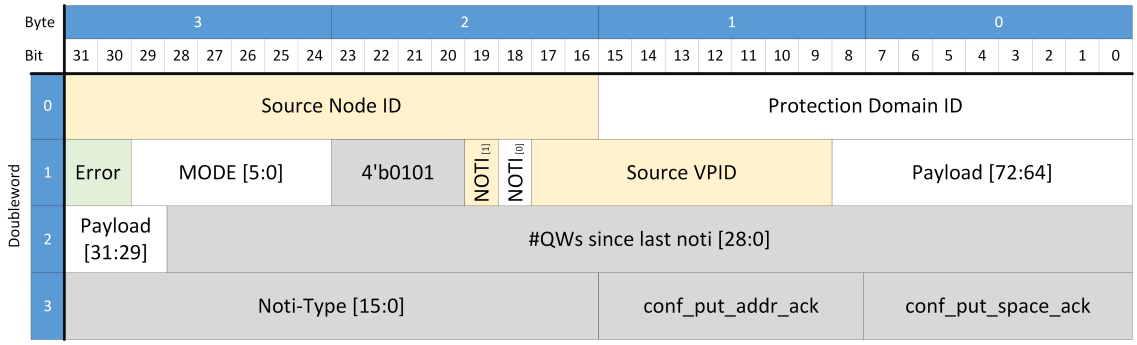


Figure 3.11: FPGA to Host notification format

As described in the Section before, the NHTL logic notifies the host software from time to time about the number of trace-data- and HICANN-configuration packets sent. A trace-notification is sent either when an EOT-marker is detected, or if the number of packets sent to the host exceeds a configured limit, or in case of low traffic when a timeout-counter asserts a configured limit of clock-cycles. HICANN-configuration notifications are sent with a similar condition except for the EOT-marker, which is not applicable to configuration responses.

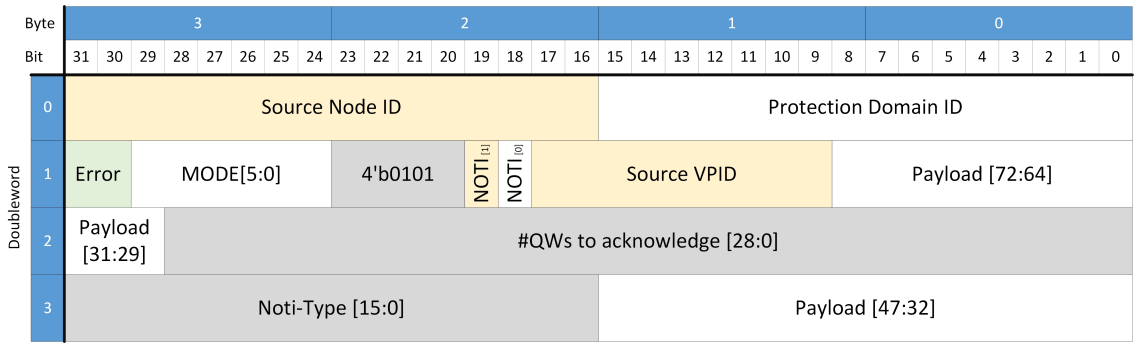


Figure 3.12: Host to FPGA notification format

These notification packets contain the number of QWs, currently sent to the host. For trace-notifications, this number will mostly be the configured limit number of packets times the maximum payload size of 62 QW. It can be less than that in case of EOT or timeout triggering the notification. For HICANN-configuration notifications this number always equals the number of sent packets, as these are always of 1 QW length. Beside the number of QWs, the notification packets also contain the number of successful attempts to

configure the put-address and host buffer-space in the registerfile for the according type. This enables the host software always to identify the correct memory-location where the notified data can be accessed. Furthermore they are tagged with the respective type, they refer to. The packet-structure of these notifications is shown in Figure 3.11.
The host software on its part must also send notifications to the NHTL logic from time to time, acknowledging the number of QWs processed from the memory buffer. The host can at the earliest send a notification after a respective notification packet from the FPGA has arrived and the thereby notified data has been processed. These notifications free the host-memory space, used in form of a ring-buffer, for continued use by the NHTL module. They are also tagged with the payload-type which they refer to. The packet-structure of this type of notification is shown in Figure 3.12. For more information on the purpose of these notifications and the buffer control see Section 4.5.

4 The NHTL-Architecture & Implementation

This chapter describes the architectural and implementation details of the Network HMF Transaction Layer (NHTL)-interface module, developed throughout this work. Section 4.1 introduces the interface-signals and signal-groups of the NHTL main-module and continues giving an overview of the submodules and data-paths through the NHTL-interface. The control-structures for these data-paths are also described in detail. The following sections then go into the details of the NHTL sub-modules, namely the NHTL-completer (Section 4.2), the NHTL-responder (Section 4.4) and the NHTL Ringbuffer Controller (Section 4.5). Section 4.3 describes the Remote Registerfile Access (RRA) architecture and goes into detail about the different configuration registers in the NHTL-interface. Last but not least, a short section about the different clock-domains and -frequencies is given in Section 4.6.

4.1 NHTL-Top

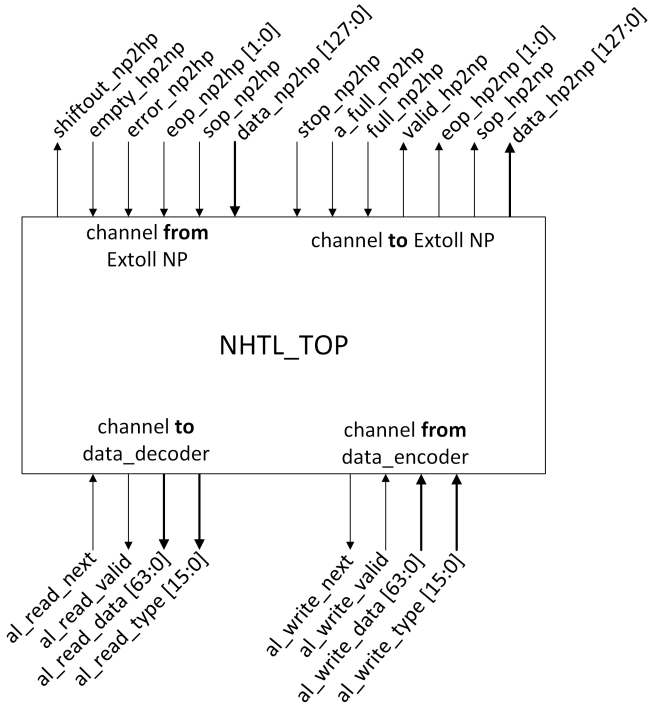


Figure 4.1: Interface block diagram of the NHTL-Top module. The Registerfile interface, shown in Figure 2.2 is not included here for the sake of simplicity.

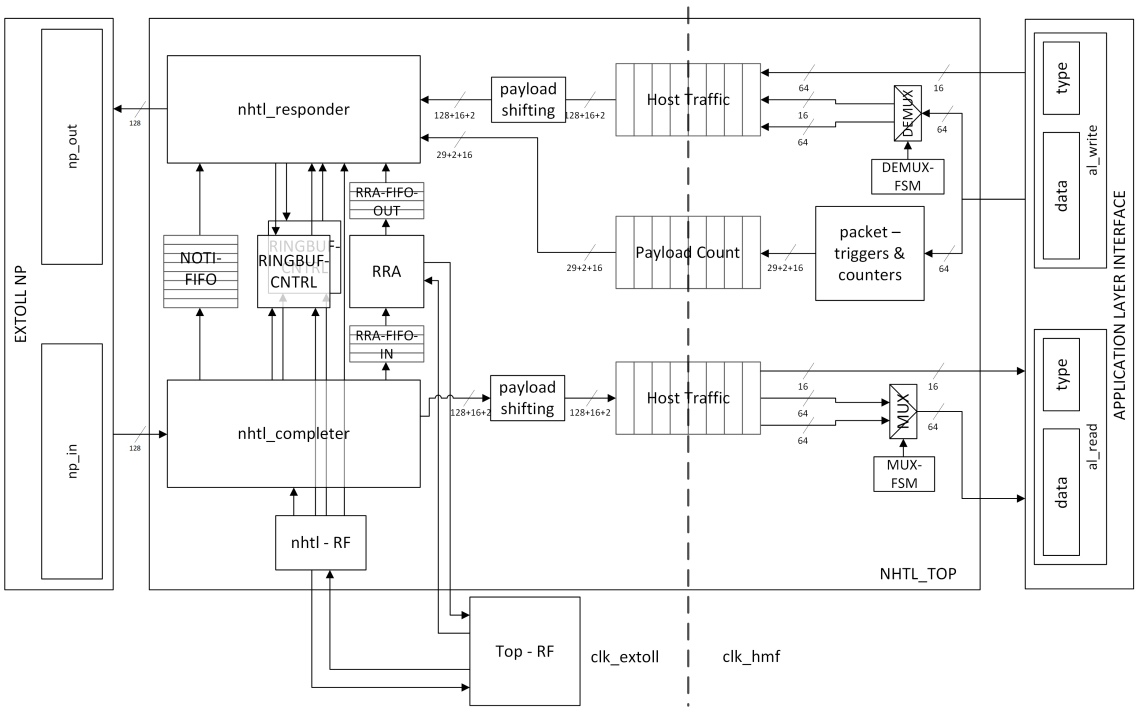


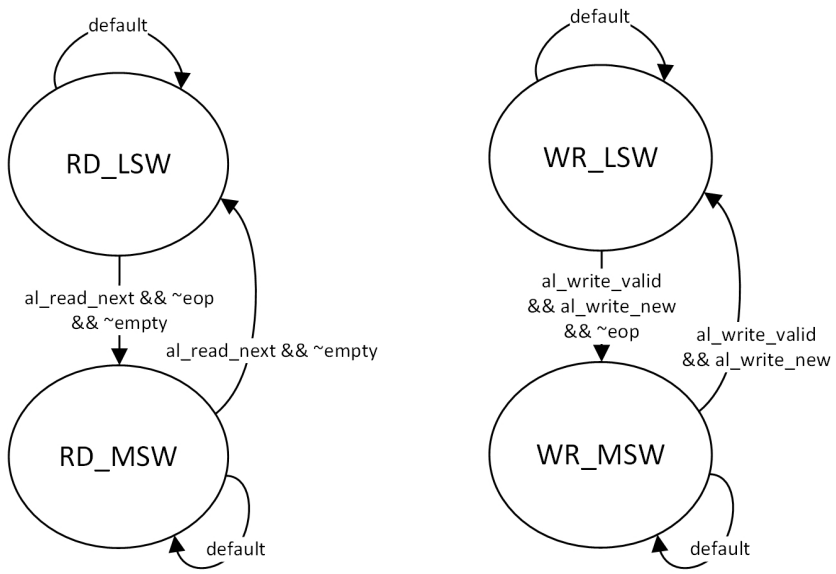
Figure 4.2: Block diagram of the overall NHTL module and its interfaces.

The general interface structure of the EXTOLL interface module, which is also called Network HMF Transaction Layer (NHTL) is shown in Figure 4.1. There are two major interfaces.
The Network-Port (NP) interface transports the network-packets to and from the EXTOLL NP. The interface is FIFO-like, i.e. it provides `empty-`, `full-`, `shift-out-` and `valid-`signals. The start of each network-packet is marked with an `SOP-`signal and the end of each packet is marked with an `EOP-`signal. The `EOP-`signal consists of two bits for being able to distinguish, whether the first or the second QW in the 128 bit data bus is the End Of Packet (EOP). The EOP signal tells the EXTOLL NP to generate an EOP cell marking the end of the network transaction. On the sending side of the interface, the `stop-`signal indicates, that the NHTL shall finish the current packet-transmission, but must not start a new one until the signal is again deasserted. The `error-`signal indicates an erroneous data-transfer since the last `SOP-`signal and is not used in the NHTL implementation. [16]
The NHTL is also connected to the AL-interface of the HMF-core-logic. This interface is also very FIFO-like, but implements a `next-`signal in both directions in addition to a `valid-`signal. Asserting the `next-`signal without `valid` asserted is not allowed. The data is divided into two buses. One transports the main payload of 64 bit, i.e. one QW. The additional data-bus provides a 16 bit data type. The possible types that can occur, are listed in Section 3.3.1 (Table 3.3). The AL-interface as well as the EXTOLL NP-interface are both bidirectional, i.e. the interfaces are each implemented once per direction. While the NP-interface denotes the direction directly in the signal-names, the AL-interface has two sub-interfaces called AL-read and AL-write. The former one *reads* data from the network while the latter one *writes* to the network. [8, p. 155]
The architecture of the NHTL-Top module is shown as block-diagram in Figure 4.2. Incoming packets from the network are processed in the NHTL-Completer submodule first.

They are sorted into different FIFOs for further handling. Registerfile accesses are shifted into the RRA-FIFO between the completer and the RRA-module. If there is a notification request bit in the packets network-descriptor, a notification-request is generated and inserted into the NOTI-FIFO. Notifications from the host are communicated to the ring-buffer-control modules to let them decrement their filling-level.
All other packets containing Host-FPGA communication payload for the HMF core-logic are shifted into an asynchronous FIFO across the clock-border, together with their type-information. Before the data is shifted into this asynchronous FIFO, the Least Significant Word (LSW) (1 QW) which is acquired together with the network descriptor has to be delayed for one cycle. This is to ensure that the LSW is shifted into the FIFO together with the Most Significant Word (MSW) (1 QW) which is acquired one cycle later, together with the next LSW. In Figure 4.2 this is depicted as "payload shifting".
Behind the clock-domain-border, the 128 bit wide data from the FIFO (LSW and MSW) have to be multiplexed onto the AL-interface, which only takes 64 bit in one clock-cycle. The multiplexer is controlled by a minimal two-state-FSM, which is shown in Figure 4.3a. It alternately reads the LSW and the MSW from the FIFO (The FIFO-format is shown in Figure 4.4 in the next Section). If the LSW has an eop-marker, the FSM waits in the RD_LSW state because in this case the MSW would not be valid. In both states the FSM waits until the AL-interface requests the next data-word. The FIFO is told to shift-out in the state RD_MSW or in case of eop also in the state RD_LSW.

Data from the AL-interface directing to the network is demultiplexed to a second asynchronous FIFO across the clock-domain-border, also together with the 16 bit type information. This demultiplexer is also controlled by a minimal two-state FSM, which is shown in Figure 4.3b. The FSM waits in WR_LSW state if there is an eop. It also waits in any state if the AL interface is not valid and new. The interface-value is said to be new when a next value is being requested. It is not new any more when a new value has been valid for one clock-cycle. The FIFO is told to shift-in always in the state WR_MSW or in case of eop also in state WR_LSW. Again this is constrained to a new and valid value.

The QWs arriving at the AL-interface are carefully monitored with different counters and triggers to define the completeness of outgoing packets and to generate notifications for the host. The problem of the unknown payload-size at the AL-interface, described in Section 3.3.3 is solved here by counting the payload words when inserting them into the asynchronous FIFO towards the NHTL-Responder. When the maximum payload-size is reached or an EOT marker is detected in the payload, the counter will be cleared and the counted number of payload words is inserted to another asynchronous FIFO across the clock-domain-border.
This FIFO can be used by three different packet-types, which can in principle occur independently of each other. These are, according to the Sections 3.3.3 and 3.3.4, AL-write-data packets and HICANN-configuration-response- or trace-data-notifications. To resolve this access conflict, an arbiter is instantiated, that decides with given priorities, which of the requested packet-types is inserted. The AL-interface has to be stalled always, when a request is pending at the arbiter to prevent race-conditions.
Back in the EXTOLL-clock-domain, now the MSW has to be delayed, as the first LSW will be sent to the network together with the packet-header. The NHTL-Responder submodule processes the packet-information from the FIFOs, including the NOTI-FIFO, the RRA-output-FIFO and the asynchronous FIFOs containing data and packet-information from the core-logic.



(a) state diagram of the MUX-FSM (b) state diagram of the DEMUX-FSM

Figure 4.3: MUX- / DEMUX-logic FSMs

The ring-buffer-control submodules provide information on whether the host has enough memory space allocated to cash the network data. If this is not the case, the packet is stalled and waits for the host to notify more space.

4.1.1 FIFO-Formats

The asynchronous FIFOs instantiated in the NHTL-Top module are designed with the data format shown in the Figures 4.4 and 4.5.

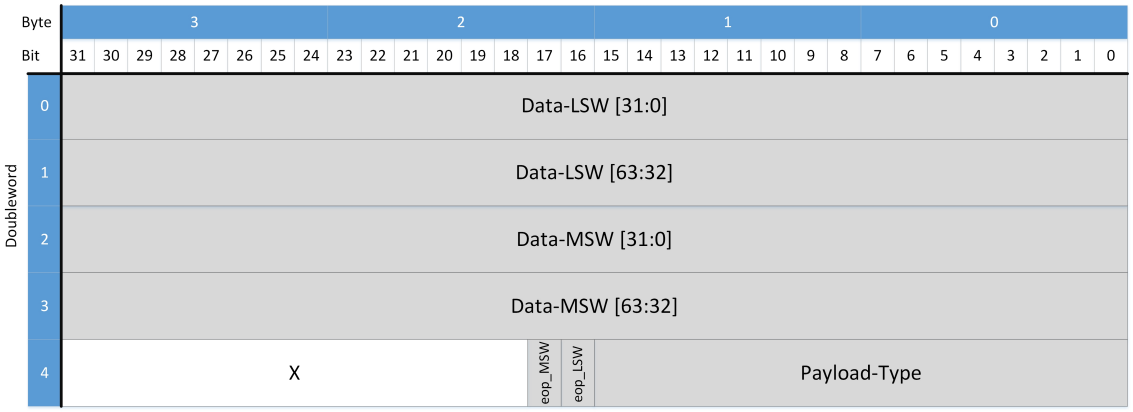


Figure 4.4: Format of the asynchronous Data-FIFOs between the AL-interface and completer / responder.

The asynchronous FIFO between the NHTL-Completer and the AL-interface, as well as the FIFO in the opposite direction, between the AL-interface and the NHTL-Responder, are designed with the same layout, shown in Figure 4.4. The first 64 bit contain the Least Significant Word (LSW) (64 bit), followed by the Most Significant Word (MSW) (64 bit).

Together this are 128 bit, which is the amount of data that is transmitted in parallel at the EXTOLL NP-interface.
The subsequent 16 bit in the FIFO (position [143:128]) carry the type-information (see Section 3.3.1). The last two bits (position [145:144]) carry the information, whether the LSW or the MSW is an End Of Packet (EOP). As the LSW is delayed between the NHTL-Completer and the AL-interface and the MSW is delayed between the AL-interface and the NHTL-Responder, this delay also has to be applied to the corresponding EOP-bits.

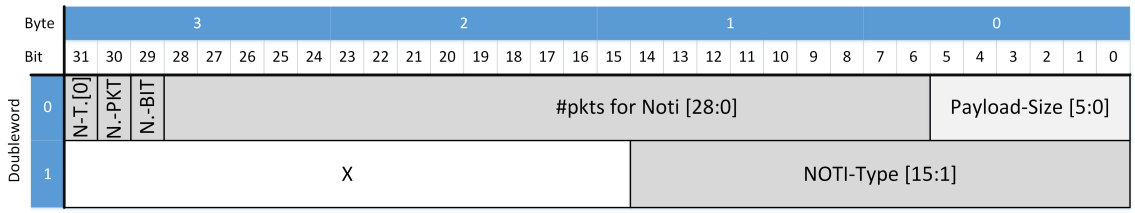


Figure 4.5: Format of the asynchronous packet-information FIFO.

The second asynchronous FIFO between the AL-interface and the NHTL-Responder buffers corresponding packet-information for the data in the first FIFO, described above. This FIFO either carries the 6 bit number of payload-QWs to include in the current trace-data-packet or the 29 bit number of packets for a notification to the host. Both numbers are packed overlapping each other, as they belong to different packet types and cannot occur together in one FIFO-entry. To be able to distinguish between these packet types, there are two bits at position {30:29} that trigger a notification-packet or a notification-bit in e.g. an FPGA-configuration-response packet. If the NOTI-PKT bit is set, the position {28:0} is interpreted as number of packets for notification. Otherwise, the position {5:0} is interpreted as number of payload-QWs for a trace-data-packet. In case of a notification packet, the bits {46:31} define the type-information for the notification-header.
For each non-notification-packet, defined in this FIFO, the data-FIFO must contain the given number of QWs until the next eop-bit is set. Otherwise the two FIFOs are not consistent with each other.

4.2 NHTL-Completer

The completer-module processes the incoming network packets and distributes them according to their protocol-content (see Chapter 3).
This module is mainly driven by a simple FSM with three states (LD_HEAD, LD_ADDR and LD_DATA). This FSM is shown in Figure 4.6. At reset, the FSM is in the state LD_-HEAD and waits there until the next two network cells are shifted out of the EXTOLL network-port. The np_shiftout signal has to coincide with a positive np_sop signal which indicates the start of a new network-packet. In this state (LD_HEAD) the completer stores the EXTOLL packet header, consisting of the SOP-cell and the network-descriptor-header, into two 64 bit registers.
The next state of the FSM is LD_ADDR. In this state the NHTL-Completer stores the request address which can, depending on the packet type, be a registerfile write- / read-address or the payload-type of the transported Host-FPGA communication data. The NHTL-Completer also determines the actual packet type from the RMA network descriptor header and updates the corresponding event- and error-counters in the register file.

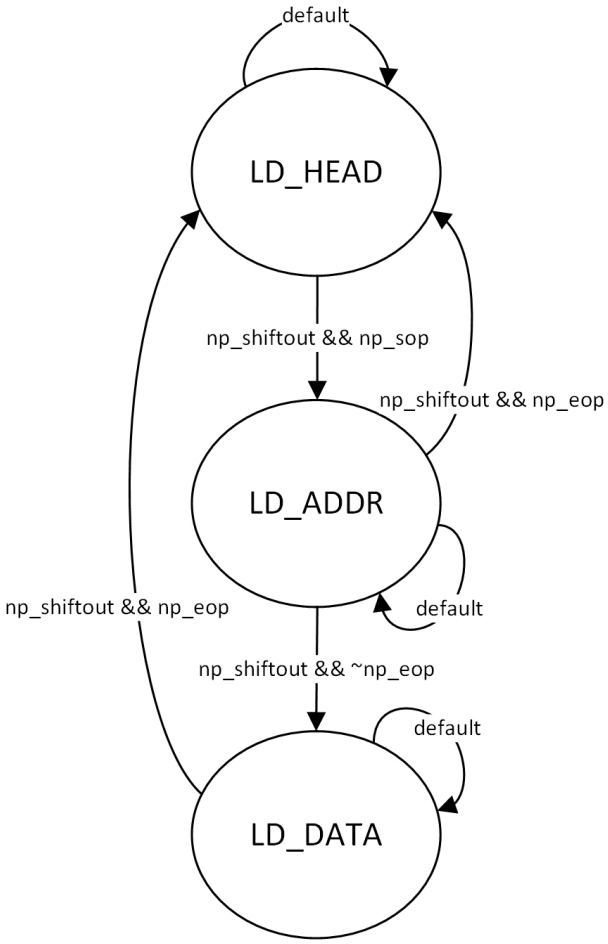


Figure 4.6: State-diagram of the completer FSM with transition conditions.

In case of registerfile access it configures the input-register for the RRA-FIFO. In case of Host-FPGA communication it configures the input- and shiftin-register of the asynchronous payload-FIFO towards the AL-interface. In case of a notification request, the respective data is shifted into the NOTI-FIFO.
From the second state the FSM can pass either back to the reset-state LD_HEAD if one of the np_eop signals is high, i.e. the packet is finished with the first payload-cell, or the next state is LD_DATA in case there is more payload-data to process and the packet is not finished yet. As for all transitions the FSM in any case waits for the next np_shiftout to occur.
In the third state (LD_DATA) the NHTL-Completer loads further data to the asynchronous payload-FIFO. The FSM returns to the reset-state (LD_HEAD) when np_eop is asserted as it does in LD_ADDR state.

4.3 Registerfile access (RRA)

The registerfile accesses are processed by the RRA-module. This forwards the request, stated per RMA_PUT_BYTE or RMA_GET_BYTE command with RRA-modifier set, to the Top-RF which is instantiated outside the NHTL on the HMF top level. From here the request travels through the RF-hierarchy down to the corresponding sub-RF,

e.g. the NHTL-RF, which is again instantiated inside the NHTL. In case of a read/-request, the RF-response travels back on the same path and is finally inserted into the RRA-response-FIFO between the RRA-module and the NHTL-Responder. Here, in the NHTL-Responder-module, the response is inserted into an RMA_GET_BYTE_RESP packet and sent back to the host using the header-information copied from the RMA_-GET_BYTE request at the NHTL-Completer.
The ring-buffer modules (see Section 4.5) start their initialisation cycles when all configuration registers are written at least once and the respective registers *config_partner_host_-3* (see Figure 4.9) or *config_partner_host_6* (see Figure 4.12) have been written with the init-bit set.

4.3.1 NHTL configuration registers

To configure the communication with the host-computer, one at first must define the desired partner-host in the registerfile. The registerfile provides eight registers for this purpose (config_partner_host_[1 – 6] and config_[trace/hicann]_noti_behav). These must all be written by software at least once. Changes can be made independently in most cases. A detailed overview of the configuration registers is given in the following figures and paragraphs. The address-map of the configuration-registers is shown in Table 4.1.

Register	Address
config_partner_host_1	0x1090
config_partner_host_2	0x1098
config_partner_host_3	0x10a0
config_partner_host_4	0x10a8
config_partner_host_5	0x10b0
config_partner_host_6	0x10b8
config_trace_noti_behav	0x10c0
config_hicann_noti_behav	0x10c8

Table 4.1: Address-map of the NHTL configuration-registerfile.

config_partner_host_1

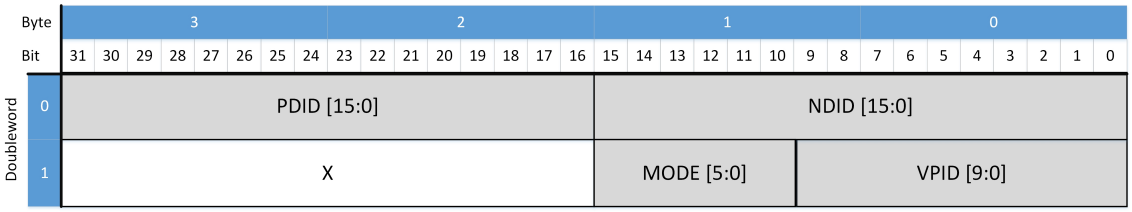


Figure 4.7: register-map of *config_partner_host_1*

This register contains the fundamental configuration data for building the communication headers when sending trace-data or configuration-responses from FPGA to host. For the MODE[5:0] field, only the third bit (MODE[2], Translate Enable) has an effect, as all

other bits are treated as don't care. This bit determines whether the address is treated as physical or virtual address (see also Section 3.1.2 and Figure 3.3).

config_partner_host_2

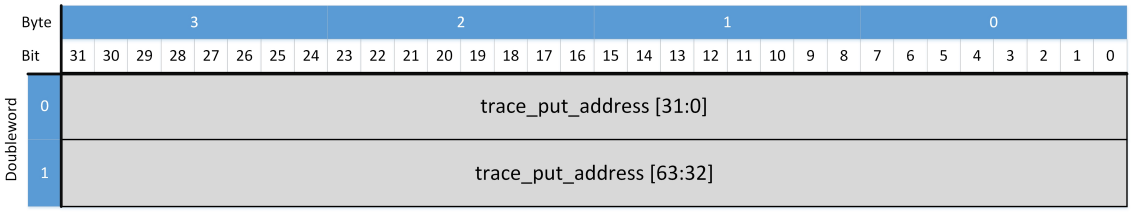


Figure 4.8: register-map of *config_partner_host_2*

This register defines the host-memory start address for the trace-data ringbuffer. It should contain a virtual address. It should be configured together with the following register, config_partner_host_3.

config_partner_host_3

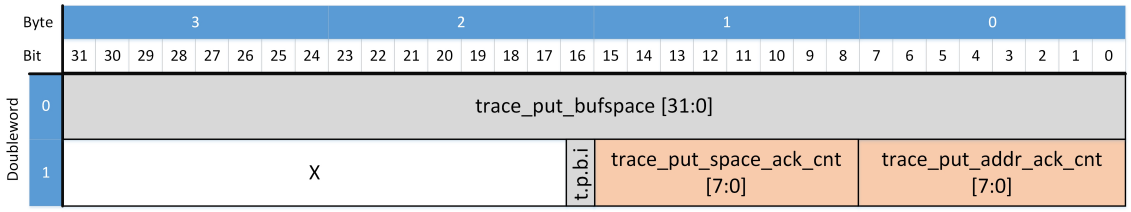


Figure 4.9: register-map of *config_partner_host_3*

This register mainly defines the allocated memory size for the trace-data ringbuffer in Byte-units. The buffer must at least be larger than 2 kB. This is because the almost-full threshold is hard-wired to be 4 maximum-sized EXTOLL-packets:

$$\text{buf\_space}\,[\text{B}] > (4 \times 62\,\text{QW} \approx 2\,\text{kB}) \tag{4.1}$$

The 1b-field trace_put_buffer_init (abbreviated in figure as t.p.b.i) triggers, when set, the initialisation process in the NHTL_ringbuffer_cntrl module for the trace-data ringbuffer. This bit must always be set when changes in ringbuffer size, or -start-address are to be configured. It must be assured, that no data-transfer is running while the configuration is done. The two fields marked in red-orange are read-only-accessible and represent the number of times that the ringbuffer-configuration-process has been run with new size and / or start-address.

config_partner_host_4

This register defines the host-memory-address for FPGA-config responses. It should contain a virtual address. The FPGA-interface writes responses of this type always to this same address.

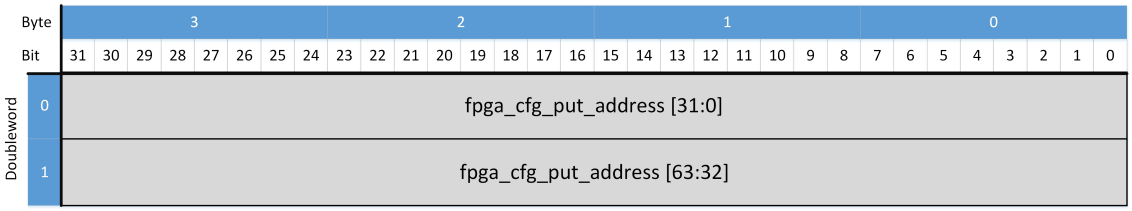


Figure 4.10: register-map of *config_partner_host_4*

config_partner_host_5

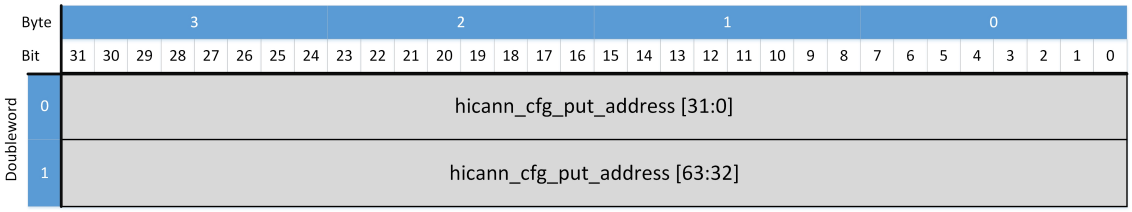


Figure 4.11: register-map of *config_partner_host_5*

This register is analogue to the register config_partner_host_2 (see Figure 4.8), but in contrast this register defines the start-address for the HICANN-config responses ringbuffer.

config_partner_host_6

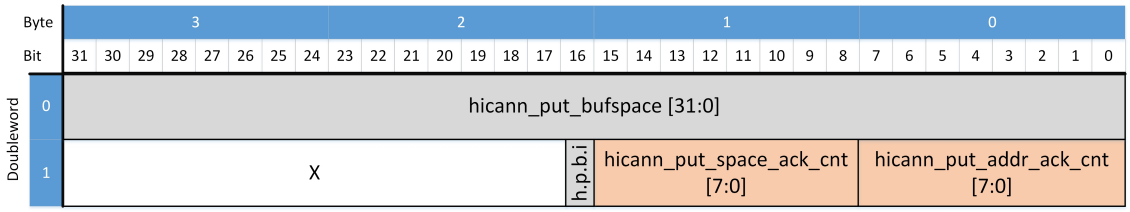


Figure 4.12: register-map of *config_partner_host_6*

This register is analogue to the register config_partner_host_3 (see Figure 4.9), but in contrast like config_partner_host_5 (see Figure 4.11) it defines the size of the HICANN-config response ringbuffer. The 1 bit field hicann_put_space_init (h.p.b.i) at position 48 triggers the initialisation process like t.b.b.i at config_partner_host_3.

config_trace_noti_behav

This register defines two things concerning the trace-data notification-behaviour: The timeout-period in [clkcyc] and the notification-frequency (period) in [pkt]. The corresponding clock-frequency is 125 MHz.
It is important, to configure the notification frequency such, that a notification is generated in any case before the ringbuffer can become (almost-) full:

$$\text{frequency}\,[\text{pkt}] < \frac{\text{buf\_space}\,[\text{B}]}{512\,\text{B}\,\text{pkt}^{-1}} - 4\,\text{pkt} \tag{4.2}$$

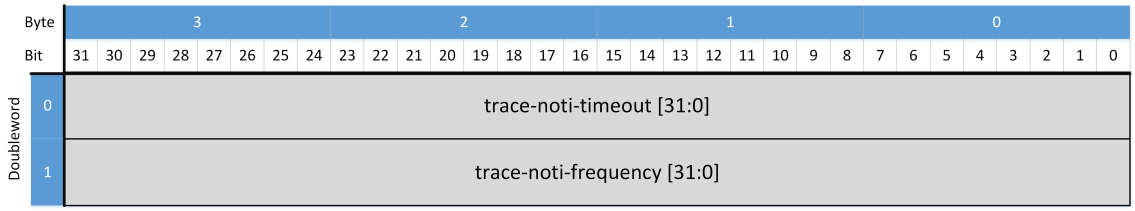


Figure 4.13: register-map of *config_trace_noti_behav*

If the notifications would be generated more rarely, it could occur, that for example a big trace-data transfer fills the host-buffer and is stalled. The host on its part cannot process the received data, because it has not received a notification telling him the amount of valid data in the buffer. This situation is called a deadlock. To solve this, it would be necessary to work with virtual channels in the NHTL-logic, so that a time-out-generated notification could overtake the stalled data-transfer. However, this is rather complicated to realise and would change most of the NHTL-architecture, not to mention the significantly increased need for hardware-logic in the FPGA. Fortunately this deadlock-situation can be avoided by correct configuration of the notification-frequency in relation to the available buffer-space and the almost-full-threshold, which is defined as four maximum-sized EXTOLL-packets ($4 \times 496\,\mathrm{B} \approx 2\,\mathrm{kB}$).

config_hicann_noti_behav

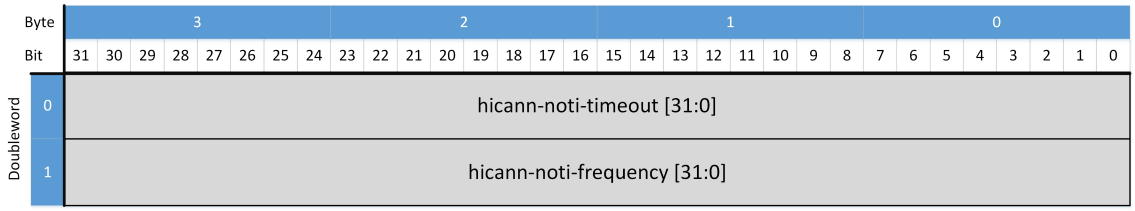


Figure 4.14: register-map of *config_hicann_noti_behav*

This register is analogue to the register config_trace_noti_behav (see Figure 4.13), but defines the timeout-period and frequency for the HICANN-response notification behaviour.

4.3.2 NHTL Error- and Performance-Counters

The NHTL registerfile also contains a number of error- and status-counter-registers. These are listed in the Tables 4.2 and 4.3 below. All of these counter registers are equipped with a reinit signal which is asserted when a special register at address `0x1048` is written with an arbitrary value.

4.4 NHTL-Responder

All outgoing EXTOLL packets are processed by the NHTL-Responder. This module merges all communication onto the EXTOLL network-port. Notification responses are generated with top priority from the requests in the NOTI-FIFO. If there are no notification requests, then the NHTL-Responder forwards the registerfile responses to the

Register	Address	Description
perf_cnt_rra_put_rcvd	0x1000	The number of RRA-put commands received in the NHTL-Completer
perf_cnt_rra_get_rcvd	0x1008	The number of RRA-get commands received in the NHTL-Completer
perf_cnt_rma_put_rcvd	0x1010	The total number of RMA-put commands received in the NHTL-Completer
perf_cnt_rma_noti_put_rcvd	0x1018	The number of RMA-notification-put-requests received in the NHTL-Completer
perf_cnt_playb_data_rcvd	0x1020	The number of playback data packets received in the NHTL-Completer
perf_cnt_fpga_conf_rcvd	0x1028	The number of FPGA-configuration packets received in the NHTL-Completer
perf_cnt_hicann_conf_rcvd	0x1030	The number of HICANN-configuration packets received in the NHTL-Completer
perf_cnt_jtag_data_rcvd	0x1038	The number of JTAG-nibble packets received in NHTL-Completer. This counter should be deprecated because JTAG data is no more transferred in form of nibbles, but directly controlled through a registerfile.
perf_cnt_ngbr_data_rcvd	0x1040	The number of Neighbour pulse-data packets received in NHTL-Completer.

Table 4.2: Status-counter registers in the NHTL registerfile.

network port.
If there are also no registerfile communication responses pending, the NHTL-Responder is free to build packets from the asynchronous payload- and packet-information- FIFOs. For details on the structure of this FIFO see Section 4.1.1.

This module is, like the NHTL-Completer (see Section 4.2) driven by a simple three-state-FSM. The FSMs state diagram is shown in Figure 4.15.

At reset, this FSM starts in the state SD_HEAD. While in this state, the NHTL-Responder waits for one of the four FIFOs to become not empty. Unless the np_stop signal is asserted, a pre-wired network descriptor header, suitable for the respective packet type (see Section 3.3.3) is buffered in the np_data output-register, now containing the header-information, delivered by the respective FIFO. The np_sop and np_valid signals are set in the corresponding output-register. The active data-source which was selected by priority is stored in a register for reference in the other states.

When the header has been sent, the FSM descends to the second state SD_ADDR. In this state the NHTL-Responder sends the pre-wired address information together with the first 64 bit data-word. The np_valid signal is asserted and in case of notification-response or RF-response also the np_eop signal. If the data-source is the asynchronous payload-FIFO

Register	Address	Description
err_cnt_invalid_command_rcvd	0x1050	Request Command was not RMA_-PUT_QW (don't care for RRA-bit) or RMA_GET_QW (with RRA-bit set) nor RMA_PUT_NOTI or RMA_PUT_-IMM.
err_cnt_invalid_type_rcvd	0x1058	The payload-type, encoded in the destination-address of RMA_PUT_QW or RMA_PUT_IMM was not a valid type for the data-decoder
err_cnt_invalid_payload_size	0x1060	The payload-size was not a multiple of 8 Bytes for RMA_PUT_QW or exceeded the exact value of 8 B with RRA-modifier set.
err_cnt_fields_error	0x1068	One of the Error-bits in the received command was set.
err_cnt_fields_mode	0x1070	The received command had ERA, NTR, EWA or INT modifiers set
err_cnt_invalid_rra_adr	0x1078	The RRA module returned an *invalid_-address* flag due to an address, requested out of the registerfile-range.
err_cnt_undefined_host	0x1080	The configuration registers were not fully configured at least once, but the FPGA tried to send data to host.
err_cnt_cfg_resp_addr_reinit	0x1088	It was attempted to reinit the FPGA-config-response-address.

Table 4.3: Error-counter registers in the NHTL registerfile.

from the AL-interface, the eop-flag is being read from a special bit of this FIFO.
The FSM then descends either back to SD_HEAD if np_eop has been asserted or to the third state SD_DATA, where further 64 bit data-words are sent until the eop-flag is asserted.

4.5 Ringbuffer Controller

Payload of the types *Trace / Pulse Data* and *HICANN Configuration* are sent to the host, addressing a ring-buffer in host-memory. By following this approach, the host-software does not need to memory-allocate the full amount of 512 MB possible trace data for each FPGA in the network. Instead only a small buffer of a few MB size has to be allocated, which significantly lowers the requirements on the host-computer.
The ring-buffer controller module ensures, that the target-address, to which the packets are sent, does not exceed the limitations configured in the registerfile. These limits comprise the start-address, at which the ring-buffer begins, and its size, given in Bytes (registers *config_partner_host_{2,3,5,6}*).
After it has been initialised, the controller continuously calculates the current write-

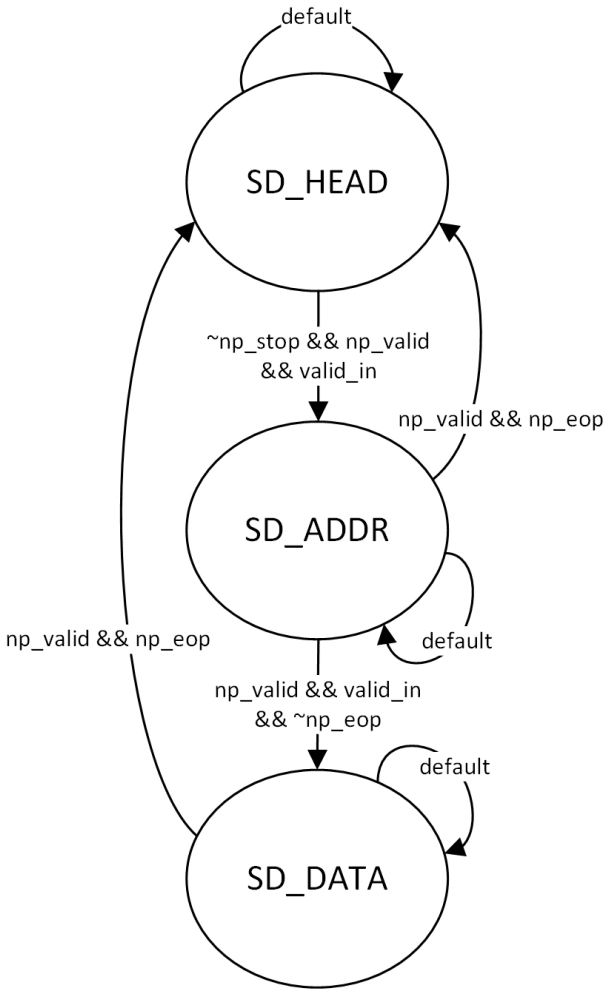


Figure 4.15: State-diagram of the responder FSM with transition conditions.

address to be used by the NHTL-Responder. To do so, it implements a fill-level-calculator that continuously monitors the filling level of the host-memory ring-buffer. For being able to increment this level together with the write-address, the NHTL-Responder has to tell the controller how many QWs of data it is going to send to the host. The host software in turn must send acknowledgement notifications (see Figure 3.12) to inform the ring-buffer controller when parts of the ring-buffer are ready for reuse. This information is always forwarded to the controller module by the NHTL-Completer when such a notification packet is received. Upon receipt of an acknowledgement notification the ring-buffer controller decrements its fill-level by the acknowledged number of QWs.

The controller-module automatically resets the address pointer when it reaches the end-address defined by the buffer-size. The pointer cannot exceed the limit before it is reset. This is ensured by a trigger-counter in front of the AL-interface, in NHTL-Top, that always triggers an EOP marker when the number of Trace-/Pulse-QWs sent to the host reaches the configured buffer-size. This approach also ensures, that no packet is wrapped around the buffer-boundary. These precautions significantly reduce the complexity of the hardware-logic and the host-software.

When the fill-level-calculator approaches the configured buffer-size by a fixed safety margin, a stop-signal is issued, to prevent the NHTL-Responder from sending another packet

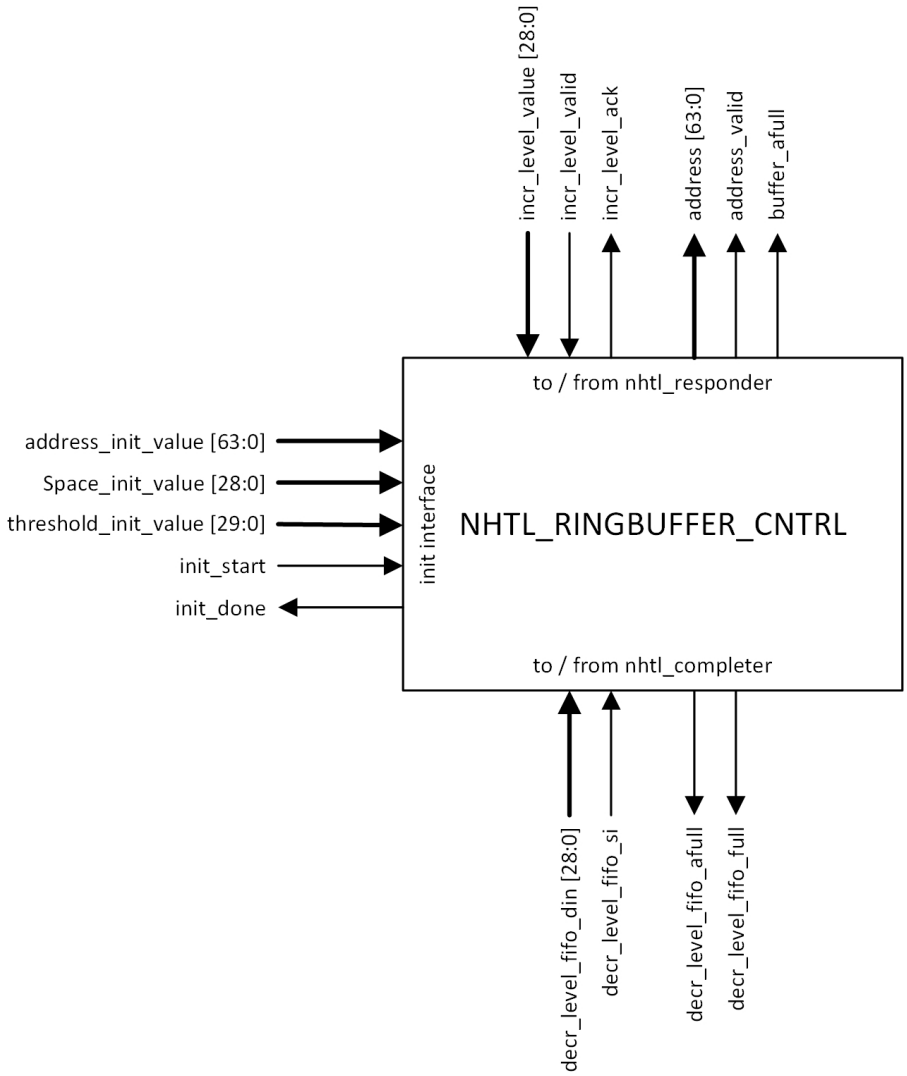


Figure 4.16: Interface block-diagram of the NHTL_ringbuffer_cntrl module.

to the host until the software has acknowledged the reusability for parts of the configured buffer.
A block-diagram of the interface-structure is shown in Figure 4.16. The interface is divided into three different signal groups. One side of the interface controls the initialisation and is connected to the registerfile in *NHTL-Top*. The incrementation interface, shown on top of the module-block in Figure 4.16 is directed to the *NHTL-Responder* and also communicates the currently valid write-address for the next EXTOLL-packet to be sent to the host-buffer.
The remaining decrementation interface is depicted on the modules bottom-edge and is directed to the *NHTL-Completer* from which it collects decrementation notifications into a FIFO to gradually lower the filling level of the buffer.
The nhtl_ringbuffer_cntrl module must be initialised with values for the start-address, the available buffer-space in QW-units and the safety-threshold for the almost-full stop-signal, also in QW-units. An initialisation sequence is started by a one-cycle-pulse on the *init_start* signal (see Figure 4.16) and involves the calculation of the end-address from the initialised start-address and available buffer-space. Once the initialisation sequence is

done, the nhtl_ringbuffer_cntrl module signals the validity of the write-address, which at this point equals the start-address, and acknowledges the completion of the initialisation phase.
When a valid increment-value is asserted at the responder-interface, this value has to be held active at the interface until the action is acknowledged by the ringbuffer-controller. This action comprises both, the incrementation of the filling-level and the write-address and is only then executed if the decrement-level-FIFO is empty. Otherwise (if the FIFO is not empty), the freeing of buffer-space has higher priority than writing the next chunk of data.

4.5.1 Implementation using DSPs

To implement the behaviour described in the preceding paragraphs, the *nhtl_ringbuffer_cntrl* module uses Digital Signal Processors (DSPs). These are given hardware blocks in an FPGA that can be used for digital calculations in hardware. Figure 4.17 shows a block-diagram of the internal structure of a DSP. DSPs can be configured in each clock-cycle using the 5 bit INMODE, the 7 bit OPMODE and the 3 bit CARRYINSEL inputs. More general settings of DSPs are available over Verilog parameters. The cascading in- and outputs, depicted on the vertical block-borders in Figure 4.17 can only be used directly between DSPs. These ports are located in the hardware so that they directly contact neighbouring DSPs in one slice. Connections to external logic and registers have to be made to the ports on the vertical block borders. Each DSP can handle up to 48 bit in parallel at its data-inputs.

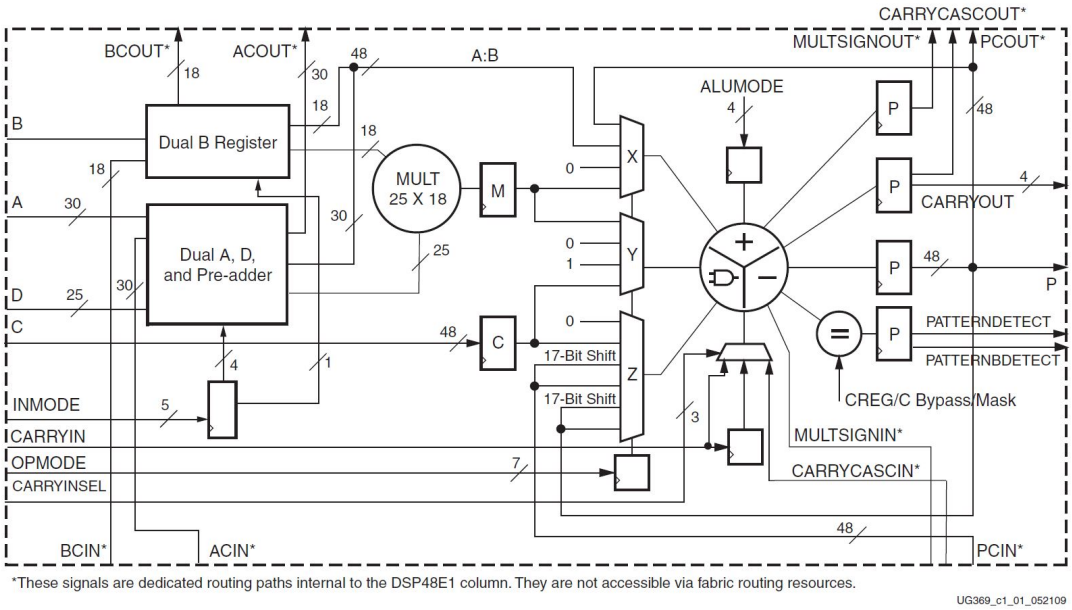


Figure 4.17: Internal block-diagram of a DSP [17, p. 8]

For the ringbuffer-controller, the DSPs are configured to calculate the next write-address, the filling-level of the ringbuffer, the end-address for wrap-around triggering, and the free-space value to trigger the almost-full stop-signal to the *NHTL-Responder*. Because the EXTOLL network, as well as the most current computer systems work with 64 bit memory-addresses, two DSPs have to be combined for address-calculations.

Address Calculation - Increment-DSP

The 64 bit address-incrementation is carried out by two DSPs which are connected by the carry-signal using the cascading ports. The use of these ports is advantageous because it allows the synthesis tool to use neighbouring DSPs directly besides each other in an FPGA DSP-slice. This implies short functional paths and facilitates timing closure for higher clock frequencies.
These two DSPs are configured to increment the result by a given value when an increment-enable signal is asserted. On assertion of a load-enable signal, the DSPs load a new value without reusing the previous result. This mode can be combined with the increment mode. The DSPs are configured with input-registers to facilitate timing when capturing the input values from the logic-stage in front.
For normal operation the address is incremented by the number of bytes sent to the host. When the address reaches the end-address, the DSPs are re-loaded with the start-address.

End-Address Calculation - Adder-DSP

The 64 bit end-address calculation is carried out by two DSPs as for the address-incrementation. In this case the DSPs are configured to simply add the input operands, start-address and buffer-size. Because there is no logical loop in the adder, there is no compelling need to guard the DSP with an enable signal as for an incrementer / decrementer.
There is also no need for input-registers at the DSPs inputs because the operands of this calculation are directly taken from registers without logic in between.

Level Calculation - Increment- / Decrement-DSP

The 29 bit level-calculation is carried out by one DSP, which is configured similar to the increment DSPs for address-calculation. But in difference, its ALUMODE is configured to switch between incrementing and decrementing the level, depending on a 1 bit mode-input. The mode is prioritised by control logic, to decrement if the decrement-FIFO is not empty.
The size of 29 bit for the level in QW-units is chosen as this corresponds to a 32 bit counter in Byte-units. This allows for buffer-sizes up to 4 GB.

Free-Space Calculation - Subtractor-DSP

To calculate the free buffer-space, the filling-level has to be subtracted from the overall available buffer-space. For this purpose one DSP is configured similar to the DSPs for the end-address calculation. The DSP is configured to calculate the difference from the input-operands. Again there is no need for guarding the operation with an enable signal and also no need for input-registers.

4.5.2 FSMs for Calculation Timing

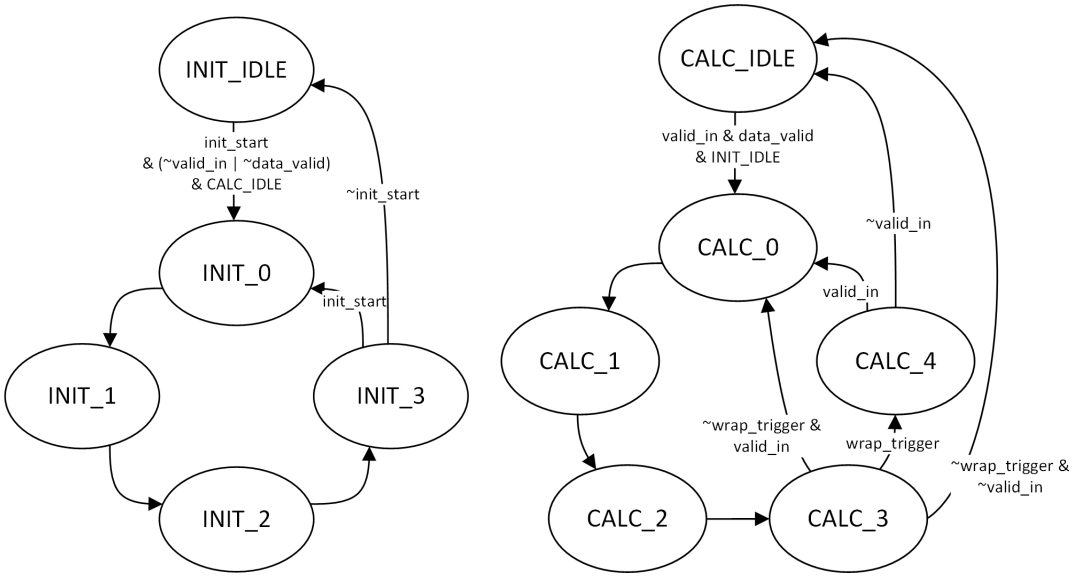
The Figures 4.18a and 4.18b show the two FSMs which have been implemented to manage the initialisation and calculation sequences and control the acknowledgement signals for init, increment, write-address and decrement-shiftout. The states of these FSMs are

presented in detail in Table 4.4. As can be seen in Figure 4.18, all states with the exception of the idle-states are only visited for one clock cycle, which is why they are called *cycles* rather than *states* in Table 4.4.

Init-State	**Init-Actions**
INIT_IDLE	The controller is not in initialisation mode and is free for calculation.
INIT_0	In this cycle, the level-DSP prepares its input-registers for a load-reset. The initialisation values are captured to registers from the interface.
INIT_1	In this cycle, the first end-address-DSP calculates the lower 48 bit of the end-address and a carry-signal for the next DSP. The level-increment-DSP loads its reset-value and the address-increment-DSPs both prepare their input-registers for loading the start-address.
INIT_2	In this cycle, the second end-address-DSP calculates the upper 16 bit of the end-address considering the carry-bit from the first DSP. The first address-increment-DSP loads the lower 48 bit of the start-address.
INIT_3	In this cycle, the second address-increment-DSP loads the upper 16 bit of the start-address. At the same time, the *address_valid* output-register is activated, *init_done* is asserted and *buffer_afull* is reset to false.
Calc-State	**Calculation-Actions**
CALC_IDLE	There is no calculation going on. The controller is free for (re-) initialisation.
CALC_0	In the first cycle, the *address_valid* output-register is set to false. The level-DSP and the two address-DSPs input-registers become valid.
CALC_1	In the second cycle, the level-DSP is active and either increments or decrements the level, depending on the mode. In case of increment-mode, the first address-DSP calculates the lower 48 bit of the incremented write-address.
CALC_2	In this cycle, the free-space-DSP calculates the difference of available buffer-space and the current filling-level, calculated in the previous cycle. In case of increment-mode, the second address-DSP calculates the upper 16 bit of the incremented write-address and the increment interface is being acknowledged. In case of decrement-mode, the decrement-FIFO shifts out. The address will be valid after this cycle, if the lower 48 bit don't equal the respective end-address-bits, i.e. surely no wraparound has to be done.
CALC_3	In this cycle the address-DSP will prepare for wraparound if the end-address is reached. Otherwise the write-address will become valid after this cycle. If the free-space is smaller than the initialised safety-threshold, the *buffer_afull* signal will assert the stop-condition for the NHTL-Responder. If no wraparound condition is detected, the next cycle will be skipped.
CALC_4	In case of wraparound this is the last cycle and both address-DSPs load the start-address. After this cycle the new write-address will finally be valid.

Table 4.4: Details on the initialisation and calculation cycles.

The state-diagrams in Figure 4.18 show two possible transitions from the respective final state back to the start. One transition goes back to the IDLE-state and the other goes directly back to the first actual cycle in case another init- or calculation-action is already issued at the interface.
The Calculation FSM (Figure 4.18b) has two possible end-states, CALC_3 or CALC_-4, depending on whether there is a wraparound or not. Although the new write-address can already be valid after CALC_2, the CALC_3 cycle has to be visited in any case to calculate the stop-condition for the NHTL-Responder.



(a) State diagram of the ring-buffer initialisation-FSM (b) State diagram of the ring-buffer calculation-FSM

Figure 4.18: FSMs for the ring-buffer controller

The Init-FSM (Figure 4.18a) only starts from INIT_IDLE when there is no calculation going on, i.e. the Calculation-FSM (Figure 4.18b) is in state CALC_IDLE, and there is no valid input, i.e. the Calculation-FSM won't start in this cycle. If this is the first initialisation, the Init-FSM starts anyway.
The Calculation-FSM analogously starts only if the Init-FSM is in INIT_IDLE and if the initialisation data is already valid. Thereby the address-calculations are prioritised over re-initialisation of the controller. This ensures, that reinit only happens when all memory-free (decrement-) operations are done.
Anyhow it can happen, that an *init_start* pulse is missed because a calculation is going on. In this case the respective registerfile-command (see Figures 4.9 and 4.12) has to be re-asserted. Generally it should be assured, that no transfer is going on while reinitialising the ringbuffer-controller.

4.6 Clock-domains

The NHTL-Top module is divided into two clock-domains. The first one, including the main NHTL-logic, runs at the speed of the EXTOLL network-port, which is 210 MHz. With 128 bit parallel data-width and 8b/10b coding this gives a network data-rate of

8.4 Gbit s^{-1} on four lanes. The second clock-domain contains the HMF core-logic and runs at a speed of 125 MHz. The border between the two clock-domains is synchronised by asynchronous FIFOs that carry data and packet-information from one domain to the other.
The overall FPGA design contains some more clock-domains controlling the differential communication with the HICANN chips and the EXTOLL link-port. All clocks in the design are generated using Mixed-Mode Clock Managers (MMCMs) and Phase Locked Loops (PLLs), instantiated as Xilinx® Intellectual Property (IP) blocks on the FPGA. All clock-domains share one common 50 MHz reference-clock that is globally generated on each Wafer Module and distributed to all FPGAs on that Wafer Module. [8, p. 139, 142]
The accuracy of the clock generation is estimated to ±100 ppm [18]. This leads to the problem, that the generated clocks on different Wafer Modules gradually run apart. With 1 ppm accuracy 1 s experiment time would lead to a clock-drift of 1 µs, which corresponds to a biological time of 10 ms with the speedup factor of 10^4. To solve this problem, the Wafer Modules also support a future many-wafer system where the reference clock is centrally generated and distributed to all Wafer Modules.

5 JTAG Optimisations

This rather short chapter describes the optimisations regarding the JTAG access to the HICANNs and some registers in the FPGA. Section 5.1 explains the general functionality of the JTAG Test Access Port (TAP) controller and its counterpart, the JTAG master-controller. An FSM based controller is used instead of the given bit-transaction-based one. Section 5.2 describes the optimisations conducted to the JTAG chain as a whole, meaning the replacement of the TAP controller in the FPGA with an EXTOLL accessible registerfile.

5.1 JTAG Controller

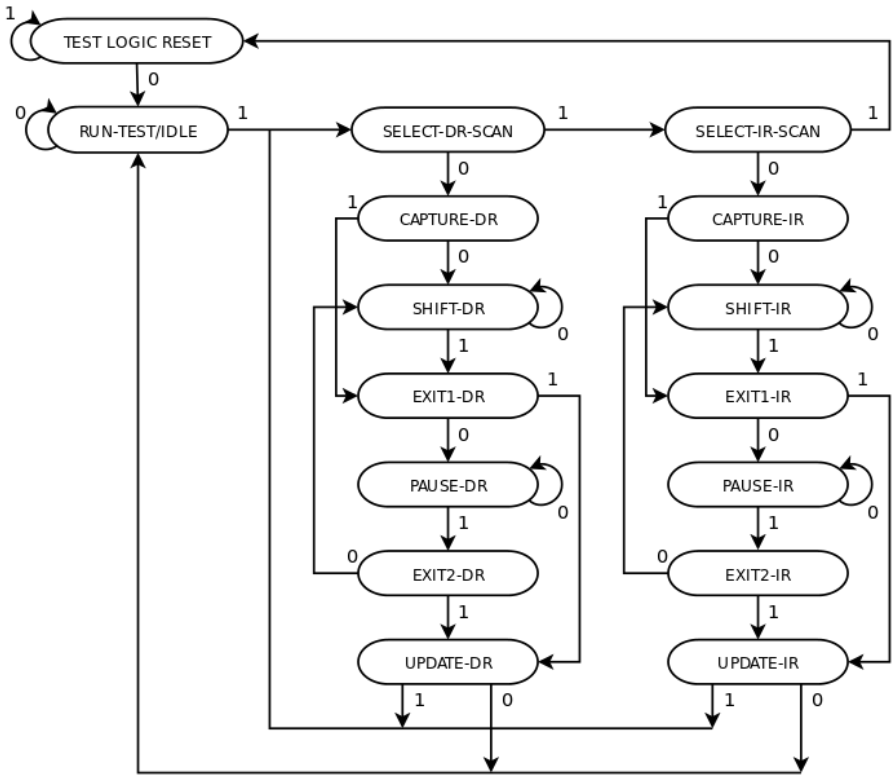


Figure 5.1: State diagram of the JTAG TAP Controller as defined by the JTAG standard. [19]

The JTAG standard defines five pins: Test Clock (TCK), Test Mode Select (TMS), Test Data In (TDI), Test Data Out (TDO) and Test Reset (TRST). The TMS pin controls the state of a finite state machine called the TAP-controller and is sampled at every posedge of TCK. The state-diagram of the TAP-controller is shown in Figure 5.1.

When the state `SELECT-DR-SCAN` is left with (`TMS == 1`) and `SELECT-IR-SCAN` is left with (`TMS == 0`), instructions can be shifted into the instruction-registers through the chain. When this is done, depending on the instructions, data-registers are selected and can be shifted by leaving the state `SELECT-DR-SCAN` with (`TMS == 0`). In the `CAPTURE-*` states the functional registers are captured to the scan-chain while in the `UPDATE-*` states, the functional registers are updated with new data from the scan-chain. The `PAUSE-*` states render the possibility to reload the data-buffer for shifting large amounts of data. [20, 21]
The TAP-FSM is meant to control the JTAG-slaves in the chain. Additionally there must also be a master-controller which controls the signals TMS, TCK and TRST respectively and manages the data flow through the chain (TDI and TDO).
In the HMF environment, JTAG (Joint Test Action Group) is used for backdoor access to the FPGAs and HICANN-chips. The JTAG-Chains connect one FPGA and eight HICANNs respectively and are accessed via network-interface. In the FPGA-design, the JTAG controller was placed directly at the UDP-interface. To ensure correct transmission of the JTAG data over the Ethernet network, up to now the controller will be moved behind the Host-ARQ module, which implements a retransmission protocol (see Section 2.1).

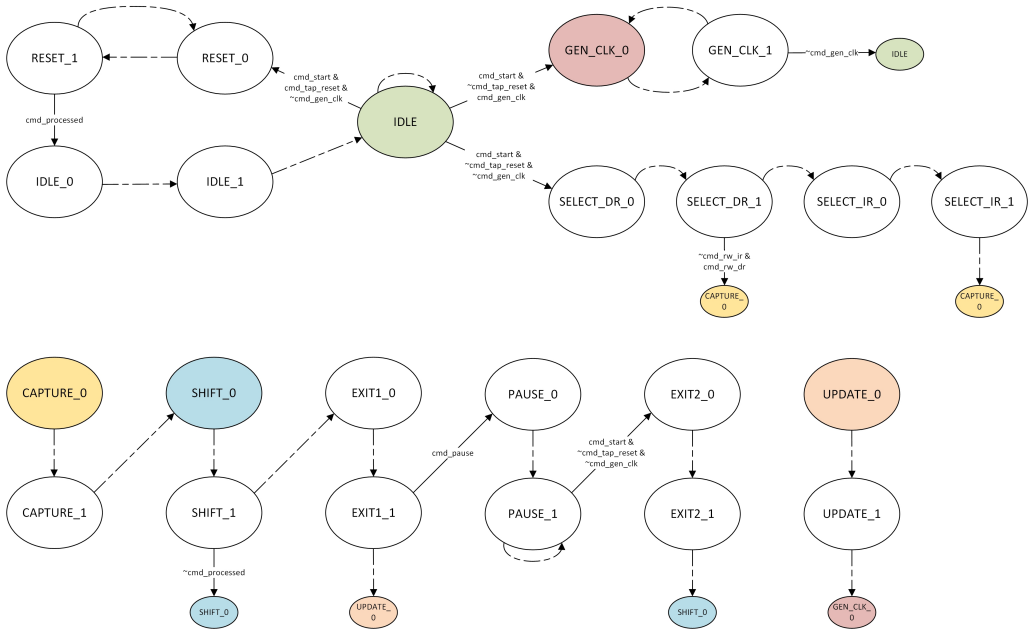
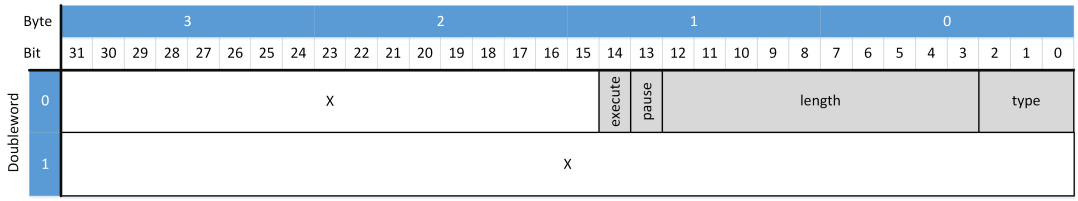


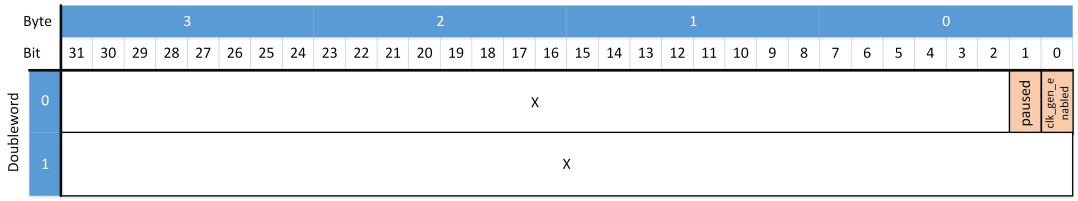
Figure 5.2: Block-diagram of the JTAG master-controller FSM designed by [22] at the Computer Architecture Group. The colour-code highlights linked state-transitions. Each logical state is divided into two functional states, to control TMS with a coincident posedge of TCK. Dashed lines represent default-transitions.

Up to now the master-controller was implemented to receive so-called JTAG-nibbles from the host via network (Type `0x06A4`, see Section 3.3.1). These nibbles include 4 bit of data indicating the state of the three JTAG output pins (TCK, TMS, TDO) and whether the input pin (TDI) shall be monitored. To make this work, each state-transition in the TAP-Controllers and each data-bit requires to transfer two nibbles, i.e. 1 B from the host to the FPGA via network, which is quite inefficient.
To improve the efficiency when controlling the JTAG-chain, the JTAG master-controller

at hand is replaced by a module which was developed by [22]. This module uses an FSM which is shown in Figure 5.2.



(a) Structure-diagram of the JTAG command-register.



(b) Structure-diagram of the JTAG status-register.

Figure 5.3: Structure-diagrams of the JTAG registers.

The used JTAG master-controller is managed through a generated registerfile which is connected to the EXTOLL registerfile interface. It can efficiently be controlled through EXTOLL-RMA packets with the RRA-modifier-bit set (see Section 3.2). This registerfile contains a command- and a status-register as well as a send- and a receive-data-buffer. The two registers are shown in Figure 5.3.

Value	Meaning	Description
`3'b000`	TAP-Reset	This command controls the TAP-FSM to go to the reset-state.
`3'b001`	Read/Write IR	This command controls the TAP-FSM to shift the instruction-register.
`3'b010`	Read/Write DR	This command controls the TAP-FSM to shift the selected data-register. Prior to this command a data-register must be selected by shifting an address into the instruction register.
`3'b100`	Enable Idle Clock Generation	This command controls the TAP-FSM to generate TCK even in IDLE state.
`3'b101`	Disable Idle Clock Generation	This command takes back the effect of the previously described command.

Table 5.1: Description of the type-field in the JTAG cmd-register.

The command register (Figure 5.3a) contains four fields. The type-field is 3 bit wide and controls the operation mode of the JTAG master-controller. For detailed description of the type-field see Table 5.1. The length-field defines the number of bits to shift from the send-buffer through the JTAG-chain and into the receive-buffer. With a 10 bit wide length-field, it is possible to shift up to 1024 bit which is also the size of the data-buffers (16×64 bit). The pause-bit tells the controller to go into `PAUSE-*` state after shifting the data from the

send-buffer. The execute-bit must be set to start the execution of the specified command. After the execution has completed, the controller resets the execute-bit to its inactive state. The status register (Figure 5.3b) contains two fields. The clk_gen_enabled field indicates, whether the clock generation is activated while the FSM is in `IDLE` state (see Figure 5.2). The second field, namely the paused-bit indicates whether the TAP-FSM is currently in `PAUSE-`$\star$ state or not.
The two data-buffers are implemented as RAM-Blocks in the registerfile and provide space for up to 16 QW (1024 bit) of JTAG-data respectively.

5.2 FPGA JTAG Registers as EXTOLL RF

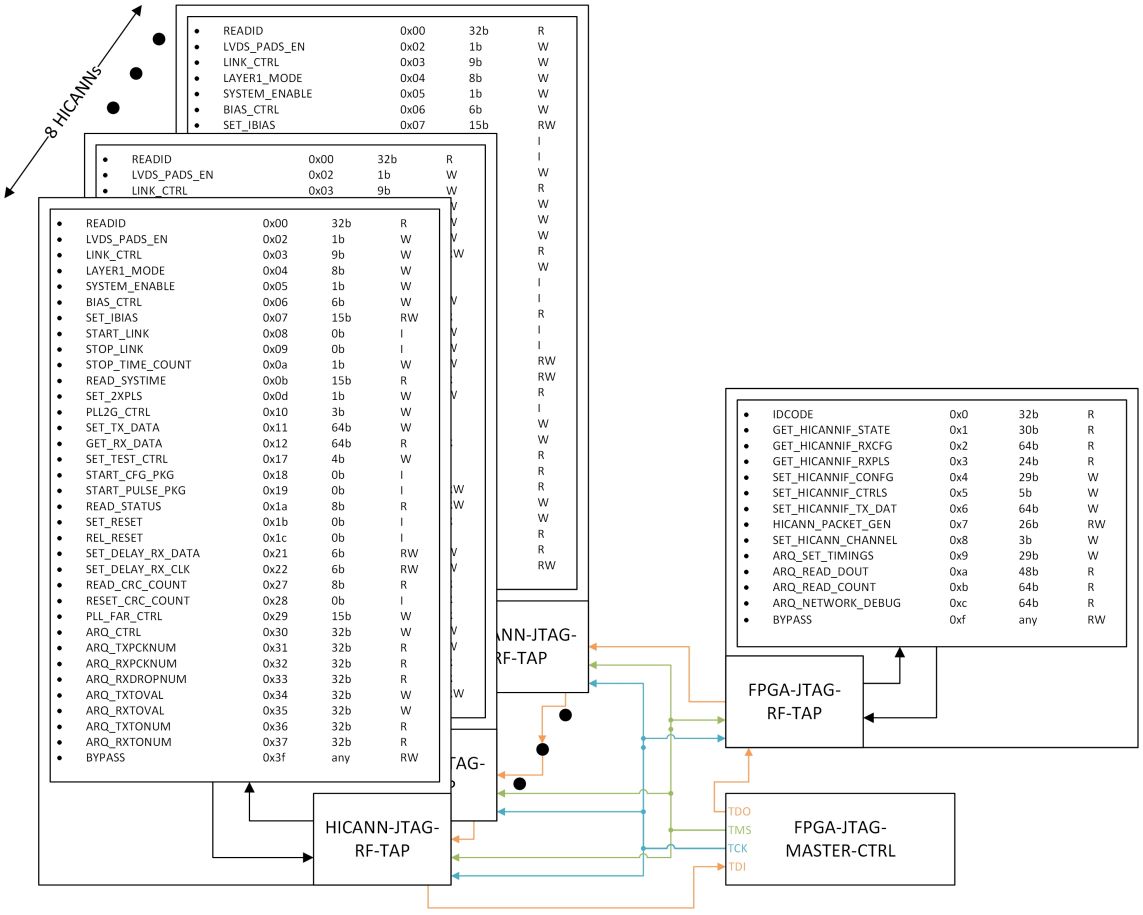


Figure 5.4: The JTAG-Chain setup and the available instructions for HICANNs and FPGA TAP.

The JTAG interface provides backdoor-access to a number of registers and actions related to the initialisation of the HICANN communication channels in the FPGA. The chain-setup is shown together with a list of the available instructions in Figure 5.4.
Instructions marked with 'R' are mapped to read-only data-registers, while instructions marked with 'W' are mapped to write-only data-registers and 'RW' instructions indicate both readable and writeable data-registers. Instructions marked with 'I' are not mapped to any data-register. Instead they trigger an action in the HICANN-chips.

To improve the transfer-efficiency, the JTAG-registers located in the FPGA can be directly implemented as EXTOLL-accessible registerfile. This supersedes the necessity of transferring an instruction for each access to one of these registers. In total this avoids at least three of four registerfile-access packets, necessary to instruct the JTAG-controller (1: Fill *send_buffer* with instruction, 2: write *cmd_reg* to send instruction (and poll for success), 3: fill *send_buffer* with data, 4: write *cmd_reg* to send data (and poll for success)). Instead, the data can now directly be sent to the respective register. Additionally the JTAG-chain to access the HICANN-TAPs is now shortened by one instruction- and data-register less. Furthermore, space in the FPGA implementation area is freed, as the TAP-controller is not needed any more. The optimised JTAG-Chain is shown in Figure 5.5.

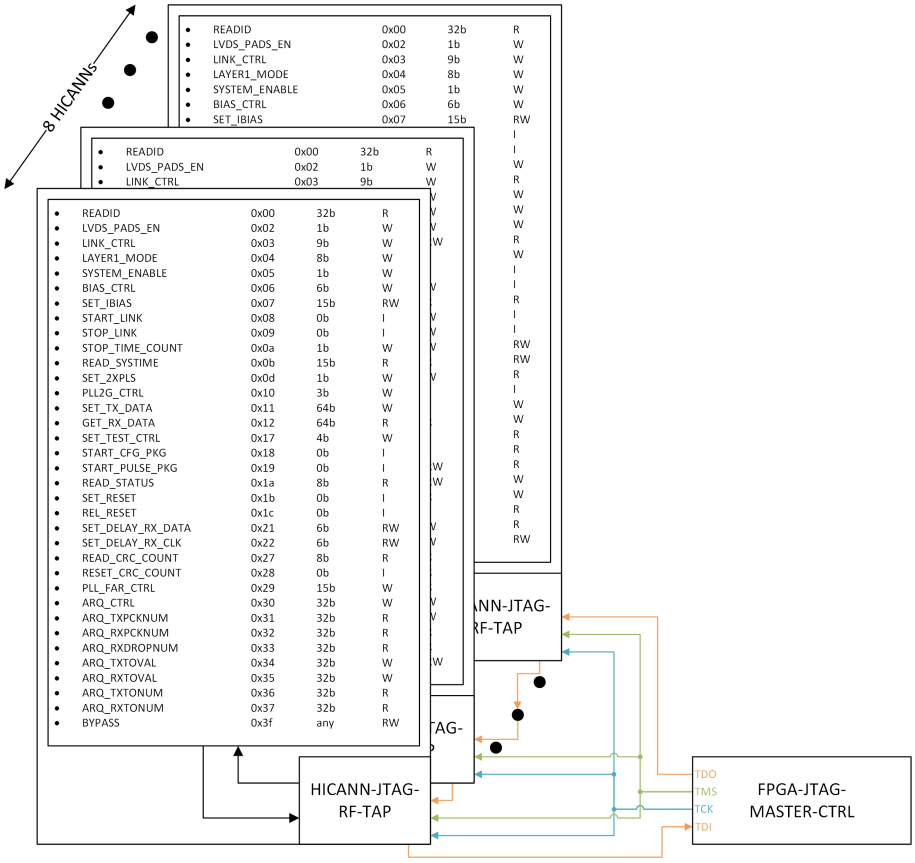


Figure 5.5: The optimised JTAG-Chain setup. The FPGA-JTAG-RF-TAP has been replaced with an EXTOLL-compatible registerfile.

6 FPGA-Implementation

This chapter first introduces the general properties of the used Xilinx® FPGA (Section 6.1) before describing the used tool-environment (Section 6.2). The distribution of the floorplan over the FPGA is explained in Section 6.3. Section 6.4 then presents the representative results of an implementation run constrained with the previously described floorplan using the mentioned tool-environment. Finally Section 6.5 summarizes the available procedures for flashing the implemented design onto the FPGAs as a bitfile.

6.1 Xilinx FPGA

The Xilinx® FPGAs used for the BrainScaleS system are of the type Kintex®-7 (XC7K160T). This device belongs to the 7 Series FPGAs. The Kintex®-7 Family is optimized for best price-performance and doubles its performance compared with the previous generation of Xilinx FPGAs. Besides the Kintex® Family, there are also the Spartan®, the Artix®, and the Virtex® Families. The Kintex® XC7K160T is the second smallest FPGA in its Family and comprises 162 240 Logic Cells, 25 350 Configurable Logic Blocks (CLBs), 600 Digital Signal Processors (DSPs), 11 700 kbit of Block-RAM (RAM) and 8 Clock Management Tiles (CMTs), each containing a PLL and an MMCM. Each CLB contains 4 Lookup Tables (LUTs) and 8 Flip Flops (FFs). [23]

6.2 Tool-Environment

To implement the design, described in the previous Chapters, onto the FPGA architecture, the Xilinx® Vivado® tool-environment is used. The source-code and constraint-files, describing the design, are gathered by a make-script that creates a Vivado® project-file and includes all sources to it.
With this project-file, the design can be evolved through the design-flow traversing the steps *Synthesis*, *Place* and *Route*. When all the implementation-steps are done, a *Bitstream* can be generated for directly programming the FPGA fabric.
Throughout these implementation steps various analysis-runs can be started and evaluated from the Vivado® GUI. Useful analysis-runs are for example the *Timing Report*, the *DRC Report*, the *Utilisation Report* or the *Power Report*. For debug purposes, there is also a *Schematic Viewer*. This can be useful to check e.g. the connections between high level module interfaces.

6.3 FPGA-Floorplan

After Synthesis, the design should be constrained to a reasonable floorplan to facilitate timing closure on the FPGA and to make the design more clearly arranged. Creating a

reasonable floorplan also helps using the available implementation space on the FPGA in an ideal way.

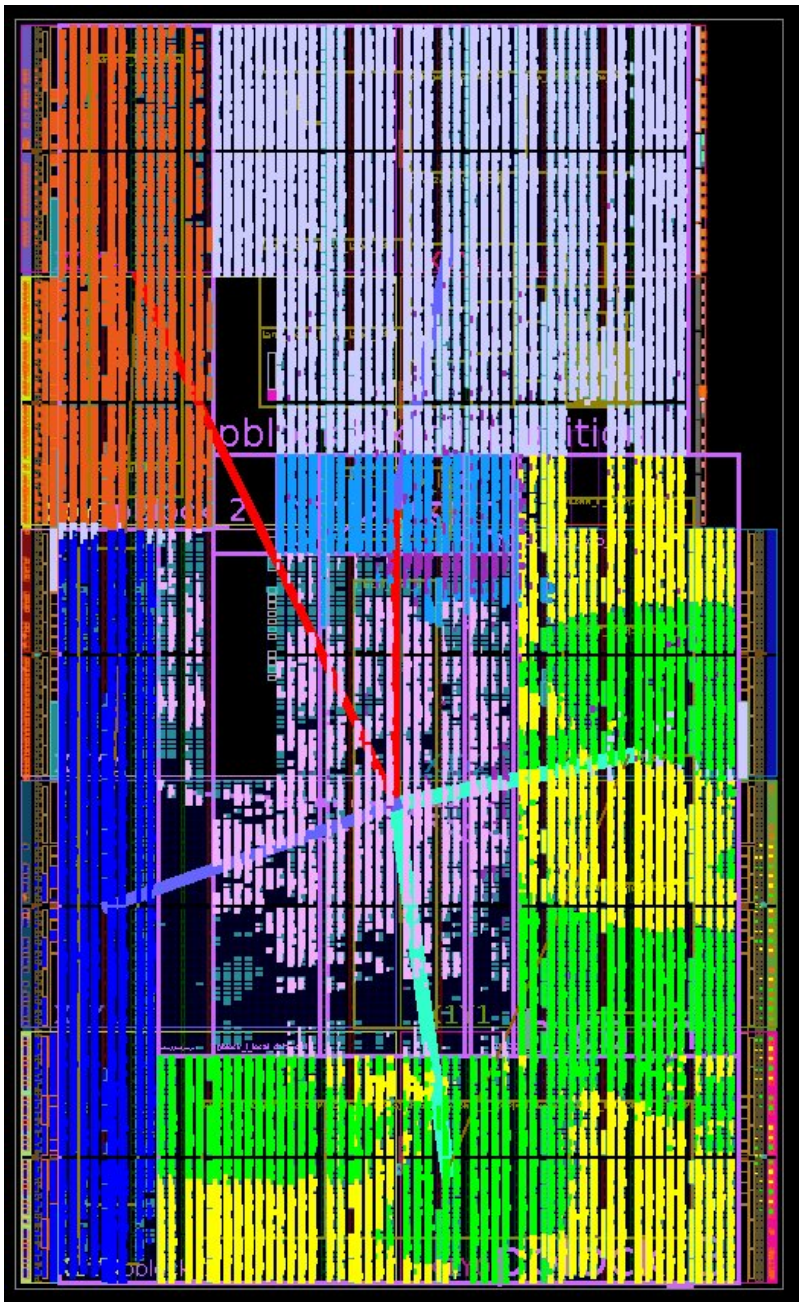


Figure 6.1: Floorplan for the BrainScaleS FPGA design. [Screenshot from Vivado]

Figure 6.1 shows a screenshot of the implemented design in the Vivado® Device view. The colour code for the module leaf-cells is explained in Table 6.1.
The main interfaces are the HICANN-IFs and the EXTOLL partition containing the EXTOLL NP and the LP. Besides these two interfaces, there are also the DDR3 memory interfaces. All these main interfaces are placed at the borders of the design. The EXTOLL partition is placed at the top-right corner of the FPGA and the eight HICANN interfaces are placed around the bottom-right corner. In Figure 6.1 the HICANN interfaces are coloured alternately in yellow and green, so they can be distinguished from each

colour	location	module / partition
grey	top-right corner	EXTOLL partition (NP, LP)
light-green / yellow	bottom-right corner	8 × HICANN-interface
dark-blue	bottom-left	playback-memory DDR3-interface
orange	top-left	trace-memeory DDR3-interface
light-blue	center-top	NHTL-top and submodules
pink	center	core-logic
purple	everywhere	registerfile

Table 6.1: Colour code from Figure 6.1

other. The memory interfaces are placed at the remaining left edge of the design. The decision, where to place the interfaces at the FPGA borders is not arbitrary, but depends on the location of the respective interface-pins on the FCP board-layout.
The remaining design parts, namely the NHTL interface and the core-logic are placed in the centre of the FPGA, whereat the NHTL interface has to be placed near the EXTOLL partition at the centre-top. The core-logic and the remaining small modules (e.g. the I2C interface) are placed automatically on the remaining space in the FPGA centre region to optimize timing. It can be seen in Figure 6.1, that the leaf-cells belonging to the registerfile (coloured in purple) are spread across the whole FPGA to where they particularly belong to.

6.4 Implementation Results

After synthesis and implementation of the design timing- and utilization-reports can be analysed. The timing-report tells, whether the constraints stated for the timing of functional paths in the design could be met by the synthesis and implementation tool. It also tells how much margin is left in case of success or how much the implementation misses the constraints. Concretely, the timing summary is shown in Figure 6.2.

Figure 6.2: Screenshot of the timing summary in Vivado®. Worst Slack respectively refers to the minimal safety-margin left for timing-closure, while the Total Slack respectively refers to the sum of negative slack Endpoints where the timing is not reached.

The report shows, that the timing is met and the design has been successfully implemented. The Setup-time is the time that a functional signal must arrive at the input of a Flip-Flop before the next positive clock-edge. In the implemented design it is satisfied for all paths ending at a Flip-Flop and the minimal margin left is 0.176 ns. However this number is not quite meaningful because the Vivado® implementation engine stops optimization once timing is met. If this number was negative, the timing would be missed.

This could have several reasons as for example inappropriate placement or to high clock frequency constraints.
The Hold-time is the time, a functional signal must stay stable after the current positive clock-edge. Again this constraint is satisfied in the implemented design for all Flip-Flops. The Pulse-Width slack is a measure for the quality of clock signals in the design. This type of constraints controls the clock-waveform and skew. Skew is referred to as the time-difference between the arrivals of the same clock-signal at different Flip-Flops in a design. The skew should be as low as possible because otherwise the output-values of different Flip-Flops are not valid at the same time and therefore cannot be combined in logical gates and functions.

Resource	Utilization	Available	Utilization %
LUT	78754	101400	77.67
LUTRAM	5034	35000	14.38
FF	67973	202800	33.52
BRAM	125	325	38.46
DSP	151	600	25.17
IO	252	400	63.00
GT	4	8	50.00
MMCM	5	8	62.50
PLL	3	8	37.50

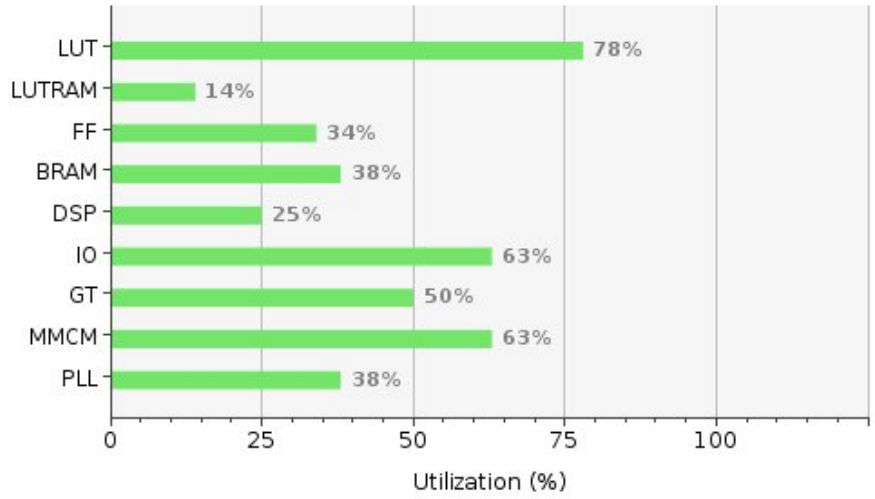


Figure 6.3: Screenshot of the utilization summary in Vivado.

A Screenshot of the utilization report summary is shown in Figure 6.3. As can be clearly seen from the table and the diagram, the design easyly fits into the FPGA and there is still some amount of space left. LUT, FF, BRAM and DSP represent the main components in an FPGA, while GT and IO refer to input and output units. MMCM and PLL are clocking resources. The remaining space can later be used for the pulse-routing implementation. A concept for this pulse-routing is described in Chapter 9.

6.5 Flashing the FPGA

Before the FPGA can be programmed, a bitstream has to be generated using the Vivado® software. This can be done after synthesis and implementation have completed successfully. The FPGA can then be flashed, using a JTAG-based USB programmer. This device can be seen in Figure 8.1a.
The firmware can either be programmed directly onto the FPGA, or into an internal Flash-Memory. In the former case, the FPGA must be reprogrammed after each power-cycle. In the latter case, the firmware stays stable in the Flash-Memory, but a boot-loader is required on the FPGA to load the design from the Flash upon power-up.
In the course of this work both methods are used: The test-system uses the direct method, while the BrainScaleS system FPGAs are programmed to Flash (for details see Chapter 8). That is because reprogramming 48 FPGAs each time that a Wafer-Module is power-cycled, is quite time-consuming. For the test-system, only containing four FPGAs, this can be accepted for the advantage of not requiring a boot-loader.

7 Verification

The Universal Verification Methodology (UVM) used to verify the functionality of the NHTL interface is described in Section 7.1 at the beginning of this chapter. The chapter continues with the description of the testbench implementation (Section 7.2) and the implemented test-cases in Section 7.3. Finally, the regression method is explained in Section 7.4

7.1 A UVM Testbench

The NHTL-design, which is described in detail in the previous chapters, is verified, using the Universal Verification Methodology (UVM) and the System Verilog hardware description and programming language.

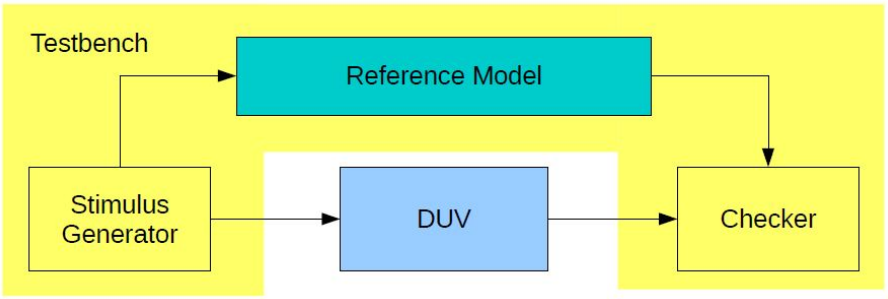


Figure 7.1: Generic structure of a testbench.[24]

The generic structure of a verification testbench is shown in Figure 7.1. The Design under Verification (DUV) is provided with input from a Stimulus Generator. The design is checked for correctness by comparing its output with a Reference Model which works on the same inputs from the Stimulus Generator.

The general structure of a UVM testbench is shown in Figure 7.2. The different interfaces of the DUV are controlled and monitored by so-called Interface Universal Verification Components (UVCs). These contain a Bus Functional Model (BFM), a Sequence Driver (Drv) for controlling the interface, and a Monitor (Mon) to observe the interface signals. The Sequence Driver feeds the BFM with programmed and random-generated test-sequences. The Interface Monitors forward their observations to the overall Module UVC. The Module Monitor collects all observations and feeds them to the Reference Model and the Scoreboard. The Scoreboard (SCB) compares the results from the DUV and the Reference Model and stores the comparison results for later analysis. Error messages are thrown for failed tests.

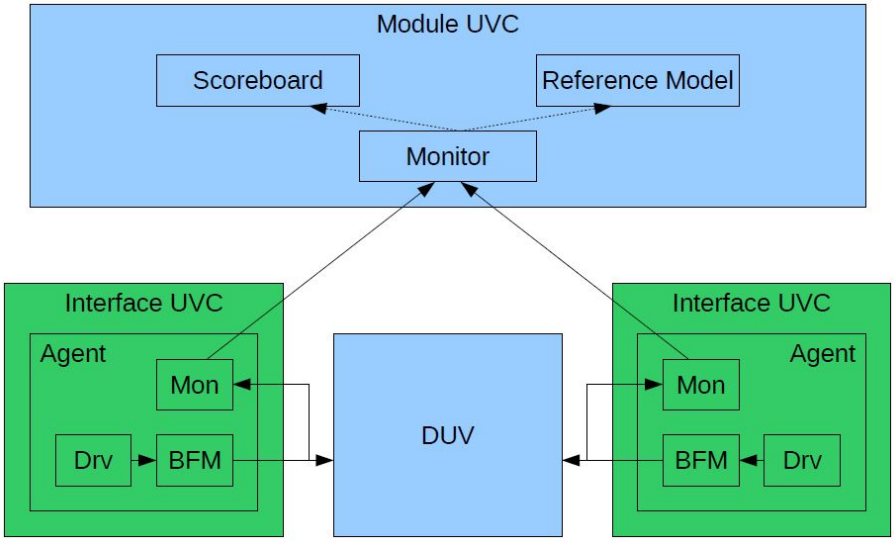


Figure 7.2: General structure of a UVM testbench. [24]

7.2 NHTL-Testbench

The structure of the NHTL testbench is shown in Figure 7.3. The DUV, which is the nhtl_-top module, mainly has two bidirectional interfaces. One of them contacts the EXTOLL network-port, the other one connects the application layer. Both interfaces are observed by Monitors and controlled by Sequence Drivers. The direction towards the nhtl_top is controlled by the master-drivers, while the opposite direction is controlled by slave-drivers. The drivers them selves are controlled by two Sequencers, fed by the different tests.

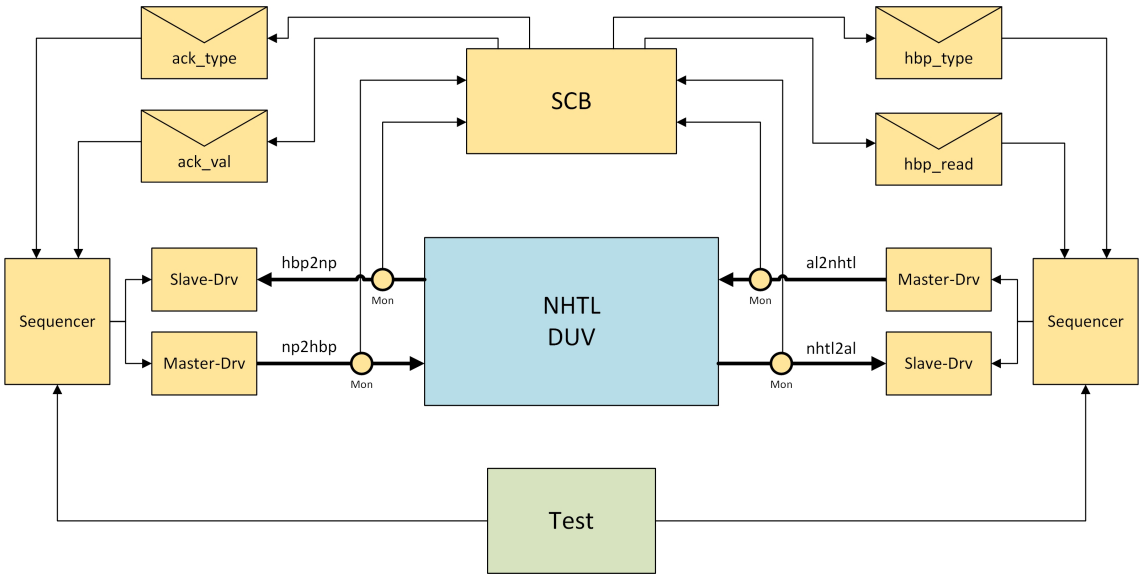


Figure 7.3: Structure of the NHTL-Testbench.

The Scoreboard (SCB) collects transactions from the interface-monitors and notifies the test-sequence about received packets trough four message-boxes. This facilitates the test to generate responses for packets at the Application-Layer and notification-acknowledges

from the network-host. The *hbp_type* message-box transports one bit of information, describing the type of the requested response packet (`1'b0`: FPGA-Config, `1'b1`: HICANN-Config). In case of FPGA-Config the request usually triggers a normal response of 1 QW length, which is signalled through the *hbp_read* message-box containing a 0. But in case of trace-data readout-request, the *hbp_read* message-box is written with a random number of requested QWs.
When the scoreboard detects a payload-notification packet at the network-port interface, it writes a `1'b1` to the *ack_type* message-box in case of a HICANN-Config notification and a `1'b0` in case of Trace-Data notification. The number of QWs to acknowledge equals the number of QWs notified and is written in both cases to the *ack_val* message-box.

7.3 Verification-Tests

The following test-sequences were implemented to verify the functional behaviour of the nhtl_top module:

7.3.1 configure_host_node

This sequence writes the configuration registers in the nhtl-registerfile through RRA network packets. The notification frequency for HICANN-Config responses is constrained to obey the configuration constraints described in Section 4.3.1 on page 31. The configuration sequence is used by the other test-sequences to initialise the DUV.

7.3.2 put_fpgaConf_test

This test-sequence charges the DUV with randomised FPGA-config requests at the network-interface and also generates responses for those requests which the scoreboard successfully detects at the AL-interface. The test will run until all asserted requests have been received at the AL-interface. In case of an error, present in the DUV, this can lead to an infinite test-loop, that has to be killed manually or with a timeout. As it happens, the generated FPGA-config packets may also trigger the generation of trace-pulse data.

7.3.3 put_hicannConf_test

This test-sequence charges the DUV with randomised HICANN-configuration packets. Those packets, that successfully reach the AL-interface trigger responses back to the network-host. Notifications arriving at the network-interface trigger an acknowledgement-notification back to the DUV. Like for the put_fpgaConf_test, this test can lead to an infinite test-loop if the DUV is incorrect and not all packets arrive at the AL-interface.

7.3.4 put_pbdata_test

This test simply feeds the network-interface with packets containing playback-data for the AL-interface. The Scoreboard will check, that the packets have all arrived at the AL-interface, after the test has finished sending them over the NP-interface.

7.3.5 put_random_test

The put_random_test combines all previously described test-sequences and randomly sends alternating FPGA-Config, HICANN-Config, and playback-data packets. Configuration packets trigger responses at the AL-interface and payload-notification packets trigger acknowledgements at the NP-interface.

7.3.6 read_test

This test sends special FPGA-config packets to the NP-interface that trigger trace-data-readout responses at the AL-interface. Notification packets are acknowledged at the network-interface.

7.4 Regression

In a regression run one or more tests are executed automatically and repeated many times with different random seeds. By doing so, the design is run under very different conditions. This helps finding bugs in the design, that usually don't occur and could easily be missed by only simulating the design a few times. When a bug is found, a report message is logged. The design can then be simulated with the respective seed that led to the bug to analyse and fix it. When the bug is fixed, the regression is rerun again to verify, that the design still works after the bug-fix.
The NHTL-regression runs the five tests, described above in Section 7.3. The configure_-host_node sequence is called by each of these tests and doesn't have to be run separately. The regression tool records the logs from all runs and sorts them into categories for successful, failed and unfinished. When a test runs for a very long time, it is aborted by timeout and logged as unfinished.

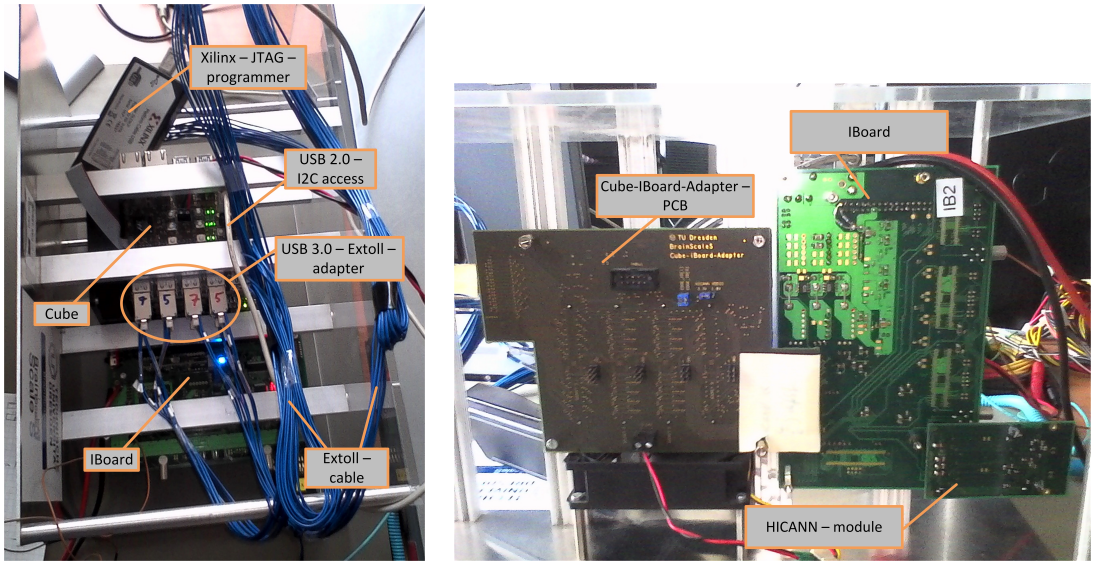
8 Testing

This chapter is about the test-setup for testing the developed FPGA-design. Section 8.1 describes the hardware test-setup also called Cube-Setup that hosts four FCPs and up to 16 HICANNs. Also the used interface adapter from USB 3.0 to EXTOLL is described here. Section 8.2 shortly describes the software environment, necessary to use the EXTOLL network and communicate with the NHTL interface in the BrainScaleS FPGAs. The individual tests to check the reachability of the registerfile and the JTAG TAP controller in the HICANN chips are described in Section 8.3.

8.1 The Test-Setup

8.1.1 Cube-Setup

For being able to test the design and its communication with Host and HICANN-chips, a special test-setup of the BrainScaleS system is provided. This setup is shown in Figure 8.1. It hosts four FCPs between an IO-board on top and an FPGA-IBoard-interface board underneath.



(a) Top-view on the Cube-Setup.

(b) Bottom-view on the Cube-Setup.

Figure 8.1: Photos of the Cube-Setup for testing of the FPGA-design and communication with Host and HICANNs.

The IO-board on top of the FCPs interfaces the network with 16 USB 3.0 plugs and four additional Ethernet connectors. Each FCP is assigned to four of these USB plugs. These four USB-ports are combined to one EXTOLL-cable adapter that directly connects the respective FPGA to the host-server containing an EXTOLL Tourmalet card.

The connection-scheme of these USB-to-EXTOLL adapters is shown in Figure 8.2. An EXTOLL cable plug provides connections for 12 lanes, each consisting of two differential signal-pairs RX+/- and TX+/-. Each of these lanes is mapped to one USB-plug respectively. Because only two of these adapters are available for testing, also only two of the four available FPGAs can be connected over the EXTOLL network to the host-server. The board also provides side-band access through a USB 2.0 connection for low level I2C configurations. The FPGAs can be programmed via a Xilinx® USB-JTAG programmer (see Figure 8.1a). The Cube-setup has to be provided with 12 V power-supply. This is realised using a conventional PC power-supply unit. The Cube-Setup is cooled by a standard PC-fan.

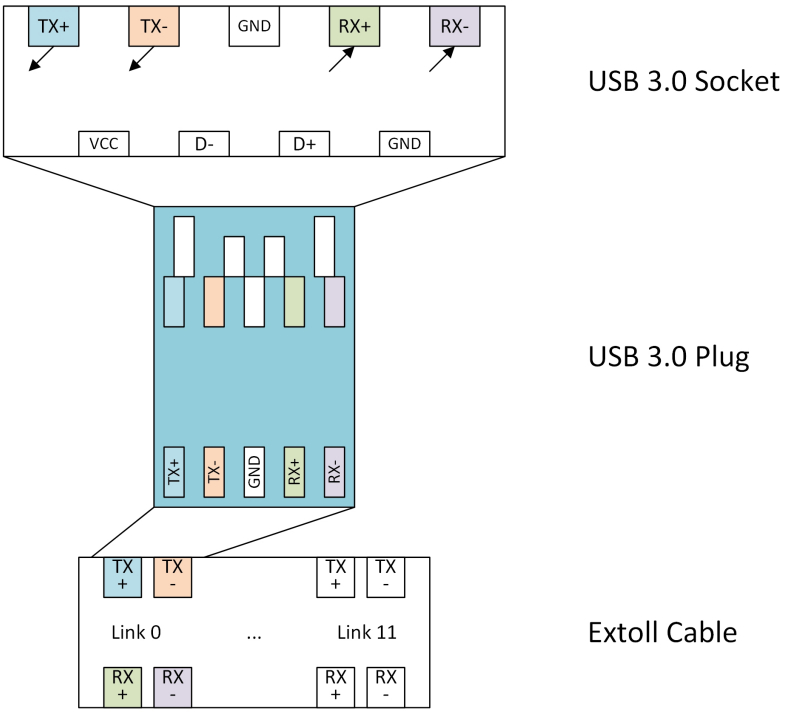


Figure 8.2: Connection scheme of the EXTOLL to USB 3.0 adapter-cable. In the USB socket part, RX and TX are defined from the FPGAs point of view. For the EXTOLL cable RX and TX are related to the host. The small arrows indicate the logic data direction.

The bottom PCB in the Cube-setup interfaces two of the FPGAs to one IBoard respectively. An IBoard interfaces the setup with up to eight HICANNs contained on four HICANN-module cards. The IBoard has to be provided with 13.8 V power-supply. To provide this voltage, an adjustable lab-power-supply unit is used. Then the IBoard can provide the HICANN-modules with the required supply-voltages. In the actual test-setup given, only one IBoard and one HICANN-module is connected. The HICANN-module used, does actually only contain one HICANN chip. The IBoard and the HICANN-module can be seen in Figure 8.1b. The IBoard is marked with a sticker, labelled *IB2* and the HICANN-module is connected to it on the bottom left in the image.
The JTAG chain, connecting the HICANN-chips with the FPGA has to be configured using a dedicated jumper-setup on the IBoard. This jumper-setup can be seen in Figure 8.3. The configuration containing the blue jumper horizontally shorts the JTAG-chain over the

non-existent HICANN-chip while the configuration with two vertical jumpers on the right connects the available HICANN to the chain.

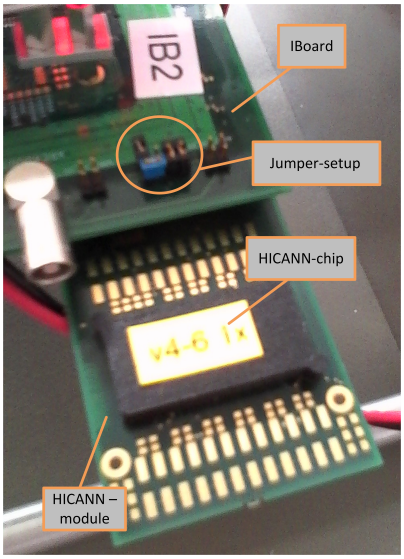


Figure 8.3: Photo of the HICANN-JTAG jumper-setup on the IBoard.

8.1.2 Wafer-Test-Setup

In addition to the Cube-Setup, which is described in the section above and located in the hardware-lab in Mannheim, also a second test setup is used. This setup connects two FPGAs of one Wafer-Module on the BrainScaleS system in Heidelberg. Each of these FPGAs is connected to eight HICANNs on the wafer. Hence, this setup allows for more realistic tests, than the Cube-Setup. As explained in Section 6.5, the FPGAs on the wafer-module are not programmed directly. Instead the Firmware is flashed into a persistent memory on the FPGA.

8.2 Software

The software required to start up the Cube-Setup for test is shortly described below.

8.2.1 HMF-Software

Before the FPGAs can be programmed using Xilinx® Vivado®, the FPGAs have to be turned on. This task can be performed via the I2C-interface. This interface is available through the USB 2.0 connector on the top-board of the Cube-setup. Alternatively the I2C-interface can directly be accessed by a Raspberry-Pi computer. When accessing the interface through the USB 2.0 connector a simple python-script on the host-computer controls the power-management functions in the Cube-Setup. When the FPGAs are all turned on, they can be programmed through the USB-JTAG programmer.
If the Cube-Setup shall use HICANNs connected to the IBoard, a second python-script has to be executed to configure the various supply-voltages provided by the IBoard for the HICANN chips. This script also uses the USB 2.0 connector on the top-PCB of the

Cube-Setup. This again can alternatively be done by a Raspberry-Pi computer directly connected to the I2C bus. This can be especially advantageous, as the Texas-Instruments® USB 2.0 chip on the Cube-Setup can sometimes behave rather unstable. However the USB-connection was sufficient for the purpose of testing in this work and therefore posed the simplest solution for configuring the Cube-Setup.

8.2.2 Extoll-Software

In order to use the Tourmalet network-card, the host-server must at first have the EXTOLL-drivers installed. When the FPGAs are turned on and programmed, the network topology as well as the node-IDs for all connected nodes have to be defined in a special file called `network.td`. When this is done, the network-discovery software called *EXTOLL-EMP* can configure the Tourmalet-card and set up the routing tables for the described topology. When the network is completely set up and configured correctly, it can now be used for communication between host and FPGAs. For this purpose a special API called *libHBP* is under development, based upon the basic EXTOLL-RMA software-API. The development of the *libHBP* is not part of this thesis work. The *libHBP* provides functions for Remote Registerfile Access (RRA) and for sending and receiving general RMA-packets. It maps the available HBP commands (described in Chapter 3) into user friendly function calls.

8.3 Individual Tests

To demonstrate the functionalities of the design, developed in the course of this work, the following tests are conducted using the Cube-setup described in Section 8.1.1.

8.3.1 Registerfile

The first simple test to be performed on the design is checking, that the registerfile is accessible over the EXTOLL network. For this purpose at first the register at address `0x8008` belonging to the sub-registerfile `info-rf` can be read, as it contains the well-known and previously configured node-ID and the Global Unique Identifier (GUID) of the respective FPGA.
When this has been successful, an arbitrary read-write register should be written and read-out in sequence. Thereby also the write-access to the registerfile can be checked.
Another test is to check, if the registerfile replacing the JTAG-TAP (`test_hicann_-if_rf`) registers in the FPGA (see Section 5.2) is correctly connected to the eight HICANN-interfaces. To check this, debug logic is inserted in the design. To do so, the respective locations have to be marked in the HDL-code before synthesis. The Xilinx® Vivado® software then automatically implements the required hardware-logic into the design. The FPGAs JTAG-programming interface can then be used with Vivado® to monitor the previously marked nets in the design at runtime. Arbitrary triggers can be defined on these nets. When the trigger asserts, the debug-logic logs 1024 clock-cycles and transfers them over the programming interface.
Now the former JTAG registers are configured using the RRA-API-commands and checked for correct implementation in the design using the debug interface.

8.3.2 JTAG

To test the JTAG interface controller (see Section 5.1), the same methodology as for the registerfile-test is used. Debug-logic is inserted into the JTAG-controller at nets defining the state of the FSM. Also the actual JTAG signals are monitored with debug-logic. Then the controller is configured and operated through RRA-commands over the EXTOLL network. For successful testing, it has to be assured, that the IBoard and HICANN-module are powered on and connected to one of the FPGAs connected to the EXTOLL network.

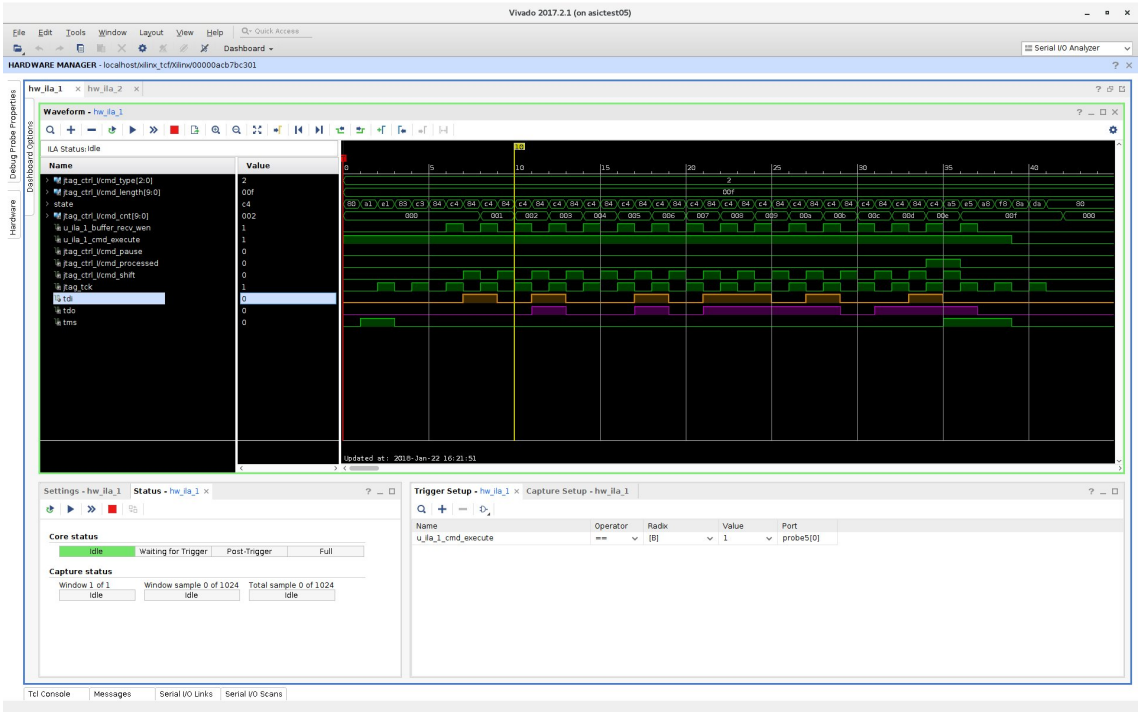


Figure 8.4: Waveform of the debug-signals while writing `0x25a5` and reading `0x77a4` to and from the SET_IBIAS register at JTAG-instruction-address `0x07` in the HICANN. The TDI-signal (output from the FPGA) is coloured orange while the TDO (input to the FPGA) is coloured magenta.

To check the functionality of the JTAG interface, the *READID* register in the HICANN registerfile is read. If the JTAG works, this register should return a value of `0x14849434`. If this was successful, a read-write register is selected to be written and read-out in sequence to check the write-access to the HICANN-JTAG interface. While doing these transfers, the JTAG-signals are monitored and verified for correctness.
A screenshot of the Vivado® Hardware Manager while monitoring the transfer of 15 bit to the SET_IBIAS JTAG register in the HICANN is shown in Figure 8.4. The trigger has been configured to start monitoring the hardware, when the `cmd_execute` bit in the JTAG-control-register has been set. The test shows, that the JTAG controller and chain work perfectly well.

9 Pulse-Routing Concept

This chapter gives a theoretic consideration of the pulse-routing design space. A first design for neighbour pulse routing has been proposed in [13], excluding the aspect of FPGA to host communication. That proposed design is going to be re-evaluated in this chapter under consideration of the developed network-interface structure.
Section 9.1 describes the generation of pulse-events and their handling at the on-wafer networks. The neuromorphic pulse-events are compared with biological spikes in neurons and synapses. Subsequently Section 9.2 gives an overview of the different routing strategies i.e. *Point-to-Point Routing*, *Multicast Routing* and *Broadcast Routing*. The different strategies are discussed in terms of applicability to the EXTOLL network. In Section 9.3, a Lookup Table (LUT) architecture is suggested for the future implementation of the pulse-routing module. Furthermore, an addressing scheme as well as a strategy for *Multicast Routing groups* is discussed. The global interrupt capabilities of the EXTOLL network are shortly described in Section 9.4 and their usability for a synchronous reset of all FPGA *systime* counters is pointed out. Finally, Section 9.5 shows the necessary modifications in the overall FPGA design to implement the pulse-event routing and to tie it directly to the EXTOLL interface.

9.1 Pulse Communication

9.1.1 Event Generation in the Neuromorphic Circuits

Biological neuronal networks handle the communication between their neurons by sending bio-electrical spikes over synapses connecting one source neuron with many destination neurons. Spikes are emitted by presynaptic neurons to the axons and received by the postsynaptic neurons at the dendrites (compare Figure 9.1). This synaptic spike communication is modelled with so-called pulse-events in the BrainScaleS NCS.
The neurons on the HICANN-chips generate pulses consisting of a freely configurable 6 bit destination synapse-address. The generated pulses are emitted into the on-wafer Level 1 (L1) network, which is configured for each experiment. These 6 bit packets are read-out and digitised into the also on-wafer-located Level 2 (L2) network. For this purpose, there are 8 Synchronous Parallel L1 (SPL1) synchronisation channels belonging to each HICANN. The 3 bit address of the respective SPL1 channel is added to the 6 bit packet, when synchronised into the L2 network. The time when these packets are generated for the L2 network is recorded and added to the pulse-event packet as a 15 bit timestamp. The resulting pulse-event packet is shown in Figure 9.2a. The L2 network now transports these packets to the FPGA HICANN-interfaces. As one FPGA concentrates the pulse-events from eight HICANNs, another 3 bit HICANN-address is added to the packet. The resulting pulse-event is shown in Figure 9.2b. [26]
When passing the HICANN-interface in the FPGA, the timestamp can optionally be modified by adding a configurable delay. This is referred to as axonal delay and results in a

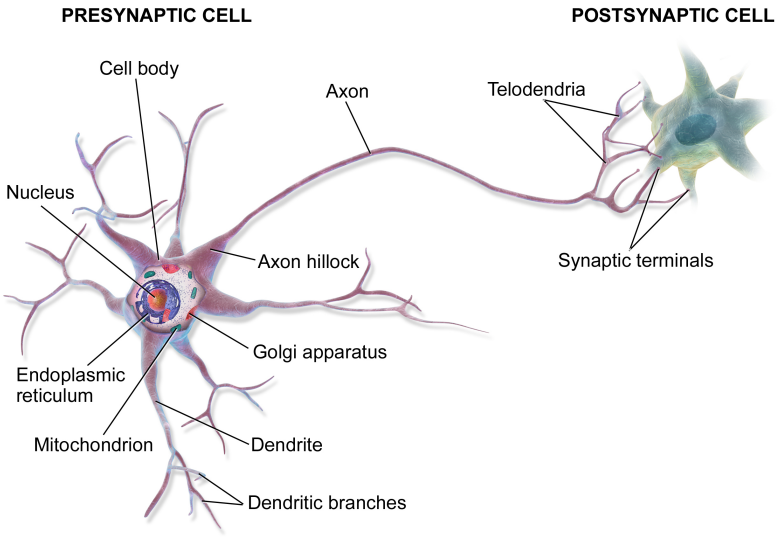
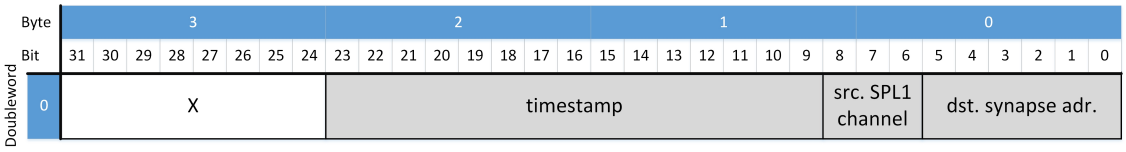
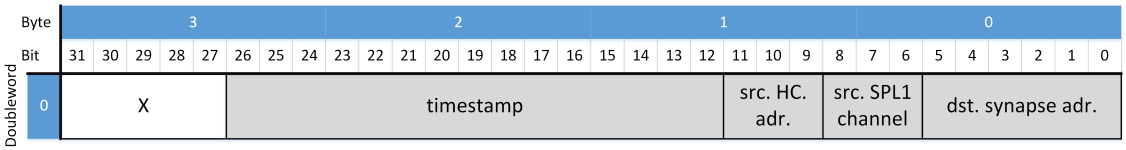


Figure 9.1: The anatomy of a multipolar neuron, showing a presynaptic and a postsynaptic cell. In particular the Axon and the Dendrite are shown. [25]

timestamp, now representing the desired arrival-time at the destination HICANN. For each source Neuron on a HICANN, a different delay may be configured in the respective FPGA HICANN-interface. [26]



(a) Pulse-event format comming from the HICANNs to the FPGA-HICANN-interfaces.



(b) Pulse-event format used in AL and network with additional source HICANN address (bits {11:9}).

Figure 9.2: Pulse-event-Formats in the BrainScaleS System.

9.1.2 Event Routing between Neuromorphic Circuits

Based on the given source HICANN- and SPL1-address, the routing logic has to transport the pulse-event to the target HICANN. The destination HICANN will then inject the received pulse-events into the SPL1-channels. Finally the 6 bit synapse-address is injected to the L1 network where the target-neuromorphic circuits can consume the incoming pulse-events. The neurons can again be configured, to listen only to a specific 6 bit target-address. [26]

Biological synapses as depicted in Figure 9.1 always have a constant propagation delay. In contrast to that, in computational packet-based networks, the propagation delay of a packet strongly depends on the stress, currently posed on the whole network. For being able to accurately model the biological networks, a Neuromorphic Computing System has to keep this temporal jitter as low as possible. This jitter should not exceed 1 ms biological time, corresponding to 100 ns with the current speedup-factor of 10^4 in the BrainScaleS NCS [26].
To achieve this, pulse-events are buffered when they arrive at a destination HICANN-interface and sorted using a custom developed heap sorting memory according to their arrival-timestamp. When the difference between the next pulse's timestamp and the current system-time is lower than a pre-configured limit, the respective pulse-event is released down to the HICANN-chip. This mechanism ensures, that the events always arrive at the destination neuron according to their configured synaptic delay. The delays must thereby be defined according to the expected network-transmission delay plus a bit of safety overhead. [8, p. 148 ff.]
Finally, the on-wafer SPL1 channel will emit the pulse-event exactly at the given destination timestamp. The hardware buffer-space for achieving this is limited to eight entries per channel. Therefore the FPGA mustn't forward "to many events to early". [26]

In some future NCS implementation it could be considered to not only model axonal delays at the sending FPGA, but also dendritic delays at the receiving FPGAs. However, this will require much more buffer-space and configurable logic to define individual dendritic delays for each neuron and to buffer the events for the configured time before they are released.

9.2 Routing Strategies

9.2.1 Neuromorphic Network Topologies

Neurons in biological neuronal networks usually fan-out to several hundreds of target neurons and fan-in from several hundreds to thousands of source neurons. Ideally these neurons are mapped to the NCS in a way that most of these connections can be realised in the on-wafer L1 network. As the modelled networks grow larger, this constraint cannot fully be satisfied and the network has to be spread over several wafers. Many requirements on the inter-wafer routing strategies can be derived from the modelled neuronal network topology. As described and analysed in [27, Chapter 5.1], there are two major topologies: Uniform Random Networks (URNs) and Local Random Networks (LRNs).
In URNs all neurons have the same probability to be connected to each other, whereas in LRNs the connection-probability decays exponentially with the connection-distance. Despite of this probability-decay, most presynaptic neurons for a given neuron in a biological neural network are located outside the "local volume". This is because the considered cylindrical volume increases quadratically with the radius. [28] Therefore there might still be a significant amount of synaptic connections to be modelled across several wafers.
A special case of LRNs are feed-forward connected neuron populations. Based on these topologies, different routing strategies can be imagined.

Point-to-Point Routing

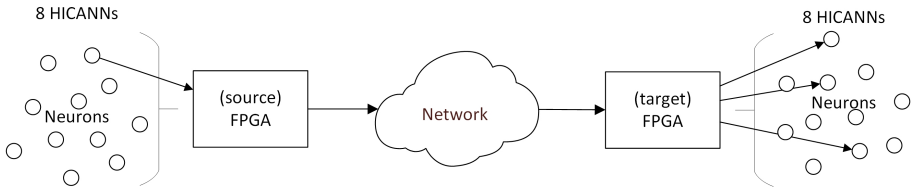


Figure 9.3: Schematic diagram of the Point-to-Point routing topology. Only one source- and destination-FPGA is shown for the sake of simplicity.

With this routing strategy, one neuron associated with a source FPGA is connected to destination neurons at exactly one target FPGA. The router at each FPGA will have to calculate a target HICANN and SPL1 channel based on the respective source HICANN and SPL1 channel. Pulse events may be replicated to several HICANNs and / or SPL1 channels at the target FPGA, realising a "local multicast" for the neuron populations. [26]
A schematic view of this topology is shown in Figure 9.3. To keep the simplicity, only one source-target pair is depicted. In a real implementation there would of course be different destination FPGAs for most of the neurons associated to the source-FPGA.
This strategy would well be suitable for Local Random Networks where neurons have a relatively low fanout, i.e. have relatively few postsynaptic connections still fitting to one FPGA.
This routing strategy requires a lookup-table on the sending side of the connections. The table has to map the incoming pulse-events to a destination Node ID (NDID). To realise the local multicast, a second mapping has to be done at the target-FPGA, distributing the incoming packets to one or multiple HICANNs and SPL1-channels.

Multicast Routing

With this strategy, in contrast to Point-to-Point routing, neurons associated with one FPGA can be connected to several neurons located at more than one target-FPGA. In addition to that, also the "local multicast" can still be implemented. This routing model is very close to biology, as connections between neurons can be modelled most flexibly. [26]
Figure 9.4 shows again a schematic view of this routing model. As can be seen, one FPGA fans out to several other FPGAs, but there are still many FPGAs in the network, that are not addressed by the pulse-event transaction from a particular neuron.
This model is, like Point-to-Point routing, suitable for LRNs, but allows for much higher neuron-fanout, as postsynaptic connections now can be distributed over several FPGAs.
The EXTOLL network already supports multicast-messages by replacing the destination NDID with a Multicast Group ID (MC-GID) and setting the MC bit in the SOP-header. The EXTOLL network only supports 64 different MC-GIDs. However, these MC-GIDs can be reused for disjunct multicast-groups giving a lot of flexibility to the EXTOLL multicast-implementation. The only actual restriction is, that the same MC-GID must not be used twice for one EXTOLL Tourmalet inport (see Figure 1.3).

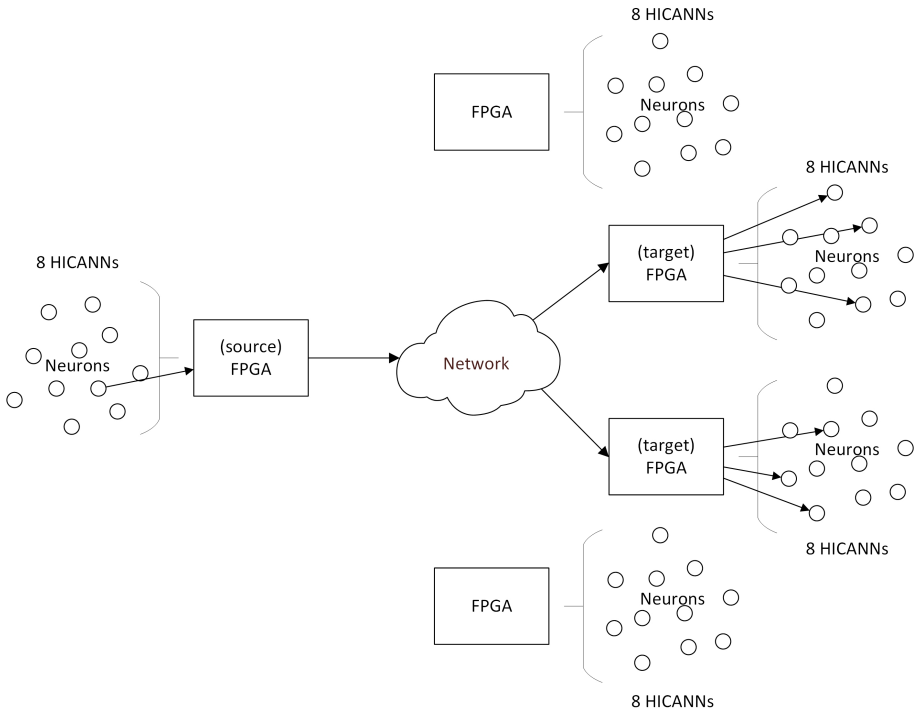


Figure 9.4: Schematic diagram of the Multicast routing topology. Only one source-FPGA is shown for the sake of simplicity.

Broadcast Routing

When the modelled neuronal network is getting very large, each neuron will send and receive pulse-events to and from several thousands of post- and presynaptic neurons. Depending on the mapping of these source- and target-neurons, the connectivity of one neuron could be distributed over almost all other FPGAs. In this case it might be the best solution to broadcast every outgoing event to all other FPGAs in the network. The target FPGAs would then have to filter the incoming stream of pulses for relevant events to forward them to the intended SPL1 channels. [26]
This is illustrated in Figure 9.5. In difference to multicast routing (see Figure 9.4) the pulses are now forwarded to all FPGAs, but filtered there, while for multicast routing they don't need to be filtered, as they are not sent to irrelevant FPGAs.
This strategy would especially be suitable for big Uniform Random Networks where the connection-probability is the same for every neuron-pair in the network.
However, broadcasting every upcoming event to all FPGAs in the network is a rather bad strategy, as any interconnection-network would easily develop massive congestion with this kind of load. Moreover, the target-FPGAs would be massively congested while filtering out relevant events and local-multicasting them to several HICANNs and SPL1-channels.

SpiNNaker-like Multicast Routing

The SpiNNaker System implements pulse routing, using a special custom-developed packet-router, which is located on the neuromorphic computing chips. The routed packets do not contain information about their destination. The routing decisions are made solely

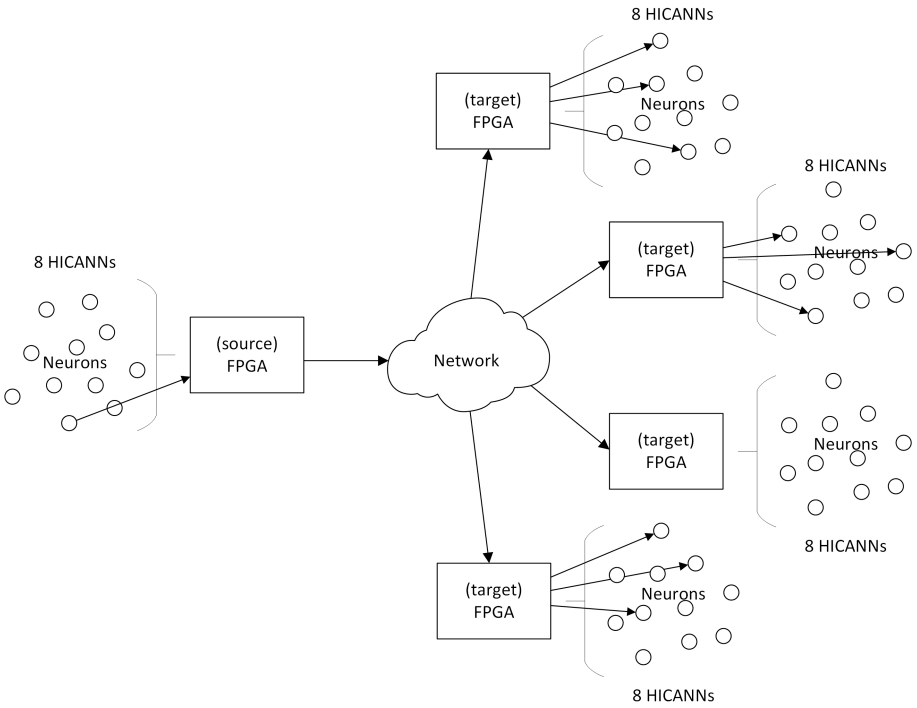


Figure 9.5: Schematic diagram of the Broadcast routing topology. Only one source-FPGA is shown for the sake of simplicity.

depending on the source neuron-address. The SpiNNaker-network is organised as a two-dimensional triangular torus. Deadlocks and lifelocks are avoided by dropping packets in the routers. The SpiNNaker network supports multicast operations and specially developed routing-algorithms are used to calculate the routing tables. [29]
Direct reuse of the SpiNNaker routing algorithms is not possible, as the EXTOLL network implements routing based on the destination Node IDs of the participating network nodes. Furthermore, the EXTOLL network-switching nodes are not able to evaluate the packet-payload directly for making the next routing-decision as SpiNNaker does. This would have to be realised by sending pulse-events hopping from one FPGA to another and evaluating the packet at each hop. This would inevitably introduce higher latencies and logic requirements for the FPGAs. Therefore this is not considered a viable solution for the BrainScaleS routing with EXTOLL.

9.3 Proposing a Routing Architecture

9.3.1 Table-based Routing

The desired routing strategy for the BrainScaleS system is a multicast FPGA-routing combined with local multicast for HICANNs and SPL1-channels. As described in Section 9.2.1 on page 62, an NCS might require very many connections between modelled neurons, which will ideally be mostly realised in the on-wafer networks of the BrainScaleS system. However it is likely, that there are still a lot of off-wafer connections to be implemented. Defining a routing-table at the sending side is not a great problem, as there are only 64 SPL1-channels under control of an FPGA, implying also 64 entries

in the lookup-table. The receiving side of the connection is more tricky, as the local multicast will have to distinguish the incoming pulses from their origin in the network. This network-source is the 16 bit NDID combined with at least the 3 bit HICANN-ID and the 3 bit SPL1-channel. This yields a total theoretically possible number of 2^{22} connection sources. Under the assumption of maximally connecting to four neighbouring Wafer Modules, there are still

$$4[\text{Wafers}] \times 48[\text{FPGAs}] \times 64[\text{SPL1-channels}] = 12\,288$$

possible connection-sources. A routing-table with such many entries would be horribly slow and would not even fit into the FPGA.

To solve this problem reasonable restrictions have to be made for the routing. It is therefore considered reasonable to restrict the possible number of connection-sources to a relatively low value. The algorithm, mapping the neural network to the BrainScaleS hardware

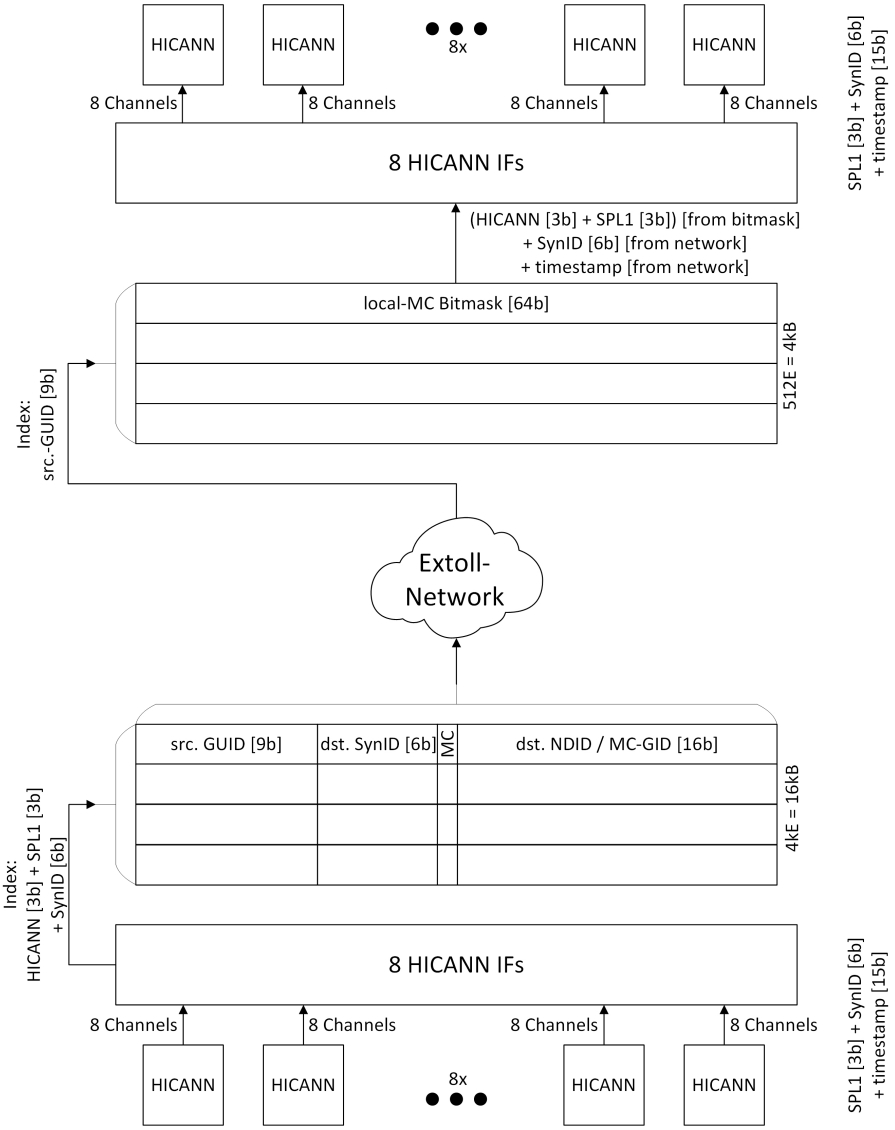


Figure 9.6: Structure of the lookup-tables for pulse-routing between source- and destination-FPGAs.

must hereby be constrained to place the connected neurons as locally as possible across the Wafer Modules to implement as much connections as possible through the on-wafer networks. The connection sources in the off-wafer network are not identified with their source-NDID and SPL1-channel number, but with a configurable GUID that uniquely identifies each connection source-FPGA for one destination-FPGA. Like the EXTOLL Multicast Group ID, these GUIDs may be reused for disjunct fanin-groups.

Figure 9.6 shows the suggested routing-table architecture. Pulse-events arising from the neuromorphic hardware at one FPGA contain a 12 bit identifier which is used for indexing-access to the first routing-table at the sending FPGA. This table contains $2^{12} = 4\,\mathrm{kE}$, each consisting of the destination NDID, a flag for the MC-bit, as well as a new destination synapse-ID and the configured source GUID. By also making the destination synapse-ID configurable, more flexibility is given to the mapping-algorithm for the neuronal network. The information from this routing-table is packed together with the timestamp from the pulse-event into a network-packet and sent through the network to the desired destination. The size of this table-buffer calculates to:

$$4\,\mathrm{kE} \times 4\,\mathrm{B\,E^{-1}} = 16\,\mathrm{kB}$$

With BRAMs having a size of 4.5 kB in the used FPGA, this can be easily implemented using four Block-RAMs.

At the target-FPGA the 9 bit source-GUID is used to index a second lookup-table containing 64 bit masking-entries. These bit-masks tell the FPGA-logic the HICANN- and SPL1-channel-IDs to forward the received pulse-event. The destination synapse-ID and the timestamp are both extracted from the network-packet. The size of this table-buffer calculates to:

$$512\,\mathrm{E} \times 8\,\mathrm{B\,E^{-1}} = 4\,\mathrm{kB}$$

This adds one more BRAM to the FPGA-implementation. If there are still BRAM-resources left, it could be considered to expand the source-GUID. With a BRAM-size of 4.5 kB up to 4 bit could be added, while still only using four BRAMs for the first routing table. This would imply $2^{13} = 8\,\mathrm{kE}$ in the second lookup-table, finally consuming 64 kB and 16 Block-RAMs.

9.3.2 Node-ID Addressing Scheme

The EXTOLL Node IDs for the FPGAs are configured during the networks setup-phase. To simplify the configuration, the NDID can be divided into different tags. One such tag would be the number of the FPGA with respect to the wafer belonging to that FPGA. As there are 48 FPGAs at each wafer, 6 bit are sufficient to encode this part of the NDID. The remaining 10 bit of the Node ID would then denote the number of the Wafer Modules. The host-nodes could either be encoded in the so far unused FPGA-IDs, or with a 1 bit flag e.g. at bit 15 of the NDID. The latter solution restricts the possible number of Wafer Modules in the system from 1024 to only 512, which is still a huge number of Wafer Modules. This layout of the NDID is depicted in Figure 9.7.

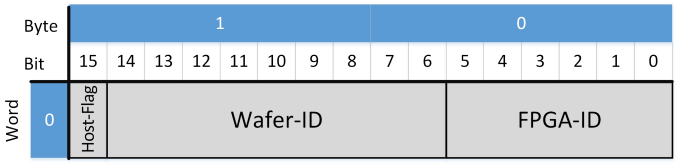


Figure 9.7: Layout of the NDID-addressing.

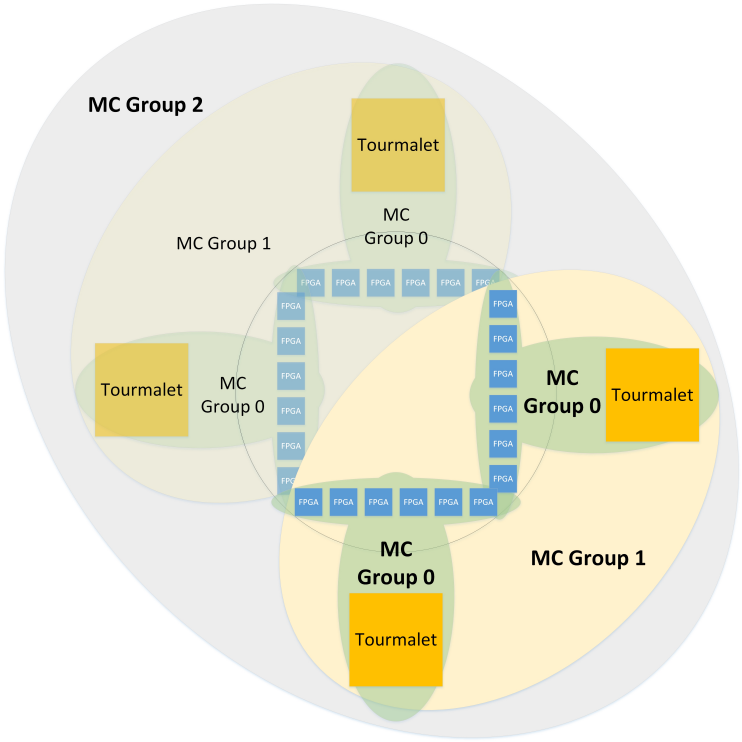


Figure 9.8: Possible EXTOLL Multicast Groups between FPGAs on one wafer.

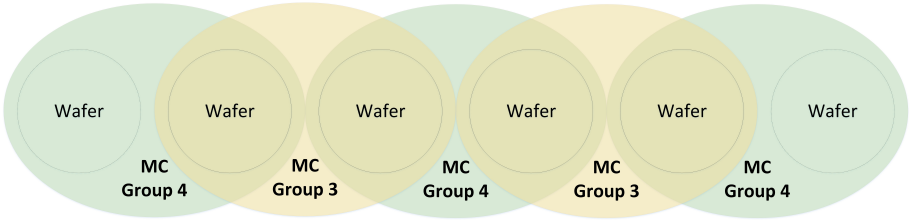


Figure 9.9: Possible EXTOLL non-overlapping Multicast Groups between different wafers.

9.3.3 Multicast Group Communication

Using the EXTOLL multicast mechanism, some possible communication schemes shall be mentioned. As multicast groups can be reused as long as they don't overlap on the EXTOLL Tourmalet inports, each concentrator-node grouping six FPGAs (see Section 1.4), could have its own disjunct multicast group. Additional multicast groups can be defined to combine neighbouring FPGAs and wafers as shown in Figure 9.8 and 9.9. Based on this grouping scheme, different communication paradigms would be conceivable. Of course, one FPGA can simply send messages to the groups, it belongs to. This can either be a local group (MC Group 0 in Figure 9.8, see Figure 9.10) or some bigger

group with more than one Tourmalet (see Figure 9.11). To use this routing paradigm, an FPGA just has to send the desired packet out to the network with the correct MC-GID set. The Tourmalet network cards will then take care of the routing automatically and broadcast the message to the correct outports.

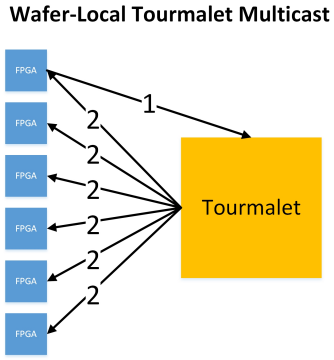


Figure 9.10: One FPGA broadcasts to its local FPGA neighbours.

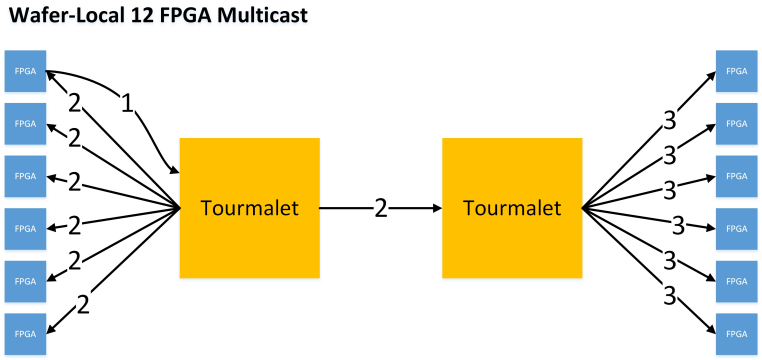


Figure 9.11: One FPGA broadcasts to local and neighbouring Tourmalet FPGA neighbours.

When introducing one FPGA routing-hop, it is also possible to broadcast a message to a remote multicast-group. To do so, the packet must at first be sent to a remote FPGA, contained in the desired multicast-group. This FPGA can then forward the message to a local multicast group (see Figure 9.12). It should be mentioned, that for this paradigm, the routing-table architecture described in Section 9.3.1 has to be adapted accordingly make the detour-hop possible.

9.3.4 Pulse-Event Accumulation

As one EXTOLL-packet has a header overhead of at least two QWs (see Chapter 3), sending one packet (29 bit payload) is quite inefficient. To improve this, [13] suggested to accumulate pulse-events with the same destination-address. However, this can only be done, if there is enough time left between the current systime and the arrival-timestamp to do the accumulation and send the message over the network. Again this requires more buffer-resources in the FPGAs.

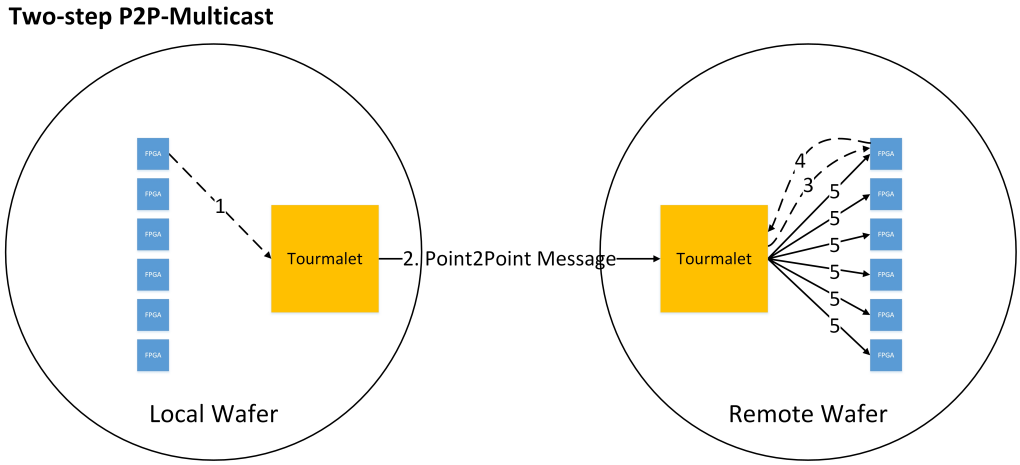


Figure 9.12: FPGA sends single message to remote FPGA - Remote FPGA broadcasts to its wafer-local FPGA neighbours.

9.4 Global Interrupt

As described in Section 9.1 and 9.3.4 above, the pulse communication and routing strategy strongly depend on a globally synchronised *systime* counter. Starting this counter at the same time on all FPGAs is not trivial. Fortunately the EXTOLL network already implements global interrupt functionality in its hardware. The global interrupt units can be used to reset and start the systime counters coincidentally on all FPGAs at the initialisation-phase of the system.
The global interrupt unit works similarly as the “down phase” of an integrated hardware barrier network (also implemented in EXTOLL). The nodes are virtually organised in a tree-structure and the root-node can notify all its child-nodes by sending one message to the network. For this interrupt-message, the EXTOLL network uses a special barrier-cell, depicted in Figure 9.13 [30, p. 25].

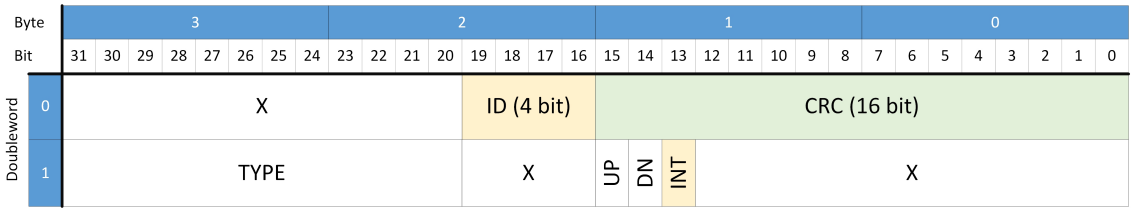


Figure 9.13: Format of an EXTOLL Barrier cell

The EXTOLL hardware supports up to 16 barriers and up to four global interrupt IDs. For the barrier, the UP- and DN-bits mark whether the message belongs to the up- or the down-phase of the barrier. An interrupt is simply marked with the INT-bit. When the interrupt message travels through the network-tree, it arrives at the network nodes at different instances in time. To regardlessly ensure a synchronous global interrupt, a delay mechanism is built into the EXTOLL hardware: In the initialisation phase, the network connections are measured for their individual transmission delays. The measurement results are stored in a local register in the hardware. When a global interrupt is issued through the network, each hardware node waits for the stored delay-time until the interrupt is triggered to the attached target logic. [30, p. 42]

Besides synchronising the *systime* reset for initialisation, the EXTOLL global interrupt can also be used to help evaluating the traced pulse timestamps from different Wafer Modules. As mentioned in Section 4.6, the oscillator-sources on the Wafer Modules run gradually apart. The global interrupt could trigger the FPGAs to send their current *systime* when the experiment is finished. The returned results can then be used to calculate the relative oscillator drift between the FPGAs.

9.5 Modifications in the FPGA Design

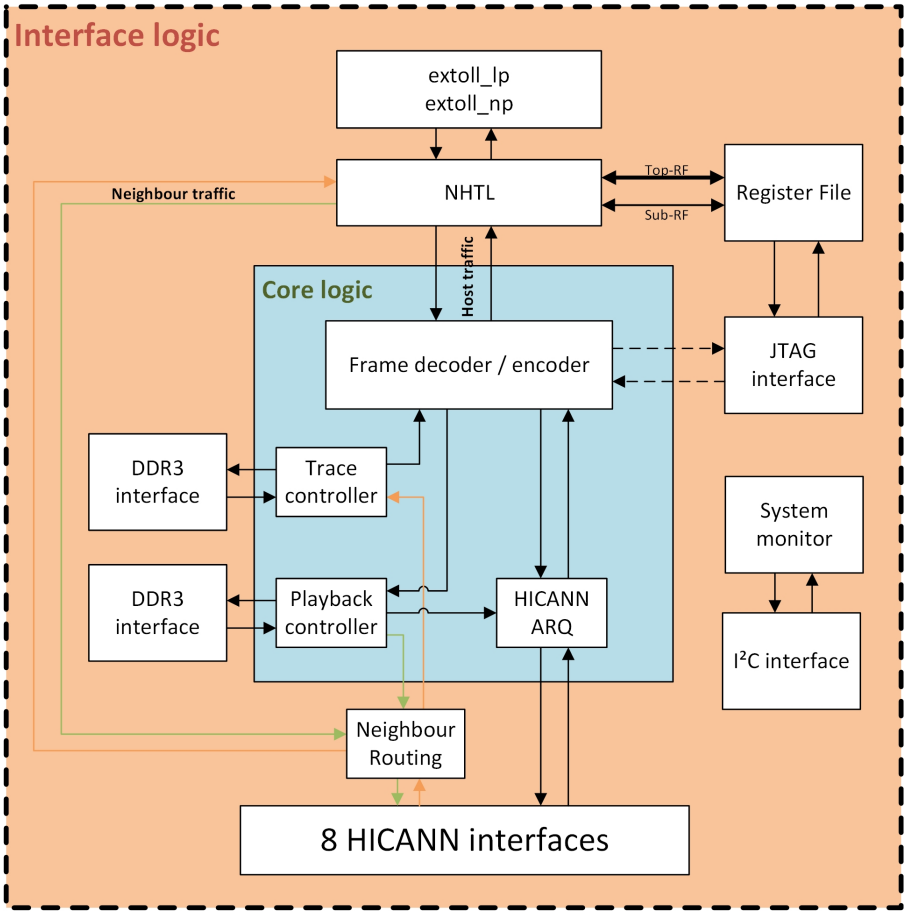


Figure 9.14: Blockdiagram of the communication-FPGA design as designed for EXTOLL-communication with pulse-routing.

To enable pulse-routing and FPGA-to-FPGA communication in the design, the modules and interfaces have to be slightly rearranged. Some necessary changes in the design-structure are depicted in Figure 9.14. Pulses coming through the network from other FPGAs have to be recognised at the NHTL-module. These pulses are forwarded directly to the HICANN interfaces, where they compete with pulses coming from the playback-memory. To solve this conflict, an arbiter has to be introduced in front of the HICANN-interfaces. Also these pulses have to be sent to the right HICANN-interface, as there are eight of them in one FPGA. In the other direction, a routing-decision has to be made depending on the source neuron-ID. The router also has to decide, whether the pulse-events shall be recorded in trace-memory or sent to another FPGA. In the latter case, the

source neuron identifier together with the source HICANN number is taken to look up the destination NDID for the EXTOLL network as described in Section 9.3.1. An alternative behaviour would be to simply trace all events, whether they are sent to a remote node or not.
To implement these changes in the design, also the interface of the NHTL module has to be extended for being able to directly interact with the HICANN interfaces.

10 Conclusion and Future Work

10.1 Conclusion

With this work, a new network interface has been developed and implemented for the BrainScaleS system, enabling the use of the EXTOLL network. The FPGAs can now communicate with a host computer over EXTOLL to send configurations, write the playback-memory or readout the experimental results from the trace-memory. For this purpose, the EXTOLL RMA protocol has been used and modified for the needs of this application. Furthermore, JTAG access to the HICANN chips has been optimized and implemented using a registerfile-driven JTAG master-controller. The design has been verified using a UVM testbench and successfully tested on the implemented FPGAs.
Last but not least, a theoretic evaluation of the pulse-routing design-space has been conducted to facilitate a future extension of the design with a pulse-event routing module. This will also enable the BrainScaleS system FPGAs to communicate not only with a host, but also with each other. Thereby the Wafer Modules containing the neuronal network models can be combined to form a larger neural network.
Generally, the EXTOLL network provides significant improvements compared with Gigabit Ethernet. EXTOLL provides up to 100 Gbps bandwidth and hop-latencies under 60 ns. Also the conventional compute cluster, which is part of the BrainScaleS system, will significantly benefit from the EXTOLL network.

10.2 Future Work

This work is not complete in the sense of using the full capabilities of the EXTOLL interconnection network within the BrainScaleS system. Therefore, as Chapter 9 already indicates, the future work for this project will comprise the following tasks:

- **Implementation of a pulse router:** A module has to be developed and implemented in the FPGA design to provide the pulse-routing functionality, described in Chapter 9. The requirements, analysed in this thesis, shall therefore be taken into account.
- **Include the global interrupt:** The global interrupt functionality, described in Section 9.4 is not yet implemented in the FPGA version of the EXTOLL LP.
- **Implement the global interrupt software:** The global interrupt has not been used before yet. The EXTOLL software-stack has to be extended to support this functionality, which is already contained in the EXTOLL Tourmalet network-cards.
- **Finish the libHBP:** Not all features of the NHTL interface are yet implemented in software for the host-computer. Especially the ringbuffers for receiving answers and trace-data from the FPGAs still have to be implemented.

- **Adapt the BrainScaleS software:** The software controlling the BrainScaleS system has to be adapted to use the libHBP to be able to cope with the newly developed FPGA design.
- **Implementation of trace memory bypass:** It is envisioned to implement a direct feedback algorithm on the current traced pulses. For this purpose, the pulses are not kept in the trace-memory until the end of the experiment, but more or less directly sent to the host. The NHTL design is already prepared for this scenario by sending the traced pulses to a ringbuffer in host memory. The HMF core-logic will have to be changed accordingly in the future to implement this behaviour.
- **Implementation of playback memory bypass (SpiNNaker interface):** In the current Ethernet implementation a SpiNNaker interface provided direct access to the HICANNs. This SpiNNaker interface is not contained in the Extoll-design because it was directly connected to the Ethernet interface module. Including the SpiNNaker interface would have been complicated, as either the NHTL would have to imitate the Ethernet interface structure, or the SpiNNaker interface module would have to be redesigned. Fortunately, the direct access to the HICANNs will already be possible when pulse routing is implemented. The host software will then have to imitate an FPGA while communicating with the FPGAs.

Appendix

A Acronyms

AL Application-Layer. 9, 10, 17, 19, 23–27, 33, 34, 52–54, 61, 77

AnaRM Analog Readout Module. 4

API Application Programming Interface. 4, 58

ARM Acorn RISC Machines / Advanced RISC Machines. 2

ARQ Automatic Repeat reQuest. 9, 42

ASIC Application Specific Integrated Circuit. 2, 6, 17

ATU Address-Translation-Unit. 12

AVC Adaptive Virtual Channel. 12

BFM Bus Functional Model. 51

BRAM Block-RAM. 46, 49, 67

CLB Configurable Logic Block. 46

CMOS Complementary Metal-Oxide-Semiconductor. 2

CMT Clock Management Tile. 46

CRC Cyclic Redundancy Checksum. 12

DDR Double Data Rate. 9, 16, 17, 47, 48

DEMUX Demultiplexer. 25, 77

DMA Direct Memory Access. 13, 18

DR Data Register. 42, 43

DRC Design Rule Check. 46

Drv Sequence Driver. 51, 52

DSP Digital Signal Processor. 36–38, 46, 49, 78

DUV Design under Verification. 51–53

DVC Deterministic Virtual Channel. 12

E Unit for table-Entries. 67

EOP End Of Packet. 12, 23, 26, 34

EOT End Of Trace. 19, 20

ERA Excellerate Read Access. 13, 33

EWA Excellerate Write Access. 12, 33

EXTOLL Extended Atomic Low Latency (ATOLL). v, vi, 1, 4–12, 17, 23, 24, 26, 29, 31, 35, 36, 39–41, 43, 45, 47, 48, 52, 55, 56, 58–60, 63, 65, 67–71, 73, 77–79

FCP FPGA Communication PCB. 2–6, 16, 17, 48, 55

FF Flip Flop. 46, 49

FIFO First In First Out. 15, 17, 23–28, 31–33, 35–38, 40, 77

FLOP Floating Point Operation. v, 1

FLOPs Floating Point Operations per second. v, 1

FPGA Field Programmable Gate Array. v, 1–3, 5–13, 15–21, 24, 26–29, 31–33, 36, 37, 40–42, 44–50, 53–74, 77–79

FSM Finite State Machine. 24–27, 32–34, 37, 39, 41–44, 59, 77, 78

GT Gigabit Transceiver. 49

GUI Graphical User Interface. 46

GUID Global Unique Identifier. 58, 67

HBP Human Brain Project. v, 1, 2, 9, 58

HDL Hardware Definition Language. 58

HICANN High-Input Count Analog Neuronal Network Chip. 2, 3, 8, 9, 16–20, 24, 30–33, 40–42, 44, 45, 47, 48, 53–67, 71, 73, 74, 78

HMF Hybrid Multiscale Facility. 3, 9, 13, 15, 17, 18, 22–24, 27, 40, 42, 74

HPAC High Performance Analytics & Computing. 1

HPC High-Performance Computing. 4

HW Hardware. 9

I2C Inter Integrated Circuit. 4, 9, 48, 55, 57, 58

IBoard Interface Board to the HICANN-Modules in the Cube-Setup.. 55–57, 59, 78

INT Interrupt. 12, 33

IO Input Output. 49

IP Intellectual Property. 40

IR Instruction Register. 42, 43

JTAG Joint Test Action Group. 7, 9, 10, 16, 17, 19, 32, 41–45, 50, 55–59, 73, 78, 79

KIP Kirchhof-Institute for Physics. v

L1 Level 1. 60–62

L2 Level 2. 60

libHBP special API, used to integrate the communication protocoll in software.. 58, 73

LP Link-Port. 9, 47, 48, 73

LRN Local Random Network. 62, 63

LSW Least Significant Word. 24–26

LUT Lookup Table. 46, 49, 60

MC Multicast. 11, 63, 67

MC-GID Multicast Group ID. 63, 67, 69

MMCM Mixed-Mode Clock Manager. 40, 46, 49

Mon Monitor. 51, 52

MSW Most Significant Word. 24–26

MTU Maximum Transmission Unit. 4, 12

MUX Multiplexer. 25, 77

NCP Neuromorphic Computing Platform. 2

NCS Neuromorphic Computing System. 2, 60, 62, 65

NDID Node ID. 63, 65–68, 71, 79

NEST Neural Simulation Technology. 4

NEURON A flexible and powerful simulator of neurons and networks.. 4

NHTL Network HMF Transaction Layer. 9, 11, 12, 17, 18, 20, 22–29, 31–36, 38, 39, 48, 51, 52, 54, 55, 71, 73, 74, 77–79

NIC Network Interface Controller. 4

NM-PM Neuromorphic Physical Model. 2, 4

NP Network-Port. 9, 12, 23, 26, 47, 48, 53, 54

NTR Notification Replicate. 13, 33

PCB Printed Circuit Board. 2, 5, 6, 55, 57, 77

PCIe Peripheral Component Interconnect Express. 5

PDID Protection Domain ID. 12, 13, 15, 18, 19

PLL Phase Locked Loop. 40, 46, 49

PyNN A Python package for simulator-independent specification of neuronal network models.. 4

QW Quad Word. 9, 11–13, 17–20, 23, 24, 26, 29, 33–35, 37, 44, 53, 69, 77

RAM Random Access Memory. 9, 44, 46, 67

RF Register File. 10, 27, 28, 32, 45, 78

RMA Remote Memory Access. 4, 11–15, 17, 18, 26–28, 32, 33, 43, 58, 73, 77, 79

RRA Remote Registerfile Access. 12, 13, 15, 17, 22, 24, 27, 28, 32, 33, 43, 53, 58, 59

SCB Scoreboard. 51–53

SOP Start Of Packet. 11–13, 15, 18, 23, 26, 63, 77

SpiNNaker Spiking Neural Network Architecture. 2, 64, 65, 74

SPL1 Synchronous Parallel L1. 60–67

TAP Test Access Port. 41–45, 55, 58, 78

TC Traffic Class. 12

TCK Test Clock. 41–43, 78

TCL Tool Command Language. 9

TDI Test Data In. 41, 42

TDO Test Data Out. 41, 42

TE Translate Enable. 12, 13, 19

TMS Test Mode Select. 41, 42, 78

TRST Test Reset. 41, 42

TU Target Unit. 11

UDP User Datagram Protocol. 9, 42

UMC United Microelectronics Corporation. 2

URN Uniform Random Network. 62, 64

USB Universal Serial Bus. v, vi, 1, 5, 6, 50, 55–58, 78

UVC Universal Verification Component. 51

UVM Universal Verification Methodology. 51, 52, 73, 78

VPID Virtual Process ID. 11–14, 18, 19

B Lists

B.1 List of Figures

B.2 List of Tables

C Bibliography

[1] (2017, November) Top 500 List. Top500. Accessed 08 Jan 2018. [Online]. Available: https://www.top500.org/lists/2017/11/

[2] (2017, November) Green 500 List. Top500. Accessed 08 Jan 2018. [Online]. Available: https://www.top500.org/green500/lists/2017/11/

[3] H. Markram, "The human brain project." *Scientific American*, vol. 306, no. 6, pp. 50 – 55, 2012. [Online]. Available: http://www.redi-bw.de/db/ebsco.php/search.ebscohost.com/login.aspx%3fdirect%3dtrue%26db%3dbuh%26AN%3d75207402%26site%3dehost-live

[4] FET Flagships. European Commission. Accessed 10 Nov 2017. [Online]. Available: http://ec.europa.eu/programmes/horizon2020/en/h2020-section/fet-flagships

[5] What is Horizon 2020. European Commission. Accessed 10 Nov 2017. [Online]. Available: http://ec.europa.eu/programmes/horizon2020/en/what-horizon-2020

[6] Human Brain Project. Accessed 10 Nov 2017. [Online]. Available: https://www.humanbrainproject.eu/en/

[7] Neuromorphic Computing. Human Brain Project. Accessed 10 Nov 2017. [Online]. Available: https://www.humanbrainproject.eu/en/silicon-brains/

[8] *Neuromorphic Platform Specification*, Human Brain Project, July 2017, git 7786ee3.

[9] (2016) Technology Overview. EXTOLL GmbH. Accessed 08 Jan 2018. [Online]. Available: http://www.extoll.de/images/pdf/Extoll_Technology_Overview_2016.pdf

[10] Technology_Diagramm_new.jpg. EXTOLL GmbH. Accessed 08 Jan 2018. [Online]. Available: http://www.extoll.de/images/Einzelbilder/Technology_Diagramm_new.jpg

[11] Tourmalet_pic_new.jpg. EXTOLL GmbH. Accessed 08 Jan 2018. [Online]. Available: http://www.extoll.de/images/Einzelbilder/Tourmalet_pic_new.jpg

[12] C. Leibig, "HBP EXTOLL Networking v2," Institute for Computer Engineering, Computer Architecture Group, Tech. Rep., 2014, Internal technical documentation.

[13] F. Beuttenmüller, "Interfacing a Neuronal Accelerator to a High Performance Computing System," Bachelor's Thesis, Department of Physics and Astronomy, University of Heidelberg, 2014.

[14] unihd-cag/odfi-rfg. github; Institue for Computer Engineering, Computer Architecture Group. Accessed 15 Jan 2018. [Online]. Available: https://github.com/unihd-cag/odfi-rfg

[15] A. Giese, B. Kalisch, and Dr. M. Nüssle, "RMA2 Specification," Institue for Computer Engineering, Computer Architecture Group, Tech. Rep., 2012, revision 2.0.4, CAG Confidential.

[16] B. Geib, "EXTOLL Network Port Specification," Chair of Computer Architecture, University of Heidelberg, Tech. Rep., 2012, Version 2.0, internal specification document.

[17] *Virtex-6 FPGA DSP48E1 Slice*, XILINX®, February 2011, User Guide, Revision 1.3.

[18] *2.5V CMOS Compatible SMD Crystal Oscillator: ASE2 SERIES*, Abracon Corporation, August 2014, datasheet. [Online]. Available: https://www.digikey.de/products/de?keywords=535-9979-6-ND

[19] Rudolph. H. File:JTAG_TAP_Controller_State_Diagram.svg. Wikimedia Commons. Copyrighted free use, accessed 22 Nov 2017. [Online]. Available: https://commons.wikimedia.org/wiki/File%3AJTAG_TAP_Controller_State_Diagram.svg

[20] "IEEE Standard for Test Access Port and Boundary-Scan Architecture," *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pp. 1–444, May 2013.

[21] Prof. Dr. U. Brüning, "Lecture: Digital hardware design," Institute for Computer Engineering, Computer Architecture Group, Tech. Rep., 2017.

[22] M. Müller, "CAG JTAG Controller," Institue for Computer Engineering, Computer Architecture Group, 2013, unpublished hardware design.

[23] *7 Series FPGAs Data Sheet: Overview*, XILINX®, August 2017, Product Specification, Revision 2.5.

[24] Dr. N. Burkhardt, "Lecture: Functional Verification," Institute for Computer Engineering, Computer Architecture Group, Tech. Rep., 2016.

[25] BruceBlaus. File:blausen 0657 multipolarneuron.png. Wikimedia Commons. Creative Commons Attribution 3.0 Unported license, accessed 08 Feb 2018. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/archive/1/10/20131220130908%21Blausen_0657_MultipolarNeuron.png

[26] A. Grübl, "Specification of Event Routing Requirements between FPGAs in the Brainscales System," Kirchhof Institute for Physics Heidelberg, Electronic Visions Group, Tech. Rep., February 2018, internal preliminary draft.

[27] V. Tanasoulis, "Analysis and Development of a Communcication Infrastructure for a Waferscale Neuromorphic System," Ph.D. dissertation, TU Dresden, October 2017, preliminary draft, Rev. 0.29.

[28] C. Boucsein, M. Nawrot, P. Schnepel, and A. Aertsen, "Beyond the cortical column: Abundance and physiology of horizontal connections imply a strong role for inputs from the surround," *Frontiers in Neuroscience*, vol. 5, p. 32, 2011. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2011.00032

[29] J. Navaridas, M. Luján, L. A. Plana, S. Temple, and S. B. Furber, "SpiNNaker: Enhanced multicast routing," *Parallel Computing*, vol. 45, pp. 49 – 66, 2015, computing Frontiers 2014: Best Papers. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819115000095

[30] N. Burkhardt, "A hardware verification methodology for an interconnection network with fast process synchronization," Ph.D. dissertation, University of Mannheim, 2012.

D Acknowledgements

Ich möchte mich bei allen bedanken, die mir geholfen haben, diese Masterarbeit zu bewerkstelligen. Ganz besonders möchte ich mich bei Dr. Juri Schmidt und Prof. Ulrich Brüning bedanken, die mich hervorragend betreut haben und mir immer mit Rat und Tat zur Seite standen. Ebenso danken möchte ich Dr. Andreas Grübel, Dr. Eric Müller, Vitali Karasenko und den Kollegen von der Electronic Visions Group in Heidelberg, sowie Dr. Johannes Partzsch von der TU Dresden, die mir stets beratend und diskutierend in vielen Meetings geholfen haben, das BrainScaleS System zu verstehen, mich in das FPGA Design einzuarbeiten und die Anforderungen des Pulse-Routings zu erarbeiten. Danken möchte ich auch Vasileios Thanasoulis für das zur Verfügung gestellte Kapitel seiner noch nicht fertig gestellten Dissertation.
Nicht zuletzt möchte ich außerdem meiner Familie danken, die mich immer bei meinem Studium unterstützt und gefördert hat.

Erklärung:

Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den .....