Department of Physics and Astronomy
University of Heidelberg

Master Thesis
in Physics
submitted by

Robin Heinemann

born in Kassel (Germany)

2025

BRISCET — a routing ASIC for hybrid neuromorphic systems

This Master Thesis has been carried out by Robin Heinemann at the
Kirchhoff Institute for Physics in Heidelberg
under the supervision of
Prof. Dr. Johannes Schemmel and Prof. Dr. Dirk Koch

BRISCET — a routing ASIC for hybrid neuromorphic systems

BrainScaleS-2 is a hybrid neuromorphic system developed at Heidelberg University, combining continuous-time analog emulation of AdEx neuron dynamics with loosely coupled digital SIMD microprocessors. In the current ASIC generation, a single chip contains 512 neurons and two microprocessors. A prospective scaled-up system connects multiple BrainScaleS-2 ASICs in a 2D mesh topology to increase capacity and extend the capabilities to solve more complex problems. This thesis presents BRISCET, a companion ASIC to the BrainScaleS-2 SoC, which facilitates the interconnection in such a scaled-up system. This interconnection poses unique challenges: spike messages have to be transported with real-time requirements and simultaneously a high throughput of non-spike messages, such as configuration data and neuron membrane voltages, which do not tolerate packet loss, must be supported. To meet these requirements, a packet-based interconnection network with a hybrid error control scheme is proposed. A custom simulation and analysis framework was developed to quantify the performance characteristics and validate the technical implementation. Formal verification techniques are employed to ensure the correctness of critical components under all possible operating conditions. Joint simulation of the newly developed BRISCET chip and the BrainScaleS-2 ASIC is used to demonstrate the operation of a minimal, two-node, scaled-up BrainScaleS-2 system.

BRISCET — ein Routing ASIC für hybride neuromorphe Systeme

BrainScaleS-2 ist ein hybrides, neuromorphes System, das an der Universität Heidelberg entwickelt wurde. Es kombiniert die kontinuierliche, analoge Emulation der AdEx-Neuronendynamik mit digitalen On-Chip-SIMD-Mikroprozessoren. Die aktuelle ASIC-Generation verfügt über 512 Neuronen und zwei SIMD-Mikroprozessoren. Zukünftig sollen mehrere dieser ASICs in einer 2D-Mesh-Topologie verbunden werden, um die Kapazität zu erhöhen und dadurch eine Anwendung auf komplexerer Probleme zu ermögilchen.

In dieser Arbeit wird BRISCET vorgestellt, ein ASIC, der die Verbindung mehrerer BrainScaleS-2 SoCs zu einem solchen skalierten System ermöglicht. Bei der Konzeption von BRISCET stellen vor allem die verschienen Nachrichtten-Typen und ihre unterschiedlichen Anforderungen eine Herausforderung dar: Spike-Nachrichten müssen in Echtzeit übertragen werden, während gleichzeitig die Übertragung von Nicht-Spike-Nachrichten, wie zum Beispiel Konfigurationsdaten, mit hohem Durchsatz und ohne Paketverlust gewährleistet werden muss. Um beide Anforderungen zu erfüllen, wird ein paketbasiertes Verbindungsnetzwerk entwickelt, das einen hybriden Mechanismus zur Fehlerkontrolle verwendet.

Um die technische Umsetzung zu validieren und die Leistungsmerkmale zu quantifizieren, wird ein im Rahmen dieser Arbeit selbst entwickeltes Simulations- und Analyse-Framework verwendet. Darauf aufbauend werden formale Verifizierungstechniken eingesetzt, um die Korrektheit kritischer Komponenten unter allen möglichen Betriebszuständen sicherzustellen.

Zur Demonstration des neuen, skalierten Systems wird eine gemeinsame Simulation des neu entwickelten BRISCET-Chips und zwei Knoten des BrainScaleS-2-Systems durchgeführt.

Contents

Acronyms

ACK Acknowledgement. 31, 32

ADC Analog digital converter. 6, 7, 73, 74, 82, 95

AdEx Adaptive exponential integrate-and-fire. 2–4

AHB Advanced High-performance Bus. 44, 48

ARQ Automatic repeat request. 5, 8, 15, 23–25, 27, 30–33, 36, 37, 51, 58–60, 64–66, 69–72, 75–77, 82, 85, 91, 94, 95

ASIC Application specific integrated circuit. 2–6, 8, 14–17, 20, 47–49, 51, 77, 79, 81–84, 89, 90

AXIS Quad Serial Peripheral Interface. 84

CRC Cycling redundancy check. 20

DMA Direct memory access. 85

DPI SystemVerilog Direct Programming Interface. 77

FIFO First-in-first-out Queue. 54, 56, 59, 70, 94

FPGA Field-programmable gate array. 4–7, 14, 16, 17, 37, 46–49, 70, 73, 77–84, 89, 90, 94–96, 99, 100

FST Fast Signal Trace. 5, 68

HDL hardware description language. 8, 51, 64–66, 71, 94

JTAG Joint Test Action Group. 4, 5, 14, 16, 17, 37, 43–45, 49, 78, 84, 90

LVDS Low-Voltage Differnetial Signaling. 4–6, 14, 17, 79, 82, 83, 89, 90

MTBF Mean time between failure. 18–21, 28, 29, 75–77, 82, 89, 91, 95

OCP Open Core Protocol. 4

PCIe Peripheral Component Interconnect Express. 23, 24

PLL Phase-locked loop. 14, 16

PPU Plasticity processing unit. 2, 4–7, 15, 16, 37, 42–44, 46, 47, 73, 74, 79, 82–84, 95

ROB Reorder buffer. 43, 60–63

RTL Register Transfer Level. 77

TCP Transmission Control Protocol. 24

UDP User Datagram Protocol. 4, 89

UT Universal translator. 49

VCD Verilog Change Dump. 68

1. Introduction

Neuromorphic computing aims to learn from the operational principles of biological brains to develop novel approaches to computation. This thesis is based on the BrainScaleS-2 (BSS2) system (Pehle et al., 2022), developed by the Electronic Vision(s) Group at Heidelberg University. This is a hybrid system that combines time-continuous analog emulation of the AdEx (Brette and Gerstner, 2005) model for neurons with digital event routing and loosely coupled on-chip digital processors (plasticity processing units, or PPUs) (Pehle et al., 2022). Compared to biological time, the system operates in an accelerated fashion by a factor of roughly 1 000. The current silicon implementation of this architecture contains two of these PPUs as well as 512 neurons and $512 \cdot 256$ synapses per chip. This silicon realization will be called the BrainScaleS-2 ASIC hereafter.

Fully digital neuromorphic systems like SpiNNaker (Mayr, Hoeppner, and Furber, 2019) often allow a dynamic trade-off between emulation speed and the number of emulated neurons and synapses. Emulating fewer neurons allows for a higher emulation speed, while more neurons can be emulated by reducing the emulation speed. The analog emulation of neuron and synapse dynamics employed by the BrainScaleS-2 ASIC means they are a fixed resource and no such dynamic trade-off is possible. Instead, scaling up the BrainScaleS-2 architecture to more complex problems is envisioned by combining multiple BrainScaleS-2 ASICs into a single, larger system. Due to the hybrid architecture of the BrainScaleS-2 ASIC, unique challenges for the interconnection network of such a scaled system arise: the interconnection network is shared by two classes of messages: Event messages are sensitive to latency and have real-time requirements due to the analog emulation of the neurons, while non-event messages do not have real-time requirements but cannot tolerate lost or corrupted messages, unlike event messages can.

This thesis presents the digital design of a companion ASIC to the BrainScaleS-2 ASIC — the BRainScaleS-2 Interconnection Switching Chip for Extended Topologies, or BRISCET ASIC — to facilitate such a scaled-up system. The BRISCET ASICs can be connected to a single BrainScaleS-2 ASIC and to neighboring BRISCET ASICs in a 2D mesh topology. It creates an interconnection network optimized for the shared usage of event and non-event messages. Special focus was placed on the verification of the proposed interconnection network architecture and its hardware implementation in the form the of the BRISCET ASIC. Three focus points were identified: the performance of the interconnection network, the correctness of its implementation, and the demonstration of a scaled-up BrainScaleS-2 system utilizing the BRISCET ASIC.

This thesis starts by providing background information on the BrainScaleS-2 ASIC, and the requirements for event and non-event messages are outlined. Section 3 starts with an introduction to Amaranth HDL, the hardware description language used in this thesis, before describing custom tools that were developed to improved the integration of Amaranth HDL into SystemVerilog-based designs. The section concludes by describing the stream interfaces used by most of the hardware blocks developed in this thesis and providing a short overview of a library of stream-based building blocks that were devloped. Next, section 4 describes the architecture and the rationale behind the BRISCET ASIC from the link layer up. Finally, section 5 describes in detail how the different verification goals

of performance, correctness, and demonstration of a scaled-up BrainScaleS-2 system were achieved.

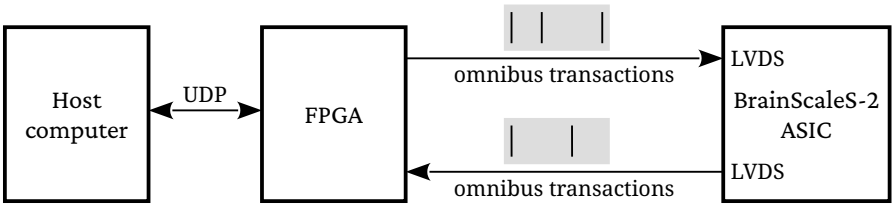

Figure 1: Schematic overview of the network-attached accelerator style deployment of the BrainScaleS-2 ASIC currently in use in the Electronic Vision(s) group. FPGA and host computer are connected using UDP, and the FPGA connects to the BrainScaleS-2 ASIC via an LVDS-based interface to exchange spikes, as well as configuration data in the form of omnibus transactions with the BrainScaleS-2 ASIC.

2. Background

In this chapter, the BrainScaleS-2 ASIC is described in more detail and an overview of an envisioned scaled-up BrainScaleS-2 system utilizing it is given. The different messages and their requirements are determined to guide the design of the BRISCET companion ASIC in such a scaled-up system.

2.1. The BrainScaleS-2 ASIC

The BrainScaleS-2 ASIC is the current in-silico realization of the hybrid neuromorphic BrainScaleS-2 architecture. It combines analog, continuous-time emulation of 512 AdEx neurons and two loosely coupled on-chip SIMD microprocessors — the plasticity processing units, or PPUs. Digital on-chip routing routes spikes emitted by the neurons to on-chip synapses or to an external high-speed interface. Configuration registers, as well as memory accesses by the PPUs, are connected to a memory-mapped bus named omnibus (Friedmann, 2013). omnibus is derived from the Open Core Protocol (OCP) bus (OCP, 2009).

Two external interfaces are used to control the BrainScaleS-2 ASIC: a JTAG interface is used for low-level initialization and debugging, while a high-speed LVDS interface is used during normal operation. To bridge this high-speed interface to conventional computer systems, an FPGA is used. This allows for different kinds of deployments, for example, as a network-attached accelerator (Müller et al., 2020), by connecting the FPGA via a network interface to a host computer, or as an edge computation platform (Stradmann et al., 2022) by utilizing combined FPGA-CPU systems. Figure 1 shows a schematic overview of the network-attached accelerator deployment.

The high-speed interface between the FPGA and the BrainScaleS-2 ASIC is used to transmit spikes to and from the FPGA, and to tunnel omnibus transactions between the FPGA and the BrainScaleS-2 ASIC. Transactions from different omnibus masters on the BrainScaleS-2 ASIC are tunneled to the FPGA, and transactions from a single omnibus master on the FPGA are tunneled to the BrainScaleS-2 ASIC:

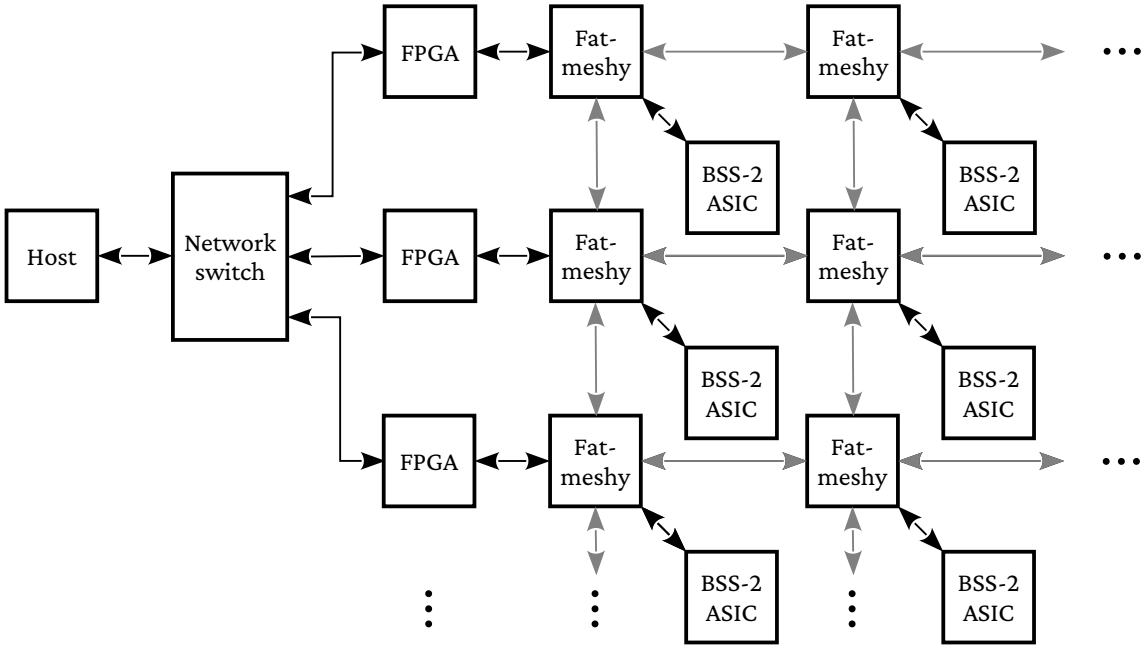

Figure 2: Schematic overview of a scaled-up BrainScaleS-2 System. Multiple BrainScaleS-2 ASICs are connected together in a 2D mesh topology, utilizing the BRISCET ASICs. One or more BRISCET ASICs are connected to an FPGA to control them from a host computer. FPGAs and BRISCET ASICs, as well as BRISCET ASICs and BrainScaleS-2 ASICs, are connected via LVDS interfaces. Connections between the BRISCET ASICs utilize a link using a wide parallel bus of the transceivers developed by Ilmberger et al. (2024).

source	target	transaction granularity [bit]	note
PPU 0 instruction fetch	FPGA	32	read-only
PPU 1 instruction fetch	FPGA	32	read-only
PPU 0 data load / store	FPGA	128	
PPU 1 data load / store	FPGA	128	
FPGA	BrainScaleS-2 ASIC	32	

Table 1: Sources of omnibus transactions that are tunneled via the high-speed interface of the BrainScaleS-2 ASIC

An in-depth description of this interface can be found in Karasenko (2020).

2.2. Scaled-up BrainScaleS-2 System

To scale the BrainScaleS-2 system to more synapses and neurons than a single BrainScaleS-2 ASIC can emulate, multiple chips should be connected. Figure 2 shows a schematic view of the envisioned scaled-up BrainScaleS-2 system, operated in a network-attached accelerator fashion.

The BrainScaleS-2 ASIC itself has no facilities to allow interconnection between multiple BrainScaleS-2 ASICs, so to combine multiple of them into a larger system, an external interconnection infrastructure is needed. In this thesis, a dedicated companion ASIC is developed, that connects to each BrainScaleS-2 ASIC via its high-speed interface. These companion ASICs can be connected in a 2D mesh topology to form an interconnection network. These chip-to-chip connections use a full-duplex

parallel bus of the IO cells developed in (Ilmberger et al., 2024). This system, again, is connected via FPGAs to a conventional host computer that operates the system.

In this thesis, a design for the digital parts of this companion ASIC is developed. The twofold nature of the data that the BrainScaleS-2 ASIC processes presents a unique challenge for the interconnection network: First, spike data has to be transported across the interconnection network. Different encoding schemes can be used to carry information using a sequence of one or more spikes. These encoding schemes use not only the number of spikes, but also temporal aspects, like the relative timing between spikes. The interconnection network, therefore, must be able to preserve this temporal information when transmitting spikes.

Second, configuration messages, which have to be transmitted over the interconnection network to configure the different BrainScaleS-2 ASICs, have different requirements and are not timing critical. These two types of messages — event messages and non-event messages — also differ in their tolerance for lost or corrupted messages. The system can tolerate event messages being lost or corrupted by the interconnection network at a low rate. A lost or corrupted event message can only cause localized transient faults, which the system can recover from. This is in contrast to the non-event messages, which carry configuration data, where a single corrupted or lost message can cause the system to enter a state that can only be recovered from by restarting it.

The nature of non-event messages carrying configuration data creates an additional requirement for their transport across the interconnection network. The configuration data is transmitted as omnibus transactions, which have a strict ordering requirement. A given set of transactions from a single master must be completed in the same order they are issued by the master. Therefore, the interconnection network must allow a flow of messages to retain their order for a given sender and receiver pair. Note that only transactions coming from the same master have this ordering requirement, transactions from different masters have no ordering requirement between them. In summary:

	event messages	non-event messages
strict ordering between messages	not necessary	neccessary
lost messages	permissible	not permissible
preservation of message to message timing	necessary	not needed
corruption of message payload	permissible	not permissible

Table 2: Comparison of the requirements for the two classes of messages that need to be transported across the interconnection network.

The required bandwidth for these messages depends on the use case. Data coming from or going to a BrainScaleS-2 ASIC is limited by the bandwidth of the high-speed interface. For non-event type messages a usable bandwidth of 2.4 Gbit to 4 Gbit is achieved when using 4 to 8 of the LVDS links. Spikes can be transmitted at a rate of 125 MHz to 250 MHz for 4 to 8 links. Each spike has a 16 bit label, yielding a bandwidth of 2 Gbit to 4 Gbit. Another notable example is the surrogate gradient training demonstrated in Cramer et al. (2022). It uses the PPUs to record ADC traces of

the membrane voltage of the neurons. These ADC samples are written to a memory attached to the FPGA. The minimum readout time for an ADC sample was 1.7 µs, yielding a data rate of 1.2 Gbit/s of write bandwidth for a single PPU for 256 neurons. Finally, the links developed by Ilmberger et al. (2024) allow a raw data rate of 2 Gbit/s per link. This establishes the approximate requirement for the data rate that the BRISCET chip needs to route.

The real-time requirements for event messages also differ depending on the use case. One limit can be derived from the time constants associated with analog neurons and synapses. In (Leibfried, 2021) these are calibrated to $\approx 500\,\mathrm{ns}$. As spikes travel over multiple hops, the jitter in their relative timing incurred from the transport over the interconnection network accumulates from hop to hop, so a link-to-link jitter of $O(10\,\mathrm{ns})$ is desirable.

3. Hardware Design Methodology

The digital design for BRISCET was implemented by using hardware description languages (HDLs) to describe digital logic. Fabrication of ASICs is both expensive and time-consuming, so verifying that a design adheres to its specification before fabrication is important. In this section, a background on the employed hardware description languages is given. Commonly used hardware description languages include SystemVerilog (IEEE, 2024) and VHDL (IEEE, 2019). In this thesis, the core components used by the interconnection network, the router and the ARQ protocol, were written in Amaranth HDL (Amaranth contributors, 2019) to allow usage of its strong meta-programming capablities. Other components were written in SystemVerilog to simplify their integration with preexisting components. Section 3.1 introduces Amaranth HDL and the meta-programming capabilities that it provides. Using these meta-programming capabilities, tools were developed in this thesis to improve the integration of Amaranth HDL into designs written in SystemVerilog as well as C++, were developed, which are described in section 3.2. Finally, section 3.3 introduces the AXI-Stream-based stream interface that was used for most of the components developed in this thesis. Section 3.4 briefly outlines some fundamental building blocks that were developed for this purpose.

3.1. Amaranth HDL

Amaranth HDL is a Python library that provides the user with tools to describe synchronous logic. Similar to traditional hardware description languages like SystemVerilog, it organizes a digital design as a hierarchical, tree-like structure of parametrizable building blocks. In SystemVerilog, these building blocks are called *modules*, while in Amaranth HDL they are called *components*. Being a Python library provides the designer with access to the full capabilities of Python for parametrization of components and meta-programming. For example, it is possible to parametrize a module using a lambda function. Furthermore, components are Python classes, allowing their organization in Python packages and modules, as well as the ability to use package managers for Python like pip to manage dependencies. It is possible to use all packages available in the Python ecosystem. For example, scipy (Virtanen et al., 2020) could be used to generate filter coefficients for a FIR filter component. A more detailed case study on the possibilities of Amaranth HDL can be found in Est´evez (2023). In order to use a design described in Amaranth HDL with other tools, Amaranth HDL can be compiled to Verilog.

An example of a component written in Amaranth HDL is as follows:

```
class StreamFilter(Component):
  input: In(stream.Signature(32))
  output: Out(stream.Signature(32))

  def __init__(self, filter):
    super().__init__()
    self.filter = filter

  def elaborate(self, _):
    m = Module()

    selected = self.filter(self.input.p)
    m.d.comb += [
      self.output.valid.eq(self.input.valid & selected),
      self.output.p.eq(self.input.p),
      self.input.ready.eq(self.output.ready | ~selected)
    ]

    return m
```

Listing 1: Example of a component written in Amaranth HDL. A hardware design in Amaranth HDL is composed of a hierarchy of components. Components are Python classes that inherit from the **`Component`** base class. Inputs and outputs are defined using Python type annotation syntax. Here, an input stream with a payload of width 32 bit and an output stream with the same payload width are defined. Parameters for components can be passed in the constructor. A component has a function **`elaborate`** that creates the synchronous or combinatorial logic determining the behavior of it. All statements are attached to a **`Module`**. In this example, a purely combinatorial component is presented that forwards an input stream to an output stream selectively-predicated by the **`filter`** given in the constructor.

In Amaranth HDL, components are Python classes that inherit from the base class **`Component`**. This snippet defines a component named **`StreamFilter`**. This component has an input stream interface **`input`** with a payload width of 32 bit and an output stream interface **`output`** with a payload width of 32 bit. It is parametrized by a function **`filter`** in the constructor. Finally, the function **`elaborate`** describes the behavior of the component. **`elaborate`** returns a **`Module`** to which a set of statements is attached. Here a purely combinatorial design is specified. The **`input`** and **`output`** stream are connected together with the **`filter`** function being used to determine which words of the stream should get passed through. This **`StreamFilter`** component could, for example, be instantiated as follows:

```
gt_32_filter = StreamFilter(lambda v: v > 32)
```

Listing 2: Example instantiation of the **`StreamFilter`** component. Here, a Python **`lambda`** function is passed as a parameter.

Here, for **`filter`**, a lambda function is passed. In this case, only payloads that are greater than 32 would be passed from the **`input`** stream to the **`output`** stream.

Amaranth HDL also provides powerful introspection capabilities, ranging from introspection into the fields of a custom data type to the ability to introspect the hierarchy of components in a design. For example, the fields of a custom data type can be introspected as follows:

```
class Coordinate(data.Struct):
  x: 8
  y: 8

print(list(data.Layout.cast(Coordinate)))
# [('x', Field(8, 0)), ('y', Field(8, 8))]
```

Listing 3: Example of the introspection capabilities of Amaranth HDL. This example shows how the fields of a custom data type can be determined.

Here, the first parameter of `Field` is the size of a field and the second is the offset.

3.2. Integration of Amaranth HDL with SystemVerilog and C++

The tools commonly used for different aspects of hardware design, including simulators like Cadence Xcelium and synthesis tools like Synopsys Design Compiler, do not natively support Amaranth HDL as one of their input languages. To use a design written in Amaranth HDL with them, it is compiled to Verilog. As Verilog does not support constructs like interfaces or custom data types, if they are used in Amaranth HDL, they get lost in this transformation. This increases the friction when combining Amaranth HDL modules with other modules written in SystemVerilog, which does support custom data types and interfaces. To reduce this friction, an automatic wrapper generator was developed that uses the introspection capabilities provided by Amaranth HDL. It creates SystemVerilog interfaces and data types based on the interfaces and data types used in an Amaranth HDL module and produces a SystemVerilog wrapper for the plain Verilog module generated by Amaranth HDL using those. Listing 4 shows an example of such an automatically generated wrapper. Custom data types like the `Coordinate` type in Amaranth HDL are translated to an equivalent type definition in SystemVerilog (matching the bit layout). Amaranth HDL interfaces like `stream.Signature(Coordinate)` are translated to equivalent SystemVerilog interfaces.

```
class Coordinate(data.Struct)
  x: 8
  y: 8

class Example(Component):
  input: In(stream.Signature(Coordinate))
  output: Out(stream.Signature(Coordinate))
```

```
package example_pkg;
typedef struct packed {
    logic [7: 0] y;
    logic [7: 0] x;
} coordinate;
endpackage

interface coordinate_stream_if import example_pkg::*;;
    coordinate payload;
    logic valid, ready;

    modport master (
        output .p(payload), valid, input ready
    );
    modport slave (
        input .p(payload), valid, output ready
    );
endinterface

module example import example_pkg::*; (
    coordinate_stream_if.slave in,
    coordinate_stream_if.master out
);
// omitted for brevity
endmodule
```

Listing 4: Example of an Amaranth HDL module and the corresponding generated SystemVerilog wrapper.

The same loss of information occurs when generating a CXXRTL model from an Amaranth HDL design. Typed input and output ports are represented as flat bit vectors by CXXRTL. To facilitate more convenient interaction with a CXXRTL model of an Amaranth HDL design, an automatic generator for C++ equivalents of the types used in an Amaranth HDL design was developed. These C++ equivalents can be assigned from a flat bit vector and converted back to the same. Listing 5 shows the C++ type definition that is generated for the **`Coordinate`** data type in listing 4. Furthermore, omitted from the listing is a **`std::formatter`** specialization that allows pretty-printing of the C++ type.

```
struct coordinate {
    uint8_t x;
    uint8_t y;
    coordinate & operator=(const value<16> & val);
    operator value<16>() const;
};
```

Listing 5: C++ struct generated automatically from the Amaranth HDL data type **`Coordinate`** shown in listing 4. Implementation of the conversion and assignment operators is omitted for brevity.

3.3. Stream-based interfaces

During the development of hardware components intended to be reusable — similar to software development — the interface chosen for a component must be given special consideration. For hardware components, the timing relationship between the inputs and outputs of a component plays a special role. In systems processing data in a streaming fashion, AXI-Stream (AMBA, 2021b) is commonly used as an interface for components. At the simplest level, an AXI-Stream stream consists of three signals:

`payload` carries the information transported over the stream,

`valid` indicates that the payload signal carries valid data,

`ready` is used by the receiver of the payload data to acknowledge reception.

The relationship between these signals is governed by a set of rules:

- Data is transported across the stream when both **`valid`** and **`ready`** are asserted.
- When the sending side asserts **`valid`**, it must stay asserted until **`ready`** is asserted by the receiver.
- When the sending side asserts **`valid`**, **`payload`** must stay the same until **`ready`** is asserted by the receiver.
- A sender is not allowed to wait for **`ready`** to be asserted by the receiver before asserting **`valid`**.

An advantage of the AXI-Stream interface is the possibility to transparently add pipelining stages between the sender and the receiver of a stream. This way, pipelining necessary to achieve a given clock frequency can be decoupled from the implementation of the individual components. The components developed in this thesis were designed using the AXI-Stream interface where possible.

3.4. Custom building blocks for stream processing

To simplify the development of stream-based components, a base library of building blocks for stream-based systems was developed in SystemVerilog and in Amaranth HDL as part of this thesis. This includes:

stream_arbiter: merges multiple input streams into a single output stream in a round-robin fashion.

stream_tee: forwards the payload of a single input stream to multiple output streams.

stream_fifo: a FIFO that accepts data from an input stream and outputs data to an output stream.

stream_filter: fowards payloads from an input stream to an output stream only they matche a predicate.

4. Architecture

This section describes the architecture and hardware implementation chosen for the BRISCET ASIC. The goal of BRISCET is to facilitate the interconnection of multiple BrainScaleS-2 ASICs in a 2D mesh topology as shown in figure 2. Figure 3 gives a block-level overview of the chosen design. BRISCET employs the same LVDS based interface for connection to an FPGA as the BrainScaleS-2 ASIC. A BRISCET ASIC further can be connected to the high-speed interface of a BrainScaleS-2 ASIC as well as the JTAG interface that is necessary for low-level initialization of the BrainScaleS-2 ASIC. Four full duplex mesh ports allow a BRISCET to connect to neighboring BRISCET ASICs in a 2D mesh topology. Limiting the total size of BRISCET to 5 mm^2 constraints the number of IO pads it can use. The two LVDS interfaces are restricted to four links in each direction. Furthermore, the number of links available for the mesh ports is limited to 11 to 13 per direction and port. For local control tasks, such as link training, BRISCET furthermore includes a RISC-V CPU. The open-source Hazard3 core was chosen for this purpose. Finally, a PLL is included for clock generation and a JTAG is used for low-level initialization, such as initialization of the PLL.

Next, a high-level overview of the chosen architecture for the interconnection network is given before its different components are described in more detail.

4.1. High-level overview

Similar to the FPGA and BrainScaleS-2 ASIC interface, both event data and non-event messages are transported over the interconnection network. However, the design constraints and requirements for the connection between the BRISCET ASICs differ from those between a BrainScaleS-2 ASIC and an FPGA in two significant ways.

The BrainScaleS-2 ASIC high-speed interface preserves the relative timing between events by attaching a timestamp on the receiving side and sorting the events according to their timestamp on the receiving side. By delaying them by a fixed amount relative to the timestamp they are received with, jitter in the transmission latency is further compensated. This scheme is described in more detail by Schmidt (2017). This scheme is employed on the high-speed interface of the BrainScaleS-2 ASIC for two main reasons. First, the channel bonding strategy can cause events to be reordered relative to each other, which is compensated by sorting according to the timestamp on the receiving side. Additionally, the transmission latency can vary according to the link congestion. This for example, can again be caused by the channel bonding architecture or by differences in the fill state of internal buffers depending on the rate of event transmissions. The timestamp-based approach of the high-speed interface of the BrainScaleS-2 ASIC has three main downsides. It increases the amount of data that has to be transported for a single event, as a timestamp has to be transmitted along with the event. Furthermore, it requires additional buffer resources on the receiver side to perform the reordering and buffering of events by a fixed delay. Finally, it increases the minimum latency for event transmissions by the fixed delay employed to correct event rate-dependent transmission latency. Therefore, for the BRISCET ASIC, this timestamp-based approach is avoided. First, by operating the

link between two BRISCET ASICs as a wide, parallel link, no channel bonding is necessary, so jitter in the latency of transmission, as well as reordering of events caused by channel bonding, is avoided. Furthermore, internal buffering of events and congestion or rate dependent latencies incured thereby are minimized.

The second difference is that these links are used to connect multiple BRISCETs together into a larger system, while the high-speed interface design for the BrainScaleS-2 ASIC was designed for a single point-to-point link. The envisioned topology of a system built using multiple BRISCET and BrainScaleS-2 ASICs is a 2D mesh. A 2D mesh topology is not full connected. This means that for transmission of data from one node of the system to another node, the data potentially has to traverse multiple nodes in between. This makes some form of routing protocol that determines how data traverses across the mesh to get from source to target necessary.

Due to these differences, for the BRISCET-to-BRISCET links and the tunneling of event and non-event data over them, a new architecture was developed from the ground up in this thesis. This is split into four different layers that build upon each other. First, starting from the data transmitted over a single link, a link-level protocol to share the link between event and non-event data was devised. Event data is transmitted without any error control, while non-event data employs a link-level ARQ protocol for error control. Using a link-level ARQ avoids the extra required buffer space of an end-to-end ARQ protocol. A trade-off of not using an end-to-end ARQ protocol, which could be used to reorder received messages according to their sent order, is that the ordering requirements for omnibus transactions coming from a single master must instead be enforced by the routing scheme. The routing scheme chosen for the non-event messages is a combination of two routing schemes: a dimension-ordered routing scheme is supplemented with a local routing table-based scheme. Dimension-ordered routing schemes have low path diversity and cannot be adapted according to link congestion. By supplementing it with a routing table-based scheme, higher path diversity and static optimization of the routing paths according to expected link congestion is possible. This combined routing scheme is not adaptive to changes in congestion at runtime, but can be statically optimized for expected link congestion, for example using the algorithm proposed by Shim (2010). The routing uses wormhole switching (William James Dally and Towles, 2004) and credit counting for resource allocation to optimize latency and required buffer space. Furthermore, two virtual channels are employed to reduce the impact of head-of-line blocking. Allocation of the virtual channels to different flows is again constrained by the ordering requirements of the non-event messages carrying omnibus transactions. The virtual channel allocation is therefore static and determined by the routing scheme (either the dimension-ordered routing scheme or the table-based one).

Building upon this routing scheme, a custom protocol that utilizes the advantages of wormhole switching is developed for tunneling omnibus read and write transactions across the interconnection network. Each BRISCET has six omnibus masters that generate transactions which are tunneled over the interconnection network using this protocol:

- One (read-only) omnibus master with a 32 bit bus width for each of the two PPUs on the BrainScaleS-2 ASIC used for instruction fetching

- One omnibus master with a 128 bit bus width for each of the two PPUs on the BrainScaleS-2 ASIC used for data loads and stores
- One omnibus master with a 32 bit bus width for the JTAG interface of BRISCET
- One omnibus master with a 32 bit bus width for the RISC-V core included on BRISCET

The BRISCET ASIC also acts as a target for omnibus transactions, with the transactions being forwarded to three different targets according to their address:

`address[31:30]` = 00_2**:** These transactions are forwarded to the connected BrainScaleS-2 ASIC via its high-speed interface

`address[31:30]` = 01_2**:** These transactions are forwarded to the local omnibus slaves on BRISCET

`address[31:30]` = $1x_2$**:** These transactions are forwarded to the FPGA via the high-speed interface between BRISCET and FPGA

The clocks necessary for the operation of the BRISCET ASIC are generated by an on-chip PLL. It uses the same PLL as the BrainScaleS-2 ASIC. For initialization, BRISCET includes a JTAG interface that can act as an omnibus master and is also used to configure the PLL. The JTAG interface of the BrainScaleS-2 ASIC is connected to an omnibus-accessible JTAG driver on the BRISCET. Due to time constraints, for the event data only a simple circuit-switching-based routing scheme was implemented.

The next sections will describe the different components and their underlying design choices in more detail.

4.2. Link-level protocol

Two classes of messages have to be transported across the interconnection network. Event messages have to be transported with minimal latency, preserving inter-event timing, and can tolerate corrupted messages. In contrast, non-event messages cannot tolerate corrupted data. The interconnection network must enable non-event messages to be transmitted error-free and without losing non-event messages. However, there is no requirement for non-event messages to preserve the timing between messages.

Both of these classes of messages share the same chip-to-chip links between two BRISCETs. The link transmits data in discrete units of l bits, also called phits. Due to their different requirements, it is desirable to separate the two classes of messages as close to the links as possible and process them independently downstream. The closest level to the links is having each phit either transport event data or non-event data, with a header indicating which type of data the phit carries.

With a size of 22 bit to 26 bit for the phits, a single phit is sufficient to carry a single event message. For non-event messages, it is desirable to combine multiple phits into a larger word before processing. As the non-event messages cannot tolerate corruption, some mechanism to detect and/or correct errors must be employed. One option for detection is a checksum, for example a CRC (Koopman

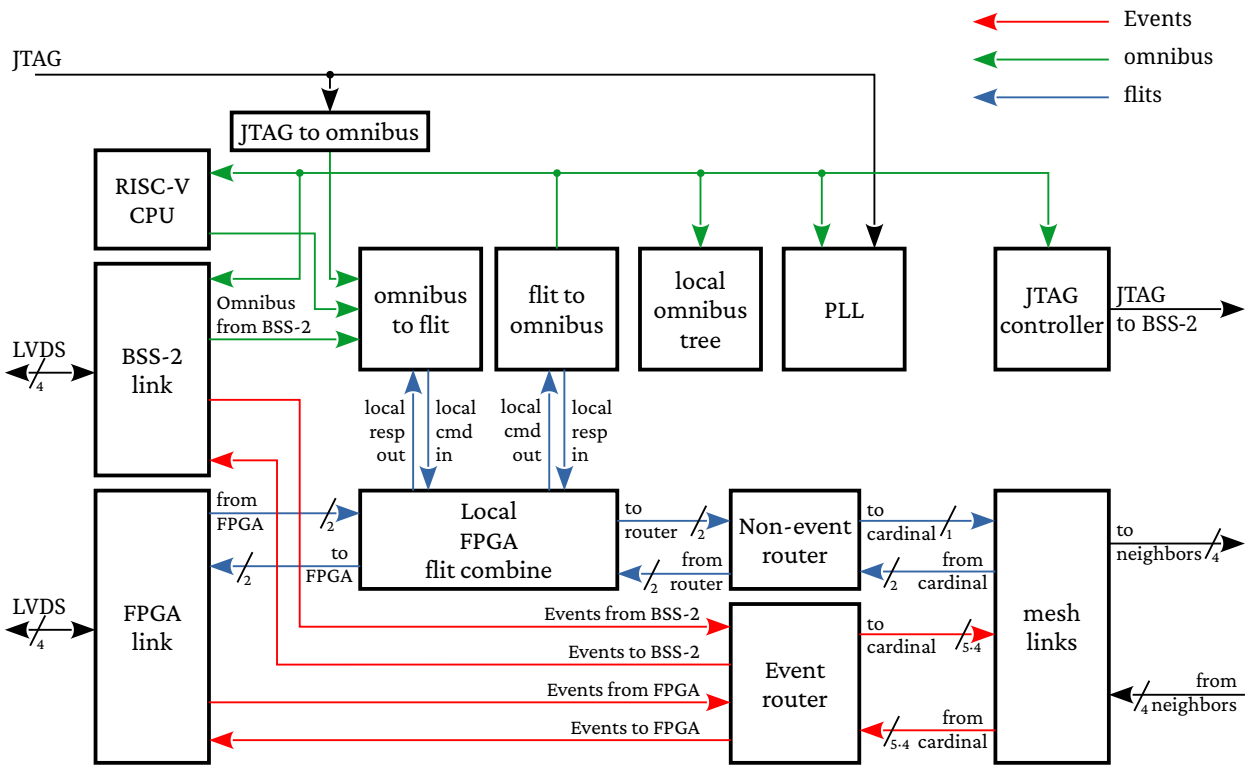

Figure 3: Toplevel overview of the BRISCET ASIC developed in this thesis. Each BRISCET ASIC has an LVDS high-speed interface to connect it to an FPGA and a similar interface to connect it to a BrainScaleS-2 ASIC. A JTAG interface is included for low-level initialization. Multiple BRISCET ASICs can be connected in a 2D mesh topology utilizing four full-duplex mesh links. Each BRISCET includes routing components to create an interconnection network for both event and non-event messages that have to be transmitted for a system. Non-event messages are split into fixed-sized flits and processed as streams of flits. The direction of arrows indicates control direction. Each arrow indicates a stream including backpressure.

and Chakravarty, 2004). To analyze the behavior of a checksum for detecting corrupted non-event messages, let us consider a theoretical model of the chip-to-chip link. Note that in the following considerations, only a single chip-to-chip link is considered. A scaled-up BrainScaleS-2 system will have many of these chip-to-chip links. For example, a 4×4 system has 9 BRISCET-to-BRISCET links, and therefore all failure rates calculated below have to be multiplied by the number of links in a system, when determining the failure rate for a complete system. Three parameters define the behavior of the link.

l number of bits that are transmitted over the link in parallel. 11 to 13 independent links yield 22 bit to 26 bit for l due to the double data rate nature of the links.

B the bandwidth of the link. The links by (Ilmberger et al., 2024) achieve a bandwidth of 2 Gbit/s, so for 11 to 13 this yields 22 Gbit/s to 26 Gbit/s

ρ the bit error rate. For this model of the link, the bit errors are assumed to be independent of each other. So ρ is defined as the probability of any bit transmitted over the link being flipped. This error rate is symmetric, the probability for a bit signalling 1 to flip is the same as the

probability for a bit signalling 0 to flip. The links by (Ilmberger et al., 2024) have been tested to $\rho \leq 1 \times 10^{-10}/\text{bit}$.

A checksum can be described by two parameters

c the number of bits used by the checksum

HD the hamming distance of the checksum. The checksum can detect any number of bit flips smaller than HD. If the message (including the checksum) has more bit flips, it is not guaranteed the checksum is able to detect the bit flips.

In the limit of c or more bit flips in a message, a lower bound for the probability that the checksum cannot detect these bit flips is given by

$$p_{\text{checksum wrong}} = 2^{-c}$$

In the simplest case a phit can then be split into a header and a payload part:

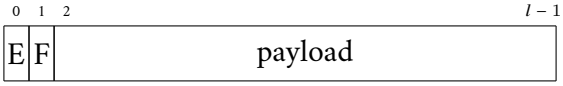

Figure 4: Simple encoding scheme to differentiate between event and non-event payloads transmitted over fixed-size phits (the unit of transmission used by the link) of l bit. A single bit E indicates the type of the payload — event or non-event — while the header bit F delimits non-event messages.

where

E indicates the type of the payload data, A one indicates event data and a zero indicates non-event data.

F is used for framing of the non-event data. A one indicates the payload contains the last part of a non-event message.

payload contains the actual data, which is interpreted according to the header bit.

For bit flips then two cases have to be considered: bit flips in the header bits and bit flips in the payload data. Let us first consider bit flips in the payload data. The checksum can detect up to $HD - 1$ bit flips. It can fail to detect them if HD or more bit flips occur. The probability of this happening for an m bit message transmitted as $n := \lceil (m + c)/(l - 2) \rceil$ phits is given by

$$\text{binomsf}(HD - 1, m + c, \rho)$$

Where $\text{binomsf}(k, n, p)$ is the survival function for k of the binomial distribution with a probability p, a number of trials n. The mean time between failures (MTBF) can then be computed as

$$\text{MTBF} = \frac{n \cdot l}{\text{binomsf}(HD - 1, m + c, \rho) \cdot B}$$

B[bit/s]	n	ρ[1/bit]	l[bit]	c[bit]	m[bit]	MTBF[d]
2.00×10^{10}	3	1.00×10^{-10}	22	7	50	8.15×10^{-3}
2.00×10^{10}	4	1.00×10^{-10}	22	16	50	4.17
2.00×10^{10}	6	1.00×10^{-10}	22	7	100	8.15×10^{-3}
2.00×10^{10}	6	1.00×10^{-10}	22	16	100	4.17
2.00×10^{10}	3	1.00×10^{-9}	22	7	50	8.15×10^{-4}
2.00×10^{10}	4	1.00×10^{-9}	22	16	50	4.17×10^{-1}
2.00×10^{10}	6	1.00×10^{-9}	22	7	100	8.15×10^{-4}
2.00×10^{10}	6	1.00×10^{-9}	22	16	100	4.17×10^{-1}

Table 4: MTBF for a bit flip in the packet header to stay undetected for different bit error rates ρ, message sizes m, and checksum sizes c.

For different values for m, ρ and c this results in:

B[bit/s]	n	ρ[1/bit]	l[bit]	c[bit]	m[bit]	HD[bit]	MTBF[d]
2.00×10^{10}	3	1.00×10^{-10}	22	7	50	2	2.39×10^{3}
2.00×10^{10}	3	1.00×10^{-10}	22	7	50	3	1.31×10^{12}
2.00×10^{10}	6	1.00×10^{-10}	22	7	100	2	1.35×10^{3}
2.00×10^{10}	6	1.00×10^{-10}	22	7	100	3	3.85×10^{11}
2.00×10^{10}	3	1.00×10^{-9}	22	7	50	2	2.39×10^{1}
2.00×10^{10}	3	1.00×10^{-9}	22	7	50	3	1.31×10^{9}
2.00×10^{10}	6	1.00×10^{-9}	22	7	100	2	1.35×10^{1}
2.00×10^{10}	6	1.00×10^{-9}	22	7	100	3	3.85×10^{8}

Table 3: MTBF for bit flips in the payload data to stay undetected, for different bit error rates ρ, message sizes m and protection level HD afforded by the checksum.

For the given cases, a checksum with $HD = 3\,\mathrm{bit}$ makes it unlikely that a failure ever occurs over the lifetime of a system.

The second case that has to be considered is a bit flip in the packet header. If a bit is flipped in the packet header, this causes the type of payload to be misidentified. Either an event payload is identified as part of a non-event message, or the framing for the non-event message is wrong. Both cases result in at least $l - 2\,\mathrm{bit}$ that are wrong. Usually, $l - 2$ will be larger than the number of bits of the checksum c, so for a message containing $m\,\mathrm{bit}$ transmitted as $n := \lceil (m + c)/(l - 2) \rceil$ phits, the probability that the checksum does not identify the corrupted message is given by

$$(1 - (1 - p(\text{bit flip in phit header}))^n) \cdot p_{\text{checksum wrong}} \approx n\,\mathrm{binomsf}(0, 2, \rho)2^{-c}$$

Again, the MTBF is then given by

$$\mathrm{MTBF} = \frac{l}{\mathrm{binomsf}(0, 2, \rho)2^{-c}B}$$

For different values for m, ρ, and c this results in: Comparing the MTBF for this case with the case of

B[bit/s]	n	ρ[1/bit]	l[bit]	c[bit]	m[bit]	MTBF[d]
2.00×10^{10}	4	1.00×10^{-10}	22	7	50	1.63×10^{7}
2.00×10^{10}	4	1.00×10^{-10}	22	16	50	8.34×10^{9}
2.00×10^{10}	7	1.00×10^{-10}	22	7	100	1.63×10^{7}
2.00×10^{10}	7	1.00×10^{-10}	22	16	100	8.34×10^{9}
2.00×10^{10}	4	1.00×10^{-9}	22	7	50	1.63×10^{5}
2.00×10^{10}	4	1.00×10^{-9}	22	16	50	8.34×10^{7}
2.00×10^{10}	7	1.00×10^{-9}	22	7	100	1.63×10^{5}
2.00×10^{10}	7	1.00×10^{-9}	22	16	100	8.34×10^{7}

Table 5: MTBF for bit flips in the header of a packet to stay undetected when using an encoding for the header bits that can correct a single bit flip. ρ is the bit error rate, c is the number of bits used for the checksum, and m is the message size transmitted.

a bit flip in the payload bits clearly shows that the impact of bit flips in the packet header is much greater than bit flips in the packet payload. In fact, a CRC checksum with $HD = 3$ for messages up to $120\,\mathrm{bit}$ (including the checksum) needs just $c = 7\,\mathrm{bit}$, but still compares favourably to $c = 16\,\mathrm{bit}$ in the case of a bit flip in the packet header.

An alternative approach to the simple encoding of the packet header analyzed above is an encoding that allows for correction of one (or more) bit flips. If one bit flip can be corrected, a misidentification of event data as non-event data or of the framing can then only happen if at least two bit flips occur in the header. To make it possible to correct a single bit flip in the two bits of the header, at least three parity bits have to be added to the header. An $m\,\mathrm{bit}$ message is then transmitted as $n := \lceil (m + c)/(l - 5) \rceil$ phit. For a bit flip that occurs in the header, the MTBF is given by:

$$\mathrm{MTBF} = \frac{l}{\mathrm{binomsf}(1, 2 + 3, \rho) 2^{-c} B}$$

This results in an MTBF for different values m, ρ, and c of: Clearly, an encoding scheme like this offers a much improved tolerance to bit flips in the header, at a cost of increased overhead. The overhead of this approach can be reduced by recognizing that with three parity bits, up to four bits of data can be protected against single bit flips. This naturally suggests the combination of two payloads of size $l - 4$ transmitted into a packet with a combined header (at the cost of increasing the transmission latency by one word):

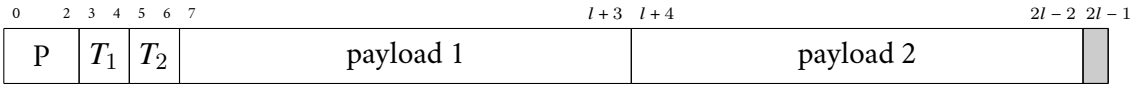

Figure 5: Optimized packet encoding that encodes up to two non-event, or event payloads into two phits. P are the parity bits to be able to correct a single bit flip in the header and T_n is the type of payload n, which can be either *none*, *event*, *non-event* or *non-event last*.

Here T_n contains the payload type of payload n, with the four possible types being *none*, *event*, *non-event*, and *non-event last*. The typical size of a non-event payload will be at least $72\,\mathrm{bit}$ (to transport a $64\,\mathrm{bit}$ write and $8\,\mathrm{bit}$ of byte enables) plus at least a $7\,\mathrm{bit}$ checksum. l is constrained by the ASIC

size and bonding constraints to $22\,\mathrm{bit}$ to $26\,\mathrm{bit}$. For bit flips in the payload a CRC with $7\,\mathrm{bit}$ is then sufficient to reach a negligible MTBF at $HD = 3$. For the number of phits required to transmit this message n, this gives a maximum of 5.

For these parameters, protecting the header bits against single bit flips with the above two payloads per packet scheme, has an overhead of $2n = 10\,\mathrm{bit}$. This can be compared to an unprotected header with an increased checksum size with the same total size, so $c = 17\,\mathrm{bit}$. The protected header scheme results in an MTBF of $O(10^7\,\mathrm{d})$, while the unprotected header scheme results in an MTBF of $O(10\,\mathrm{d})$. So for this set of parameters, the protected header scheme is drastically superior in terms of MTBF for the same overhead, so it is chosen for the encoding of the link data.

Finally, the encoding is optimized by recognizing that only a subset of combinations for the payload type is valid. If T_1 is idle, T_2 will also always be idle. Furthermore, a T_1 of non-event cannot be followed by idle and, finally, a non-event message is always made up of at least two payloads, so a non-event last payload cannot be followed by a non-event last payload. These restrictions can be used to construct the following prefix code:

payload type 1	payload type 2	encoding
event	event	111000_2
event	none	1001100_2
event	non-event	0111100_2
event	non-event last	0101010_2
non-event	event	1011010_2
non-event	non-event	1101100_2
non-event	non-event last	0010110_2
non-event last	event	1000011_2
non-event last	non-event	0001111_2
none	none	0000000_2

Table 6: Optimized encoding for the packet header. The constructed prefix code only requires 6 bit to encode a packet that contains two events, while all other packets require 7 bit. Each code differs by atleast three bits from every other code, so a single bit flip can be corrected.

Each code differs from every other code by at least 3 bit, allowing a single bit flip to be detected and corrected. This results in four different packet encodings. Packets containing two event messages:

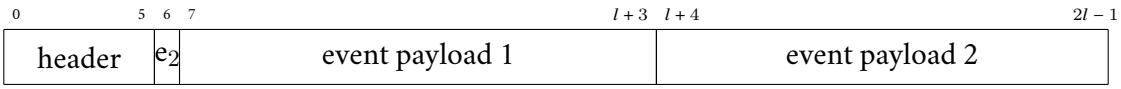

Figure 6: Optimized encoding scheme for a packet carrying two event payloads. Each event payload has $l - 3\,\mathrm{bit}$. The first bit of the second event payload is carried in e_2. Here l is the number of bits transported by the link in parallel. For BRISCET this is $22\,\mathrm{bit}$.

Here the first bit of the second event payload is carried in e_2. Packets containing first an event payload and then a non-event payload:

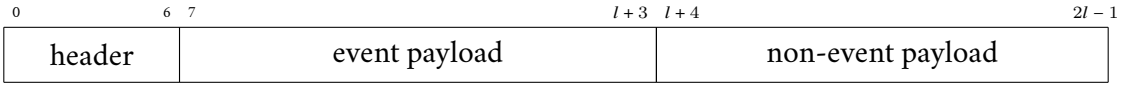

Figure 7: Optimized encoding scheme for a packet carrying a event payload followed by an non-event payload. The event payload has a size of $l - 3\,\mathrm{bit}$ while the non-event payload has a size of $l - 4\,\mathrm{bit}$. Here l is the number of bits transported by the link in parallel. For BRISCET this is $22\,\mathrm{bit}$.

Here the first bit of the second event payload is carried in e_2. Packets containing first a non-event payload and then an event payload:

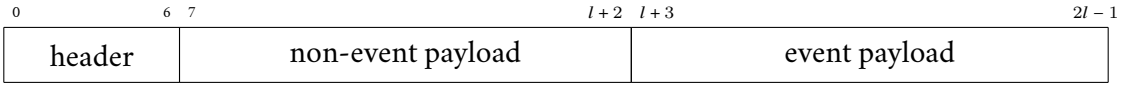

Figure 8: Optimized encoding scheme for a packet carrying a non-event payload followed by a event payload. The event payload has a size of $l - 3\,\mathrm{bit}$ while the non-event payload has a size of $l - 4\,\mathrm{bit}$. Here l is the number of bits transported by the link in parallel. For BRISCET this is $22\,\mathrm{bit}$.

And finally, packets containing two non-event payloads:

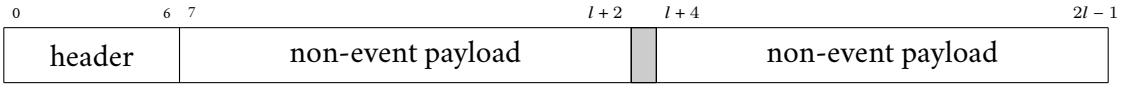

Figure 9: Optimized encoding scheme for a packet carrying two non-event payloads. Both non-event payloads have a size of $l - 4\,\mathrm{bit}$. A single bit is unused in this encoding. Here l is the number of bits transported by the link in parallel. For BRISCET this is $22\,\mathrm{bit}$.

Using this encoding, the overhead of the packet header per event can be reduced to $3\,\mathrm{bit}$, allowing event data to have a payload size of $l - 3\,\mathrm{bit}$. For non-event data, the overhead stays at $4\,\mathrm{bit}$ allowing $l - 2\,\mathrm{bit}$ of payload data per link word.

Figure 10 shows a schematic overview of the hardware implementation of the encoding for transmission of the event / non-event packets for $l = 22\,\mathrm{bit}$. The links operate at a $1\,\mathrm{GHz}$ clock rate. On the transmitting side, two streams are accepted: a stream of non-event messages with a size of up to $83\,\mathrm{bit}$ and a stream of event messages with a size of $17\,\mathrm{bit}$. Each non-event message gets combined with a 7 bit CRC checksum and a gearbox splits the non-event message into up to 5 fixed-sized 18 bit words and a framing bit. Furthermore, the gearbox performs clock domain crossing from the 200 MHz domain that the non-event data is processed at into the clock domain of the links. Arbitration between these non-event data words and event data is performed by a configurable weighted arbiter.

This arbiter accepts a 4 bit weight w, which controls how event and non-event data words get arbitrated. For $w = 0$, it operates as a priority arbiter that gives the event data priority. For $w \neq 0$ it still gives event data priority, but ensures a minimum bandwidth for non-event data of $1/w$ per clock cycle. The output of the arbiter goes to the link word encoder, which combines two words into a packet and generates the header according to the encoding described above. If not enough data is passed to the link word encoder, it inserts empty (idle) words as necessary to fill up the two payload slots of a packet, ensuring the data is sent with minimal latency. On the receiving side, these packets

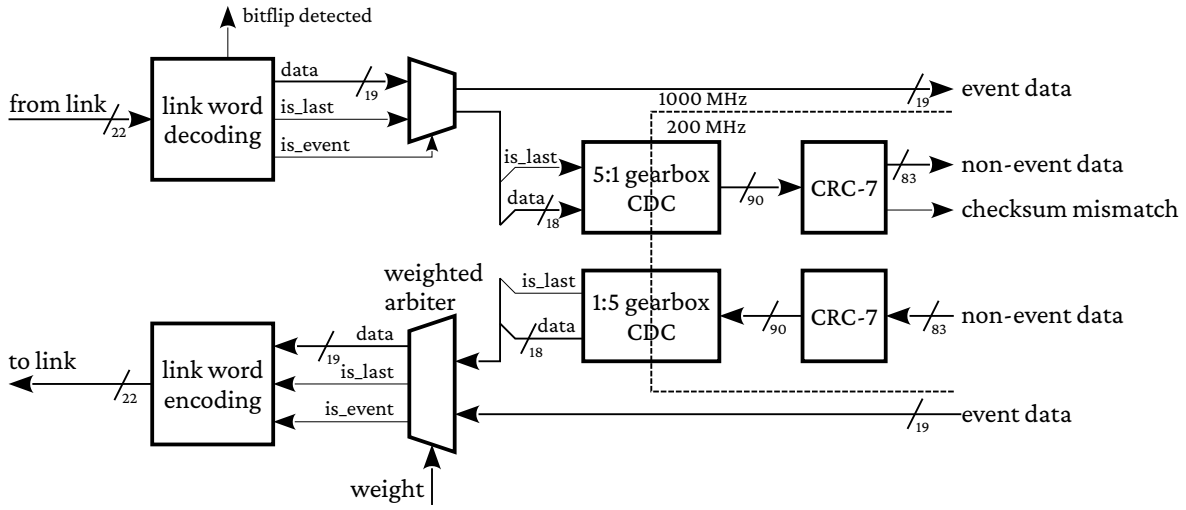

Figure 10: Block diagram of the hardware implementation of the link-level protocol. Incoming non-event data gets combined with a checksum before it is split into up to five link-level words by a gearbox, which simultaneously performs clock domain crossing. A weighted arbiter arbitrates between these link-level words and outgoing event data. In a final step, two link-words are combined into one packet and the header is added by the link word encoding block. Data received on the links goes through these steps in reverse: the packet header is decoded to determine the payload types. Up to five non-event data words get combined by a gearbox and the checksum is checked. Event data words get passed to the event handling downstream as is.

are decoded and non-event data is forwarded to a gearbox, which uses the framing information to combine up to 5 words into a non-event message. Again, this gearbox is also responsible for crossing the clock domain from the link clock domain to the slower non-event data clock domain. For every message, the received checksum is checked and, if correct, the message data is forwarded. Additionally, the receiving side outputs a signal that is asserted if a non-event message was received but the checksum was not correct.

4.3. Error control scheme

The link-level scheme fulfills two responsibilities: it allows sharing of the link by event messages and non-event messages and detects any corruption due to flipped bits for non-event messages. While these mechanisms prevent processing of any non-event messages that are erroneous, they do not ensure that all non-event messages sent by the sender are eventually received by the receiver, as messages can get lost due to bit flips corrupting their payload.

Similar to the high-speed interface of the BrainScaleS-2 ASIC, an ARQ protocol is employed to guarantee eventual error-free reception of all data sent by the sender.

4.3.1. Background on ARQ protocols

In network communication, automatic repeat request (ARQ) protocols are commonly used, for example in the TCP network protocol (Forouzan, 2003) and the PCIe protocol (*PCI Express® Base Specification Revision 6.* 2021), to enable reliable data transmission over a link that can loose or corrupt

data. It is also used by the high-speed interface of the BrainScaleS-2 ASIC. ARQ protocols are used for two responsibilities: first, to ensure any data sent by the sender is eventually received by the receiver and second, they ensure data is received on the receiving side in the same order as on the sending side. In general, ARQ protocols follow a scheme where the sending side stores and repeatedly transmits a data word until it receives an acknowledgement from the receiver of successful reception of the transmitted data word. The usage of ARQ protocols in network communication can be grouped into two categories:

1. Link-level ARQ protocols are employed on a link-level, independently protecting every link between components of the network. This is, for example, used by PCIe.

2. Point-to-point ARQ protocols protect data on a point-to-point or flow basis. This is, for example, used by TCP.

Important for the hardware implementation of ARQ protocols is the achievable throughput T for a given buffer size B on the sending side. As the sending side has to store the sent data until it receives an acknowledgement from the receiving side, an upper bound for the throughput of any ARQ protocol is given by

$$T = \frac{B}{t_{\mathrm{ACK}}}$$

where t_{ACK} is the time that is required for the sent data to arrive at the receiver and for an acknowledgement sent by the receiver to arrive at the sender. This can be further decomposed into the round-trip time t_{RTT} between the sender and the receiver and processing overheads on the sending and receiving side.

$$t_{\mathrm{ACK}} = t_{\mathrm{RTT}} + t_{\mathrm{sender,overhead}} + t_{\mathrm{receiver,overhead}}$$

For a network where a sender and a receiver are separated by multiple hops and therefore data needs to be transmitted over multiple links, the t_{RTT} for a point-to-point ARQ protocol will always be greater than the t_{RTT} for a link-level ARQ protocol.

The necessary buffers for an ARQ protocol furthermore depend on the scheme used for the acknowledgements. This includes Go-Back-N schemes, which employ no buffer on the receiving side. This means that if a single message is corrupted in transmission from sender to receiver, the receiver must stop accepting any subsequent messages until the corrupted message is received, to ensure that messages are be processed downstream in the same order as they were sent. This means the sender has to resend all messages starting with the corrupted one. A different approach is a Selective-Repeat scheme, which includes a buffer on the receiving side to allow messages received after a corrupted message to be stored. Using this scheme, the sender can selectively resend only the messages that were actually corrupted.

The high-speed interface of the BrainScaleS-2 ASIC uses an ARQ protocol that combines concepts from Go-Back-N ARQ protocols and Selective-Repeat ARQ. It has a receiver-side buffer like in Selective-Repeat ARQ, however, resends are not selective and instead resend all unacknowledged values. The receiver buffer instead has its main use as a reorder buffer, as the channel bonding

technique employed by the BrainScaleS-2 ASIC can cause data to be sent out of order. Finally, there is no separate channel for acknowledgements, acknowledgements and normal data packets are multiplexed over the same link. Acknowledgements are cumulative, with their frequency determined dynamically from the frequency of data packets received.

4.3.2. Proposed ARQ protocol scheme

Multiple factors were considered to design a suitable ARQ protocol:

- The size of the required buffers should be minimized to minimize the area of a hardware implementation.
- The protocol should be able to sustain a throughput close to the link bandwidth.

A link-level ARQ protocol has two advantages. First, for a given throughput T, the buffer size B increases with increasing t_{ACK}. t_{ACK} for a point-to-point ARQ protocol will always be at least as large as t_{ACK} for a link-level protocol, as the data has to traverse at least one link to move from sender to receiver. So a link-level ARQ protocol always needs at most the same buffer size B as a point-to-point ARQ protocol for a given throughput, and usually much less if sender and receiver in the point-to-point case are separated by multiple hops. The second advantage is its simpler implementation. A link-level ARQ protocol only processes a single flow of data at once. To allow a sender to target multiple receivers in parallel, a point-to-point ARQ protocol implementation would have to track multiple independent data flows at the same time.

The disadvantage of a link-level ARQ protocol is that it operates on the link-level and cannot use the independence of the (potentially multiple) flows transmitted across a single link. While data belonging to the same flow has to be transmitted in-order, data belonging to different flows is allowed to be reordered. A point-to-point ARQ protocol allows independent flows to progress independently, while with a link-level ARQ protocol an error in data belonging to one flow also blocks the transmission of data belonging to different flows.

Due to the increased complexity and required buffer sizes, a link-level ARQ protocol was chosen. Furthermore, a Go-Back-N scheme is used to avoid a buffer on the receiving side. The behavior of the protocol design in this thesis is determined by four parameters:

W: the window (buffer) size of the sender, limited to powers of two

$t_{timeout,sender}$: the timeout time of the sender

$t_{timeout,receiver}$: the timeout time of the receiver

n_{ack}: the number of words that are acknowledged cumulatively

The sender and the receiver operate in the usual sliding window fashion. Each message that should be sent by the sender gets assigned a consecutive sequence number and gets stored in a buffer with W entries. The sender can accept up to W entries. Each message entered into the buffer is

sent to the receiver. Acknowledgements sent by the receiver are cumulative and carry the highest consecutively received sequence number. Whenever the sender receives an acknowledgement, it removes all messages with a sequence number smaller than or equal to the sequence number carried in the acknowledgement message from the buffer. A resend of all messages in the buffer by the sender is triggered in two cases if the sender is not already in the process of resending all messages:

1. If no acknowledgement was received for a time of $t_{\text{timeout,sender}}$
2. If a negative acknowledgement was received

The receiver has no buffer and accepts a message whenever its sequence number is consecutive to the previously received sequence number. The transmission of acknowledgements to the sender is controlled by two variables. First is `n`, the number of successfully received messages since the last (positive or negative) acknowledgement was sent. Second is `nack_scheduled`, which is set, whenever a negative acknowledgement is sent and cleared, whenever a message is successfully received. There are five cases that cause transmissions of acknowledgements:

1. Whenever `n` equals n_{ack}, a positive acknowledgement is sent.
2. If no acknowledgement was sent for a time of $t_{\text{timeout,receiver}}$ and $\texttt{n} > 0$, a positive acknowledgement is sent.
3. Whenever a checksum mismatch is detected and `nack_scheduled` is not set, a negative acknowledgement is sent.
4. Whenever a sequence number that is larger than the next consecutive sequence number and `nack_scheduled` is not set, a negative acknowledgement is sent.
5. Whenever a sequence number that was already received is received, a positive acknowledgement is sent.

For the hardware implementation it is important to note that at most W messages can be outstanding, so restricting the sequence numbers to $\mathbb{Z}/2W\mathbb{Z}$ is sufficient to encode the sequence number.

To analyze the behavior of this protocol, let us consider a fixed rate R of messages that are provided to the sender, a link between the sender and the receiver that has a maximum message rate of $R_{\text{max}} \geq R$ and an error rate ρ_{link} per message. Finally, let $t_{\text{ACK,min}}$ be the minimum possible time between transmission of a message word by the sender and reception of an acknowledgement for this data word.

First, let us consider the case where $\rho_{\text{link}} = 0$. n_{ack} messages have to be received by the receiver to trigger an acknowledgement. Between the sending of the message that causes the receiver to send an acknowledgement and the reception of the acknowledgement, a time of $t_{\text{ACK,min}}$ passes, during which a further $R \cdot t_{\text{ACK,min}}$ of messages are provided to the sender. The maximum amount of unacknowledged messages is then

$$\text{buf}_{\text{max,no error}} = n_{\text{ack}} + t_{\text{ACK,min}} \cdot R$$

Now, if $\rho_{\text{link}} \geq 0$ a message can get corrupted in transmission. Initially, the influence of a single transmission being corrupted is considered. There are two cases: First, if an acknowledgement gets corrupted, the sender will be unable to remove any successfully received messages from the buffer until the next acknowledgement gets received, which is sent by the receiver after a further n_{ack} messages are received by it. The maximum number of unacknowledged messages on the sender side is therefore

$$\text{buf}_{\text{max,ack error}} = 2n_{\text{ack}} + t_{\text{ACK,min}} \cdot R$$

If a message gets corrupted, any message sent after it is ignored by the receiver until the corrupted message is received correctly. Upon reception of a corrupted message, the receiver sends a negative acknowledgement immediately to the sender. This acknowledgement acknowledges the successful reception of all messages up to and excluding the corrupted one. The time elapsed between transmission of the corrupted message by the sender and the reception of the negative acknowledgement is given by $t_{\text{ACK,min}}$. After reception of the negative acknowledgement, the sender will therefore have $1 + R \cdot t_{\text{ACK,min}}$ unacknowledged messages stored in its buffer. The negative acknowledgement causes the sender to resend all these unacknowledged messages, at the maximum rate R_{max} possible. In the worst case, it takes $n_{\text{ack}}/R_{\text{max}} + t_{\text{ACK,min}}$ for the next acknowledgement to be received by the sender. This means that a single message being corrupted will cause the sender to have a maximum number of unacknowledged messages $\text{buf}_{\text{max,message error}}$ of

$$\text{buf}_{\text{max,message error}} = 1 + R \cdot t_{\text{ACK,min}} + \left(\frac{n_{ack}}{R_{\text{max}}} + t_{\text{ACK,min}}\right) \cdot R \leq 1 + 2 \cdot t_{\text{ACK,min}} + n_{\text{ack}}$$

So for the protocol to be able to accept a message rate of R if a single error occurs, the sender has to have a

$$W \geq \max(\text{buf}_{\text{max,message error}}, \text{buf}_{\text{max,ack error}}) := W_{\text{min}}$$

Note that this analysis is unchanged if instead of a Go-Back-N a Selective-Resend ARQ protocol is employed and therefore would also have the same constraint on W. Finally, let us consider the effect of multiple transmissions being corrupted. Considering the case where an arbitrary number of consecutive transmissions is corrupted, it is clear that in general no hard bound on the necessary buffer space for a given message rate R exists. Instead, here a lower bound for the mean time between failure — the buffer of the sender being full — for a sender with $W = W_{\text{min}}$ is determined. As outlined above, a corrupted message causes up to

$$\Delta\text{buf}_{\text{message error}} = \text{buf}_{\text{max,message error}} - \text{buf}_{\text{max,no error}} = 1 + R \cdot t_{\text{ACK,min}}$$

more buffer space to be used. The link limits the maximum rate the sender can send messages at to R_{max} and data enters the sender at a rate of R, so the maximum rate that the number of unacknowledged messages in the sender buffer can decrease is $R_{\text{decrease}} := R_{\text{max}} - R$. An upper bound for the time

required to receive an acknowledgement for an arbitrary data word is furthermore given by

$$t_{\text{ACK,max}} = \frac{n_{\text{ack}} - 1}{R} + t_{\text{ACK,min}}$$

This means, if no additional errors occur, an upper bound for the time required for the sender to remove the additional unacknowledged messages in the buffer caused by a message being corrupted, $t_{\text{recover,message error}}$, is given by

$$t_{\text{recover,message error}} = \frac{\Delta\text{buf}_{\text{message error}}}{R_{\text{decrease}}} + t_{\text{ACK,max}}$$

After this time, the sender has enough buffer space to accommodate an error during a transmission again. During this time, the number of transmissions is given by

$$n_{\text{recover,message error}} = (1 + 1/n_{\text{ack}})(t_{\text{recover,message error}} \cdot R + \Delta\text{buf}_{\text{message error}})$$

So the buffer of the sender is guaranteed not to fill up if a corrupted message is followed by at least $n_{\text{recover,message error}}$ transmissions without error. An upper bound for the probability $p_{\text{full, message error}}$ for the buffer of the sender to fill up can therefore be determined from the probability that at least one of $n_{\text{recover,message error}}$ transmissions following a corrupted message is corrupted:

$$p_{\text{full, message error}} = \rho_{\text{link}} \cdot \text{binomsf}(0, n_{\text{recover, message error}}, \rho_{\text{link}})$$

The MTBF for this case is then given by

$$\text{MTBF}_{\text{message error}} = 1/(R p_{\text{full, message error}})$$

Analogously, the case of a corrupted acknowledgement can be analyzed:

$$\begin{aligned}
\Delta\text{buf}_{\text{ack error}} &= n_{\text{ack}} \\
t_{\text{recover,ack error}} &= \frac{\Delta\text{buf}_{\text{ack error}}}{R_{\text{decrease}}} + t_{\text{ACK,max}} \\
n_{\text{recover,ack error}} &= (1 + 1/n_{\text{ack}})(t_{\text{recover,ack error}} \cdot R + \Delta\text{buf}_{\text{ack error}}) \\
p_{\text{full, ack error}} &= \rho_{\text{link}} \cdot \text{binomsf}(0, n_{\text{recover, ack error}}, \rho_{\text{link}}) \\
\text{MTBF}_{\text{ack error}} &= \frac{n_{\text{ack}}}{R \cdot p_{\text{full, ack error}}}
\end{aligned}$$

Finally, a third case needs to be considered, which is a negative acknowledgement being corrupted. In this case, a resend is only triggered after $t_{\text{timeout,sender}}$. This case is again considered as leading to the buffer of the sender filling up. For any message being sent, the probability of a negative acknowledgement being corrupted $p_{\text{nack error}}$ is given by

$$p_{\text{nack}} = p(\text{negative acknowledgement}) \cdot p(\text{negative acknowledgement corrupted}) = \rho_{\text{link}}^2$$

and the MTBF in this case is

$$\mathrm{MTBF}_{\text{nack error}} = 1/(R\rho_{\text{link}}^2)$$

The joint MTBF for these three cases is finally given by

$$\mathrm{MTBF}_{\text{full}} = (\mathrm{MTBF}_{\text{message error}}^{-1} + \mathrm{MTBF}_{\text{ack error}}^{-1} + \mathrm{MTBF}_{\text{nack error}}^{-1})^{-1}$$

Figure 11 shows this resulting MTBF for different values of R and minium link latencies $t_{\text{ACK,min}}$ for a word error rate of the link of $\rho_{\text{link}} = 10^{-8}$ a maximum transmission rate of the link $R_{\text{max}} = 200 \times 10^6/\text{s}$ and cumulative acknowledgements for $n_{\text{ack}} = 8$ messages. As expected, the MTBF increases significantly if R is significantly smaller than R_{max}. Furthermore, even if R is close to R_{max}, up to about $R = 0.925R_{\text{max}}$, the MTBF is greater than 1 d.

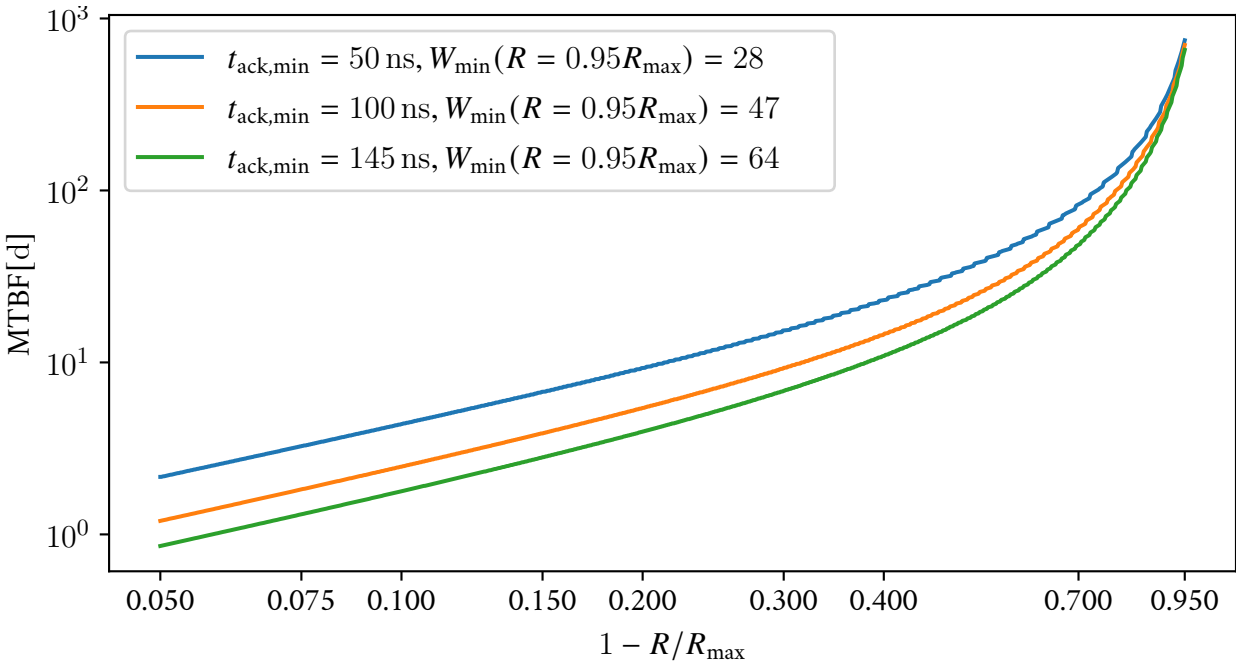

Figure 11: MTBF for different $t_{\text{ACK,min}}$ and message rates R for a $\rho_{\text{link}} = 1\times10^{-8}$, $R_{\text{max}} = 200\times10^6/\text{s}$ and $n_{\text{ack}} = 8$. The x-axis gives the message rate R as the relative difference to the maximum data rate of the link R_{max}.

For the hardware implementation, a W of 64 was chosen to allow for a $t_{\text{ACK,min}}$ of up to 145 ns to sustain a throughput of $0.95R_{\text{max}}$. This throughput can then be sustained with a lower bound for the MTBF of $\mathcal{O}(1\,\text{d})$. As a failure of this kind does not compromise system stability, but instead only causes a transient drop in the sustained throughput, this MTBF is considered acceptable. Acknowledgements are transmitted from the receiver to the sender as independent link-level non-event messages. For the encoding as a non-event message as used by the link layer protocol, the sequence number is then transmitted as a 7 bit number and a header bit is used to differentiate between messages containing payloads and messages containing acknowledgements. The secured messages are transmitted with a format of:

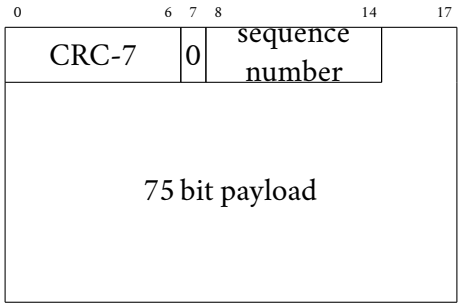

Figure 12: Format of the messages handled by the ARQ protocol.

4.4. Credit flow control

Processing downstream from the ARQ receiver is split into two independent data flows, the two independent virtual channels. These virtual channels are generally not guaranteed to be able to process messages at the same rate as the ARQ receiver can receive them. Furthermore, the rate of data that can be processed by the two virtual channels is not necessarily the same. This is, for example, the case when the packets in one virtual channel are routed to a highly congested link, but the packets in the other virtual channel are routed to a mesh link with low congestion. The ARQ receiver has no buffer space of its own, so if it receives a message that cannot be processed downstream immediately, it has to drop the received message, causing retransmission of it down the line. To avoid the sender sending messages that cannot be processed by the receiver and therefore cause unnecessary (re-)transmissions, a credit-based flow control (William James Dally and Towles, 2004) scheme is employed to stop the flow of messages from the sender if they cannot be processed on the receiving side.

Credit-based flow control schemes operate by the sender keeping track of an account of credits. Data is only allowed to be sent (in this case, enter the ARQ sender) if the sender has credits available. For every message sent by the sender, a unit of credit is subtracted from the account of credits the sender keeps. Credits are returned to the sender by the receiver for every message it has processed. The return of credits to the sender by the receiver again needs a back-channel, similar to the acknowledgements in the ARQ protocol.

For the implementation for BRISCET, to avoid extra messages used purely for the credit flow control, the return of credits to the sender piggybacks on the acknowledgements sent by the ARQ protocol, by adding the credit information to them. Furthermore, instead of returning units of credits one-by-one, the credit messages are cumulative, similar to the cumulative acknowledgements sent by the ARQ receiver. In addition to piggybacking on the acknowledgements sent by the ARQ protocol, the credit counting mechanism on the receiving side also causes acknowledgements (and therefore credit messages) to be sent. An acknowledgement is triggered whenever more than k messages have been processed since the last acknowledgement was sent.

Finally, the credit information is not encoded as a relative change (the number of messages received since the last credit message), but instead as an absolute count of the received messages since the start of the system. This allows the credit return to be tolerant against lost update messages, as a later message always includes the credits returned in previous messages. Again, equivalent to the sequence

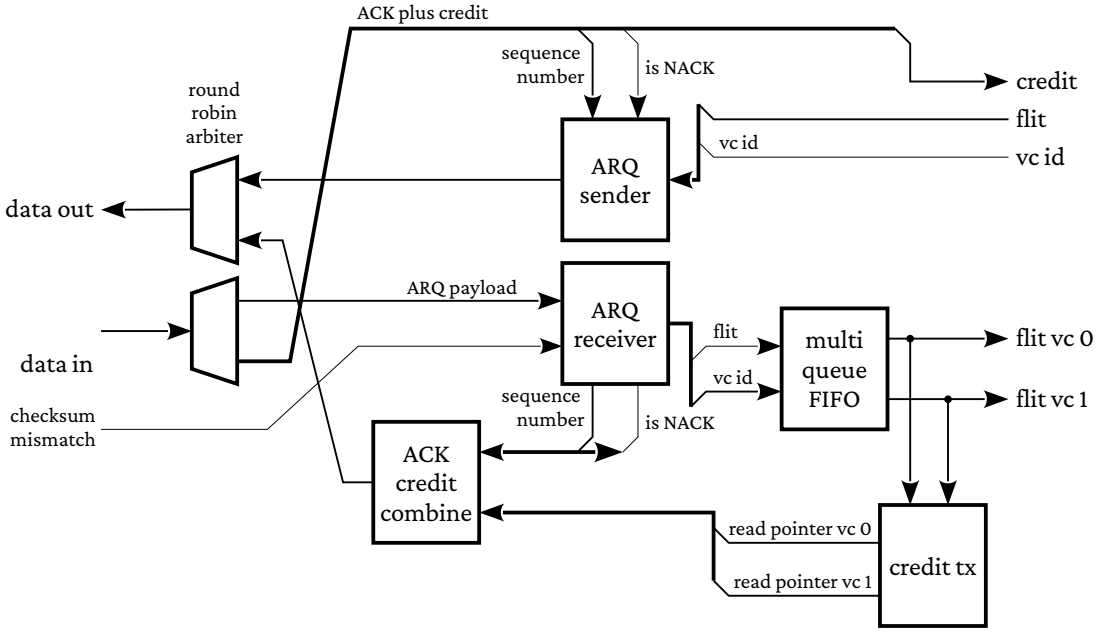

Figure 13: Block-level diagram of the error and flow control schemes implemented for the mesh links. Non-event messages are processed in fixed-sized chunks called flits. A flit that is sent over a mesh link enters the ARQ sender and gets forwarded to the link-level encoding described in figure 10. This stream of payloads is combined with acknowledgement plus credit messages created by the ACK-credit combine unit using round-robin arbitration. Data coming from the link-level protocol gets split up by type. ACK-credit messages get forwarded to the ARQ sender and the credit counting mechanism further downstream of the ARQ sender. Data messages are forwarded to the ARQ receiver. If valid and in-order, the ARQ receiver forwards the received data to an input buffer, which has two independent queues for the two independent virtual channels. These queues are connected to the routing component. By having two independent queues, two indenpedent flows of packets can progress at different speeds. The flits sent downstream are monitored to create the credit information that is sent back to the sender across the mesh link.

numbers in the ARQ protocol, if the available buffer space b on the receiver is a power of two, it is sufficient to transmit this running count c as $\bar{c} = (c \mod 2b)$, requiring $1 + \log_2(b)$ bits.

Again, the processing downstream from the ARQ receiver is split into two independent virtual channels with their own buffer resources, which are allowed to progress independently. Each of these two buffers uses its own credit account on the sender side, and the credit messages returned by the receiver always include the count of received messages for both.

figure 13 shows how this is implemented for the hardware design.

The ACK-credit combiner unit receives the credit information and the ACK information and combines this into a single message. This is encoded for the link-layer as follows:

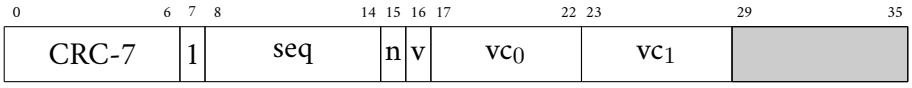

Figure 14: Encoding used for messages carrying acknowledgements and credit information. Here seq carries the sequence number of the acknowledgements as determined by the ARQ receiver and vc_n carry the credit counts for the two virtual channels. Finally, n indicates a negative acknowledgement, for example, when the receiver detects a checksum mismatch and v indicates the validity of the sequence number field.

where

seq is the highest consecutively received sequence number by the ARQ receiver.

n is set for negative acknowledgements.

v is set to indicate the validity of the sequence number field.

vc_n is the running count of messages processed on virtual channel *n*.

If backpressure from the link layer causes an ACK-credit message remain unsent when then next acknowledgement is triggered by the ARQ receiver, or the credit counting mechanism in the combiner unit, the old message is discarded and replaced with the new one, avoiding unnecessary transmission of outdated information.

The input buffers for the two virtual channels are implemented using a shared buffer statically partitioned into two equally sized halves. Sharing the buffer decreases the required area due to fixed overheads associated with a buffer. The buffer has a single read port and a single write port. A single write port is sufficient in all cases, as at most one flit can be received between the two input channels. On average, a single read port is also sufficient to remove flits from the buffer at the same rate they can enter. For the buffer size, 32 messages for each virtual channel was chosen as a tradeoff between required area and provided buffer space.

4.5. Non-event router

The topology of the interconnection network created by the BRISCET is a 2D mesh. To be able to send messages from one node in this mesh to any other node, a message may traverse multiple nodes of this mesh to go from sender to receiver. This makes a routing scheme necessary which allows intermediate nodes to determine where a received message has to be forwarded to. Each BRISCET has five (logical) ports on which messages are received: the four mesh ports connecting it with the neighbors and a local port it uses to inject its own messages. Similarly, it has five output ports, four to the neighbors and one to accept local messages. Each of these ports is further divided into two virtual channels whose flow of messages is independent of each other. The routing header adds an overhead to each message that is sent over the mesh. To reduce this overhead, messages are transmitted over the mesh in the form of variable-length packets. Each packet has a single routing header and some amount of payload. The link-level and ARQ protocol operate on units with a maximum size of 75 bit, therefore a packet is split up into multiple of these units called flits. A variable-length packet is then encoded as a sequence of four different types of these flits:

`start` contains the routing information and further payload data. It is followed by at least one further **`payload`** or **`tail`** flit.

`payload` contains payload data and is followed by either a further **`payload`** flit or a **`tail`** flit.

`tail` contains payload data and signifies the end of a packet. Only **`start`** and **`start and end`** flit can follow this flit.

start and end encodes a packet composed of a single flit, it contains routing information and payload data and signifies the end of a packet. Only **start** and **start and end** flit can follow this flit.

start and **start and end** flits are encoded into the 75 bit payload data processed by the ARQ as:

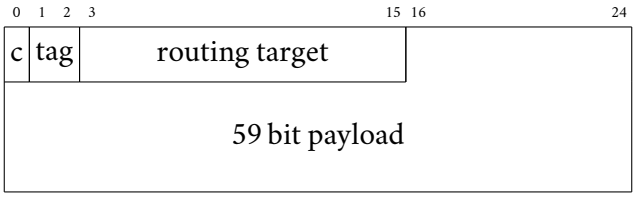

Figure 15: Encoding of **start** and **start and end** flits used by the router. They carry a virtual channel id c, a tag identifying the flit type, and a 13 bit routing target that is interpreted downstream by the route computer to determine the target of the packet. Each of these flits carries a 59 bit payload.

payload and **tail** flits are encoded into the 75 bit payload data processed by the ARQ as:

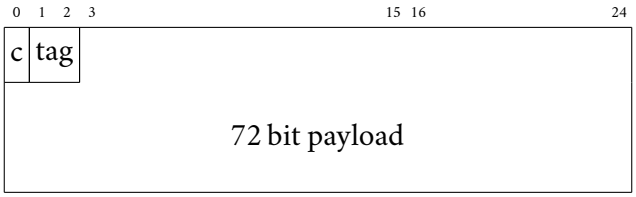

Figure 16: Encoding of **payload** and **tail** flits used by the router. They carry a virtual channel id c, a tag indentifying the flit type, and finally a 72 bit payload.

Here,

c encodes the virtual channel,

tag encodes the flit type

BRISCET uses the classical architecture for the routing part of an interconnection framework, as described for example in William James Dally and Towles (2004), consisting three steps:

1. Route computation determines the target output port and virtual channel for an incoming packet.
2. Virtual channel allocation allocates the output virtual channel to the packets that target them.
3. Packets are forwarded to the correct output port by a crossbar.

Furthermore, it employs wormhole switching, first described in William J Dally and Seitz (1986), which performs routing not on a packet basis, but instead on a flit basis. The flits that compose a packet are forwarded to the output port before the complete packet is received. This decreases the latency and required buffer space. Figure 17 shows a schematic overview of the specific hardware implementation chosen for BRISCET. Each input port has a corresponding input channel, that processes both virtual channels for this port. The target port and virtual channel are determined for each virtual channel

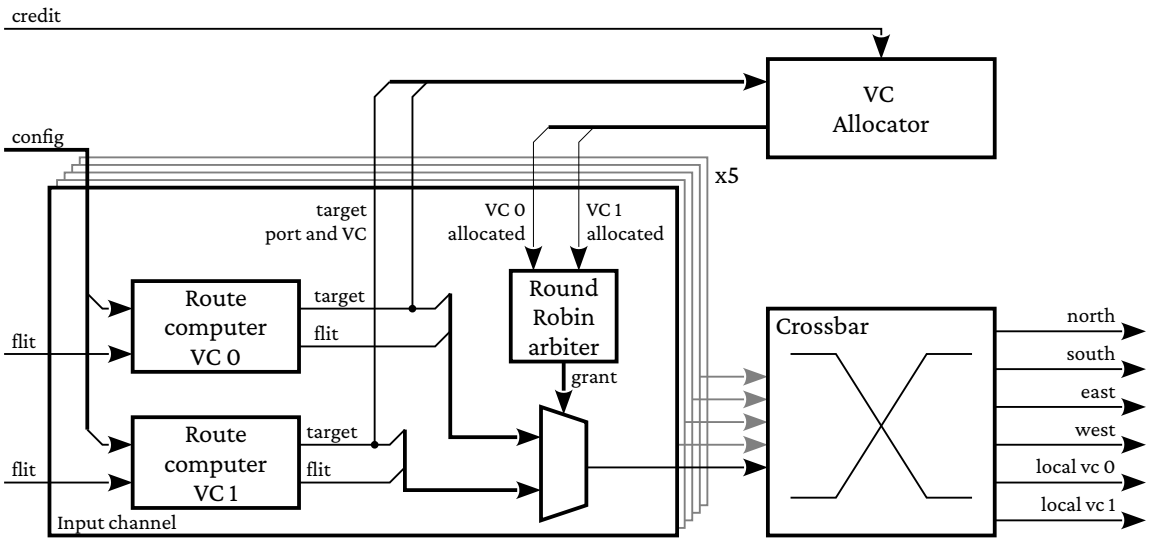

Figure 17: Block-level diagram of the router architecture implemented for BRISCET. Variable-length packets are processed by the router in the form of fixed-sized chunks called flits. Incoming flits go through a route computer to determine their target port and virtual channel. Each possible target virtual channel gets uniquely allocated to a single packet at any point in time by a central virtual channel allocator. A flit that has an allocation for its target virtual channel finally traverses a crossbar to its target. This crossbar has four output ports, one for each of the mesh links and one output port for each local virtual channel used by flits targeting the local mesh node.

separately by a route computer. Allocation of the target ports and channels determined by the route computer is requested from a central virtual channel allocator and upon successful allocation, the flits are forwarded to the crossbar. The crossbar only has a single input port per input channel, so between the two virtual input channels round-robin arbitration is used to determine which flit gets sent to the crossbar. Note that the two virtual channels of a port use a shared buffer with a single output port, so sharing the input port between the two virtual channels does not impose an additional constraint on the possible throughput. Finally, the crossbar has six output ports, one for each mesh port and one for each local channel. This allows two flows targeting the local node to progress separately. Figure 18 shows the implementation of the route computer. It uses the 13 bit routing information contained in the initial flit of a packet to determine the correct output port and virtual channel for the packet. The routing computer uses a combination of two routing algorithms. The first bit of the 13 bit routing information selects between these two algorithms. If it is zero, the coordinate is interpreted as a pair (x, y) of two 6 bit numbers, corresponding to the coordinate of the node the packet is intended for:

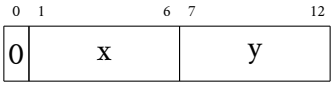

Figure 19: Encoding of routing information for packets routed using dimension-ordered routing. x and y give the target coordinate of the packet in the 2D mesh.

In this case, the target port of the packet is then determined by the route computer using dimension-ordered routing. The router determines the target port by comparing the coordinate encoded in the routing header with the coordinate of the node it is itself located at. It then selects an output port that

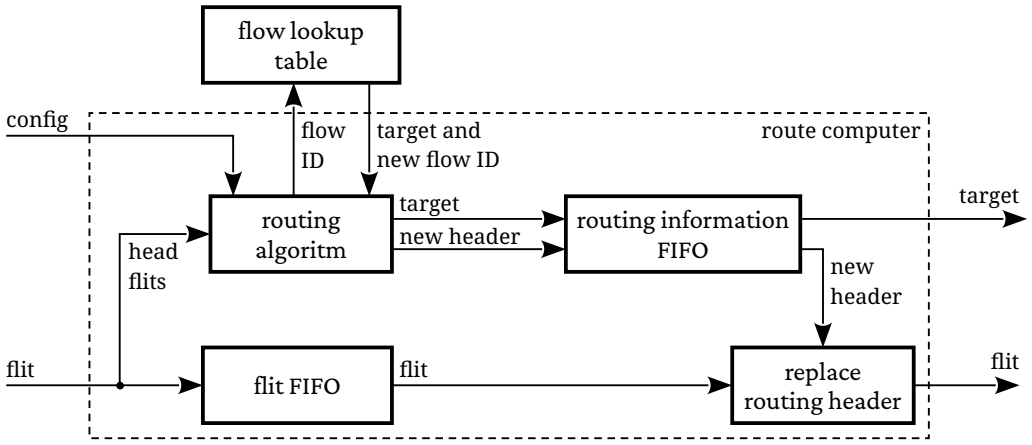

Figure 18: Block-level schematic of the architecture of the route computer implemented for BRISCET. Packets are routed via two different algorithms depending on their header. They are either routed using dimension-ordered routing or using a lookup table on each node, shared by all the route computers on the node. The route computer determines the target port and virtual channel as well as a new routing header that replaces the old one. The router is configured using omnibus-accessible configuration registers.

is connected to a node that is closer to the target coordinate. The north port goes to decreasing y coordinates, the west port goes to decreasing x coordinates, and so on. Configurably, either the x or the y direction is prioritized. This prioritization has to be the same for every node in the mesh to avoid deadlocks. Finally, the target virtual channel is determined from the virtual channel the flit was received on. Note that the target virtual channel has to be determined using a deterministic algorithm (and cannot, for example, be dynamically chosen based on congestion), in order to fulfill the in-order requirement of tunneling omnibus transactions. For the dimension-ordered routing of packets, the target virtual channel is simply statically assigned a configurable target virtual channel. If the first bit of the coordinate is set, the coordinate is interpreted as a flow ID:

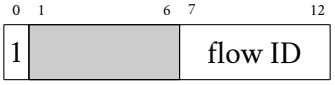

Figure 20: Encoding of routing information for packets routed using the local route table. The flow ID gives an index into the local route table, which contains the target port and a replacement routing information header that replaces the old one.

The target port and virtual channel for this ID are determined from a lookup table using the ID as index. Each entry of the lookup table contains the target port and virtual channel as well as a replacement coordinate:

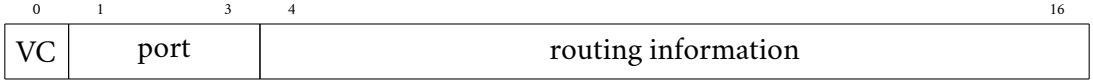

Figure 21: Encoding of the entries of the local lookup table. Each entry contains the target virtual channel and port for the packet as well as a replacement for the routing information carried in the packet header.

The route computer replaces the routing header in the head flit with the replacement routing

information stored in the lookup table. The lookup table containing these entries is shared between all route computers of all input channels. The flow ID was limited to 6 bit in the current implementation of BRISCET to match the total size of the lookup table of 64 entries, which was chosen as a tradeoff between required area and the number of different flows a single node can route this way. Using this lookup table-based scheme allows packets to be routed with greater flexibility than the dimension-ordered routing. For example, if all flows and their required bandwidth are known, the algorithm proposed in Shim (2010) can be used to construct an optimized in-order routing scheme with higher path diversity than a dimension-ordered scheme.

Finally, the hardware implementation of the route computer is pipelined, allowing the route computation of a subsequent packet to be performed while the current packet is still being sent.

Figure 22 shows a schematic overview of the architecture for the virtual channel allocator. The virtual channel allocator is responsible for arbitrating access to the target virtual channels. The router processes 10 independent flows of data — two virtual channels for each of the four cardinal directions and two virtual channels for the local port. Similarly, there are 10 different targets that these flows can be routed to — two virtual channels for each of the mesh neighbors and two virtual channels for the local output port. For each of the input channels, the VC allocator receives the target channel. This target is a tuple of target port (north, east, west, south, and local) and virtual channel ID. Each of these targets is encoded into request lines for each of the target virtual channels. This encoding is one-hot, a given input flow can at most have a single target. If two or more input flows have the same target channel, the allocation is performed in a round-robin fashion on a packet level, and the allocation of a target channel to an input flow is kept until the last flit of a packet is transmitted. For the mesh ports, the two virtual channels on the target side share a single link with a single ARQ sender and receiver. The allocation of the target channels is therefore contingent on buffer space being available, as determined by the credit counting flow control scheme described in section 4.4. The hardware implementation of the virtual channel allocator is optimized by restricting the targeted port of an input channel to be different from its own port. For example, the north input channel is not allowed to send data to the north neighbor. Finally, figure 23 shows a block diagram of the crossbar implementation. The crossbar has five inputs, one for the local input and four for the mesh inputs, and six outputs, one for each mesh port and one for each of the local channels. Each possible target virtual channel is uniquely allocated to a single input by the VC allocator. Nonetheless, a single one of the mesh output ports of the crossbar can be the target of multiple inputs, as the two virtual channels per output port share the link. In this case, the crossbar performs round-robin arbitration on a flit basis. For the local output channels, no arbitration is necessary, as these get uniquely allocated to a single input by the VC allocator. Again, the hardware implementation of the crossbar is optimized by omitting the connections between input and output ports that are equal: the north input port is not connected to the north output port, and so on.

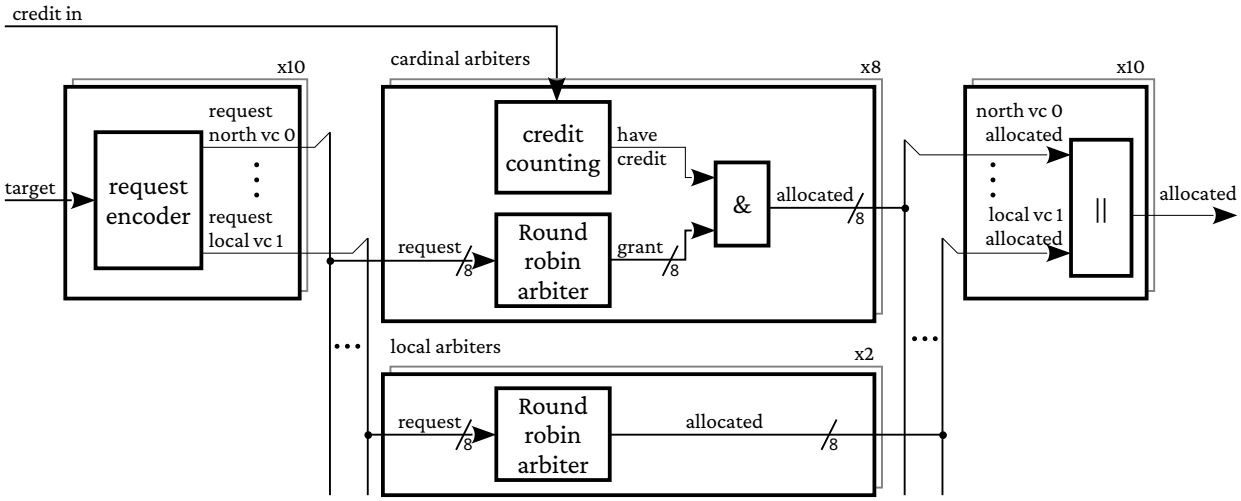

Figure 22: Block-level diagram of the virtual channel allocator. It receives the requested target port and virtual channel of all incoming packets (two for each mesh link and two for the packets locally injected into the network). Among these requests, it allocates a target port and virtual channel uniquely to a single incoming packet, using round-robin arbitration when multiple incoming packets request the same target port and virtual channel at the same time. Furthermore, it uses the credit counting information to block allocation of virtual channels that do not have buffer space available on the receiving side.

4.6. Tunneling of omnibus transactions

The components described above — the link-level protocol, the ARQ protocol, and the routing scheme — form the basis of the interconnection network. There are multiple different types of data flows that are transmitted over this interconnection network, including omnibus transactions. For a given BRISCET node, there are six omnibus masters:

- The BRISCET node includes a RISC-V core, which can generate omnibus transactions.
- The JTAG interface of BRISCET.
- One (read-only) source of transactions for each PPU used for instruction fetching.
- One (read and write) source of transactions for each PPU used for data loads and stores.

Furthermore, each BRISCET can also be a sink for omnibus transactions. These can either target the omnibus tree on the BRISCET node, the omnibus tree of the BrainScaleS-2 ASIC, or the FPGA. The high-speed interface of the BrainScaleS-2 ASIC splits the load and store PPU omnibus transactions into two different packet types: one packet type containing the address, transaction type (read or write), and the byte enables, and one packet type containing write payload data. Each write transaction generates three packets: one with the address, type, and byte enables, and two containing 64 bit of write data each. However, executing a write omnibus transaction needs both the address, type, and byte enable information as well as the write payload. This means buffers for both of these packet types are necessary, as they need to wait for packets of the other type before they can be processed. The high-speed interface of the BrainScaleS-2 ASIC tries to minimize the necessary size of this buffer by

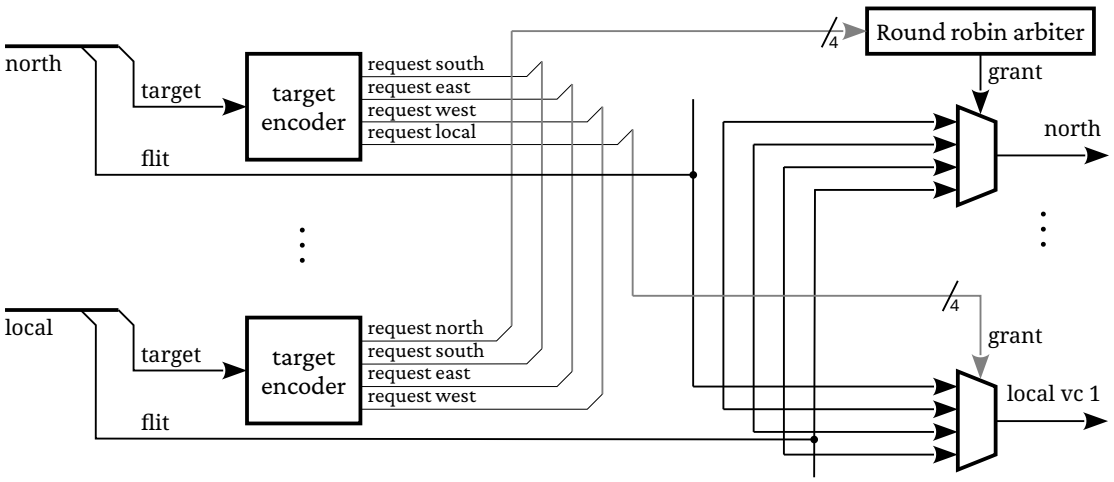

Figure 23: Block-level diagram of the crossbar implementation. Each input port of the router (north, east, west, south and the local port) is routed to the possible output ports, with round-robin arbitration in case of conflicts. It has six output ports, four for the mesh links and two for packets that target the local node.

using round-robin arbitration between these two packet types on the sending side, however, this still requires a buffer for up to eight packets containing the addresses, transaction type and byte enables, as every packet of this type requires two packets containing payload data to be processed. This kind of arbitration is not possible when tunneling the transactions over the mesh network. Even if the source node of a transaction arbitrates between the packet types with the same strategy as employed on the BrainScaleS-2 ASIC, such that the required packets are sent back-to-back to the target node, they are not guaranteed to be received back-to-back by the target node, as packets from other nodes targeting the same target could be mixed in between them. To avoid the need for buffers, it is therefore desirable to transmit each omnibus transaction as a single packet. A similar consideration must be made for the response data of transactions. omnibus transactions are ordered, the response data has to be provided to the source of the transactions in the same order as the transactions were issued. For a given pair of source and target nodes, the routing protocol guarantees that the transmission order of packets is maintained between source and target node. The same, however, is not true for packets sent to different targets from the same source node. Therefore, if multiple transactions to different targets are allowed to be in flight at the same time, the response data from the targets for these transactions can arrive in a different order than the transactions were issued or sent out by the source node. Allowing only a single transaction to be in flight at any given time alleviates this problem, but is undesirable, as it dramatically reduces the possible bandwidth. The same issue is also encountered when considering purely local omnibus topologies. A bus can have multiple slaves, with potentially different latencies between accepting transactions and providing response data. For local topologies, the bus splitter component developed in (Friedmann, 2013) is responsible for forwarding transactions from a single master to multiple slaves. It ensures that the response data is returned to the master in the same order as the transactions, by forwarding the response data from the multiple slaves in the same order as the transactions were issued. If a slave generates a response out of order, its response is stalled until it is the slave's turn. The solution of exerting backpressure until the correct response is received is

not possible in general for the tunneling of transactions over the mesh protocol. There are only two independent channels (the two virtual channels), with each channel being an ordered queue. If two responses *A* and *B* are expected, with *A* coming before *B*, but *B* was received before *A* on the same virtual channel, *B* needs to be stored in order to access (or receive) *A*. Keeping these considerations in mind, the following protocol for tunneling omnibus transactions over the mesh was devised. Read or write transactions and read responses are transmitted as a single packet, made up of one or more flits. Similar to the high-speed interface of the BrainScaleS-2 ASIC, write responses are not transmitted. All addresses are interpreted as word addresses with a $32\,\mathrm{bit}$ granularity. The first flit of a packet always carries a header, with subsequent flits carrying (header-dependent) payload data. The first two bits of the header `HT` identifying the packet type:

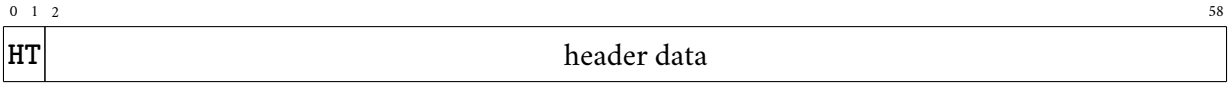

Figure 24: Encoding used for the **`start`** flits used for the omnibus tunneling protocol. HT defines the type of packet, which can be a read or write transaction or a packet carrying read response data. The further header data is then interpreted accordingly.

Three values are valid for `HT`. A read transaction is indicated by 00_2, an write transaction by 01_2 and a read response by 10_2. For write transactions, the header data contains:

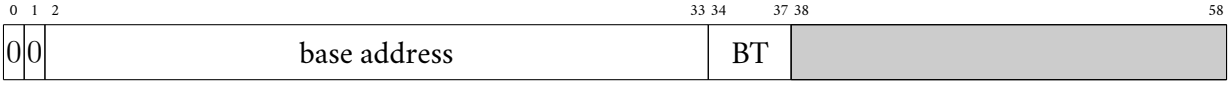

Figure 25: Encoding used for the **`start`** flits used for omnibus write transactions. The base address determines the base address of the transaction and BT the burst type, which can be incrementing or fixed.

The header flit containing the header of a write transaction is followed by one or more payload flits with the format:

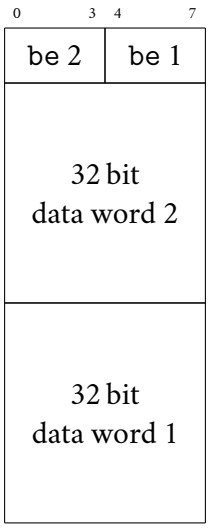

Figure 26: Encoding used for the flits carrying the payload of omnibus write transactions. The two data words carry the data to be written and the two `be` fields carry the corresponding byte enables.

These payload flits carry the data words to be written, and the corresponding byte enable bits `be`. The data is interpreted in chunks of 32 bit, and the base address as well as the burst type BT determines the address the data chunks get written to. There are four options for the burst type:

- 00_2: **`INCR`**. Then data word n is written to base address + n
- 01_2: **`FIXED4`**. Then data word n is written to base address + ($n \mod 1$)
- 10_2: **`FIXED8`**. Then data word n is written to base address + ($n \mod 2$)
- 11_2: **`FIXED16`**. Then data word n is written to base address + ($n \mod 4$)

Where data word $2n$ is the *data word 1* received in payload flit n and $2n + 1$ is *data word 2* of flit n. For read transactions, the header data contains:

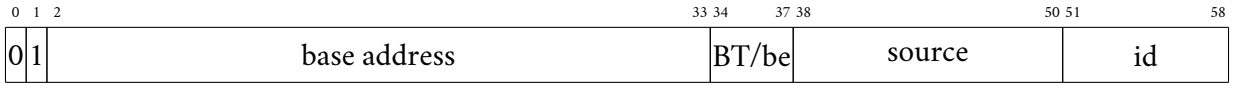

Figure 27: Encoding used for the **`start`** flits used for omnibus read transactions. The base address determines the base address of the transaction. id contains an id that is used by the response data sent to the source to determine which read transaction caused the response data. Interpretation of BT / be is either a burst type or byte enable bits depending on the amount of data read. See text for a more detailed description.

Read transactions are followed by zero or more payload flits with the format:

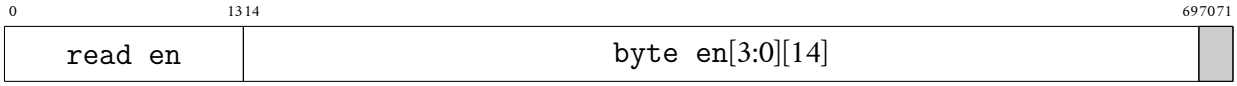

Figure 28: Encoding used for the flits carrying the payload of omnibus read transactions. Up to 14·32 bit read transactions are encoded, with the `read en` bits determining which read transactions should be performed and the `byte en` bits containing the corresponding byte enables for the read transactions.

The interpretation of the header depends on the number of payloads that follow it. If no payloads follow it, it is interpreted as a single 32 bit read at the given base address with the byte enables given by the BT/be field. If the header is followed by at least one payload flit, the BT/be field is interpreted as a burst type. Every payload flit contains up to 14 pairs of an enable bit and a byte enable. Only reads that have the enable bit set generate a transaction, and only for these bits is response data returned. The address for each read is determined in the same way from the base address and the burst type as for write transactions. The response data is sent as a single packet with the routing information given by the source field, and forwarding the ID specified in the header. Finally, for read responses the header data contains:

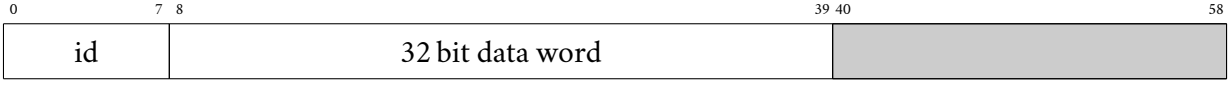

Figure 29: Encoding used for the **`start`** flits used for read response data. The id-field contains the id that was sent along with the read transaction that created the response data and the data word contains the response data for single 32 bit reads.

Read responses are followed by zero or more payload flits containing:

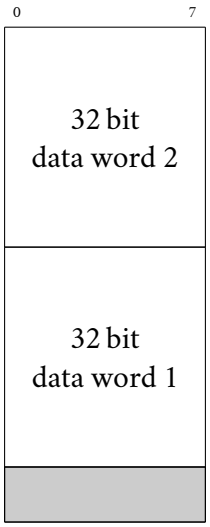

Figure 30: Encoding used for the flits carrying the payload for omnibus read responses. Each flit carries two 32 bit response data words.

Again, the interpretation of the data transmitted in the header depends on the number of payload flits following the header. If no payload flit follows the header, the "data word" field of the header contains a single 32 bit response value. If at least one payload flit follows the header flit, the "data word" field in the header is meaningless, and the response data is contained in the data fields of the payload flits.

The above described protocol has multiple advantages compared to the tunneling of omnibus transactions employed by the high-speed link of the BrainScaleS-2 ASIC. First, as outlined above, only a single packet for each transaction is generated, so it is not necessary to buffer multiple packets before processing one of them is possible. Furthermore, the protocol is designed such that the flits making up the packet can be processed in a streaming fashion, and it is not necessary for the whole packet to be received before the transactions can start to be processed. Note that while the interpretation of the read and read response header depends on the number of payload flits that follow them, this can be determined without waiting for the next flit from the type of the flit containing the header. If it is a **`start and end`** flit, no further flit will belong to the same packet. On the other hand, if it is a **`start`** flit, at least one payload flit will follow. Multiple in-flight read transactions to different targets by a single source are supported by the association of every read transaction with an ID. This ID is returned together with the read response data by the target and can be used to reorder the read response data on the sender side. Finally, this protocol is not specific to a fixed transaction size, but instead allows tunneling of different transaction sizes. The following number of flits are necessary to tunnel the different transaction sizes and types:

type	number of flits
32 bit read	1
32 bit read response	1
32 bit write	2
128 bit read	2
128 bit read response	3
128 bit write	3

Table 7: Summary of the number of flits required to encode the different omnibus transactions and response types using the protocol described above.

Additionally, in contrast to omnibus, the interpretation of the address is not dependent on the transaction size. omnibus uses addresses with the same granularity as its bus size. This means the same location in the address space has a different address depending on the omnibus bus size. In contrast, in the outlined protocol, the combination of the routing header of the packet and the address contained in the header for read and write transactions uniquely identifies a target location.

Figure 31 shows a schematic overview of the hardware implementation of the conversion from omnibus transactions to the described protocol. Each of the six omnibus masters uses a separate instance of the shown design. The omnibus transactions are accepted by and converted into three separate streams, a command stream containing the target address, transaction type, and byte enables. Second, a stream of write data and finally, a stream of read response data. For the omnibus masters tunneled by the high-speed interface of the BrainScaleS-2 ASIC (the PPU instruction fetches and data loads and stores), this step is skipped, the omnibus transactions are already tunneled as these three separate streams.

First, the commands are processed by the retarget block. The responsibility of the retarget block is to determine the routing header for a transaction. All six omnibus masters use $32\,\mathrm{bit}$ addresses to identify the target of a transaction. The goal is to allow any master to generate transactions targeting any node. While not all $32\,\mathrm{bit}$ of the addresses are used by the current address map, not enough are unused to directly encode an arbitrary routing header into the address. Instead, the retarget block uses a lookup table to determine the target coordinate. The entries of this lookup table are accessible from the BRISCET omnibus topology. If the address of an incoming command is given by

$$\texttt{fffi iixx xxxx xxxx xxxx xxxx xxxx xxxx}_2$$

then $\texttt{iii}_2$ is used as the index into the lookup table. Each entry contains four values. An entry $\texttt{ooo}_2$ that is used to replace the fill bits $\texttt{fff}_2$ bits, an entry $\texttt{rrr}_2$ that is used to replace the $\texttt{iii}_2$ bits, the virtual channel of the local input port of the router that should be used, and the 13 bit routing header. The outgoing address is then transformed according to these entries:

$$\texttt{ooor rrxx xxxx xxxx xxxx xxxx xxxx xxxx}_2$$

Not all of the omnibus masters use all of the $32\,\mathrm{bit}$ of the address. For example, the PPU instruction

fetch master does not use the upper three bits, and their value cannot be controlled from a program running on the PPU. By programming the appropriate entries in the lookup table, it is possible to generate any $32\,\mathrm{bit}$ address using the retarget block. The bit ranges used for the rewrite index and the filled bits for the different omnibus masters are given by:

source	fill bit range	rewrite index bit range
PPU instruction	31:29	28:26
PPU data	29:26	25:23
JTAG		30:28
Hazard3	31:29	28:26

Table 8: Bit ranges of the address that are used to determine the target of an omnibus transaction. The rewrite bits are used as an index into a lookup table containing the target mesh node while the fill bits are overwritten according to the entry in the lookup table.

The fill bits were chosen such that all $32\,\mathrm{bit}$ addresses can be generated by every omnibus master. JTAG can already control every bit of the address, so it does not need any fill bits. The PPU instruction master and the Hazard3 omnibus transaction can not control the uppermost three bits, so these can be controlled using the retarget block. Finally, the PPU data master cannot control the uppermost six bits, but as the addresses have a $128\,\mathrm{bit}$ granularity, the two top-most bits get remove in subsequent blocks, so for these no fill bits are necessary. The rewrite bits were chosen to use the highest three bits that are controllable by the different masters, allowing each to create transactions targeting 8 different targets without needing to change the entries of the lookup table.

The commands are next processed by a reorder buffer with a capacity of c that attaches an ID to read commands. These IDs are generated sequentially on $\mathbb{Z}/c\mathbb{Z}$. Internally, the reorder buffer has a buffer with enough capacity for the read response data of up to c commands. For the $32\,\mathrm{bit}$ wide PPU instruction fetch master, each command generates a single $32\,\mathrm{bit}$ response, so the ROB has space for c 32 bit responses. In contrast to that, a single PPU data load and store transaction generates two $64\,\mathrm{bit}$ responses, so the ROB for them has space for $2c$ 64 bit responses. The ID attached to the read response that is received by the read reorder buffer is used as an index into its buffer to store the read response data, and the contents of the buffer are forwarded sequentially in order of the buffer entries back as responses to the omnibus master. As outlined above, it is not permissible for the system to generate backpressure for the read response data stream, as this could cause read response data that has to be forwarded to the omnibus master earlier to become stuck behind read response data that has to be forwarded later. Therefore, the reorder buffer only allows read commands to progress to the next step if enough entries in its buffer are free for the response data of the command to be possible to receive without backpressure. The maximum number of read transactions that can be in flight is therefore given by c. The following values for c were used for the different omnibus masters:

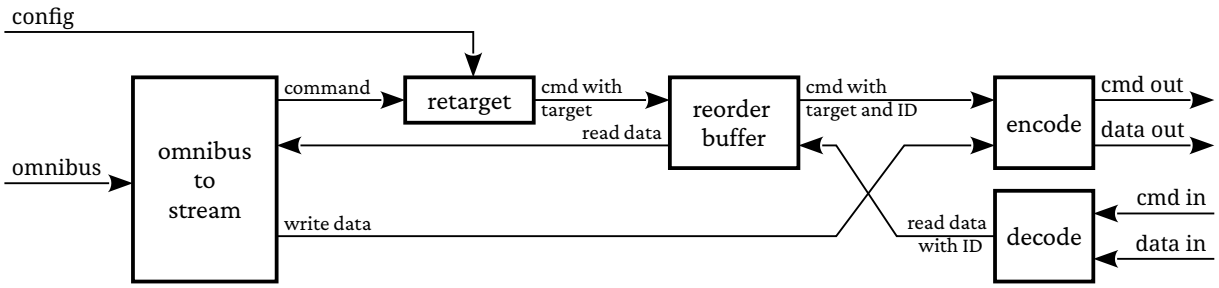

Figure 31: Block diagram of the hardware that converts omnibus transactions to the network protocol. omnibus is split into three separate streams of commands, write data, and response data. This step is omitted for the omnibus masters located on the BrainScaleS-2 ASIC as they are already transmitted to the BRISCET as these three separate streams. The command addresses are used to determine the mesh node that the transaction should be sent to according to the scheme described above. All commands enter the reorder buffer, which allocates buffer space for the response data caused by the commands. Only commands that have buffer space available are allowed to progress to the encoding step, which combines the commands with the write data. This encoding step moves the byte enable information from the command stream to the data stream. The two resulting streams are then processed further and later injected into the router. Incoming streams of commands and data contain the read response IDs and read response data, which are combined into a single stream by the decoding step before being forwarded to the reorder buffer. The reorder buffer reorders the read transactions according to the ID before returning it to the omnibus master in the order the read transactions were issued.

master	c	data width [bit]	responses per command
PPU instruction	16	32	1
PPU data	8	64	2
JTAG	4	32	1
Hazard3	1	32	1

Table 9: Size c of the reorder buffer for read response data for the different omnibus masters.

For the PPU omnibus masters, c was chosen to match the maximum number of transactions in flight supported by the BrainScaleS-2 ASIC. The AHB used by the Hazard3 microcontroller only supports a single outstanding transaction, so $c = 1$ was selected. The chosen values for c furthermore limit the number of bits used by the ID to $4\,\mathrm{bit}$ of the eight available.

Finally, the commands and write payloads are transformed by an encoding step. Here, the address of the commands is converted into an address with $32\,\mathrm{bit}$ granularity, and the byte enable data is removed from the command and added to the write payload data. Furthermore, for read transactions, payload data containing the byte enable bits are generated from the byte enables included in the read commands.

Incoming commands carrying the ID and payloads carrying the read response data are combined into a single stream of read response data that includes the ID by the decoding block and forwarded to the read response buffer. The six different omnibus masters on each BRISCET have to share the two local input ports of the router. Figure 32 shows a schematic diagram of how this is done. First, each of the six pairs of command and payload streams — one for each of the omnibus masters — is split according to the targeted virtual channel ID of the transactions (as determined by the retarget

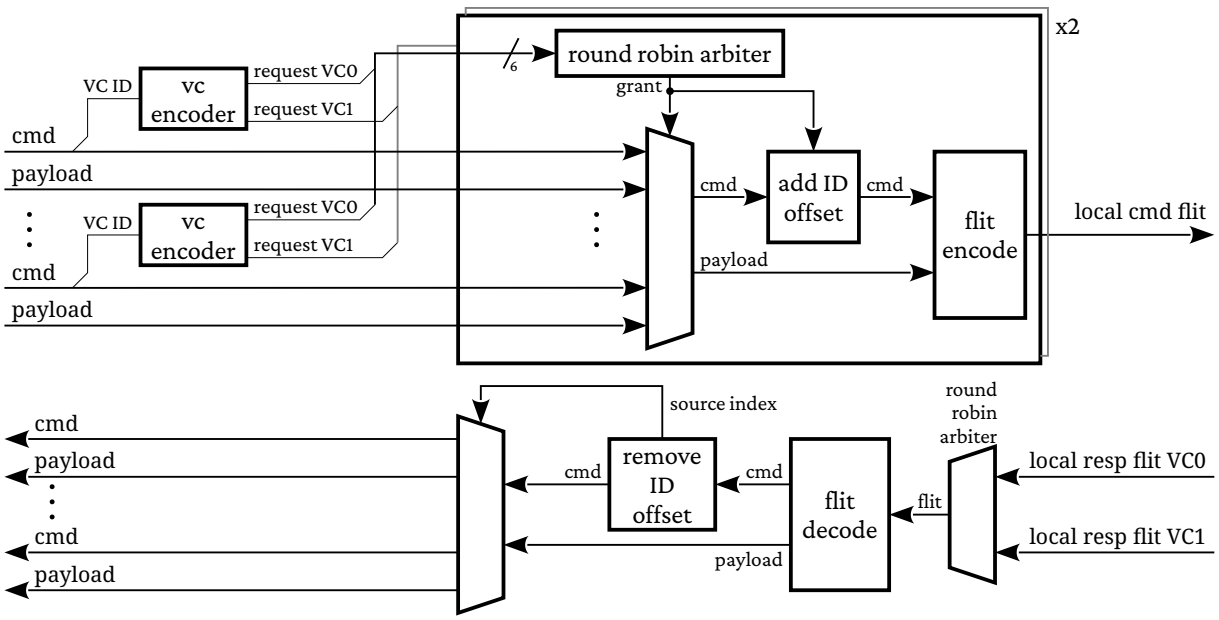

Figure 32: Block-level diagram of the arbitration between the multiple omnibus masters to the local input channels of the router. Each omnibus master is converted to a pair of command and data streams as described in figure 31. Each incoming pair is split according to the target virtual channel. Then, for each channel, round-robin arbitration determines the omnibus master that gets forwarded to the flit encoding step. This encoding step creates a packet in the form of a sequence of flits which get forwarded to the router. For read transactions, the index of the omnibus master that created the transaction gets encoded into the id field of the packet. Incoming flits containing response data from the two virtual channels of the router are combined into a single stream using round-robin arbitration before passing through a flit decoder that creates a separate command stream containing the response ID and a payload stream containing the read response data. The response data is forwarded to the correct omnibus master using the index of the master that caused the read transaction encoded into the id.

step described previously). Separately for each of the two virtual channels, a round-robin arbiter determines which stream gets forwarded to the corresponding local input port of the router. Write transactions are passed on unmodified, but again special attention has to be paid to read transactions. The read responses that are caused by the read transactions need to be forwarded to the omnibus masters that caused them. Again, as the read responses can arrive in a different order than the read transactions were sent, the only information that identifies the read responses is the ID they include. Therefore, for read transactions, the ID has to be modified to be able to identify the omnibus master that the corresponding read transaction was emitted by. Up to four bits are used by the read reorder buffers, which keeps the four most significant bits unused. Three of the unused bits are used here to encode the index of the omnibus master that a read transaction originated from. Finally, the command and the payload data are encoded into a sequence of flits which are forwarded to the local input channel of the router. Incoming flits containing the read response data get decoded into a stream of read response data and commands containing the response ID. The three bits used to encode the index of the master that created the read transaction are then used to forward the response data to the correct master. The omnibus master driven by the JTAG interface is furthermore handled as a special case, where the highest address bit selects between the transaction being forwarded to the

local omnibus tree without going through the tunneling process. This is used to initialize the retarget entries and other configuration.

Incoming flits containing read or write transactions are processed separately and get translated to transactions on either the local omnibus tree or the omnibus tree of the BrainScaleS-2 ASIC accessible via its high-speed interface. The flits for each virtual channel are processed separately by a separate omnibus transactor.

4.7. BRISCET FPGA interface

The FPGA interface of BRISCET has three responsibilities. First, it should allow events to be sent to and received from the BRISCET. In addition to that, it should be possible for nodes in the system to send omnibus transactions to the FPGA. This is, for example, used to extend the memory available to the PPUs. Finally, the FPGA also needs to be able to act as an omnibus master to the system, targeting any component of the system to read and write configuration data. For the general architecture, the same architecture that was developed to connect the BrainScaleS-2 ASIC with an FPGA was reused, described in Karasenko (2020).The scheme for tunneling events across this link between the FPGA and the BRISCET is kept identical to the scheme employed by the high-speed interface of the BrainScaleS-2 ASIC. To allow the FPGA to use the protocol used for tunneling omnibus transactions outlined previously with full flexibility, the interface between the FPGA and the BRISCET is chosen to directly tunnel flits. To allow the FPGA to utilize both input channels, each flit is furthermore given an additional bit that indicates the target virtual channel ID. In total, there are four different sources and sinks for flits on a given BRISCET node. Sources are flits received from the FPGA, flits containing read or write transactions generated by the local omnibus masters, flits containing read response data generated by local omnibus slaves, and finally flits that exit the router. These flits can have four different targets. They can target the FPGA, the local omnibus slaves, the local omnibus masters (when they contain response data), and the router, if they have a target different from the local node. Figure 33 shows how these sources and sinks are connected together. For packets that are sent by the FPGA, there are two cases. A packet can either contain read or write transactions coming from the FPGA or read response data that was generated by the FPGA in response to a read transaction it received. If an FPGA generates a read transaction, at some point the local BRISCET node will receive a packet containing read response data for this transaction. As this read response data is identified purely by the ID it forwards from the read transaction that it was generated in response to, it is necessary to modify the ID of read transactions received from the FPGA to identify the read responses for transactions generated by the FPGA. The six omnibus masters local to each BRISCET node only use 7 of the 8 bit available in the ID field, so here the highest bit is set for any read transactions coming from the FPGA. Therefore, there are 128 unique IDs that can be used by the FPGA, allowing up to 128 concurrent read transactions to be issued by the FPGA. For each of the packet sources, the sink that the packet has to be transmitted to is determined from three components. First, the target coordinate of a packet. Packets with a target coordinate that does not match the coordinate of the local node are sent to the router. Packets that target the local node are processed

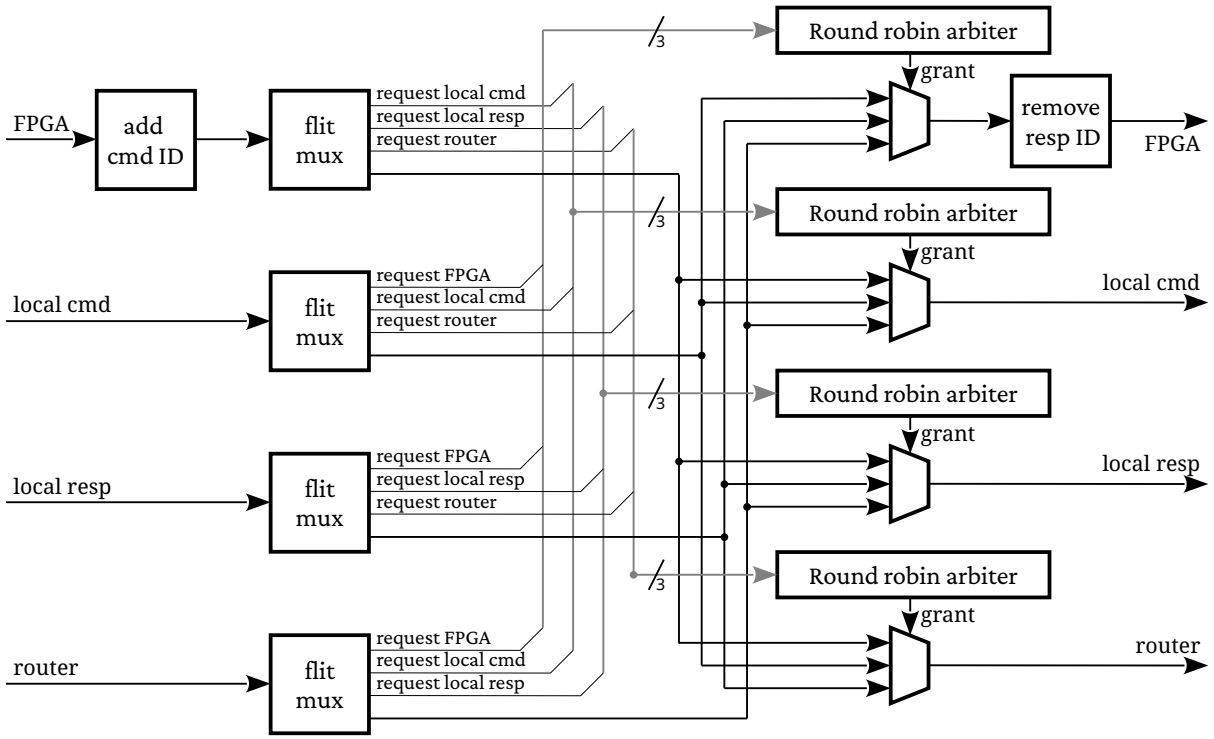

Figure 33: Block diagram showing the architecture employed to route the different local sources of flits to their local destination on a node. For every node, flits can come from the FPGA, from the tunneling of local omnibus masters, from the local omnibus slaves or from the router. According to their target, they are forwarded to four destinations. Flits targeting a different node are forwarded to the router, flits targeting the local node are forwarded either to the local omnibus masters, if they contain response data, or to the local omnibus slaves, if they contain read or write transactions. Finally, the ID of read transactions coming from the FPGA is modified to allow identification of response data to read transactions created by the FPGA.

locally. These packets can contain either read / write transactions or read response data. The target of read and write transactions is determined from their base address. Transactions that have the highest bit of the address set target the FPGA and are forwarded to the FPGA. The remaining transactions are forwarded to the local omnibus slaves. For read response data, the target is determined from the ID. If the highest bit is set, the read response data is forwarded to the FPGA after clearing the highest bit. All other read response data is forwarded to the local omnibus masters. Arbitration between the multiple possible inputs for each of the sinks is performed in round-robin fashion on a packet basis. Each BRISCET node contains two separate instances of the shown diagram, one for each virtual channel to allow two separate flows to progress independently.

4.8. RISC-V microcontroller

To perform local control tasks, like link training of the chip-to-chip links, the BRISCET ASIC integrates a RISC-V microprocessor. The RISC-V microprocessor design chosen for this is the open-source Hazard3 (Wren, 2019) design. It has a three-stage pipelined, in-order architecture, is silicon-proven and can achieve 3.84 coremark / MHz. For comparison, the PPU used on the BrainScaleS-2 ASIC achieves a coremark score of 0.75 coremark / MHz. Furthermore, it supports clock gating to

reduce the power consumption when idle. The Hazard3 CPU core uses a 32 bit AHB (AMBA, 2021a) bus to access external memory for instructions and data. To integrate the Hazard3 CPU core, a bus bridge between the AHB bus and omnibus was developed. The CPU core is furthermore coupled with a memory of 4 kbit and a software interrupt device conforming to the RISC-V SWI device specification (Favor and Patel, 2022) that is accessible via omnibus was implemented. This memory can be initialized via omnibus and contains the program data that is executed upon reset by the CPU.

4.9. Clock synchronization

Several further components are needed to complete the BRISCET design. As described in section 4.1, events that are transmitted to the BrainScaleS-2 ASIC are combined with a timestamp by the sender, BRISCET in this case, and the receiving side uses these timestamps to sort the events according to their timestamps. For this to work, BRISCET and BrainScaleS-2 ASIC have to have a synchronized clock relative to which the timestamps of the events are interpreted. By passing both BRISCET and the BrainScaleS-2 ASIC the same reference clock, it can be ensured that both have clocks that run at the same speed. Nonetheless, the starting point of these clocks has to be synchronized. To facilitate this, the BrainScaleS-2 ASIC accepts two messages: first, a **`systime init`** message, that causes it to reset the local clock to a predefined value and respond with an acknowledgement **`systime`** message containing this reset value. Second, a **`systime read`** message that also causes an acknowledgement containing the current clock value to be returned, but without resetting the local clock.

For the FPGA-BrainScaleS-2 ASIC systems, the clock on the FPGA and the BrainScaleS-2 ASIC clock are synchronized by the FPGA sending a **`systime init`** message to the BrainScaleS-2 ASIC, which causes it to reset the clock to a predefined value and to transmit back a **`systime`** acknowledgement message containing the reset value of the clock. By measuring the time that elapses between the transmission of the **`systime init`** message and the reception of the **`systime`** acknowledgement, the FPGA can determine the round-trip time and reset its local clock to the received reset value of the clock offset by half of the round-trip time. The round-trip time between the FPGA and the BrainScaleS-2 ASIC can vary in practice, for example depending on the link congestion. This can lead to this clock synchronization being imperfect if the **`systime init`** message takes a different amount of time to travel from the FPGA to the BrainScaleS-2 ASIC compared to the acknowledgement message.

For the clock synchronization between the BRISCET and the BrainScaleS-2 ASIC, this scheme is therefore adapted slightly. In a first step, by sending multiple **`systime init`** messages and measuring the time that elapses until the corresponding acknowledgement message is sent, an average value for the round-trip time is measured. In a second step, by sending multiple **`systime read`** messages and recording the difference between the local clock and the returned clock value, an average value for the offset between the clock on the BRISCET ASIC and the BrainScaleS-2 ASIC can be measured. The expected value for this offset is half of the average round-trip time measured before. If the offset differs from this expected value, finally the clock running on the BRISCET is offset by the difference between half of the round-trip time and the measured average offset. This scheme is implemented as

a combination of a hardware block, that has an omnibus slave interface which can be used to measure a single round-trip time, a single difference between the local and the BrainScaleS-2 ASIC clock, and can be used to add an offset to the local clock, and controlling software that calculates the averages and the necessary offset to the local clock. This software can for example be run on the Hazard3 CPU core. Note, finally, that for a given system, the round-trip time between the BRISCET and the BrainScaleS-2 ASIC is not expected to change, so the value for this can be reused from experiment to experiment. The same clock synchronization scheme is also used between the BRISCET ASIC and the FPGA.

Finally, events are transmitted from BRISCET-to-BRISCET without the use of timestamps, so no clock synchronization is necessary between them. If an event is routed from a BRISCET to an attached FPGA to add them to the trace data, they get a timestamp attached by the high-speed link interface used between the BRISCET ASIC and the FPGA. Note that by configuring the event routing appropriately and measuring the round-trip time for an event on the FPGA, the round-trip times between two BRISCET ASICs can still be determined, without dedicated support for clock synchronization messages.

4.10. omnibus-accessible JTAG driver

After reset, the high-speed link of the BrainScaleS-2 ASIC is not active, but instead an initialization sequence is necessary before it can be used. This initialization sequence is performed via the JTAG interface. The FPGA design includes a JTAG driver that is controlled by UT (Karasenko, 2020) encoded messages for this purpose. In systems using the BRISCET ASIC, the JTAG interface of the BrainScaleS-2 ASIC is connected to the BRISCET ASIC instead. The JTAG driver used by the FPGA is therefore adapted for usage as part of the BRISCET ASIC to have an omnibus-accessible interface.

4.11. Dummy data generators

Finally, the BRISCET ASIC includes two dummy data generators that can be used to measure performance aspects of the system. The first is a dummy data generator that is already included in the FPGA design for the `loopback` messages that can be sent to the BrainScaleS-2 ASIC.

The second is a dummy data generator for omnibus transactions. It shares the omnibus transaction to flit packets conversion with the JTAG omnibus master. This dummy data generator generates data according to five parameters:

base address: target address of the transactions

address increment: number of words the address is incremented by for every transaction.

mode: zero for read transactions and one for write transactions

count: number of transactions to generate

wait: number of clock cycles to wait between transactions

For the set of transactions this dummy data generator generates, it measures the time it takes for all transactions to be submitted, the time it takes to receive the first read response, and the time it takes to receive the last read response.

5. Verification

The verification of the BRISCET ASIC focused on three goals:

- Verification of the correctness of the implementation of the different components
- Verification of the performance of the interconnection network
- Demonstration of the operation of a scaled-up BrainScaleS-2 system

These three different goals were approached with three different strategies. The first goal was achieved with formal verification of the fundamental building blocks of the BRISCET ASIC. Section 5.1 starts with a background on formal verification and then describes how the correct operation of some of the more complex building block of BRISCET were verified using this technique.

To verify the performance of the interconnection network, a dedicated simulation and analysis framework was built for this thesis. A description of this framework and how it is also used to aid in debugging the interconnection network architecture is given in section 5.2.

Finally, section 5.3 describes a joint simulation of two BRISCET ASICs and two BrainScaleS-2 ASICs demonstrating a minimal scaled-up BrainScaleS-2 system.

5.1. Formal verification

Formal verification is a method that uses automated theorem proving techniques to prove adherence of a system to a set of properties. Applied to digital design, formal verification can be used to prove properties of a system described by an HDL.

Extensive formal verification was used in this thesis to verify the correctness of many of the individual components. In this chapter, first a background on formal verification and the formal verification sublanguage of SystemVerilog — SystemVerilog Assertions — is given, followed by a background on formal verification of FIFO-like systems. Afterwards, the formal verification strategy that was developed in this thesis for some of the more complex components — the crossbar, the ARQ protocol implementation, and the reorder buffer — is described. Many more of the components developed in this thesis were verified using formal verification. For example, for modules with stream-based interfaces, this includes adherence to the stream handshaking rules described in section 3.3. These were omitted for brevity. All formal verification was performed using the Cadence Jasper Gold tool.

Formal properties can be classified into two categories: *safety* properties and *liveness* properties. The proof of a *safety* property guarantees that a *bad* state is unreachable, while *liveness* properties verify that a *good* state is reachable.

In this thesis, the properties for formal verification were written in SystemVerilog using the SystemVerilog Assertions sublanguage. The SystemVerilog Assertions sublanguage has two fundamental building blocks. The first are *sequences*, which are used to describe sequential behavior. The simplest sequence is a single boolean expression. Such a sequence *matches* at a given point in time if the boolean

expression is true at that point in time. More complex sequences can be built, among others, from concatenations and repetitions. For example, the sequence:

```
A ##1 B[+]
```

matches whenever the boolean expression `A` is true and in the following clock cycles the boolean expression `B` is true for at least a single clock cycle. Note that matches for a sequence can be overlapping. In the given example, if first `A` is true and then `B` is true for two clock cycles, the sequence matches twice: once for the first time `B` is true and once for the second clock cycle `B` is true.

The second component is *properties*. Properties are used as arguments to **`assert`** (to specify properties that *should* always hold) and **`assume`** (to specify properties that can be *assumed* to always hold). The simplest properties are *sequence properties*. Let `S` be a sequence. Then:

```
SP: assert property(S)
```

specifies a sequence property `SP`. For each clock cycle, an evaluation of property `SP` is started. The property evaluates to true if and only if it is impossible for the design to enter a state that makes it impossible for the design to match `S` starting at the evaluation point at any point in the future. Furthermore, a commonly used property type is implications. An (overlapping) implication can be written as:

```
IP: assert property(S |-> P)
```

Here `S` is again a sequence and `P` is a property. Again, an evaluation of this property is started at every clock cycle. For a given starting point, as determined by the evaluation point of the property, the sequence `S` can have zero or more matches. If there are zero matches, the property evaluates (vacuously) to true. If there are more than zero matches, for each match `P` is evaluated starting at the same clock cycle as the match. The evaluation of `IP` is then true if and only if `P` evaluates to true for every match. Note that the evaluations of `P` can overlap, they happen in parallel and can be thought of as independent threads. The same is true for the evaluation attempts of `IP` that are started every clock cycle. In other words, implications are used to precondition the evaluation of a property to cases where an antecedent sequence matches. For a more detailed description of the SystemVerilog Assertions sublanguage, refer to chapter 16 of (IEEE, 2018).

Finally, as an example, the rules governing the AXI-Stream handshake can be described in SystemVerilog Assertions with the following two *safety* properties:

```
valid_stable: assert property(valid && !ready |=> $stable(valid));
payload_stable: assert property(valid && !ready |=> $stable(payload));
```

The **valid_stable** property asserts that whenever **valid** is asserted, it stays asserted until **ready** was asserted (Ref 2.2.1 of AMBA (2021b)) and the **data_stable** property asserts that **payload** does not change until a handshake occurs.

For many stream-based systems, certain *liveness* properties are also desirable. For example:

```
1  eventually_ready: assert property(valid |-> s_eventually ready);
```

If **eventually_ready** holds for a system, there is no sequence of inputs that can cause **ready** to never be asserted while **valid** is asserted, or in other words, it is not possible for the system to enter a state that will cause it to stop processing data.

Commonly used in formal verification are concepts called free and rigid variables. Free variables are additional inputs to the system that are unconstrained. This means that when a free variable is used in a property or as an input to the system, for a property to be proven to hold, it must hold for any possible sequence of values of the free variables. Rigid variables are free variables that are constrained to not change from clock cycle to clock cycle. For a property that uses a rigid variable to be proven to hold, this means it must hold for any possible value of the rigid variable. A rigid variable **v** can be written in SystemVerilog Assertions as:

```
1  assume property(@(posedge clk) $stable(v));
```

Finally, simplifications like reduction of data widths or buffer sizes are commonly used to reduce the runtime of a formal verification proof.

5.1.1. Formal verification of FIFOs

Many, especially stream-oriented, systems can be described as a FIFO, or FIFO-like. In this context, FIFO-like means that data words leave the system in the same order as they enter the system and data words do not get lost or duplicated. FIFOs, especially large ones, pose a challenge to formal verification due to their large state space. A naive way to formally verify a FIFO is the comparison of the FIFO outputs with a reference FIFO model being fed the same inputs. However, this technique is not feasible for larger FIFOs due to an exponential increase in state space. Different techniques have been suggested to allow formal verification of FIFO-like components nonetheless. In this thesis, two were used. The first is a transaction counting-based method proposed in (Darbari, 2019). A SystemVerilog implementation of it can, for example, look like:

```
module fifo_checker #(parameter type DATA_T = logic) (
    input wire clk,
    input wire rst_n,
    input wire input_en,
    input DATA_T input_data,

    input wire output_en,
    input DATA_T output_data,
);
    DATA_T symbolic_data;
    symbolic_data_stable: assume property($stable(symbolic_data));
    wire trigger;

    logic sampled_input;
    logic sampled_output;

    let incr = input_en && !sampled_input;
    let decr = output_en && !sampled_output;

    let next_sampled_input = incr && trigger && (symbolic_data == input_data);
    let next_sampled_output = decr && sampled_input && (counter == 1);

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            sampled_input <= 0;
            sampled_output <= 0;
            counter <= 0;
         end else begin
             sampled_input <= sampled_input || next_sampled_input;
             sampled_output <= sampled_output || next_sampled_output;
             counter <= counter + incr - decr;
         end
    end

  data_matches: assert property(next_sampled_output |-> output_data ==
  ↪  symbolic_data);
endmodule
```

Listing 6: SystemVerilog implementation of the formal verification method for FIFOs described in Darbari (2019). A rigid variable **symbolic_data** determines a single data word which gets tracked as it travels through the FIFO. The number of data words inside the FIFO before the tracked data word is counted by **counter**. When no data word is ahead anymore and a next data word exits the FIFO, the output is asserted to match the tracked data word.

The core idea is to track how a single arbitrary data word chosen by the rigid variable **symbolic_data** moves from the input to the output of the FIFO. **counter** counts how many data words are ahead of this tracked data word and have not yet left the FIFO. When no other data word is ahead anymore, the expected value of **output_data** is exactly **symbolic_data**. Finally, the free variable **trigger** is used to extend the data word tracked by the counter from the first occurrence of an **input_data** matching the **symbolic_data** to an arbitrary occurrence. A formal proof of the **data_matches**

property is guaranteed to hold for arbitrary **symbolic_data** and sequences of **trigger**, verifying the integrity of any data word through the FIFO. Of note is that this technique to verify a FIFO does not catch all input-output mismatches of a FIFO in simulation, because in simulation the free and rigid variables only get assigned a single value. Finally, this method can be extended to also verify that data cannot get stuck in the FIFO once it entered by adding the liveness assertion **will_output**:

```
will_output: assert property(@(posedge clk) sampled_input |->
    s_eventually sampled_output);
```

The second technique can only be used for FIFO-like systems that are data independent, meaning that the behavior does not depend on the value of the data. For this class of systems, it is possible to reduce the width of the checked data to a single bit, as proposed by (Wolper, 1986). Proving the data integrity for all input sequences of bits that can be described using the regex `0*110*` is sufficient to prove the data integrity for all possible input sequences. **wolper_state_machine** is a module that implements a state machine that checks a sequence of bits for matching this regex and enters the **ERROR** state if it does not:

```
typedef enum { START, ONE, END, ERROR } state_t;
module wolper_state_machine (
    input wire clk, rst_n, input_en, input_data,
);
    state_t state, next_state;

    always_ff @(posedge clk or negedge rst_n) begin
        if (~rst_n) state <= START;
        else state <= next_state;
    end

    always_comb begin
        next_state = state;
        if (input_en) begin
            unique0 case (state)
                START: if (input_data) next_state = ONE;
                ONE: next_state = input_data ? END : ERROR;
                END: next_state = input_data ? ERROR : END;
            endcase
        end
    end
endmodule
```

Listing 7: SystemVerilog module implementing a state machine that accepts words of the form `0*110*`. If the input word does not match, it enters the **ERROR** state.

Verifying a FIFO using this technique is split into two steps. The state machine is instantiated for the input data stream, which are free variables. The assumption **input_color_assumption** restricts the input data stream to sequences that match the `0*110*` regex. This process of restricting

the values of free variables is also called *coloring*. Finally, the output data stream is connected to a second instantiation of the state machine and the assertion **`output_color_check`** that the output data stream also matches the regex. In SystemVerilog, this could be written as:

```
module fifo_checker_wolper(
    input wire clk, rst_n,
  input wire input_en, input_data,
  input wire output_en, output_data,
);
    wolper_state_machine input_color(
        .clk, .rst_n, .input_en, .input_data
    );

    wolper_state_machine output_check(
        .clk, .rst_n, .input_en(output_en), .input_data(output_data)
    );

    input_color_assumption: assume property (input_color.state != ERROR);
    output_color_check: assert property (output_check.state != ERROR);
end
endmodule
```

Listing 8: SystemVerilog module implementing a formal verification check for a FIFO, as described by Wolper (1986). The input is constrained to follow the `0*110*` pattern, and the output is asserted to follow the same.

This again can be extended to prove that once a data word enters the system, it will eventually leave the system by adding the following two liveness properties:

```
output_one_one_live: assert property(
  (input_color.state == ONE) |->
    s_eventually (output_check.state == ONE)
);
output_two_ones_live: assert property(
  (input_color.state == END) |->
    s_eventually (output_check.state == END));
```

Listing 9: Additional liveness properties describing a FIFO, where data cannot get stuck using the technique by Wolper (1986) for formal verification of a FIFO. Any state entered by the input state machine has to be eventually reached by the output state machine. If not, a data word entered the FIFO, but never exited it.

5.1.2. Packet crossbar

For the crossbar used in the router, three properties were verified:

`prop_cb_fifo`: flits entering the crossbar on one port should leave the crossbar with FIFO-like properties, potentially interleaved with flits from other input ports, but not internally reordered and without lost or duplicated flits.

prop_cb_live: If a flit has entered the crossbar, it will exit it at some point.

prop_cb_fair: If an input port has valid data, it will eventually be accepted by the crossbar.

The crossbar has five inputs (one input for each mesh port and one for the local port) and six outputs (one for each mesh port and one for each local virtual channel). In general, data from any of the inputs can flow to any of the outputs of the crossbar, according to the target output port that an input selects. The approach chosen for formal verification is to track a single possible of these flows, from one specific input port to one specific output port, selected by rigid variables:

```
int selected_input, selected_output;
selected_input: assume property ($stable(selected_input) && (selected_input <
↪  INPUT_PORT_COUNT));
selected_output: assume property ($stable(selected_output) && (selected_output <
↪  OUTPUT_PORT_COUNT));
```

Listing 10: Helper rigid variables used for the formal verification of the crossbar. A single flow as chosen by the **selected_input** and the **selected_output** is checked.

prop_cb_fair can then be verified by verifying FIFO-like properties for this flow. The flits belonging to this checked flow are all flits entering the crossbar on the selected input and with a target equal to the selected output:

```
let interesting = inputs[selected_input].p.target == selected_output;
assign checked_input_valid = inputs[selected_input].valid & interesting;
assign checked_input_ready = inputs[selected_input].ready & interesting;
```

Listing 11: The input data words for the flow that is verified are given by all data words that enters the crossbar on the selected input port and target the selected output port.

Determining which flits exiting the crossbar belong to the checked flow needs additional information, as flits from different inputs (and not only the selected input) can exit the crossbar on the selected output. The approach chosen here is to *color* a subset of the payload data of the flits according to the port they enter the crossbar on:

```
for (genvar i = 0; i < INPUT_PORT_COUNT; i++) begin
  color_input: assume property (inputs[i].payload.flit[2:0] == i);
end
```

Listing 12: Helper assumptions used to *color* (mark) the input payload to the crossbar, so that the input port for each payload that exits the crossbar can be determined.

This is permissible, as the crossbar implementation is data independent with regard to the flit payload. (Note it is not data independent with regard to the targets, so this has to be kept unconstrained

for the formal verification). The flits exiting the crossbar belonging to the checked flow can then be determined from this coloring of the payload and the port they exit on:

```
let interesting = outputs[selected_input].p.flit[2:0] == selected_input;
assign checked_output_valid = outputs[selected_output].valid & interesting;
assign checked_output_ready = outputs[selected_output].ready & interesting;
```

Listing 13: The payloads exiting the crossbar belonging to the checked flow are given by all payloads exiting the crossbar on the selected port and that entered the port on the selected input port, which is determined from the coloring of the payload described above.

The FIFO-like properties of the checked flow can then be verified using the transaction counting-based method described in section 5.1.1. In the same way, `prop_cb_live` is verified by verifying them for the checked flow using the liveness properties described in section 5.1.1. Due to the optimized nature of the crossbar, where the diagonal elements are not connected, it is necessary to restrict `prop_cb_live` to non-diagonal flows:

```
let not_diagonal = (selected_input != selected_output);
cb_live: assert property(
  not_diagonal && checked_input_valid |-> s_eventually checked_input_ready);
```

Listing 14: Liveness assertion that verifies that if the checked flow has valid data, it can enter the crossbar at some point. This is restricted to non-diagonal (data entering the crossbar on the same port it wants to exit at) flows, as diagonal flows are not connected in the crossbar.

5.1.3. ARQ implementation

For the implementation of the ARQ protocol, the following properties were identified for formal verification:

prop_arq_fifo: The ARQ sender and ARQ receiver pair should be FIFO-like, even in the presence of a lossy link.

prop_arq_live: If a data word enters the ARQ sender, it will eventually leave the ARQ receiver.

prop_arq_nodata: If no data words are outstanding, the sender should eventually stop sending data.

prop_arq_noack: If no data words are outstanding, the receiver should eventually stop sending acknowledgements.

prop_arq_throughput: If the window size is large enough compared to the link latency, it should be possible to achieve $100\,\%$ throughput.

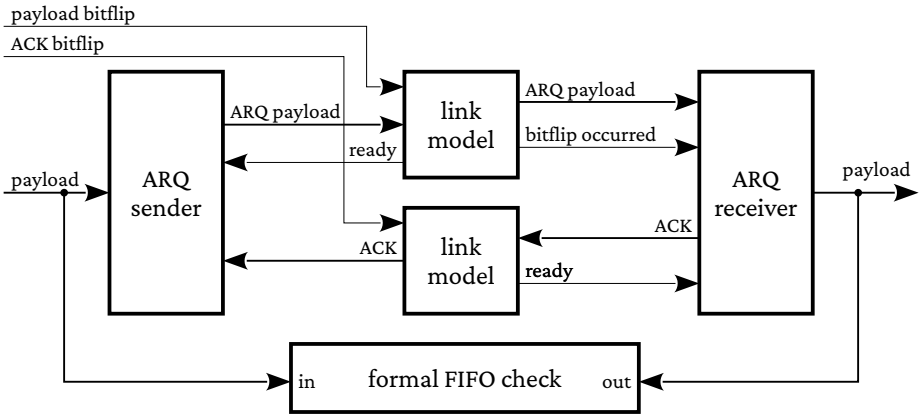

Figure 34: Schematic overview of the testbench used for formal verification of the ARQ protocol implementation. The link models a fixed latency link, which can arbitrarily lose or corrupt data controlled by the free variables **`payload bitflip`** and **`ACK bitflip`**. For the formal verification of the FIFO property, again the method described by Wolper (1986) is used.

The link between the ARQ sender and receiver is modelled as a blocking fixed latency lossy link. Both, the occurrence of a link error and the backpressure from the link are free variables. To reduce the complexity, on the receiving side, the occurrence of a link error is not detected as in the real design using a checksum, but instead directly determined for the occurrence of a link error. Figure 34 shows a schematic overview of the setup used for the formal verification.

The ARQ sender and receiver are data independent. Their control logic does not depend on the content of the input data, so **`prop_arq_fifo`** is verified using the wolper coloring technique described in section 5.1.1 and the data width of ARQ sender and receiver was reduced to a single bit. The properties **`prop_arq_live`**, **`prop_arq_nodata`** and **`prop_arq_noack`** can be specified using a counter of the outstanding data words to be transmitted:

```
int outstanding;
always_ff @(posedge clk or posedge rst) begin
    if(rst) outstanding <= 0;
    else outstanding <= outstanding + (in.valid && in.ready) - (out.valid &&
    ↪  out.ready);
end

no_unneccessary_traffic_data: assert property (
    (always (outstanding == 0)) implies
        (s_eventually (always !sender_to_link.valid)));
no_unneccessary_traffic_ack: assert property (
    (always (outstanding == 0)) implies
        (s_eventually (always !ack_from_receiver.trigger)));
eventually_received: assert property (
    outstanding > 0 |-> s_eventually out.valid;
```

Listing 15: Properties used to verify that if all data that entered the ARQ sender exited the ARQ receiver, at some point the sender will no longer send data and the receiver will at some point stop sending acknowledgements. Furthermore, an assertion that verifies that data cannot get stuck and never be received is added.

For these properties to hold, the error behavior of the link cannot be arbitrary, however. For example, consider the case where the link never transmits any word without error. Clearly, **prop_arq_live** does not hold then. Instead, some fairness assumptions have to be made about the error behavior of the link. Here, the assumption that any specific data word that gets transmitted infinitely often is transmitted by the link without error at some point is used. This assumption should also hold in the real world for errors that are not correlated to the content of the data that is transmitted by the link. In SystemVerilog Assertions this is described as:

```
for (genvar i = 0; i < 2 * window_size; i++) begin
    let want_to_send_next = sender_to_link.valid && (sender_to_link.payload.seq
    ↪  == i);
    let can_send_next = sender_to_link.ready && !tx_link.in_error;

    link_error_is_fair: assume property(
        (always s_eventually want_to_send_next)
            implies s_eventually (want_to_send_next && can_send_next));
end
ack_link_is_fair: assume property(
    (always s_eventually ack_receiver_to_link.valid)
        implies s_eventually (ack_receiver_to_link.valid &&
        ↪  ack_receiver_to_link.ready && !ack_link.in_error));
```

Listing 16: Assumptions used to restrict the error behavior of the link. Any payload that gets sent infinitely often is at some point transmitted without error. This is necessary for the ARQ protocol to be able to make forward progress. The same is assumed for the acknowledgements sent by the receiver.

link_error_is_fair guarantees that if a packet with any specific sequence number is sent infinitely often, it will eventually be sent without error. **ack_link_is_fair** guarantees that if the receiver tries to send acknowledgements infinitely often, the acknowledgements will eventually be sent without error. With these additional assumptions, all properties were proven for two specific cases:

1. A link delay of 4 and a window size of 2. This case has a link delay that is bigger than the window size. This means it is expected that **prop_arq_throughput** does not hold.

2. A link delay of 1 and a window size of 4. In this case, the window size is bigger than the round trip time. Here, **prop_arq_throughput** does hold.

5.1.4. Read response reorder buffer

For the read response reorder buffer (ROB), the following properties were formally verified:

prop_rob_fifo: The path from command input to command output is FIFO-like, with the only modification of the output command being that it contains an ID.

prop_rob_rdata_ready: The read response data input is always able to accept response data.

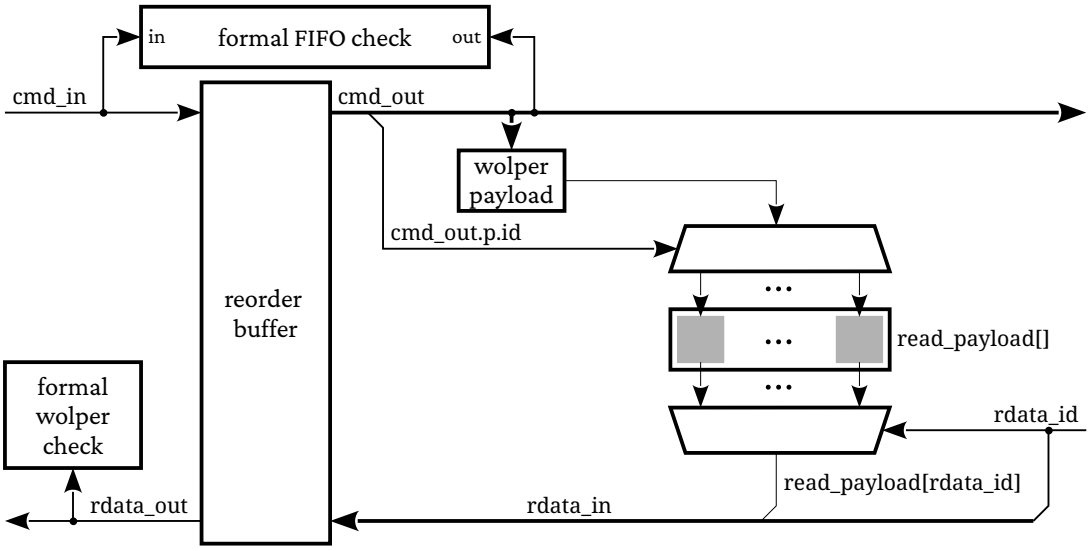

Figure 35: Overview of the formal verification testbench used to verify the reorder buffer. The commands entering and exiting the reorder buffer are checked using the method described by Darbari (2019). The correct behavior of the reorder buffer regarding the reordering of the response data in the order of the read commands is verified using a method based on Wolper (1986). It uses *wolper coloring* to prepare response data in the order of the commands, that follow the wolper coloring. The prepared response data is returned to the reorder buffer in arbitrary order determined by the free variable **`rdata_id`**. The verification is completed by verifying that the response data returned from the reorder buffer fulfills the wolper coloring.

`prop_rob_rdata_fifo:` The read response data input is reordered so that the read response output order matches the command output order.

Figure 35 shows a schematic overview of the testbench developed to formally verify these properties. To verify **`prop_rob_fifo`**, the transaction-counting based method described in section section 5.1.1 is used. Here, the wolper coloring technique cannot be used as the command input is not data independent. The ROB processes read commands and write commands differently. Property **`prop_rob_rdata_ready`** can be expressed as:

```
1 rdata_always_ready: assert property(@(posedge clk) always rdata_in.ready);
```

Finally, **`prop_rob_rdata_fifo`** is more complicated to express. It can be reduced to verifying a FIFO-like property between the sorted (according to the command IDs) response data input and the response data output. This FIFO-like property can be verified using the wolper coloring technique, as the behavior of the ROB is data independent with regard to the response data payload. So the approach chosen here is to fill an array of possible response data according to the commands emitted from the ROB. The response data is generated from free variables, restricted to obey the wolper coloring rules. Because the ROB can be configured to expect more than one read response per command, this coloring has to be unrolled multiple times in one clock cycle:

```
typedef enum { START, ONE, END, ERROR } wolper_state_t;
wolper_state_t input_state, next_input_state;

logic read_payload[num_in_flight][words_per_cmd],
 ↪  next_read_payload[num_in_flight][words_per_cmd];
logic read_input[words_per_cmd];

always_comb begin
    if (cmd_out.valid && cmd_out.ready && cmd_out.p.is_read) begin
        for (int i = 0; i < words_per_cmd; i++) begin
            next_read_payload[cmd_out.p.id][i] = read_input[i];
            next_input_state = next_wolper_state(next_input_state,
             ↪  read_input[i]);
        end
    end
end
rdata_wolper_color: assume property (next_input_state != ERROR);
```

Listing 17: Preparation of the **read_payload** array according to wolper coloring from the read commands that exit the reorder buffer. Each command produces **words_per_cmd** response data words, so the wolper coloring state machine is unrolled **words_per_cmd** per cycle. The code was simplified for brevity.

The read response data transmitted to the ROB is then chosen from the populated array by the free variable **rdata_id** used as index into it. Finally, the response can be arbitrarily delayed according to the free variable **delay**. If the ROB is configured to expect multiple read responses per command, they have to be in order, so **read_offset** is incremented accordingly and **rdata_id** is constrained to only change once the last read response for a command was transmitted:

```
wire delay;
int read_offset;
assign rdata_in.p.id = rdata_id;
assign rdata_in.p.data = read_payload[rdata_in.p.id][read_offset];
always_ff @(posedge clk or negedge rst_n) begin
    if (rdata_in.valid && rdata_in.ready) begin
        if (read_offset == (words_per_cmd - 1)) begin
            read_offset <= 0;
        end else begin
            read_offset <= read_offset + 1;
        end
    end
end
rdata_id_stable: assume property(
    rdata_in.p.valid && (!rdata_in.p.ready || (read_offset != (words_per_cmd -
    ↪ 1)))
    |=> $stable(rdata_id));
```

Listing 18: Read responses prepared in the **read_payload** array are returned to the reorder buffer in an arbitrary order determined by the free variable **rdata_id**. An arbitrary delay in the return of response data is introduced by the free variable **delay**. Code was simplified for brevity to omit the handling of the valid signal for the stream interface of the ROB.

Finally, the FIFO-like property of the response data can be verified by asserting that the response data follows the wolper coloring scheme.

```
always_ff @(posedge clk or negedge rst_n) begin
    if (~rst_n) begin
        output_state   <= START;
    end else begin
        if (rdata_out.valid && rdata_out.ready) begin
            output_state <= next_wolper_state(output_state, rdata_out.p);
        end
    end
end
read_resp_data_check: assert property(output_state != ERROR);
```

Listing 19: The read response data returned by the reorder buffer is verified to match the wolper coloring to verify its correct order and that no data words getting get lost or duplicated.

The properties were verified for two different configurations of the reorder buffer:

- **num_in_flight = 4** and **words_per_cmd = 1**
- **num_in_flight = 4** and **words_per_cmd = 2**

5.2. Interconnection Network Verification

Not every design and design property is suited to formal verification. This can have different reasons. For example, formal verification might be infeasible due to the size of the design, or the properties that should be verified are unsuited for expression in a formal framework. This might, for example, include properties related to performance metrics, especially aggregated performance metrics over many clock cycles. To verify the performance of the interconnection network, therefore, a different strategy was employed and a dedicated simulator and accompanying analysis tooling was developed. Interconnection networks can be modeled at different layers depending on the simulation performance and precision required. The approach chosen in this thesis is a simulation that directly uses the HDL implementation of the different components of the interconnection network like the ARQ protocol and the routing logic while using more generic models for components like the chip-to-chip links. This reduces the effort required to build a simulation of the interconnection network while also providing a very precise model of the behavior of a hardware realization. In the next section, the simulation part of this framework is introduced. This is followed by a description of different interactive graphical visualizations developed to aid in understanding the behavior of the interconnection network. Finally, the performance of the interconnection network is analyzed in section 5.2.4.

5.2.1. CXXRTL-based simulation framework

To faithfully reproduce the behavior of a fabricated (digital) hardware design, the simulation has to model details of the hardware implementation of the described circuit like propagation delays and metastability. For many cases, however, simulation to this detail is not necessary. The class of cycle-based simulators ignore the timing information of cells to achieve higher simulation performance. Examples of cycle-based simulators include Verilator (Snyder, 2003) or CXXRTL (whitequark, 2019). Verilator reports a speed-up factor over commercial event-based simulators of 6.0 to 11.1 (Snyder, 2020).

In this thesis, CXXRTL was used. It is part of the open-source synthesis toolchain Yosys (Wolf, 2013) and translates a digital design into a cycle-accurate C++ model providing an interface to modify the input ports, read the value of the output ports as well as advancing the state of the design according to the values of the input ports. For example, the SystemVerilog module

```
module A(input clk, rst_n, a, output b);
// module contents
endmodule
```

is translated to a C++ model by CXXRTL as:

```
struct p_A : public module {
  value<1> p_b;
  value<1> p_a;
  value<1> p_rst__n;
  value<1> p_clk;

  size_t step();

  void debug_eval();
  void debug_info(debug_items *, debug_scopes *);
};
```

Listing 20: Example for the interface of the C++ model created by CXXRTL.

Access to the ports of the design is provided by the variables prefixed with **`p_`**. The **`step`** function updates the values of internal registers as well as output ports according to the values of the input ports. Finally, **`debug_info`** can be used to discover registers, memories and wires internal to the design and their values can be calculated on demand using **`debug_eval`**. As outlined above, cycle-based simulation offers performance advantages over event-based simulation, so for the simulation CXXRTL was used to create a cycle-accurate model of a mesh node. Included in this model were the ARQ implementation, the flow control implementation as well as the non-event flit-based routing. The input and output ports of this mesh node are the local input and output ports for flits, as well as the mesh ports carrying the ARQ payload. Using this mesh node model, a C++ simulation framework was developed that connects multiple of these models together. By using a higher-order programming language like C++, compared to writing the simulation framework completely in an HDL, a more convenient way to specify the stimulus and evaluate the interconnection network is provided. This is supported by the rich standard library of C++, for example containing random number generators for different standard distributions. Figure 36 shows how two mesh nodes in a 2×1 mesh are connected in this simulation framework. The chip-to-chip links are modelled as a fixed latency link that can have bit errors. As the focus is not on analysis of the checksums used to detect bit flips, the checksums are not modelled, but rather the occurrence of any bit flip is directly forwarded as **`error occurred`** to the node models. For the gearbox and the arbiter between event and non-event data used by the link-level protocol, a simplified approach with priority arbitration of event data over non-event data is used. Finally, the behavior of a simulation is then determined by three exchangeable models:

- The *non-event traffic model*
- The *event traffic model*

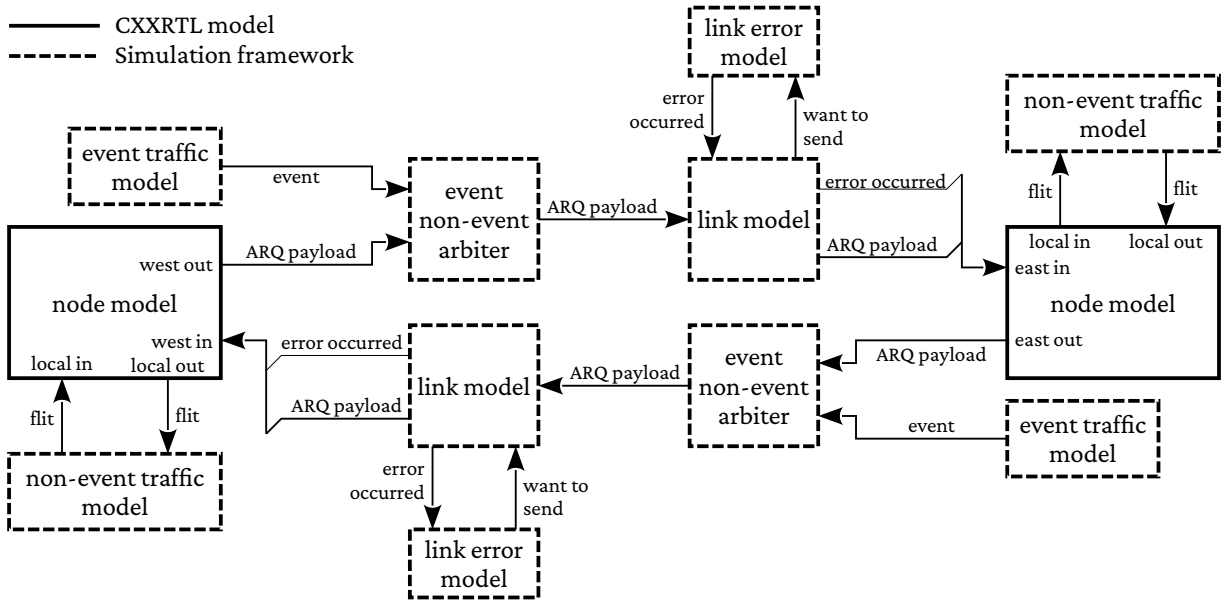

Figure 36: Block diagram of the C++ simulation framework developed for simulations of the interconnection network. A CXXRTL-created cycle-accurate model of the HDL implementation of the ARQ protocol, flow control and non-event message routing is combined with models for the link, the event traffic and non-event traffic patterns that are written in C++

- The *link error model*

For each mesh node, an instance of the non-event traffic model controls the local input and output ports of the mesh node and generates the non-event traffic. For example, a mesh node that tries to send a flit every cycle to the mesh node with the same x coordinate as itself and y coordinate zero can be written as:

```
class Flood {
    NodeInfo i;
    TraceFPGABandwidth(const NodeInfo& node_info) : i(node_info) {}

    bool step() {
        int src_vc = 0;
        routing_target target{
            .target{
                .x{i.x},
                .y{0}
            }
        };
        *i.payload_in[src_vc] = flit{
            flit_start_and_end{
                .target = target,
                .payload{}
            }
        };
        i.payload_in_valid[src_vc]->set(1);
    }
};
```

Listing 21: Example for a C++ model of the non-event message traffic.

Here, the C++ data types such as **`flit`** are automatically generated from the Amaranth HDL data types used in the mesh node implementation as described in section 3.2. The **`step`** function is called every clock cycle by the simulation framework. The **`NodeInfo`** structure passed to the class in the constructor contains, among others, the wires that represent the local inputs to the interconnection network. These are stream interfaces, so **`payload_in`** sets the payload transported over the stream and **`payload_in_valid`** sets the valid signal. This example ignores the ready signal of the stream.

For each chip-to-chip link, the simulation framework uses an independent instance of the event traffic and the link error model. First, the event traffic model determines when events are sent over the chip-to-chip link. For example, a model for poissonian event traffic could be written as:

```
class PoissonEventTraffic {
    xoshiro256pp rng;
    std::poisson_distribution<> dist;
public:
    using Params = PoissonEventTrafficParams;

    PoissonEventTraffic(uint64_t seed, Params params) :
        rng(seed), dist(params.e) {}

    bool did_send() { return dist(rng) > 0; }
};
```

Listing 22: Example for a C++ model of the event message traffic.

Here, **`params.e`** is the expected number of events per clock cycle and **`did_send`** is called by the framework for every clock cycle to determine if an event is sent over the link. Note that further parameters passed to the constructor of the model by the simulation framework such as position and direction of the link are omitted for brevity.

Similar to the event traffic model is the link error model, which is used by the link model to determine which payloads are transmitted over the chip-to-chip link with bit flips.

```
class BinomialErrorModel
{
    xoshiro256pp rng;
    std::biominal_distribution<> dist;
public:
    BinomialErrorModel(uint64_t seed, uint8_t link_bits, Params params) :
        rng(seed), dist(link_bits, params.bit_error_rate) {}

    bool should_error(bool) { return dist(rng) > 0; }
};
```

Listing 23: Example for a C++ model for the link error distribution.

5.2.2. FST support for CXXRTL

The output of a simulation using this simulation framework is a trace of the value of all ports, internal registers, wires, and memories for each clock cycle of all the simulated mesh nodes. This is also called *waveform data*. Furthermore, the different models also add additional variables to the waveform data. For example, the link models add a variable that gets asserted whenever a link error occurs.

To store the waveform data of a design, CXXRTL includes a VCD (IEEE, 2018) writer. Due to the nature of VCD as a text-based, uncompressed format, the resulting file size grows quickly. For example, a simulation with two mesh nodes for 10 000 steps generates a VCD file with a size of 122 MB. This large generated file size can also become a bottleneck during simulation if the storage for the VCD file is not fast enough. The FST format (Bybell, 2008) is an open-source alternative to the VCD format, that offers strong compression, resulting in drastically smaller file sizes. In FST format, the same 10 000 step simulation only needs 2.4 MB, a $\approx 51\times$ reduction. To utilize the FST format with CXXRTL, an alternative to the CXXRTL-provided **`vcd_writer`** that uses the open-source FST implementation (Bybell, 2009) was developed.

5.2.3. Graphical analysis tool for 2D mesh simulations

Traditionally, simulations of hardware designs are analyzed using a waveform viewer. During a simulation, the value of all signals for all clock cycles is captured and a waveform viewer visualizes this trace of values as a timeline. An example of this representation can be seen in figure 37. Analyzing and debugging communication on a 2D mesh using this kind of visualization is cumbersome due to the large amount of data that needs to be considered. For example, every mesh node has 4 bidirectional

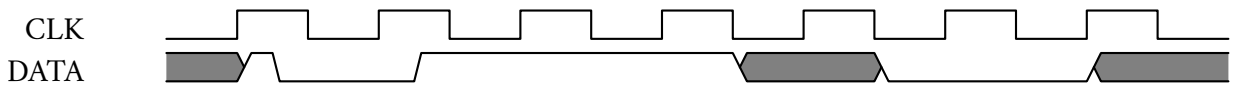

Figure 37: Example of the waveform representation used by waveform viewers.

ports that connect to neighboring mesh nodes, each of these ports has an ARQ sender and receiver, buffers, a credit counting mechanism, and more. This is combined with the potential occurrence of link errors and random event traffic. In addition, to understand performance problems of the interconnection network, a higher-level view of the nodes in the interconnection network is usually helpful. For example the exact contents of different buffers are usually less interesting than the fill state of a buffer. To reduce the difficulty of analysis and debugging, a graphical visualization tool for waveforms of the mesh traffic simulations was developed. The core of this analysis tool is written in C++ using Imgui (Cornut, 2014) to render the user interface. To be useful for design space exploration during which the internal architecture of the mesh nodes can change, the core of the analysis tool is only coupled loosely to the internal architecture of the mesh nodes, and analysis and visualizations that are implementation-specific can be added using a Python interface.

This analysis tool offers three main ways it aids with debugging and analysis. The first is an interactive graphical representation of the mesh nodes and their internal state at a specific point in time in the simulation. This representation is controlled by a Python script to allow changes to the visualization without recompiling the main C++ tool. To generate the graphical representation, a function **`process`** is called for every mesh node that was simulated, with an object that gives access to the node data. For example:

```
def process(node)
    clk_var = node.data.variables["clk"]
    to_send = node.data.variables["packets_to_send"]
    sent = node.data.variables["packets_sent"]

    async def outstanding_hist(node):
        ts_times, ts = await node.read_values(to_send, clk_var)
        s_times, s = await node.read_values(sent, clk_var)
        n.add_hist("outstanding packets", [to_send, sent], ts_times, ts - s)

    imgui.text("packets_sent: " + n.get_current_var_value(sent))
    if imgui.button("outstanding hist"):
        n.enqueue_task(outstanding_hist)
```

Listing 24: Example of a Python extension for the interconnection network analysis tool that controls the graphical representation of a mesh node.

This example draws text containing the current value of the **`packets_sent`** variable. Below that, a button is rendered with the label **`outstanding hist`**. To perform processing in the background (to avoid blocking the graphical user interface), a thread pool is provided by the C++ core that tasks can be submitted to. On the python side, this is exposed by the **`enqueue_task`** that accepts an **`async`**

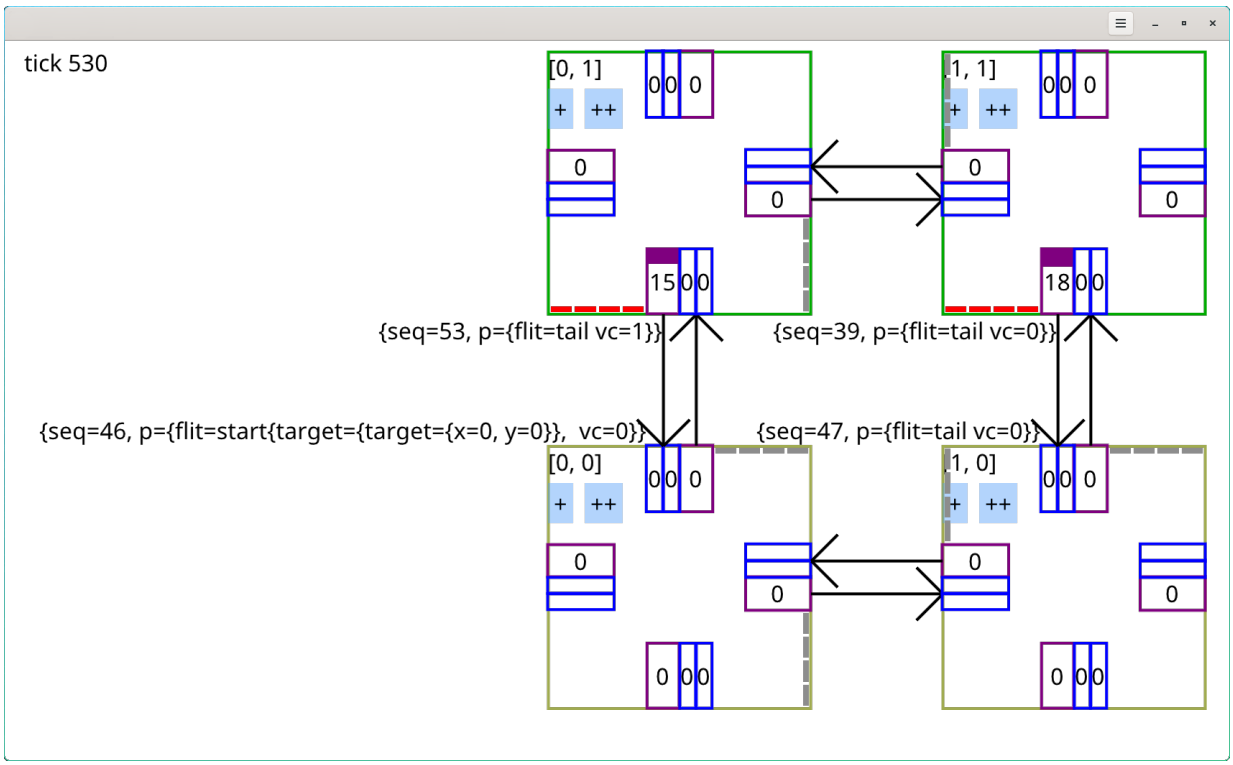

Figure 38: Screenshot of the graphical representation of the mesh nodes rendered by the custom developed analysis tool.

Python function. In this example, when the button is clicked, the Python code submits a task that extracts the values of the variables called **`packets_to_send`** and **`packets_sent`** on each rising edge of **`clk`** from the waveform data. This data is returned to the Python code as numpy (Harris et al., 2020) arrays. Finally, it instructs the C++ core to create a new window containing the histogram of the difference of these values.

An example of how this graphical representation looks with more internal state visualized is shown in figure 38. Here, a simulation containing 4 mesh nodes in a two by two grid was simulated. Mesh nodes that do not have an FPGA connected are represented by green rectangles, while mesh nodes that do by yellow rectangles. For each mesh port, the internal buffer state of the ARQ sender as well as the input queues of the two virtual channels are visualized. In addition, the current incoming and outgoing flits are parsed from their binary representation and displayed in a human-readable form.

To reduce the coupling between the implementation of the graphical representation and the internal implementation of the mesh node design, the mesh node implementation can add metadata to internal signals that gets forwarded to the analysis tool. Internally, this metadata is translated into attributes on the Verilog signals representing the Amaranth HDL signals, which the simulator forwards to the waveform dump, which is read in by the analysis tool. For example, the total capacity of a FIFO could be attached to the signal that contains the current fill level of a FIFO as follows:

```
outstanding = Signal.like(read_ptr)
outstanding.attrs["capacity"] = self.window_size
m.d.comb += outstanding.eq(write_ptr - read_ptr)
```

Metadata attached in this way in the hardware description can be accessed from the Python side of the analysis tool then as follows:

```
outstanding = var(f'north.arq_sender.outstanding')
capacity = outstanding.attrs["capacity"]
```

Listing 25: Example of the metadata mechanism that allows Python code for the analysis to access metadata created by the HDL code.

This same mechanism for metadata is also used to automatically translate the formatting instructions for custom data types on the Amaranth HDL side into formatting instructions that can be used by the analysis tool, which is used in figure 38 to print the contents of the incoming and outgoing flits in a human-readable way.

This visualization aids in understanding the state of the mesh at a specific point in time. To perform analysis of the mesh over time, a histogram view of signals in the design or of custom data using the `add_hist` functionality is provided. This histogram view furthermore allows interactive correlation between the histogram and a waveform view of the values. For example, in figure 39 on the right side, a histogram of the number of outstanding (not acknowledged) flits in the ARQ sender of the north port of mesh node `[1,0]` is shown. In purple, a user-drawn rectangle is shown, that controls which values of the waveform view of the outstanding variable on the left side get highlighted. Here, large values of unacknowledged flits are highlighted, and it is visible that these are correlated with the occurrence of bit errors on the link that this ARQ sender transmits data on.

Flit flow visualization While the previously described visualizations aid in understanding the state and behavior of a mesh at a coarse level, either at a specific point in time or statistically over the whole simulation, they do not help to understand how a single flit travels from source to destination. For a visualization like this to be possible, it is necessary to be able to track when a specific flit enters and exits a mesh node or one of its subcomponents.

This information is generated in two parts. First, additional output ports are added to the mesh node design for each of the subcomponents that flits enter or exit. For each internal port that receives or transmits a flit or a different datatype containing a flit, two toplevel ports are added. The first contains the payload of the flit, and the second a strobe signal that gets asserted whenever a flit enters or exits the component via this port. Additional metadata is added that contains the direction, the port name, and the location of the module in the module hierarchy of these ports. For example, for the first input port of the array of input ports `local_in`:

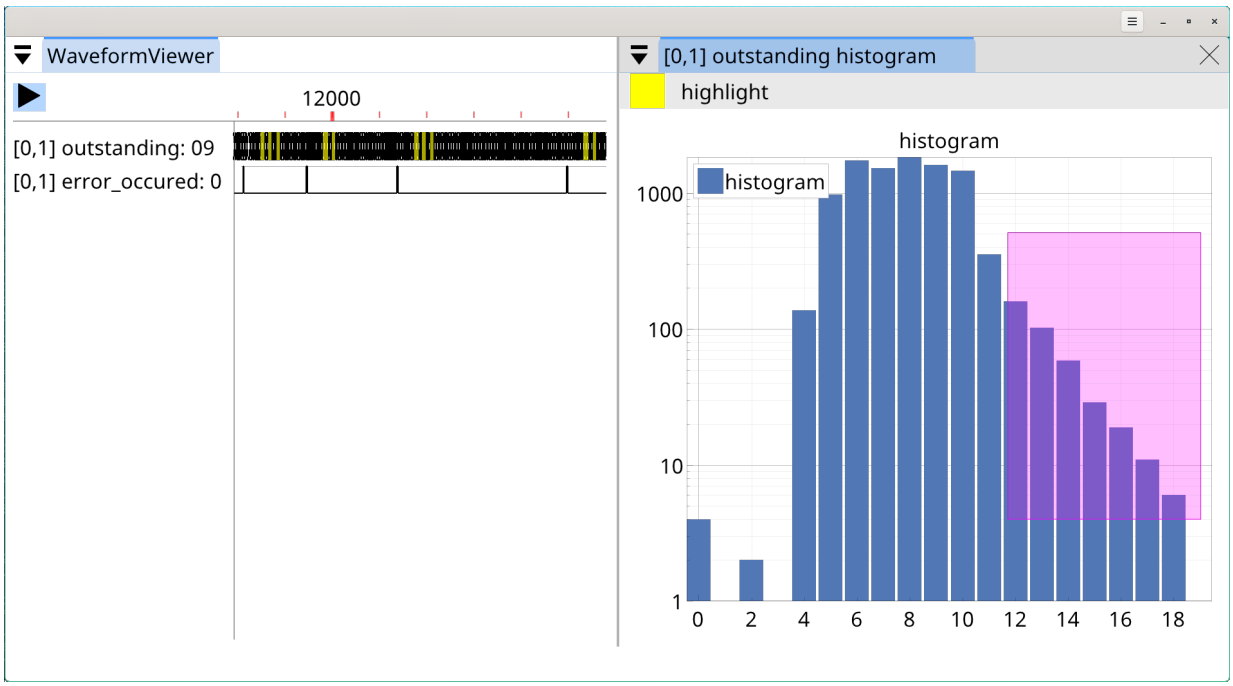

Figure 39: Screenshot of the histogram feature of the custom analysis tool. On the right, the histogram of the number of outstanding (unacknowledged) flits in the ARQ sender of one of the ports of one mesh node. The purple rectangle is a user-specified range of values that determine which values of the outstanding signal get highlighted in the waveform view on the left. In this case, large values are highlighted. It is apparent that these correlate with the occurrence of link errors, which can be seen in the waveform representation on the left side.

```
class RouterTop(Component):
    local_in: In(stream.Signature(Flit)).array(Config.N_VC)
```

the following additional debug ports are generated:

```
(* signal_flow_sample *)
(* module = "[]" *)
(* interface = "[\"local_in\", \"0\", \"payload\"]" *)
(* direction = "in" *)
output wire[31:0] trace_sample_0

(* signal_flow_sample_strobe *)
output wire trace_sample_0_strobe
```

Listing 26: Example of the additional ports automatically created in the Amaranth HDL design of the router to facilitate the tracing of flits through the design.

These additional output ports are automatically generated using the meta-programming capabilities of Amaranth HDL by recursively traversing the design hierarchy and adding additional output ports for any input or output stream of a module that contains a flit as payload, or contains a datatype that has a flit as one of its subcomponents. Furthermore, these additional output ports are only added as part of the steps used to convert the Amaranth HDL implementation of the node into a

CXXRTL model. The Verilog compilation of the Amaranth HDL parts for synthesis does not add these additional ports.

As a second step, the traces generated by a simulation of a system containing mesh nodes with these additional output ports are analyzed by the analysis tool. To be able to track a flit across subcomponents and mesh nodes using the payload and strobe signals, the payload of them has to be chosen uniquely by the simulation stimulus. Here, the sender coordinate and virtual channel ID combined with the current clock cycle is sufficient as such an identifier, as any virtual channel can only accept a single flit per clock cycle.

For a specific port of a specific submodule, the clock cycle that a specific flit enters or exits the submodule through that port is then determined by determining the clock cycle for which the debug output containing the payload of the flit matches the flit of interest and for which the strobe output is asserted. A combination of this information, combined with the metadata containing the direction and location of the submodules in the design hierarchy, can be used to produce a visual summary like figure 40. The visual summary is composed of a timeline that shows the clock cycles on the x-axis and a set of colored rectangles that represent the different submodules the flit travels through. For each rectangle, the starting point on the x-axis is given by the clock cycle the flit entered the submodule and the end point by the time the flit exited the submodule. Note that these rectangles can be overlapping, even for the same submodule. In the given example, the flit enters the `arq_sender` once, but exits it twice, producing two rectangles of different length. (This example shows a flit that encountered a bit flip on the first time it was sent over the link from node `[0,1]` to `[0,0]`). On the y-axis, the rectangles are first sorted according to the mesh node that contains the submodule they represent and secondly by the depth in the hierarchy the submodule is located. Finally, hovering on a rectangle gives more detailed information on the submodule and ports it represents.

Batch analysis The same Python interface to the waveform data and metadata, as well as the nodes, is also provided in a non-graphical batch mode to facilitate automated analysis of simulations. Using this batch analysis interface

5.2.4. Performance Analysis

The bandwidth of the interconnection network for non-event messages under different rates of event messages is analyzed. An important use case for the non-event messages that are transported over the interconnection network are ADC samples written by the PPU to FPGA memory. In a 2D mesh of BRISCET and BrainScaleS-2 ASIC nodes, a single FPGA is shared by multiple BrainScaleS-2 ASICs, typically one row or column of BRISCET and BrainScaleS-2 ASIC shares a single FPGA. The achievable data rate of these ADC samples for different event rates for a different number of BRISCET nodes sharing the same FPGA is investigated using simulations. The non-event messages generated by the PPU writing ADC samples to memory attached to the FPGA are simulated as non-event messages that get periodically injected at a fixed rate. The event messages are simulated as a bernoulli process. Figure 41 shows a schematic overview of the analyzed system. Nodes $[0, 1]$ to $[0, n]$ inject non-event

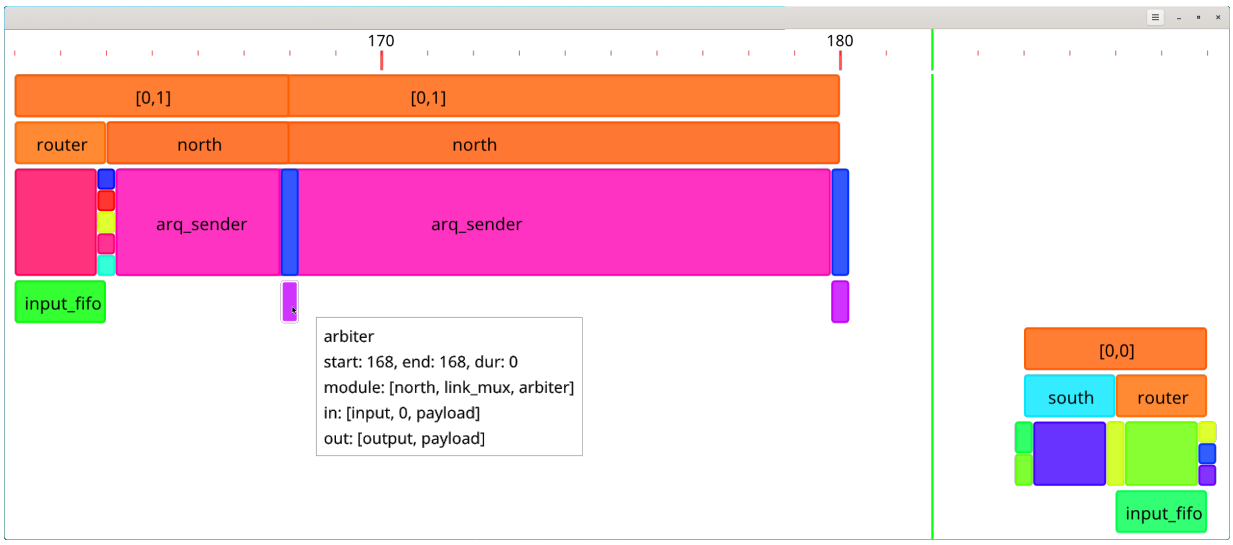

Figure 40: Screenshot of the representation of the flow of a flit across several mesh nodes and through the subcomponents of one mesh node. At the top, a timeline of clock cycles is given. Each colored rectangle represents a submodule that the flit entered and exited. The start time of a rectangle is given by the clock cycle the flit entered the submodule while the end point is given by the clock cycle the flit exits the submodule. Note that these rectangles can be overlapping. In this case, the flit exits the `arq_sender` of node `[0,1]` twice, which in this case is caused by the occurrence of an error during the transmission from node `[0,1]` to `[0,0]`.

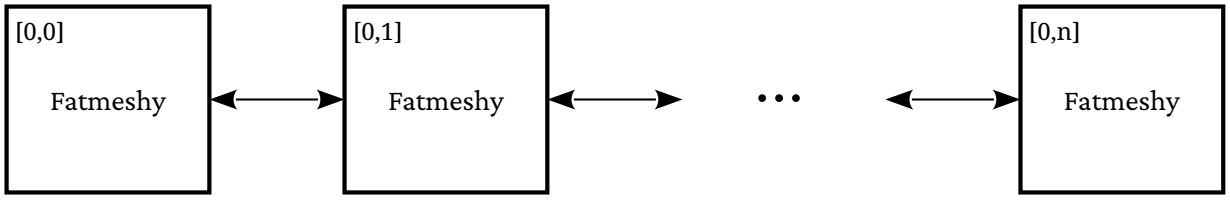

Figure 41: System used to analyze the bandwidth of the interconnection network. For this analysis traffic is emulated that mimics the periodic sampling of the ADC by the PPUs on nodes $[0, 1]$ to $[0, n]$ sending their samples to node $[0, 0]$ under different rates of event traffic.

messages at a rate of p with a target of node $[0, 0]$. As write transactions that get tunneled by these non-event messages do not have an ordering requirement between them, they get injected alternating between the two virtual channels. Each chip-to-chip link furthermore transports event messages at a rate of e injected with a bernoulli process. In this scenario, node $[0, 1]$ receives non-event messages at a rate of $n \cdot p$ with a target of node $[0, 0]$. Figure 42 shows the maximum for the latency for each flit to arrive in this scenario. The x-axis gives $n \cdot p$ as a fraction of the total useful link bandwidth, while the y-axis shows the event rate e as a fraction of the link bandwidth. Cases where the sum of both rates exceeds the link bandwidth are not simulated, as they are impossible to satisfy. In red, parameters are shown where, for any of the nodes, the local input port of the router fails to accept flits at the given rate p.

Each simulation is allowed to reach steady state for $1\,000$ cycles and then simulated for $99\,000$ further cycles during which the maximum latency for a flit is determined. The error rate for the links was simulated at zero and the link latency between sender and receiver was simulated at 27 clock cycles or equivalently $135\,\mathrm{ns}$.

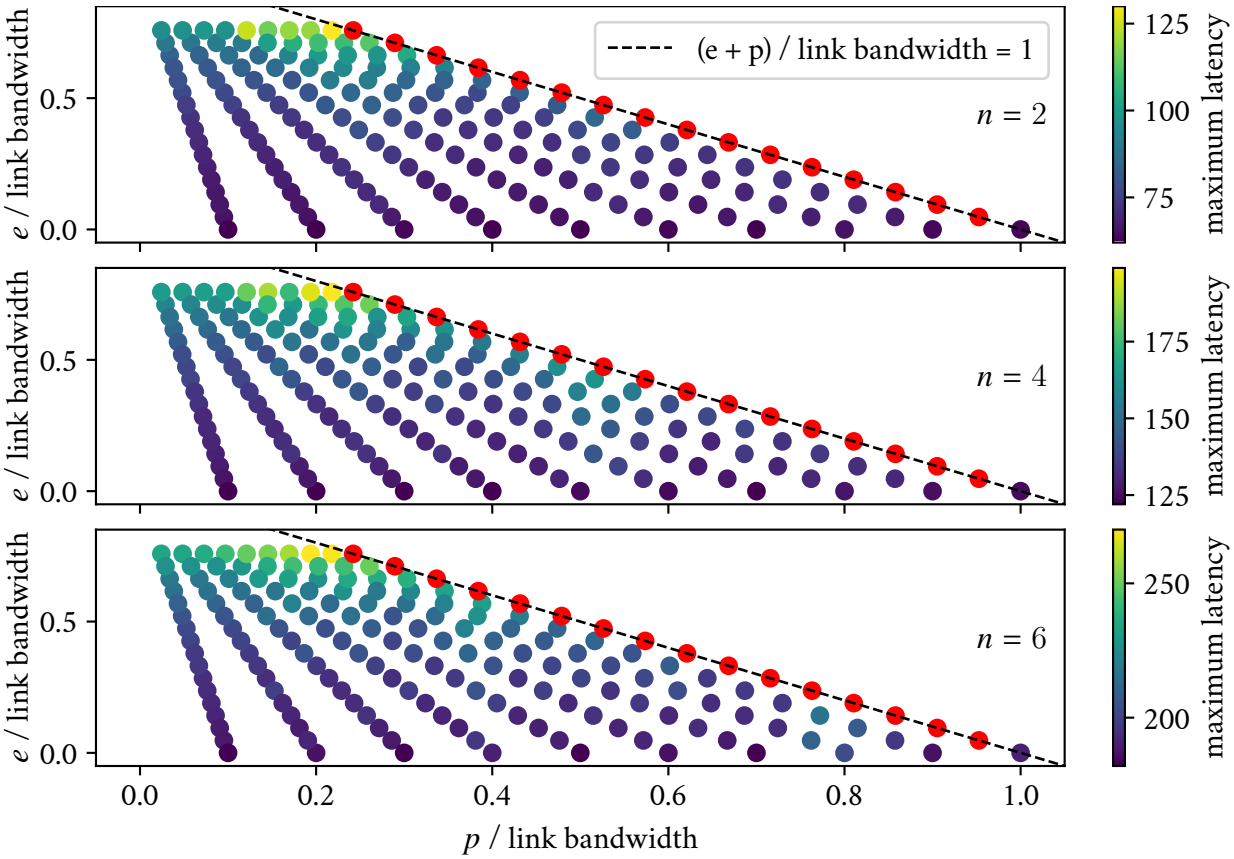

Figure 42: Maximum latency for a flit to reach its target in the system show in figure 41. Here p is the injection rate of non-event messages given in terms of the link bandwidth required by node $[0, 1]$ to send all data to node $[0, 0]$. e gives the rate for the event traffic that is injected as a bernoulli process in terms of the maximum link bandwidth. All links use the same value for e.

The simulations show that the interconnection network is able to handle this traffic pattern at data rates that almost saturate the link bandwidth. Only the cases where $e + p = 1$ fail to accept data on the local port. Furthermore, it shows that the maximum latency for a flit to reach its destination increases with the rate of events transmitted over the link. This is expected, as event messages are transmitted with priority over non-event messages and can therefore prevent non-event messages from progressing.

Analysis of the behavior of the interconnection network under error for this scenario is infeasible for realistic error rates. For a bit error rate of $\rho = 1{\times}10^{-10}/\text{bit}$, the error rate for a flit is $\leq 1{\times}10^{-8}$ flit. On average, a simulation would have to be performed for 1×10^8 steps to observe a single error for a given link. Instead, the upper bounds derived in section 4.3.2 are used to extrapolate simulations performed with a higher error rate to lower error rates. Recall that the failure for these upper bounds was defined as the ARQ sender being unable to accept data at a given rate. The upper bound derived for the MTBF depends on the behavior of the ARQ protocol in the form of the time t_{recover} and number of words n_{recover} necessary to recover to the steady state after an error has occurred.

To analyze the implementation of the proposed ARQ protocol and the interaction with the event message traffic, t_{recover} and n_{recover} are extracted from simulations. This is done by simulating a system with the same setup as before and $n = 1$, so it consists just of two nodes $[0, 1]$ and $[0, 0]$.

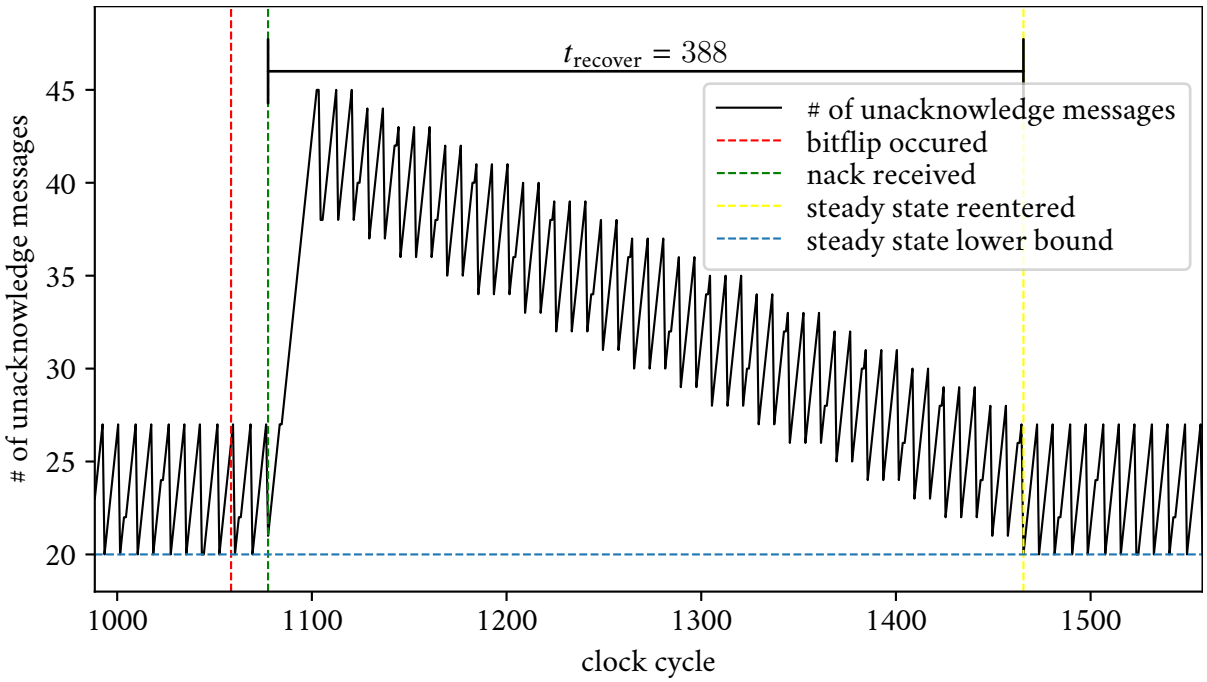

Figure 43: Scheme used to extract the number of messages necessary for the number of unacknowledged messages in the sender of the ARQ buffer to return to the baseline level n_{recover} and the corresponding time required t_{recover} from simulation. The number of unacknowledged messages in the buffer of the sender is drawn in black. t_{recover} is determined as the time required for this number to return to the lower bound of the baseline after a negative acknowledgement is received. n_{recover} is then determined by counting the number of words transmitted over the link in that timespan (not shown here).

Node $[0, 1]$ injects non-event messages at a fixed rate p targeting node $[0, 0]$ and events are injected using a bernoulli process on both link directions connecting the two nodes at a fixed rate e. t_{recover} is then extracted from the changes of unacknowledged messages in response to an error occurring on the link. Figure 43 shows how this is performed for an example simulation. First, the number of unacknowledged messages is determined in the steady state case without any errors. This buffer level varies due to the cumulative acknowledgements. Therefore, the lower bound of it is determined. After a bit flip occurs, a negative acknowledgement is received at some point by the sender. t_{recover} is then determined from the number of cycles necessary for the number of unacknowledged messages to reach the lower bound in the steady state again. n_{recover} is then measured from the number of words sent over the link in this timespan. Figure 44 shows the maximum n_{recover} and corresponding upper bound for the MTBF at a bit error rate of $\rho = 1 \times 10^{-10}/\text{bit}$ that was determined this way for different parameters of p and e across 200 simulations for each point. Again, note that this MTBF is merely a lower bound. The actual MTBF will always be larger. Here, a transmission delay from sender to receiver of 17 clock cycles equivalent to 85 ns was used. The red points show parameter configurations where the buffer of the sender filled up after a single error was injected. Here, the MTBF can be seen to increase with decreasing rate of non-event messages, as predicted by the theoretical model. Furthermore, an increase in event rate also increases n_{recover}. This is expected, as an increase in event message rate effectively decreases the maximum link bandwidth available to non-event messages, due

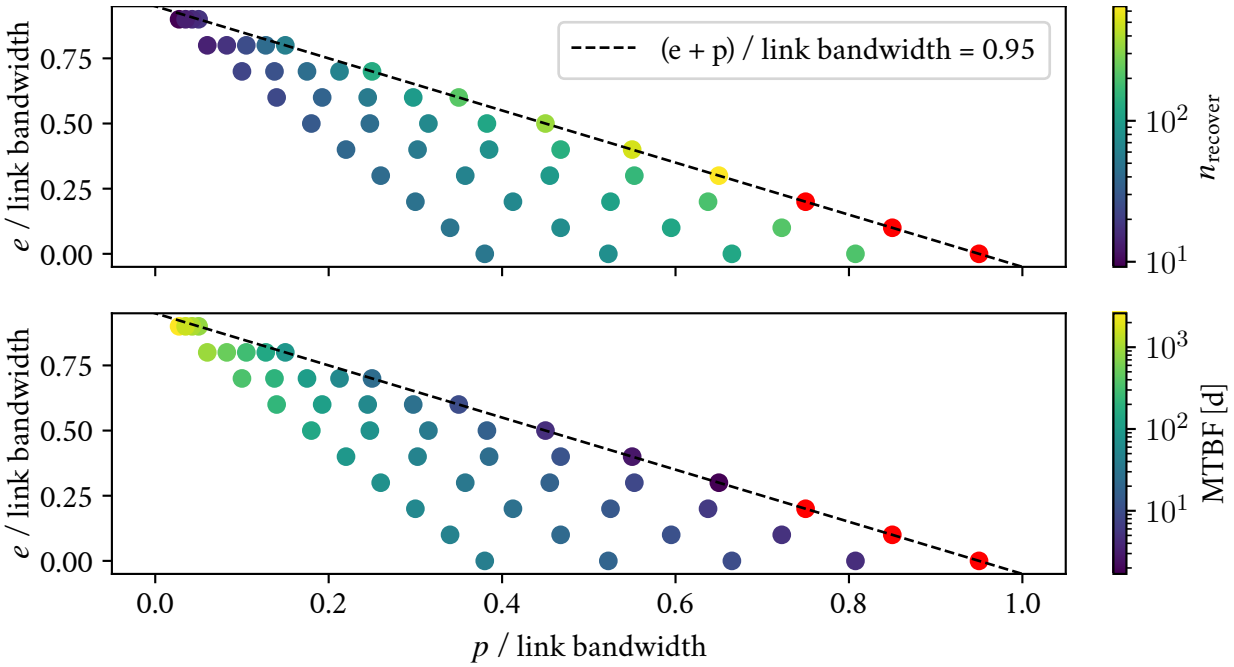

Figure 44: Maximum values of n_{recover} and corresponding MTBF values determined from simulation for different event (e) and non-event (p) message rates. n_{recover} was extracted from the number of unacknowledged non-event messages with the scheme described by figure 43. The MTBF values were calculated from the measured n_{recover} using the lower bound estimation derived in section 4.3.2 for a bit error rate of $\rho = 1 \times 10^{-10}/\text{bit}$. In red, parameters where a single error caused the buffer of the sender to fill up are drawn. Here a minimum round trip time of 85 ns was simulated.

to the prioritization of event messages over non-event messages. Nonetheless, an MTBF greater than 1 d is observed for all simulated cases where a single error does not immediately cause the buffer of the ARQ sender to fill up. The order of magnitude of the MTBF determined this way from simulations furthermore closely matches the purely theoretical estimates derived in section 4.3.2

5.3. End-to-end simulation tests

In addition to the block-level formal verification of the developed hardware design and the simulation-based analysis of the interconnection network, simulations of minimal two-node systems were used for end-to-end tests focused on testing the integration of the various components. As described in Grübl et al. (2020), hardware and software co-development for the BrainScaleS-2 ASIC and the software that controls it is enabled using co-simulation. The hardware design of the BrainScaleS-2 ASIC and the FPGA is simulated in an RTL simulator and connected to the software used to perform experiments on the BrainScaleS-2 ASIC via the SystemVerilog DPI interface. This allows all testbenches to run against simulation and later the fabricated hardware. The setup for this co-simulation was modified from simulating a single pair of BrainScaleS-2 ASICs and FPGAs to simulating a minimal BRISCET-based system containing a single FPGA, two BRISCET and two BrainScaleS-2 ASICs. Figure 45 shows a block diagram of this modified co-simulation setup. The FPGA design was modified to the interface of the BRISCET, which transports omnibus transactions via the flit-based protocol described in section 4.6. This is done by reusing the components developed for the BRISCET ASIC that translate

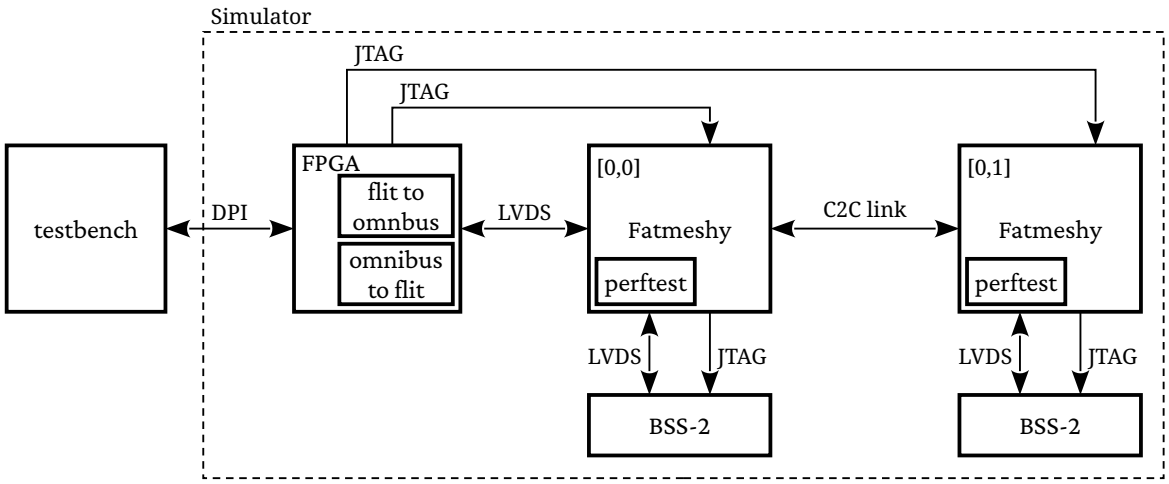

Figure 45: Schematic overview of the end-to-end tests. The testbenches were written using the software stack that is used to operate the BrainScaleS-2 ASIC, which is co-simulated with a system of two BRISCETs, two BrainScaleS-2 ASICs and a single FPGA. This setup reuses the co-simulation framework for the BrainScaleS-2 ASIC described in Grübl et al. (2020).

an omnibus master to the flit-based protocol on the FPGA. Testbenches verifying several system aspects were developed:

- Initialization of BRISCET via JTAG
- Initialization of a BrainScaleS-2 ASIC via the JTAG driver located on BRISCET
- Clock synchronization for the event timestamps between the FPGA and BRISCET
- Tunneling of omnibus transactions by the six different omnibus masters for each BRISCET to different nodes in the mesh.
- Transmission of events from FPGA to BRISCET
- Transmission of events from BRISCET to BrainScaleS-2 ASIC
- Transmission of events from BRISCET to BRISCET
- Simultaneous transmission of event and non-event data between the two BRISCET

An annotated example of one of the testbenches developed for this can be found in appendix B. Furthermore, the achieved bandwidth of different omnibus masters with different transaction targets was measured in the simulation. For reads, an estimate for the steady-state bandwidth is obtained by issuing 100 read transactions and measuring the time that elapses between the reception of the first and the last response. For writes, the steady-state bandwidth is estimated by measuring the time necessary to issue 500 and 1 000 write transactions and using the difference between these two durations as an estimate of the steady-state time necessary to issue 500 write transactions. Using the difference of the two durations counteracts the effects caused by write transactions being buffered on intermediate nodes.

Karasenko (2020) measures the usable bandwidth of the BrainScaleS-2 ASIC high-speed interface as 2.4 Gbit/s when employing four LVDS links. Each 128 bit write transactions by the PPU transmits a total of 177 bit over this interface (128 bit of payload data, 16 bit byte enables, 32 bit for the address and 1 bit for the transaction type) for a total write achievable bandwidth of $\approx$ 1.7 Gbit/s. Read transactions by the PPU are limited to 8 in-flight transactions.

The interface between BRISCET and FPGA reuses the high-speed interface of the BrainScaleS-2 ASIC, but uses payloads that are slightly larger than the ones used in Karasenko (2020). Therefore the useable bandwidth is expected to be slightly more than 2.4 Gbit/s for the four LVDS links employed. 128 bit write transactions are encoded as 3 flits of size 72 bit, leading to an expected usable write bandwidth of $\approx$ 1.4 Gbit/s. 32 bit write transactions are encoded as 2 flits, leading to an expected bandwidth of 0.5 Gbit/s.

The interface between BRISCET and BRISCET has a usable bandwidth of 14.4 Gbit/s. For 128 bit write transactions this yields an upper bound for the possible bandwidth of $\approx$ 8.5 Gbit/s and for 32 bit write transactions 3.2 Gbit/s.

Finally, for local transactions, a single flit can be processed per cycle per input channel, leading to an expected bandwidth of 3.2 Gbit/s.

For read transactions, the achievable bandwidth depends on the number of read transactions that are allowed to be in flight simultaneously and the time required for a response to a read transaction to be received. The simulation does not include a model of the chip-to-chip links, so it is expected for the simulated latency to differ from the actual latency. Therefore, the read bandwidth should merely be interpreted as an upper bound. For the LVDS interface between the BrainScaleS-2 ASIC and BRISCET as well as the BRISCET ASIC and the FPGA, the latency was measured to $\approx$ 950 ns and a resulting upper bound the read bandwidth for PPU vector loads, which support up to 8 128 bit outstanding transactions, of 1.08 Gbit/s.

The bandwidth obtained this way is given by:

source	target	granularity	read bandwidth [Gbit/s]	write bandwidth [Gbit/s]
PPU [0,0]	FPGA	128	0.38	1.65
PPU [0,1]	FPGA	128	0.38	1.59
BRISCET [0,0]	BRISCET [0,0]	32	2.85	3.2
BRISCET [0,0]	BRISCET [0,1]	32	1.02	3.2
BRISCET [0,0]	FPGA [0,1]	32	0.163	0.62

Table 10: Read and write bandwidths obtained in simulation for different sources and targets of transactions in the system shown in figure 45.

The write bandwidths achieved in the end-to-end simulations matche closely the expected bandwidth. As expected for the write bandwidth, the achievable write bandwidth does not depend strongly on the number of hops necessary, as the write bandwidth does not depend on the latency. For the reads originating on the BrainScaleS-2 ASIC, the read bandwidth is also roughly what is expected. The latency is double for the PPU to BRISCET to FPGA path compared to a direct FPGA to BrainScaleS-2 ASIC link and the measured read bandwidth is roughly half the one for a direct system.

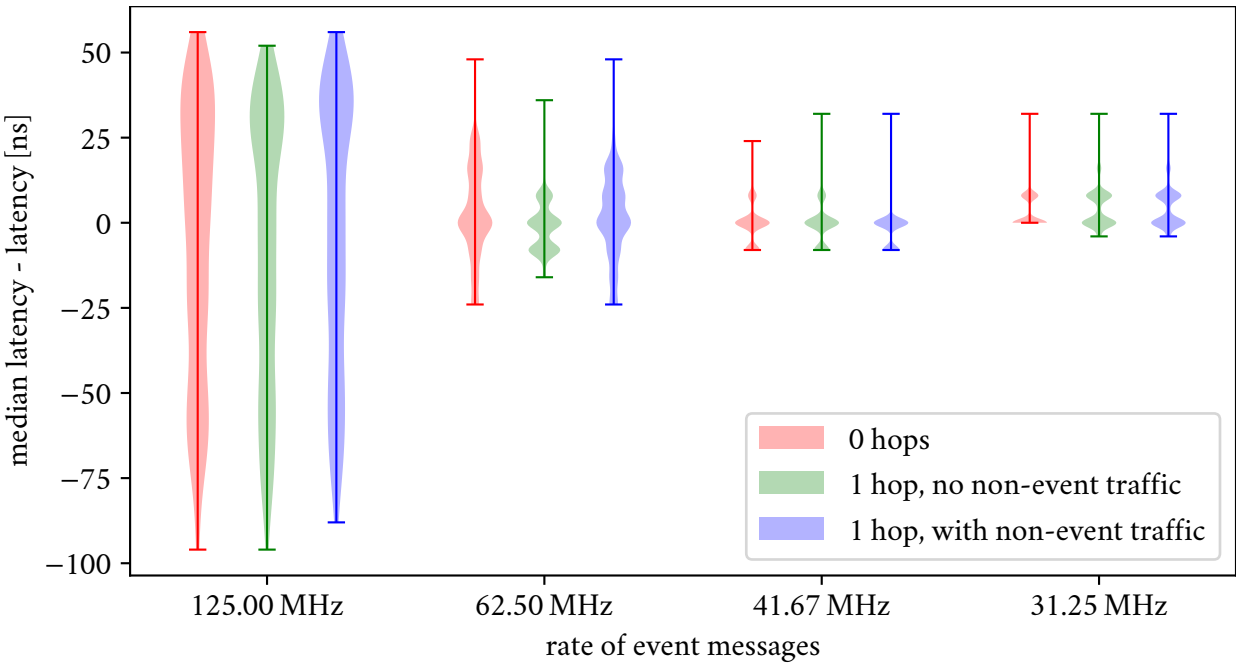

Figure 46: Difference between the median latency for an event to be looped back and the indivdual latencies summarized as a series of violin plots. Three different scenarios were considered. Events coming from the FPGA looped back by the BRISCET node [0,0], events coming from the FPGA looped back by node [0,1] routed via node [0,0] and finally the same scenario, but with additional non-event traffic on the link, utilizing the link at maximum permissible rate.

Finally, to get an initial estimate of the impact on the relative timing on event messages, three different loopback scenarios are simulated for different rates of events with constant spacing between them:

- Events coming from the FPGA being looped back directly to the FPGA by BRISCET node [0,0]
- Event coming from the FPGA are routed to BRISCET node [0,1] by BRISCET node [0,0]. BRISCET node [0,1] which loops them back to BRISCET node [0,0], which forwards them to the FPGA.
- Event coming from the FPGA are routed to BRISCET node [0,1] by BRISCET node [0,0]. BRISCET node [0,1] which loops them back to BRISCET node [0,0], which forwards them to the FPGA. Simultaneously write transactions are initiated by the node [0,0] targeting node [0,1] at the maximum rate they get accepted.

Note, that the routing for the event messages was kept as simple circuit switching in this thesis. A implementation of event-routing for the final scaled-up system will use a more complex routing scheme. Furthermore these simulations simulate the chip-to-chip links as simple wires. Therefore these simulations should be treated as an rough lower bound for the impact on the relative timing of event messages, which can in a real implementation still increase. To estimate the impact on the relative timing, the latency between transmission of an event message and reception of the looped back message was measured. In total for each point, 1 000 spikes were sent. Figure 46 visualizes the

difference between the median latency across all 1 000 event messages and the individual latency for each event message in a series of violin plots. Note, no drop of events was observed during these simulations, which also validates that expected maximum supported event rate of 125 MHz. For the baseline scenario, event message do not travel over the chip-to-chip links. In this case the jitter in the timing is purely determined by the interface between FPGA and the BRISCET ASIC. Rettig (2019) presents a detailed analysis of this jitter at different event rates. Comparing the first scenario with the other two, little to no impact on the spread of spike latencies by the BRISCET-to-BRISCET routing can be observed.

6. Conclusion

In this thesis, the digital design and verification of BRISCET is presented — a routing ASIC that facilitates a scaled-up multi-chip BrainScaleS-2 system.

The architecture for an interconnection network that can transport event messages with real-time requirements while supporting high bandwidth for non-event messages with strong integrity requirements was proposed, implemented, and verified. The core of the interconnection network was implemented in Amaranth HDL, utilizing its strong meta-programming capabilities. The state of the art for integration of Amaranth HDL designs with SystemVerilog was improved by the development of automatic wrapper generators translating Amaranth HDL interfaces and data types to SystemVerilog.

For the verification of the interconnection network, a dedicated simulation and analysis framework with a focus on flexibility was developed. Being extensible using Python scripting and using the introspection capabilities of Amaranth HDL, it allows enabled exploration and interation of the interconnection network architecture. Graphical debugging and analysis tools were developed to aid in understanding the behavior of a simulated interconnection network. Using this simulation framework, the interconnection network was verified to be able to saturate the bandwidth of the chip-to-chip links to $\geq 95\,\%$ for shared event and non-event message patterns, while having an MTBF of $\geq 1\,\mathrm{d}$ for drops in the throughput of non-event messages.

Formal verification was used to prove correct operation of the implementation of critical parts of the design, including the ARQ protocol, the crossbar implementation, and a reorder buffer.

The implementation of the interconnection network was furthermore integrated into a top-level design, that connects to a BrainScaleS-2 ASIC and an FPGA, includes a RISC-V CPU, and will prospectively be taped out in a 65 nm process.

Co-simulation of the BrainScaleS-2 ASIC, FPGA, and BRISCET design with the software used to operate the BrainScaleS-2 ASIC was used to verify the correct operation of all fundamental low-level operations used to operate the BrainScaleS-2 ASIC, including event transport, programs running on the PPU, and configuration of the BrainScaleS-2 ASIC.

Nonetheless, the proposed architecture is limited in several ways. First, the event routing is kept very simple and does not make use of the configurable labels of the events at all. A scaled-up BrainScaleS-2 system will need more sophisticated event routing options. A further limitation is the bandwidth between BrainScaleS-2 ASIC and BRISCET as well as the FPGA that provides a connection to the host computer and is attached to memories often used to hold experiment data like ADC samples. Due to size constraints, BRISCET cannot use all 8 LVDS lanes of the BrainScaleS-2 ASIC, but instead limits itself to 4, which reduces the maximum possible event rate to half of the maximum theoretically possible by the high-speed interface of the BrainScaleS-2 ASIC and to $\approx 62.5\,\%$ of the lossless rate reported by Karasenko (2020) for the off-chip direction. The bandwidth for non-event data is in turn $\approx 65\,\%$ of the theoretically possible. This same limitation of bandwidth is also present for the connection to the FPGA, which has a raw link rate of 4 Gbit/s compared to the 22 Gbit/s that each BRISCET has. A major usage of the high bandwidth from a BrainScaleS-2 ASIC to an FPGA are surrogate-gradient-based experiments, like those described by Cramer et al. (2022), which can saturate

the bandwidth between the BrainScaleS-2 ASIC and an FPGA in current single-chip deployments. In a mesh-like deployment, the bandwidth between FPGA and BRISCET ASIC will be shared by mulitple PPUs, reducing the available bandwidth per PPU. However, the bandwidth between the FPGA and the host computer is just 1 Gbit/s, which is dramatically lower than the bandwidth between a BRISCET ASIC and an FPGA. Therefore, this limited bandwidth is not expected to be a major limitation for a scaled-up BrainScaleS-2 system in practice. A limitation also arises from the reduced read bandwidth for omnibus transactions originating from the PPU due to the increased latency, when compared to a single-chip FPGA-BrainScaleS-2 ASIC system. This is especially relevant for PPU read transactions for instruction data, as the increased latency reduces the speed PPU programs can operate at. However, not all PPU programs need to fetch instructions from external memory, as the BrainScaleS-2 ASIC also include on-chip memory for the PPU. By optimizing performance-sensitive parts of a program to be located on this on-chip memory, the impact of the increased latency can be mitigated.

A further limitation of the work presented is the missing integration of the various verification components into continuous integration. This work will be completed after this thesis. Finally a more accurate co-simulation, which includes proper models of the BRISCET-to-BRISCET links would allow a more accurate estimate of the jitter incured by the chosen interconnection scheme.

Unfortunately it was not possible in the time frame of the thesis to perform a full physical implementation or synthesis of the complete toplevel design. However, the single largest component is expected to be the combination of ARQ implementation for the four mesh links, credit counting, input channel buffers and routing implementation. When using a 64 entry buffer for the sender of the ARQ protocol and 32 entries for each two input buffers per port an estimate for the required area is given after synthesis using Synopsys Design Compiler of $\approx$ 170 μm^2 in 65 nm technology, with $\approx 75\,\%$ of this being utilized by the buffers. Most of the different design parameters of the toplevel design, like the number of entries used by the omnibus retarget lookup tables, by the read response reorder buffer and more are easily configurable and therefore can be tuned depending on the available area and the final requirements, so the author expects this proposed design to fit within the envelope of the area available on a 4 mm^2 to 5 mm^2 ASIC carrying the 8 LVDS links and 11 of the links by Ilmberger et al. (2024) for each port and direction.

7. Outlook

Several of the limitations could be alleviated in the future. Furthermore, several further improvements are desirable.

Integration of verification into continuous integration Currently, the different verification strategies described in section 5 were manually employed whenever the implementation or the architecture of the interconnection network or of the BRISCET ASIC was changed. In the future, these should run automatically using the continuous integration setup already employed by the Electronic Vision(s) group. An initial integration of the formal verification parts was already started, but could not be completed due to time constraints.

More capable event routing The event routing implemented in this thesis is kept simple due to time constraints. Instead of a simple circuit-switching scheme that statically connects inputs and output ports, a routing scheme that uses the (configurable) labels assigned to events by the BrainScaleS-2 ASIC for a distributed routing scheme would offer greatly improved flexibility. For example, mimicking the L1 event routing architecture employed by the wafer-scale BrainScaleS-1 (Schemmel, Fieres, and Meier, 2008) system could be promising.

Dynamic assignment of LVDS pairs The current design uses a static assignment of four LVDS pairs to the FPGA-BRISCET interface and four LVDS pairs to the BRISCET-BrainScaleS-2 ASIC interface. It would be desirable to allow the assignment of the LVDS pairs to the BrainScaleS-2 ASIC interface or the FPGA interface in a dynamic fashion instead, to be able to build systems optimized for different demands with regard to the bandwidth of the FPGA and the BrainScaleS-2 ASIC interface of BRISCET.

Support for failed omnibus transactions The protocol used for transmitting read and write transactions across the mesh network and the local omnibus masters and slaves does not support read and write responses indicating errors. A system cannot tolerate any errors as there is no way to communicate or even replay failed transactions. It is likely that building systems composed of many BRISCET and BrainScaleS-2 ASICs will encounter an increase in reliability problems, whether due to the increased system complexity or simply the increased number of components that could be faulty or enter faulty states. To be able to operate such a system, it would be desirable for the system to have local fault tolerance, where a single faulty or misbehaving component does not prevent the whole system from operating. For this, it is necessary to support read or write transactions that can fail and whose failure can be handled and communicated by their initiators.

Standalone mode While the primary focus of the BRISCET ASIC is the development of systems connecting multiple BrainScaleS-2 ASIC a separate use case for the BrainScaleS-2 ASIC is edge deployment. Currently, such an edge deployment requires an FPGA to interact with the BrainScaleS-2 ASIC, which has energy efficiency downsides. By extending the BRISCET ASIC with a way to

generate input event patterns and capture output events, it could be used to enable standalone operation of a BrainScaleS-2 ASIC without an FPGA. Additionally, an external memory interface like AXIS could be useful to extend the amount of memory available in such a standalone system. Such a prospective system could then be operated with a simple microcontroller using the JTAG interface of the BRISCET ASIC. An extension of the available memory would also be useful for mesh deployments, to store data that is not centrally needed and therefore reduce the bandwidth required to the FPGAs of a system, which are currently used to store all data.

FPGA-based verification The simulation and the formal verifications presented in this thesis were used to analyze and verify the behavior of the proposed design for small test cases. Due to the components presented here being purely digital, they can be emulated at much higher speed on an FPGA. Furthermore, the BrainScaleS-2 ASIC can already be connected to an FPGA. In fact, it is the main way it is operated. Therefore, the FPGA emulation of the BRISCET design could be connected to real BrainScaleS-2 ASICs, allowing end-to-end test cases that actually use the analog parts of the BrainScaleS-2 ASIC to be verified.

Optimization Several of the components implemented in this thesis could be optimized. For example, the tunneling of the PPU data loads generates two flits per 128 bit read. Some specific vector fetches by the PPU, however, always generate a total of 8 of these back-to-back, generating a total of 16 flits. By recognizing this type of load and packing the read transactions into a single packet, the number of flits required could be reduced to 4 flits. A similar optimization includes the design of a dedicated DMA engine with the ability to generate larger packets. Further optimizations include dynamic buffer partitioning. Currently, buffers for input channels and ARQ are dedicated to each port and fixed in size. For many use cases, the bandwidth requirements are not expected to be the same for each port, so the amount of buffers could be optimized by allowing them to be shared dynamically. The same is true for the reorder buffers. Each of the reorder buffers uses its own buffer, which could instead be replaced by a central buffer.

Silicon prototype After implementing atleast a more flexible event-routing scheme, this design is expected to be taped out in 65 nm technology to allow a first operation of the described 2D mesh scaled-up BrainScaleS-2 system.

References

Amaranth contributors (2019). *Amaranth HDL: A modern hardware definition language and toolchain based on Python*. URL: https://github.com/amaranth-lang/amaranth.

AMBA, ARM (2021a). "AMBA AHB Protocol Specification." In: URL: https://developer.arm.com/documentation/ihi0033.

– (2021b). "AMBA AXI-Stream Protocol Specification." In: URL: https://developer.arm.com/documentation/ihi0051.

Brette, R. and W. Gerstner (2005). "Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity." In: *J. Neurophysiol.* 94, pp. 3637–3642. DOI: 10.1152/jn.00686.2005.

Bybell, A. (2008). *Implementation of an Efficient Method for Digital Waveform Compression*.

– (2009). *gtkwave/libfst*. URL: https://github.com/gtkwave/libfst.

Cornut, O. (2014). *Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies*. URL: https://github.com/ocornut/imgui.

Cramer, Benjamin et al. (2022). "Surrogate gradients for analog neuromorphic computing." In: *Proceedings of the National Academy of Sciences* 119.4.

Dally, William J and Charles L Seitz (1986). "The torus routing chip." In: *Distributed computing* 1.4, pp. 187–196.

Dally, William James and Brian Patrick Towles (2004). *Principles and practices of interconnection networks*. Elsevier.

Darbari, Ashish (2019). "Smart Formal for Scalable Verification." In: *Design and Verification Conference and Exhibition (DVCon) United States*.

Est´evez, D. (2023). *Amaranth in Practice: a case study with Maia SDR"*, URL: https://www.openresearch.institute/2023/04/11/april-2023-inner-circle-newsletter/.

Favor, G. and A. Patel (2022). *RISC-V Software Interrupts Specification*. URL: https://github.com/riscv-non-isa/riscv-swi.

Forouzan, Behrouz A. (2003). *TCP/IP Protocol Suite (2nd ed.)* McGraw-Hill. ISBN: 0-07-246060-1.

Friedmann, Simon (2013). "A New Approach to Learning in Neuromorphic Hardware." PhD thesis. Ruprecht-Karls-Universität Heidelberg.

Grübl, Andreas et al. (2020). "Verification and Design Methods for the BrainScaleS Neuromorphic Hardware System." In: *Journal of Signal Processing Systems* 92.11, pp. 1277–1292. DOI: 10.1007/s11265-020-01558-7.

Harris, Charles R. et al. (Sept. 2020). "Array programming with NumPy." In: *Nature* 585.7825, pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

IEEE (2018). "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language." In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315. DOI: 10.1109/IEEESTD.2018.8299595.

IEEE (2019). "IEEE Standard for VHDL Language Reference Manual." In: *IEEE Std 1076-2019*, pp. 1–673. DOI: 10.1109/IEEESTD.2019.8938196.

– (2024). "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language." In: *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)*, pp. 1–1354. DOI: 10.1109/IEEESTD.2024.10458102.

Ilmberger, Joscha et al. (2024). "A flexible multi-standard I/O interface for chip-to-chip links in 65 nm CMOS." In: *2024 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pp. 424–427. DOI: 10.1109/APCCAS62602.2024.10808294.

Karasenko, Vitali (2020). "Von Neumann bottlenecks in non-von Neumann computing architectures." PhD thesis. Universität Heidelberg. URL: http://archiv.ub.uni-heidelberg.de/volltextserver/28691/1/KarasenkoPhD.pdf.

Koopman, P. and T. Chakravarty (2004). "Cyclic redundancy code (CRC) polynomial selection for embedded networks." In: *International Conference on Dependable Systems and Networks, 2004*, pp. 145–154. DOI: 10.1109/DSN.2004.1311885.

Leibfried, Aron (2021). "On-chip calibration and closed-loop experiments on analog neuromorphic hardware." Masterarbeit. Universität Heidelberg.

Mayr, Christian, Sebastian Hoeppner, and Steve Furber (2019). "Spinnaker 2: A 10 million core processor system for brain simulation and machine learning." In: *arXiv preprint arXiv:1911.02385*.

Müller, Eric et al. (Mar. 2020). *Extending BrainScaleS OS for BrainScaleS-2*. Tech. rep. Heidelberg, Germany: Electronic Vision(s), Kirchhoff Institute for Physics, Heidelberg University, Germany. DOI: 2003.13750.

OCP (2009). *Open Core Protocol Specification 3.0*. URL: http://www.ocpip.org/home.

PCI Express® Base Specification Revision 6. (2021). URL: https://pcisig.com/specifications/pciexpress/.

Pehle, Christian et al. (2022). "The BrainScaleS-2 Accelerated Neuromorphic System with Hybrid Plasticity." In: *Frontiers in Neuroscience* 16. ISSN: 1662-453X. DOI: 10.3389/fnins.2022.795876. arXiv: 2201.11063 [cs.NE]. URL: https://www.frontiersin.org/articles/10.3389/fnins.2022.795876.

Rettig, Marco (2019). "Characterizing the event interface of the hicann-x." In.

Schemmel, J., J. Fieres, and K. Meier (2008). "Wafer-Scale Integration of Analog Neural Networks." In: *Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN)*.

Schmidt, Alexander (2017). "Design und Charakterisierung einer Routing-Schnittstelle für Neuromorphe Hardware." Bachelorarbeit. Universität Heidelberg.

Shim, Keun Sup (2010). "Static and dynamic virtual channel allocation for high performance, in-order communication in on-chip networks." PhD thesis. Massachusetts Institute of Technology.

Snyder, Wilson (2003). *Verilator open-source SystemVerilog simulator and lint system*. URL: https://github.com/verilator/verilator.

– (2020). *The accelerated development of Verilator with case study of accelerating SweRV core*. OSDA2020. URL: https://veripool.org/papers/Verilator_Accelerated_OSDA2020.pdf.

Stradmann, Yannik et al. (2022). "Demonstrating Analog Inference on the BrainScaleS-2 Mobile System." In: *IEEE Open Journal of Circuits and Systems* 3, pp. 252–262. DOI: 10.1109/OJCAS.2022.3208413.

Virtanen, Pauli et al. (2020). "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." In: *Nature Methods* 17, pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

whitequark (2019). *CXXRTL Simulation Engine*. URL: https://github.com/cxxrtl.

Wolf, C. (2013). *Design and implementation of the yosys open synthesis suite*. Bachelorarbeit.

Wolper, Pierre (1986). "Expressing interesting properties of programs in propositional temporal logic." In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '86. St. Petersburg Beach, Florida: Association for Computing Machinery, pp. 184–193. ISBN: 9781450373470. DOI: 10.1145/512644.512661. URL: https://doi.org/10.1145/512644.512661.

Wren, Luke (2019). *Wren6991/Hazard3*. URL: https://github.com/wren6991/hazard3.

List of Tables

List of Figures

Appendix

A. Hazard3 configuration

The configuration options used for the Hazard3 RISC-V core are summarized here:

RESET_VECTOR	32'h00000040
MTVEC_INIT	32'h00000000
EXTENSION_A	0
EXTENSION_C	1
EXTENSION_E	1
EXTENSION_M	1
EXTENSION_ZBA	0
EXTENSION_ZBB	1
EXTENSION_ZBC	0
EXTENSION_ZBS	0
EXTENSION_ZBKB	0
EXTENSION_ZBKX	0
EXTENSION_ZCB	1
EXTENSION_ZCMP	1
EXTENSION_ZCLSD	0
EXTENSION_ZILSD	0
EXTENSION_ZIFENCEI	1
EXTENSION_XH3BEXTM	0
EXTENSION_XH3IRQ	0
EXTENSION_XH3PMPM	0
EXTENSION_XH3POWER	1
CSR_M_MANDATORY	1
CSR_M_TRAP	1
CSR_COUNTER	0
U_MODE	0
PMP_REGIONS	0
PMP_GRAIN	0
PMP_HARDWIRED	0
PMP_HARDWIRED_ADDR	0
PMP_HARDWIRED_CFG	0
DEBUG_SUPPORT	1
BREAKPOINT_TRIGGERS	4
NUM_IRQS	1
IRQ_PRIORITY_BITS	0
IRQ_INPUT_BYPASS	0
REDUCED_BYPASS	0
MULDIV_UNROLL	1
MUL_FAST	1
MUL_FASTER	1
MULH_FAST	1
FAST_BRANCHCMP	1
BRANCH_PREDICTOR	1

B. End-to-end testbench example

The following testbench verifies that the RISC-V CPU can access memory attached to the FPGA or in other words, that omnibus transactions created by the RISC-V CPU can be tunneled to the FPGA. First, a test program for the RISC-V CPU is specified in assembly. This test program loads the address **`riscv_om.external_memory`** into register **`t0`**, loads an 32 bit word from this address into register **`t1`**, adds a fixed value of 123 to register **`t1`**, and finally stores the result of this addition at an offset of four bytes to the **`riscv_om.external_memory`** address before entering an infinite idle loop.

```
def riscv_data_rw_fpga():
    val = 123
    add = 100
    prog = f"""
    lui t0, {riscv_om.external_memory >> 12} // extmem
    lw t1, 0x0(t0)
    addi t1, t1, {add}
    sw t1, 0x4(t0)
    end_loop:
        wfi
        j end_loop
    """
```

Next the connection to the simulator is initialized and the initialization sequence necessary to bring up the high speed links between the FPGA and the BRISCET is performed.

```
    with create() as (conn, builder):
        generate_init(builder, reset=True, init_fm_pll=True, init_fpga_l2=True)
```

Then the memory located on the FPGA is populated with a test value. Next, the retarget block on BRISCET that determines where omnibus transactions initiated by the RISC-V CPU are configured to send the transactions to coordinate 0 ($x = 0, y = 0$) and to set the highest bit of the address (100_2), which causes the transaction to be sent to the FPGA by BRISCET with coordinate $[0, 0]$. A read to this entry is performed, and a barrier is used to ensure the completion of the transactions that configure the retarget block before the next operations are performed.

```
        retarget_addr = fm_om.riscv_retarget
        # populate the word to read
        om_write_l2(builder, fpga_om.ppumem_sram, val)
        om_write_l2(builder, retarget_addr + 0, \
                    riscv_rewrite_config(0b100, 0b000, 0, 1))
        om_read_l2(builder, retarget_addr + 0)
```

```
21          # the retarget has to be setup before we start the riscv program
22          builder.write(BarrierOnFPGA(), Barrier(Barrier.Value.omnibus))
```

Next, the **`fm_load_and_run_riscv`** helper function is called, which compiles the RISC-V assembly code, writes the resulting data to the memory attached to the RISC-V core and brings the RISC-V out of reset. After some time has elapsed, finally the second word in the FPGA-attached memory is read and compared to the expected value.

```
15          fm_load_and_run_riscv(builder, prog)
16
17          builder.write(TimerOnDLS(), Timer())
18          builder.write(WaitUntilOnFPGA(), WaitUntil(1000))
19
20          result = om_read_l2(builder, fpga_om.ppumem_sram + 1)
21          builder.write(BarrierOnFPGA(), Barrier(Barrier.Value.omnibus))
22
23          run(conn, builder.done())
24          assert result.get()[0].get().word == (val + add)
```

C. Software used in this thesis

repository	commit	changeset
chip-briscet	519dbe099b482813e528c8aba5c44ead5d441c2d	25259
hicann-dls-private	740f92bfb7b712932a590cc8a2a191c2ece31c02	25260
hxfpga	9558680e851cde1560f1025cacf57e14dad68b29	25261
visionary-rtl-utils	3121d8ac0a05e2c144c81a7d7d22a140f2e849dd	25267
lib-vhdl-utils	712e8c5c9ec19d400adee5137a1288aee93be57a	25262
flange	522fdd23830d76a072691946e656e6843c43d00d	
hate	b8c8fd03b6fdc103022848a47bc6838d46b3d3ae	
hmf-fpga	1de3dbba1e34e46b52dd3ff9ef8476bd7dc7de41	
lib-rcf	741e85b36b94929df200240b5fc2ef9c460dcf73	
verilog-i2c	9ebe47566c81828b2428c67251bc7ae2b51ccc2e	
verilog-uart	d7e9f305e1d12bdaebade995997834d2555f5918	
bss-hw-params	81bb81ec5ea72a50ba204420a9919f69d87d90ea	
code-format	512f1fafb884a7da30a1943793a3cae6441462be	
fisch	bc3e8887a8479253c87503e5a32846efa83843f6	
halco	8af1bad29c681f05bbf2e7dc57ea182a52bedd9e	
haldls	0248e7aec1158eb338f7e14997ec695a09cdd421	
hwdb	ecd48af8908ed9d9b75407c5b7efaa1f4872f5a1	
hxcomm	b0c6a6406573582a193a87b479022d6a6315193d	
lib-boost-patches	136c5b41cb046afe2c726aa4646928bf5190622e	
libnux	be48df4602176ddcb7172d3b0bbff656bf7ce938	
logger	73dadb3ce413c521845ef7d36f818073eee4fefa	
pywrap	5e2af30e9593882b471d3cd02df00b93f13ff479	
rant	53199ee94cae1e1c2e4db10e88d570a761b14e0f	
sctrltp	7a94faf9af3d38b202276d11de44182e3602b8cd	
visions-slurm	8f41ea4f5bd1573d8f4623e9ed698a29f30036a3	
ztl	c2d4faee05f497010ee55e35bf9c9607eecbf884	

Acknowledgements

I would like to thank

- Prof. Dr. Johannes Schemmel for the opportunity to write my thesis in the Electronic Vision(s) group and Prof. Dr. Dirk Koch for agreeing to be my second examiner.
- Joscha for answering all my questions.
- Yannik, Julian, Elias, and Eric and for great discussions and insights.
- Everybody who helped with proofreading this thesis, especially Joscha, Yannik, Simon, Jaro, and Anna.
- The Electronic Vision(s) group for providing a great place to work.
- My friends and family for their support during my studies.

The work carried out in this Master Thesis used systems which received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreements Nos. 720270, 785907, and 945539 (Human Brain Project, HBP) and Horizon Europe grant agreement No. 101147319 (EBRAINS 2.0).

Erklärung

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

Heidelberg, den 14. August 2025,