Dissertation

submitted to the Combined Faculty of

Natural Sciences and Mathematics

of Heidelberg University, Germany

for the degree of

Doctor of Natural Sciences

Put forward by

Vitali Karasenko

born in: Kryvyi Rih, Ukraine

Oral examination: 2020 May 27th

Von Neumann bottlenecks

in

non-von Neumann

computing architectures

—

A generic approach

Referees:
Dr. Johannes Schemmel (Heidelberg University)
Prof. Dr. Ulrich Brüning (Heidelberg University)

Neue Blicke durch die alten Löcher.

Georg Christoph Lichtenberg

Abstract

Der Begriff „neuromorphe Hardware“ bezieht sich auf eine breite Klasse von Rechenmaschinen, die verschiedene Aspekte von kortikaler Informationsverarbeitung nachzubilden suchen. Sie instanziieren Neuronen, entweder physikalisch oder virtuell, die über zeit-singuläre Impulse (Spikes) miteinander kommunizieren. Die vorliegende Arbeit präsentiert eine generische Implementation eines Punkt-zu-Punkt (P2P) Kommunikationsprotokolls, welches gut geeignet ist, die besonderen Anforderungen an die Ein-/Ausgabe von neuromorphen Computern im Bezug auf spikebasierte Kommunikation zu erfüllen, insbesondere im Kontext von beschleunigten analogen Systemen. Ein solches Protokoll wurde auf dem neuesten Chip der neuromorphen BrainScaleS-2-Architektur namens HICANN-X implementiert, wo es ihn mit einem benutzergesteuerten FPGA verbindet. Bidirektionale Spikeraten von bis zu 250 MHz zusammen mit mehreren datenflussgesicherten Speicher- und Konfigurationskanälen werden über $8 \times 1\,\mathrm{Gbit\,s^{-1}}$ low voltage differential signaling (LVDS) double-data rate (DDR) Serialisierer ermöglicht. Da die vorgelegte Protokollfamilie unabhängig von der Implementation der Serialisierer ist, ist sie auch jenseits von neuromorpher Hardware anwendbar, etwa um die Modularisierung von Zielvorhaben zu unterstützen, oder um die Entwicklung von generischen Protokollbrücken zu ermöglichen.

The term "neuromorphic" refers to a broad class of computational devices that mimic various aspects of cortical information processing. In particular, they instantiate neurons, either physically or virtually, which communicate through time-singular events called spikes. This thesis presents a generic register-transfer-level (RTL) implementation of a point-to-point (P2P) chip interconnect protocol that is well-suited to accommodate the unique I/O requirements associated with event-based communication, especially in the case of accelerated mixed-signal neuromorphic devices. A physical realization of such an interconnect was implemented on the most recent version of the BrainScaleS-2 neuromorphic hardware architecture—the HICANN-X system—to facilitate a high-speed bi-directional connection to a host FPGA. Event rates of up to 250 MHz full-duplex as well as several stream-secured configuration and memory interface channels are transported via $8 \times 1\,\mathrm{Gbit\,s^{-1}}$ low voltage differential signaling (LVDS) double-data rate (DDR) serializers. As the presented approach is entirely independent of the serializer implementation, it has applications beyond neuromorphic computing, such as enabling the separation of concerns and aiding the development of serializer-independent protocol bridges for system design.

Contents

Part I

Introduction

Chapter 1

Motivation and Outline

The human thirst for knowledge was always accompanied by a desire to process and store information throughout the ages. From mechanical devices like the Antikythera mechanism (Jian-Liang and Hong-Sen, 2016) or Charles Babbage's Difference engine (Swade, 2002), through electromechanical devices, such as the famous Bombe computers (Smith, 2014), to modern complementary metal-oxide-semiconductor (CMOS) processors, both the architecture and realization of computers evolved with our understanding of the universe, as well as the mathematical grasp of information processing itself. While the modern computing landscape is dominated by devices that track their ancestry to the von Neumann architecture (von Neumann, 1993), the realization that the mammalian brain is both a very powerful computer, and also works based on entirely different principles, has always fueled research looking to complement or even replace von Neumann machines with biologically inspired devices.

Attempts to realize *neuromorphic computers* have increased in recent years following both the rekindled interest in artificial neural networks (ANN), as well as the decline of single-threaded performance growth. Starting in the early 2000's, the semiconductor industry found itself in an increasingly paradoxical situation: as Moore's law steadily provided more and more usable transistors per area, it became harder and harder to get the same computing performance boost from them as processor core frequencies peaked at around 3-4 GHz, mostly due to thermal reasons. While conventional computers marched on to leverage the extremely high transistor counts of modern CMOS manufacturing technologies for multi-core processors (together with a paradigm shift towards concurrent programming), neuromorphic devices seek to build inherently parallel computing architectures based on small building blocks (neurons) that exchange activation events (spikes) through a routing bus fabric.

The initial vision of Carver Mead (Mead, 1990) was interpreted and implemented in many ways by various research labs that explored the utility, feasibility and scalability of different approaches to neuromorphic computing. We could not hope to provide a comprehensive review of the various proposed architectures, but rather point to literature, such as (Furber, 2016; Indiveri et al., 2011; Thakur et al., 2018). We will however point out a core challenge that these devices all have to face, namely the immense amounts of data that they are capable of producing and consuming, both locally on chip, as well as externally when connecting to a host computer or forming a larger system.

I/O has always been the Achilles heel in computation regardless of the underlying architecture. The speed at which a computational unit can process data has significantly outmatched the speeds at which this data can be made available. This is the *von Neumann bottleneck*, which has only become more pronounced since its first observation sometime in the 1970's, after the speed increase of transistor logic started to outpace the bandwidth improvements of chip interconnects. Neuromorphic devices suffer from the von Neumann bottleneck as well, but we will argue that they require a different way of addressing it than conventional von Neumann architectures in Chapter 7. The availability of chip interconnect technology, both commercially and in literature, is heavily biased by the strong drive to focus on the needs of conventional processor architectures due to their sheer dominance for the last fifty years.

While neuromorphic computers, as prominent examples of non-von Neumann architectures, are in the exploratory stage of finding optimal implementations, as well as carving ecosystem niches, now is also a good time to start thinking about how an optimized communication infrastructure for neuromorphic hardware might look like, which represents the core topic of this work. To this end however, we have taken a generic approach and will first lay the groundwork by discussing how any digital devices exchange data from a high-level perspective and introduce both the concept of Queues as well as abstract methods to bundle and transport their data between chips in Chapter 2.

In Chapter 3 we focus on the state of the art, i.e, various serializer technologies and their ability to serve as tunnels for Queues. We will see how more accessible technologies often lack desired features for generic data transport, while high-end interconnects tend to be monolithic in nature, which makes them fast and feature-rich, but at the expense of flexibility when used in non-conventional scenarios.

Part II introduces a generic method to tunnel arbitrary bundles of Queues through virtually any kind of serializer using hardware description language (HDL) implementations of sum types and the Universal Translator (UT) encoding scheme. We discuss not only the method itself, but also motivate the immense benefits offered by the generic approach as opposed to a monolithic bespoke solution, as it orthogonalizes many design choices and simplifies verification, thus freeing development time to add features or fine-tune parameters. In particular, we describe a generic method to transform almost any module interface into a bundle of Queues which can then be serialized via the UT. This opens up the possibility to build serializer-independent feature-rich bridges for common bus interfaces, which can streamline the development effort for open-source hardware.

Lastly, Part III describes the communication infrastructure for the current-generation neuromorphic hardware at the Electronic Vision(s) group in Heidelberg, which was successfully manufactured and already used for various experiments. It uses the concepts and modules introduced previously to create a complex bi-directional interconnect between the neuromorphic chip and a host field programmable gate array (FPGA) using an independently provided serializer. Here, we leveraged the generic, yet powerful sum type technique to implement features like link layer channel bonding, link health checking and failure resistant configuration while also providing a bi-directional high transaction rate event transport channel between the devices.

Chapter 2

Queues

Like with virtually any system, designing hardware begins at the block diagram level. The design is broken down into sub-modules, each performing some task, which pass information between each other. *Information passing* is somewhat ambiguous in this setting, so let us clarify further.

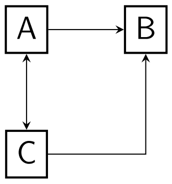


Figure 2.1: Example block diagram with three modules. The arrows represent the direction in which the modules pass information between each other

On one hand, especially in RTL design, we define the direction of information flow by which side of a wire the driver is. This is also reflected in RTL code during module declaration in statements such as '`output logic valid`' where the qualifying statement `{input/output}` describes whether the module is sender or receiver of information on that port. This is, however, not that simple when attempting to precisely define the direction of information passing in a more general context.

To illuminate the issue, let us first discuss one of the simplest non-trivial circuits that has a clear direction of data flow, the first-in-first-out buffer (FIFO) module. Its main purpose is to provide a *blocking interface* between two modules so that data may pass from one to the other. An example block diagram is shown in Figure 2.2, the declaration of a possible SystemVerilog interface can be found in Listing 1. We notice that the modports `push` and `pop` are each bi-directional, i.e, they feature both `input` and `output` ports in their declaration. This is of course necessary to provide a blocking interface to the FIFO, which requires information exchange from the FIFO to the user to notify it that it is ready for the next transaction. Still, from a transactional point of view we define the direction of a FIFO to be from the `push` modport to the `pop` modport.

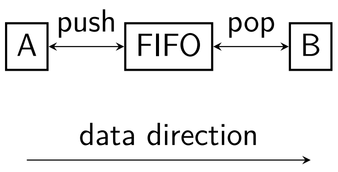


Figure 2.2: A FIFO connecting modules A and B. Module A connects at the push side of the FIFO, and module B at the pop side. Hence the data flow is from module A to module B

Usually, the term FIFO refers to a particular implementation involving some memory and control logic that stores data from the `push` modport into memory and presents data read from memory to the `pop` modport of the module. To be able to talk about directional information transfer between modules in an implementation-free manner, we introduce the following definition:

Definition 2.0.1. Queues Any pair of interfaces (a, b) that respectively perform identical functionality to a `push` and `pop` modport of a FIFO interface belongs to a *Queue* which we will write as $Q(a, b)$. The following parameters are associated with any Queue:

- **latency** as the expected time between a data push into the Queue and its arrival at the `pop` interface.
- **throughput** as the expected number of transactions across the Queue per unit of time.
- **depth** as the maximum amount of in-flight transactions within the Queue.

This definition explicitly includes pairs of interfaces that are not in the same clock domain or even the same device. It obviously follows that any FIFO module is an implementation of a Queue that stays within a device[1]. We further categorize Queues depending on the admissible transaction patterns between their endpoints.

[1]But not necessarily in the same clock domain as in the case of asynchronous FIFOs

```
interface fifo_if(
  input logic wrclk,
  rdclk
  );

parameter int WIDTH = 1;

logic full, push, wrinit, empty, pop, rdinit;
logic [WIDTH-1 : 0] wrdata, rdata;

modport push (
    input full,
    output wrinit, push, wrdata
);

modport pop (
    input empty,
    output rdinit, pop, rdata
);

modport fifo (
    output full, empty, rdata,
    input wrinit, rdinit, push, pop, wrdata
);

endinterface;
```

Listing 1: Example of a parameterized FIFO interface with the modport `fifo` being accessed by the FIFO module itself, `push` by the user on the write side and `pop` by the user on the read side. The signals `wrinit` and `rdinit` perform the initialization of the FIFO at its respective side. The behavior is implementation-dependent, but generally defines a point in time after which transactions at the `push` modport correlate with transactions at the `pop` modport. The `WIDTH` parameter allows to re-size the FIFO to fit any message in any format by casting it into a packed bit-array of the appropriate size.

Definition 2.0.2. Stream-secured Queues A Queue that is guaranteed to show the same sequence of data at its output as the sequence of data pushed into it is stream-secure. The following parameters are additionally associated with any such Queue:

- **mean time to failure (MTTF)** as the expected time duration at which the above guarantee will be broken.

This definition seems unnecessary at first glance when coming from the usual FIFO perspective, as indeed any correctly implemented FIFO will always fall under this category because we usually operate under the presumption that data can never be corrupted on-chip in RTL design methodologies. On the other hand, the concept of a Stream-secured Queue appears naturally in the context of data transfers between devices where it is usually accepted that a wide range of effects can corrupt transactions and extra care needs to be taken to ensure that these can be detected, discarded and ultimately repaired. Stream-secured queues are usually discussed in the context of their implementation within a *Transport Layer* of some interconnect, where this functionality usually resides. The MTTF denotes the expected timescales at which one of the following error cases may happen:

- The `pop` interface stops emitting words that are in-flight.
- The `push` interface is never ready to accept new words.
- Data is re-ordered or otherwise corrupted.

As we will see later when considering implementations of stream secured Queues, the first case is usually due to an internal breakdown of the transport mechanism within the Queue. Blocking on the `push` side can be also either due to transport issues or because the `pop` side is never accessed which means stalls once the Queue depth is reached. Data corruption is the worst case and usually well guarded against, as for all practical purposes this case must never happen. In some sense, we much rather prefer for a Queue to stall than to start emitting wrong data, because while we can establish timeouts to detect stalls, there is no way to detect data corruption within a presumed secured Queue except adding meta data which amounts to tunneling a stream secured Queue through another.

We keep the definition purposefully implementation-free and simply note that a user does not necessarily need to know the precise implementation or internal behavior as long as she is provided with the guarantee that all transactions are appearing in-order at the receiving end.

Relaxing the requirements on data integrity, we introduce the

Definition 2.0.3. word-secured Queues A Queue that has a macroscopic probability to re-order or drop words, but provides a guarantee that individual words are correctly transported, is called word-secure. The following parameters are additionally associated with any such Queue:

- **mean time to stream corruption (MTTSC)** as the expected time duration at which a word-secured Queue deviates from a stream-secured Queue.

We point out the obvious implication that any stream-secured Queue is also word-secure but not necessarily vice versa. Because stream-security is a combination of two properties, namely that data is neither lost nor permuted, the loss of any of these automatically downgrades the Queue to be at most word-secured. We deem it impractical for our purposes to further sub-divide the classification into cases where messages can be either lost or permuted because these properties are highly implementation-dependent. Still, any implementation of a word-secured Queue should be done in awareness of the distinction.

At last, we define the

Definition 2.0.4. Unsecured Queues A Queue that can neither guarantee stream-, nor word security is called unsecured.

Example transaction patterns are shown in Figure 2.4.

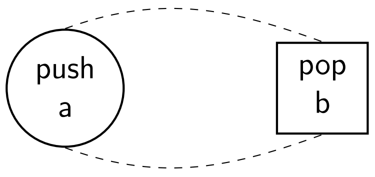


Figure 2.3: A conceptual diagram of a Queue $A(a, b)$. Data is transported from the `push` interface to the `pop` interface in a blocking manner.

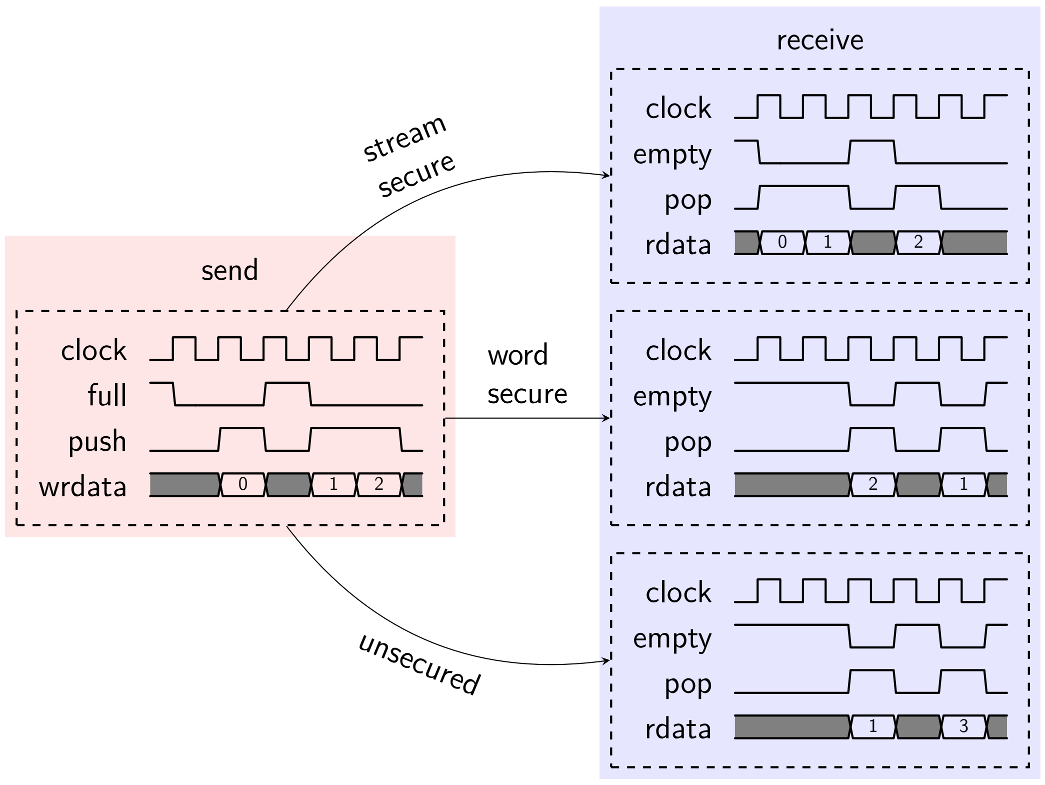


Figure 2.4: Example transactions of different types of Queues. The colored regions separating the send and receive side of the Queue emphasize that the interfaces can reside in different clock domains or devices. For the word- and unsecured Queues only the best case scenario is shown where data is fetched out of the pop interface as soon as new data arrives.

To motivate these definitions, let us now briefly discuss some of their possible realizations in the context of chip interconnects. As we have mentioned previously, moving data between chips is considered to be unsecured unless extra precautions are taken as we will see in Chapter 3. To improve the security of such a Queue, internal mechanisms like checksumming may be introduced during data transport. This additional data provides redundant information that can be used to verify the integrity of the message at the receiving end which then can filter out messages it deems to be corrupt, making the Queue word-secured. Because checksumming can only drop erroneous data but can neither restore nor correct it by itself, additional measures must be taken to achieve stream security. Error correction techniques are usually distinguished into forward error correction (FEC) and backward error correction (BEC) depending on whether they need additional information by the sender or can use already received, redundant data for word recovery.

We will further note a few other properties of Queues.

If a Queue Q is composed out of a series of Queues $Q_N = [Q_0, Q_1, \ldots, Q_n]$, the security of Q is at most the security of the lowest secured Q_i in Q_N (see Figure 2.5).

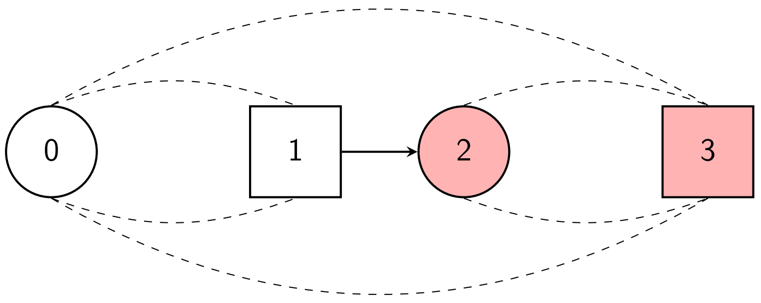


Figure 2.5: An example of Queue concatenation. $Q(0,1)$ and $Q(2,3)$ form together Q(0,3). Because $Q(2,3)$ is unsecured, so is $Q(0,3)$ regardless of the security of $Q(0,1)$. Interfaces 1 and 2 are directly connected, thus forming a trivially stream secured Queue.

A collection of Queues with endpoints in the same hardware domain can be tunneled via a single Queue and additional circuitry for encoding and multiplexing (see Figure 2.6).

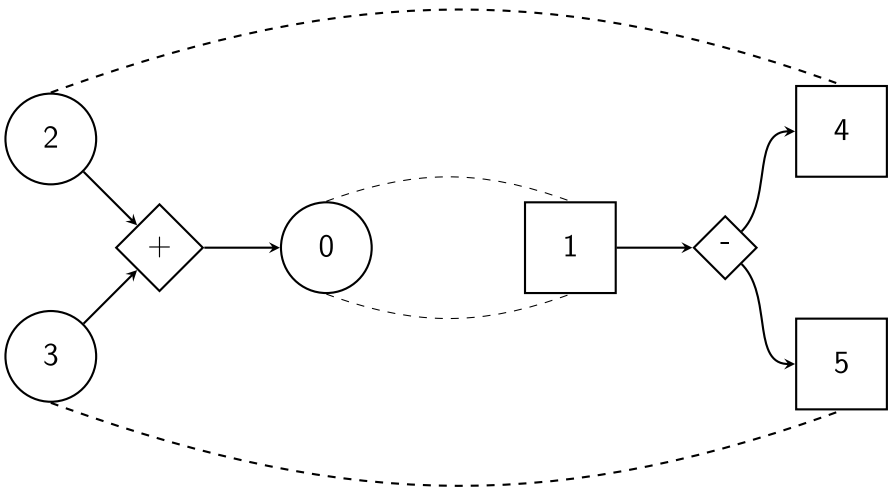


Figure 2.6: An example of a Queue sum. $Q(2,4)$ and $Q(3,5)$ are formed via $Q(0,1)$ and additional logic that merges their data such that it can be distinguished and split after transport. We call it a sum because the words of $Q(0,1)$ transport data from either $Q(2,4)$ or $Q(3,5)$.

Finally, it is always possible to increase the security of a Queue by adding circuitry at the end points without any further information about the Queue itself (see Figure 2.7). A common method is to generate metadata from the outer Queue and then summing it together with the Queue content itself into a tunnel that is less secure than required.

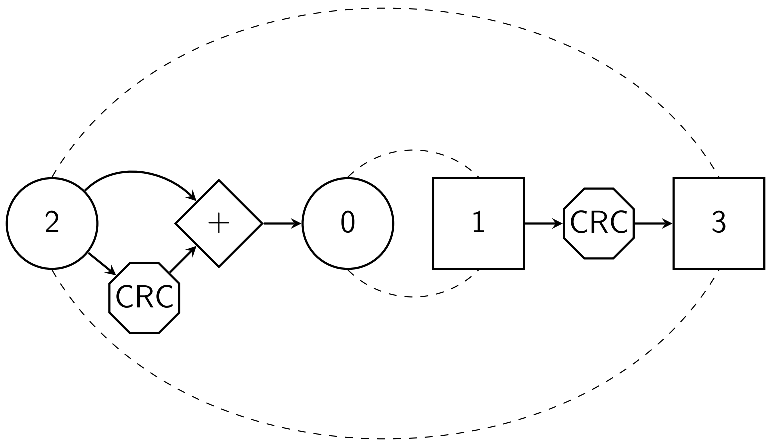


Figure 2.7: An example of Queue enhancement. $Q(2,3)$ is word-secured although $Q(0,1)$ is not due to additional logic that derives cyclic redundancy check (CRC) words which are then added together with the data of $Q(2,3)$. At the receiving side is then a filter that filters the incoming data for correct CRC and then passes it to the `pop` interface after stripping any checksum information. Note that no interface need to be modified to achieve this upgrade in transport security.

2.1 Protocol tunneling

Now that the groundwork is laid out we can begin to discuss transporting higher-level protocols in terms of Queues. Any module provides access to its inner state via ports that are grouped into interfaces such as the FIFO interface shown in Listing 1. A module often provides several interfaces which is a convenient way to achieve *separation of concerns*, as for example in the FIFO module case the `push` interface usually does not need to know about the `empty` flag that is exposed on the `pop` interface. Interfaces can always be split into a *master* and a *slave* side which simply denotes the direction of transactions passing through them. For example, a FIFO module is the slave at its `push` interface as the direction of the transaction is into it, while the opposite happens at the `pop` interface, where data is being read out of the FIFO thereby making it the master.

When we talk about *protocol tunneling* we mean the case where some module A tries to access another module B that may reside neither in the same clock domain nor on the same device altogether. This is achieved by placing circuitry that accepts the request of A acting as the slave, and transports it in some way to the target clock domain where it acts as a master towards B. Any response of B is then transported in the reverse manner towards A. An interface is called *blocking* if there are handshake signals that are not part of the request itself but rather notify the master and slave that a request has been made. If an interface is not blocking there can be no certainty that a transaction has been successfully registered by the slave unless so guaranteed by the specification. It should be clear by now that *any interface can be faithfully represented using one or several Queues and hence can be tunneled as long as the communication infrastructure implements Queue sums.*

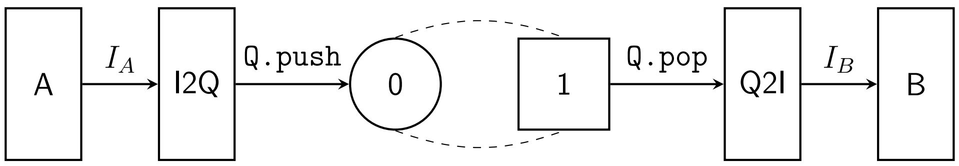


Figure 2.8: Structural view of an interface tunnel between modules `A` and `B`. Module A connects via its interface I_A to an adapter interface to queues (I2Q) which translates the transaction into a series of messages which it puts into the tunneling Queue `Q` via the `push` interface. These messages are then assembled by the queues to interface (Q2I) module into a request to module `B` via I_B.

2.2 Example: OCP tunneling

The Open Core protocol (OCP) is a popular standardized bus protocol for interconnecting modules on a chip. While more recent protocols like Advanced eXtensible Interface (AXI) are now more commonly used, it is functionally very similar and can serve as a fitting example to demonstrate its tunneling via Queues. The interface is comprised of two sub-interfaces called `BusMaster`, which performs an address-mapped request, and `BusSlave` which responds to these requests.

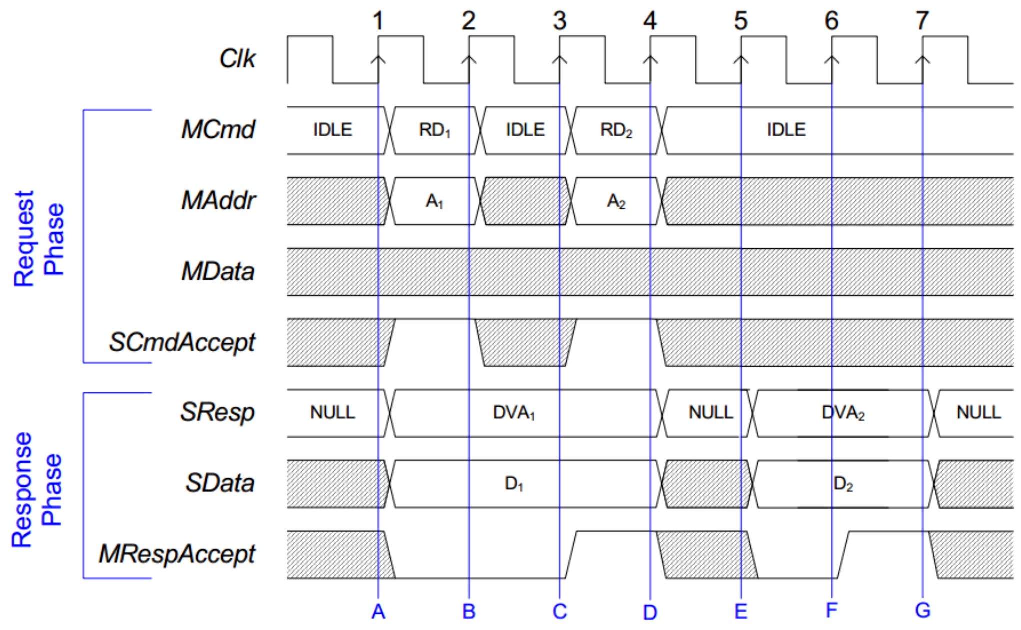


Figure 2.9: Example OCP transactions.

Conceptually, a module performs a blocking request transaction at the `BusMaster` interface into the bus and receives a response at the `BusSlave` interface some time later[2]. Looking at Figure 2.9 it is fairly easy to see that the `BusMaster` interface can be thought of in terms of two Queues C and D, while the `BusSlave` is represented by one Queue in the opposite direction. We have a Queue C whose messages contain the combination of the Fields `MCmd` and `MAddr` and a Queue D that contains the `MData` field. The response side is modeled by a Queue S that contains the `SData` field. Because Queues can be blocking, their interface already contains the handshake of the OCP interface which is represented by the `'MCmd != IDLE && SCmdAccept'` condition, and even allows us to remove the `IDLE` symbol from the `MCmd` field in C. Similarly, as long as the `SResp` field only contains the symbols `{NULL, DVA}` which simply model a handshake together with the `MRespAccept` field, the data in S does not need to contain it because it is already a blocking interface. While it is also possible to represent the `BusMaster` interface with just a single Queue, that approach also misses on the opportunity to realize that a push into D is only necessary if the command was a write, while a push into C is always necessary. Two Queues represent the interface more faithfully, clearly and efficiently and only need trivial logic to model the transaction besides the Queues themselves as can be seen in Listing 3

[2]E.g, the `master` and `slave` modports respectively in Listing 2

```
interface Bus_if #(
  parameter int addr_width,
  parameter int data_width,
  parameter int sdata_width
) ( input logic Clk );

//Request Phase ports
enum{IDLE, RD, WR} MCmd;
logic[data_width-1:0] MData,
logic[addr_width-1:0] MAddr;
logic SCmdAccept;

//Response Phase ports
enum{NULL, DVA}  SResp;
logic [sdata_width-1:0] SData;
logic    MRespAccept;

modport master(
  input Clk,
  output MAddr, MCmd, MData, MRespAccept,
  input SCmdAccept, SData, SResp
);
modport slave(
  input Clk,
  input MAddr, MCmd, MData, MRespAccept, MByteEn,
  output SCmdAccept, SData, SResp
);
 endinterface
```

Listing 2: Example OCP interface declaration. The ports are prefixed with either 'M' or 'S' to indicate whether they are driven by the `master` or `slave` modport.

```
//BusMaster logic
assign C.wrdata = '{Bus.MCmd == READ, BusMAddr};
assign D.wrdata = Bus.MData;
assign SCmdAccept = C.push;
always_comb
begin
        C.push = 1'b0;
        D.push = 1'b0;
        if ( Bus.MCmd != IDLE && !C.full ) begin
                if ( Bus.MCmd == READ )
                        C.push = 1'b1;
                else if ( !D.full ) begin
                        C.push = 1'b1;
                        D.push = 1'b1;
                end if
        end if;
end
//BusSlave logic
assign Bus.SResp = !S.empty;
assign Bus.SData = S.rdata;
assign S.pop = MRespAccept;
```

Listing 3: An example SystemVerilog implementation of a module that acts as a Slave on an OCP `Bus` interface and translates the transactions into three Queues C, D and S.

The reverse process, namely the construction of an OCP interface out of three Queues is entirely symmetric. Listing 4 shows an example HDL implementation of the necessary control logic. Note how no sequential logic is needed here either, and yet the interfaces are blocking and will for instance wait until data in D is available before asserting `'Bus.Mcmd = WR'` if data in the command Queue C has been interpreted as containing a write instead of a read.

```
//BusMaster logic
//highest bit encodes read-not-write, the rest is MAddr
assign Bus.MAddr = C.rdata[$high(C.rdata-1):0];
assign Bus.MData = D.wrdata;
assign C.pop = Bus.SCmdAccept && Bus.MCmd != IDLE;
always_comb
begin
        Bus.MCmd = IDLE;
        D.pop = 1'b0;
        if ( !C.empty) begin
                if ( C.rdata[$high(C.rdata)] ) begin
                        Bus.MCmd = RD;
                else if ( !D.empty ) begin
                        Bus.MCmd = WR;
                        D.pop = Bus.SCmdAccept;
                end
        end if;
end
//BusSlave logic
assign S.wrdata = Bus.SData;
assign S.push = !S.full && SResp == DVA;
assign MRespAccept = S.push;
```

Listing 4: An example SystemVerilog implementation of translating three Queues C, D and S into an OCP interface `Bus`.

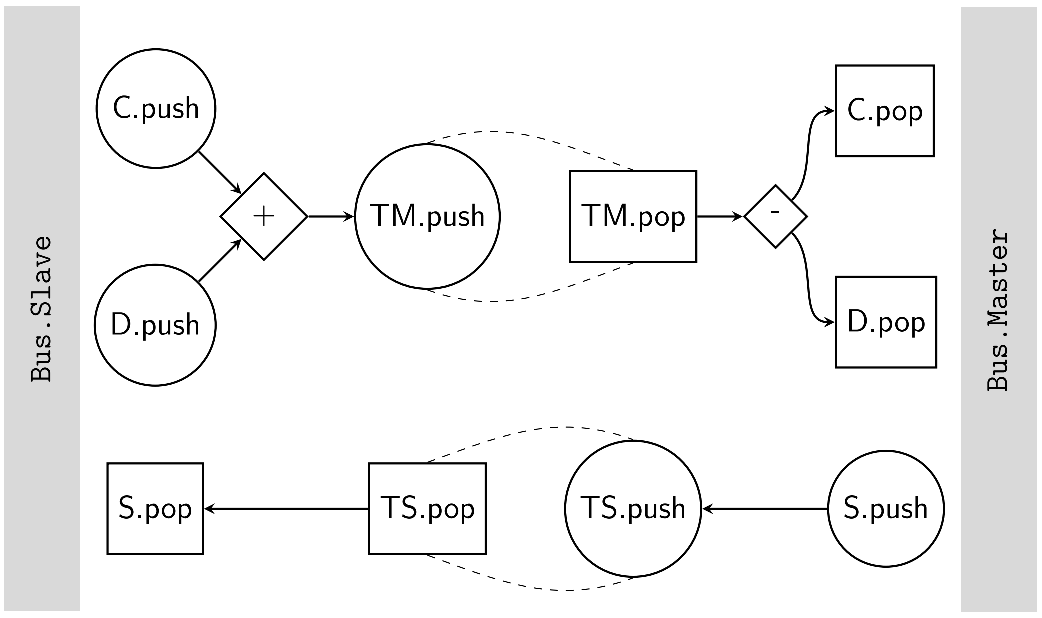


Figure 2.10: Example Queue diagram of an OCP tunnel. Since data flows in two directions during the `request` and `response` phase, at least one tunnel per direction is needed. Queues C and D are merged into TM which acts as a tunnel and ensures the necessary security. Queue TS tunnels the OCP slave responses contained in S and also provides suitable security.

The point of this exercise is to demonstrate that when it comes to protocol tunneling a simple yet generic and comprehensive design flow can be established. The design process must address the following questions:

- How many Queues are needed?
- What security do the Queues require?
- What is the available serialization method?

In this thesis we will demonstrate that it is possible to build a link layer that efficiently satisfies these design parameters using generic modules. But first, let us discuss the various techniques for data transport between devices that are currently in use.

Chapter 3

State of the Art

Any information-processing device, be it a brain or a chip, needs a way to communicate with the outside world to be deemed useful. For chips, this usually means talking to other devices via some protocol. As we outlined earlier, an *interconnect* needs to at least facilitate the tunneling of a Queue across chip boundaries for a point-to-point connection. *Communication* usually implies a bi-directional information exchange, so if the protocol is not bi-directional by nature, it needs to be instantiated twice on the device, as a sender and receiver respectively. Here, we will review some common interconnects and technologies that connect chips and compare their strengths and weaknesses in implementation and performance. We will focus on P2P connections because as it is the main application we are interested in, and thus omit discussing protocols like Ethernet which were designed to be easily routable through anonymous networks.

3.1 Raw Serialization

This technique is probably the simplest way to exchange data between chips. It consists of a shift register of some width that is continuously shifted out by the sender. The shifted out bit is then connected to a general purpose In/Out (GPIO) pin and an internal counter ensures that the next parallel word is loaded into the shift register when the last bit has been sent. Since the data does not have any clocking information, the sender and receiver side of the link must be externally synchronized, which usually means sharing the clock via a dedicated pin. Adding a bit of complexity, the serializer can be upgraded to DDR mode by shifting the data on both edges of the clock. And, of course, the data can be transmitted using the LVDS standard to improve electrical characteristics.

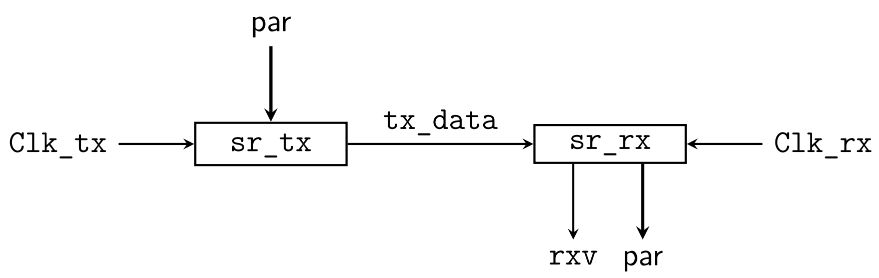


Figure 3.1: Shift register based serializer pair block diagram. Parallel data is loaded into `sr_tx` which shifts it bitwise into `sr_rx` via the `tx_data` wire. After the appropriate number of bits is shifted into `sr_rx` it asserts `rxv` to notify the user that a word is available. `Clk_tx` and `Clk_rx` must be synchronized to ensure correct data capture.

The obvious upside of raw serializers is their simplicity. They are very easy to implement, do not require external IP, and are small both in the gate- and pin count. More complex protocols can build upon raw serializers and simply use them as their PHY layer. An important advantage is the arbitrary width of the shift register, which aids the *separation of concerns* by not imposing any restrictions on the encoding scheme of the upper layer protocols.

However, using raw serializers also has quite a few drawbacks.

The speed is limited by the phase alignment between the clock and the data pins. While internal delays can be in principle compensated for in the chip, the external wire lengths on the PCB are not known in advance and impose a mismatch between the clock and data edges. When not compensated for by using programmable delay circuits—which add to the device cost either because of development effort or IP licensing costs—the serializer will only reliably work up to bit rates significantly below the worst possible delay mismatch.

If programmable delays are used, there has to be a way to calibrate the link by sampling the data until a known pattern is recognized. This in turn implies that after reset the link is at first in a training phase for some time until the receiver has calibrated the delays to reliably sample the training pattern. Only then should the link transmit payload data. A robust synchronization and transition between these link phases is best achieved via bi-directional communication between link partners which makes duplex connections all the more useful. Runtime variations such as temperature and voltage drifts can also affect the phase relationship between data and clock significantly, which makes it not only nearly impossible to statically compensate for it at the receiver, but also suggests constant link monitoring and/or re-training if necessary to ensure the best quality of service (QoS).

Raw serialization is also vulnerable to bit-errors, both on the clock and data pins. If the data pin is corrupted, i.e, sampled incorrectly at the receiver, the transmitted word has a bit flip at the respective position after de-serialization. If however the clock pin is corrupted, the receiver loses the word synchronization with the sender and de-serializes garbage data until the link is reset. A related problem is how to find the correct word boundary in the first place after the link is powered on. Furthermore, the interface that a raw serializer provides is not automatically blocking, as there is no way to distinguish between active and idle link states at the reception side. This can be achieved by e.g turning off the sampling clock (`Clk_rx` in Figure 3.1) as is done for instance in the Joint Test Action Group (JTAG), Serial Peripheral Interface (SPI) and similar interconnects[1].

Therefore, while raw serialization is a good technique to build upon more complex protocols, it is not suitable as a reliable high-speed link by itself and is the prototype implementation of an unsecured Queue whose MTTF strongly depends on the data rate.

3.1.1 Line Codes

Line codes are a popular technique to address some of the problems mentioned previously. While they represent a large family of different encoding schemes, we will discuss the 8b/10b (Widmer and Franaszek, 1983) encoding specifically as a commonly used way to imprint higher-level information onto raw binary data. A review of various line codes can be found in, e.g, (Schouhamer Immink, 2001).

The 8b/10b encoding takes, as the name suggests, 8b blocks of data and encodes them into 10b code words. The mapping is chosen such that *DC balance* is maintained within two encoded words, and aims to avoid long runs of ones and zeroes. The encoding also defines several code words that have no corresponding data words, but still obey the above criteria. These *control symbols* can be used to find word boundaries as well as signal the begin and end of a data transmission. They also allow for a decoupling of the client interface from the SerDes logic, since the SerDes can autonomously inject comma symbols that indicate a link idle state when no client data is available which can then be filtered out by the receiver. This decoupling also allows for different clock domains for the SerDes and the client, and consequently also some, albeit limited, choice in data width. The encoding also helps detect bit errors, as there are many 10 bit sequences that are

[1]Speaking of these, we will not discuss either protocol further as they are barely a step up from raw serialization and do very little—if at all—encoding that aids link security. While they certainly have their use, we will regard them as unsecured Queues at most and use them as such as building blocks for a more secure and feature rich link.

considered illegal by the scheme, so they can be recognized as such at the receiver. However, many bit errors remain undetected, hence one should not rely too much on code word security if data integrity is absolutely essential.

The 8b/10b encoding is still widely used in a variety of communication systems. Still, it is important to understand that it is an encoding for PHY layer data, and should be used in conjunction with a link layer protocol. The PHY layer should contain the 8b/10b codecs, and use the control symbols to perform tasks like word synchronization, idle commas and packet delimiting upon link layer requests. The link layer uses a suitable interface to transmit its data serialized to bytes which are encoded as data words at the PHY layer.

From our perspective, a serializer that employs line coding can be regarded as a weakly word-secured Queue. Comma characters provide the blocking interface regardless of whether the clock is running and the illegal data characters will catch certain kinds of bit errors.

3.2 Gigabit Transceivers

An evolution from raw serializers, multi-gigabit transceiver (MGT)s are now the backbone of almost all high-speed serial protocols. The use of clock-data recovery (CDR) eliminates the need for a dedicated clock pin and thus the phase alignment problem. Virtually all MGTs employ a scrambling line code, be it 8b/10b or something more modern like the 64b/66b variant to increase bit efficiency. This is important to ensure the phase-locked loop (PLL) used in CDR sees enough transitions to properly lock while also providing the benefits outlined in Section 3.1.1, such as flow control via comma symbols.

While MGTs are common in most modern FPGAs, where they are widely used, they appear less frequently in ASICs. This is mostly owed to their complexity which is necessary to achieve the high bit rate that sets the MGTs apart from more simple SerDes approaches as discussed e.g in Section 3.1. Exacerbated by the sorely underdeveloped Open Source space for hardware design, it is often a rather expensive endeavor for a design team to procure an MGT macro since they will either have to develop it in-house or purchase from an IP vendor. Still, MGTs are currently the only feasible choice to achieve high data rates over a low pin count, and are thus the preferred PHY layer choice for virtually all modern high-speed communication protocols.

3.3 PCI Express

Designing and implementing a custom Transport Layer protocol is not always necessary. There are several suitable standards defining a high-speed full stack communication protocol between devices using MGTs as PHYs. Here, we will briefly discuss the perhaps most ubiquitous high-speed device interconnect, the PCI Express protocol. It will serve as the main point of comparison to our work, so we must first evaluate Peripheral Component Interconnect Express (PCIe) from a conceptual point of view.

The design philosophy behind PCIe was to specify a scalable feature-complete general-purpose interconnect architecture. The user communicates over the interconnect via transaction layer packets (TLPs) that have a fixed definition. Additionally defined data link layer packets (DLLPs) transport protocol-internal information and implement features like flow control, data integrity and error handling. The Physical Layer then serializes the TLPs and DLLPs using one or several MGTs which employ 8b/10b encoding and also handles tasks like link training. The Base Specification Document (Group, 2010) provides an in-depth description of the interconnect.

3.3.1 Monolithic Design

While PCIe aims to provide a general-purpose interconnect, there are still applications which do not fit well into its design space. PCIe is not intended to be a customizable protocol apart from offering several backwards-compatible versions which mostly concern the number of lanes and their speed. Therefore, all available implementations, both commercial and Open Source[2], offer PCIe as a monolithic block in a take-it-or-leave-it fashion. This is a reasonable design decision since for instance, if one designs a custom accelerator that is used by an off-the-shelf desktop computer via its PCIe slot, the PCIe block instantiated in the accelerator must be fully compatible to the desktop which is easier to ensure in a monolithic specification. On the other hand, if, for example, we want to build an ASIC that is connected to an FPGA, we fully control both sides of the interconnect and are thus free to choose a protocol that fits our needs best. And while PCIe certainly fulfills its promise of being a general-purpose interconnect, there may be constraints like power, area, I/O or latency that make technology desirable which allows for trade-offs in these areas while still offering a static user interface. It is important however to not fall into the *Not Invented Here* mentality and, if possible,

[2]There are only FPGA-based Open Source PCIe implementations, which instantiate hard MGTs for the PHY layer.

evaluate the desired features compared with their counterpart defined in the PCIe specification.

3.3.2 User Interface

PCIe is a byte-aligned protocol at all layers, which is not surprising since an 8-bit byte is typically the smallest accessible unit in most CPU architectures and programming languages. It also naturally follows that the user-side data path can be somewhat parameterized to a width that is also commonly found in the microprocessor world as long as it is still byte-aligned. Typical use cases are 32 or 64 bit wide user data paths which simultaneously fit the common granularity of most current programming models and keep the user clock at manageable speeds while still achieving high throughput.

Packet types are encoded via the first byte in a TLP, with currently 20 such types being defined. These packet types facilitate transactions in 4 address spaces, `Memory`, `I/O`, `Configuration` and `Message`. While the packet types and the corresponding address spaces provide an interface that can be used to transport almost any kind of data over any network topology, it comes at a cost of rather high overhead independently of the actual application particularly for small transaction sizes. All user packets are also secured via a replay buffer and a CRC to ensure stream coherence, which is an important feature but also rather costly in terms of protocol timings, bit efficiency and hardware usage. If we again evaluate PCIe by its capability to tunnel Queues, we find that it provides facilities to tunnel many of them by either distinguishing them via Packet types, or address spaces since any suitably formatted TLP can be interpreted as a push into the corresponding Queue. PCIe guarantees stream security without the option of downgrading in exchange for performance or hardware real-estate, and also forces at least byte alignment for the Queue width. It is also not suitable for transporting small transactions due to the high overhead within a TLP.

3.3.3 PHY

As mentioned previously, PCIe relies upon MGT serializers which use scrambling[3] to transmit and receive Transaction Layer and Link Layer packets. Their behavior is strictly specified including the link training procedure and the supported speeds. *Channel bonding* is also implemented at the PHY layer, allowing for increased throughput at the cost of higher I/O footprint. It works by distributing the individual bytes of a TLP in an alternating pat-

[3]Both the 8b/10b code, as well as its 128b/130b variant in recent versions

tern on the available lanes during transmission. The obvious benefit is that no two TLPs can overtake each other during serialization because only one TLP is transmitted at a time. Furthermore, adding links not only improves the throughput, but also the latency of the link. As a drawback however, the PHY layer needs to make sure that all the individual lanes are delay-aligned during link training to exclude the possibility of a faulty reconstruction of the packet at the receiving side. Only 1,2,4,8 and 16 lanes per link are currently supported since these allow for easy distribution of the TLP and DLLP in a byte-wise fashion.

3.4 Conclusion

When looking at the individual solutions to transporting data from one chip to another, we notice two sharp extremes. Those who have the resources, both financial and in the design space, will often opt for a proven solution and use one of the common high-speed serial protocols like PCIe, Universal Serial Bus (USB) or Serial AT Attachment (SATA). These projects often afford a practical point of view that it is more efficient to adopt an existing ecosystem even if it does not fit the bill exactly instead of building a custom solution. This adaptation however makes it hard to build a well parameterized design because requirements like word alignments and data segmentation are now an important consideration to make. When the requirements of the core logic are too much at odds with the interconnect, an intermediate layer is built which will attempt to translate the interface of the core logic into the user interface of the interconnect. These adapters are often treated as an afterthought and are prone to being buggy and inefficient in both chip real-estate and encoding.

On the other end of the spectrum we find projects that choose to build a custom interconnect to serve their needs. There are many reasons to do so, ranging from lacking the means to procure a suitable IP or not being able to fit the block into the chip real-estate, to determining that the user interface is too much at odds with that of the core logic. In most of these cases the design team will start from a bottom-up approach, i.e, they will pick a suitable serializer and then build the rest of the Link and Transaction Layer from there to ultimately connect to the core logic. This approach often runs into the realization that the finished interconnect will have to replicate many of the core features of, say, PCIe, but still being different enough that reusing existing modules is impractical or outright impossible in the case of commercial IP blocks. Since the design and verification of blocks like PCIe nodes takes many person-years, the design team is then often forced to implement a solution that is sub par in both feature scope or even performance. Thus, these projects often end up with interconnects that are not thoroughly verified, lack features, extensibility and parametrization, and have their relative simplicity and thus small chip real-estate footprint as the only upside.

To improve upon this situation where a group is stuck between the choice of either adopting a big monolithic block that is not parameterizable in any meaningful way or trying to build something in-house that stands almost no chance of competing against foreign IP, we can formulate the requirements for a desirable solution as a consequence of the previous discussion.

Modularity The layers in the interconnect should be clearly separated and easily modifiable and even exchangeable with minimal changes to the surrounding layers.

Serializer independence The interconnect should provide a simple and generic interface towards the PHY layer which enforces minimal requirements on how data is transported over the air gap. In particular, this interface must admit free parametrization of the data width that is passed between the PHY and the link layer to account for any serialization ratio. The link layer must be able to optionally generate idle characters upon request if the PHY expects a continuous stream of user data after initialization. The link layer must also be able to optionally generate and inject CRCs for any subset of data packets in case the PHY layer does not provide data integrity checking on its own. It should be possible to go from a simple shift register to an MGT as a PHY layer via simple parameter adjustments. Together with the *Modularity* requirement, this also implies that low level electrical properties like line emphasis should be encapsulated at this level such that the user side of the serializer entirely consists of synchronous digital logic.

Channel Bonding The interconnect should be able to take advantage of multiple links if they cannot be merged already at the PHY layer. This should be reflected as a simple parameter with a wide range and must be transparent to the user interface.

User interface The user interface must be able to tunnel any number of independent Queues of any configuration as described in Chapter 2 in both directions. It should be trivial to add or delete a Queue, adjust its width or change the configuration from e.g unsecured to stream-secured independently from all others.

Synthesis-friendliness The design should be easy to pipeline in order to simplify implementation. Where possible, there should be free parameters which can gradually trade performance for chip real-estate.

Performance While we obviously desire a high throughput and low latency for our application, we acknowledge that no design with such a high grade of flexibility can ever hope to achieve the performance of an optimized monolithic design like PCIe. Therefore, we downgrade performance considerations to *best effort* for any given parametrization with the expectation that a well-fitting parametrization will make up for the overhead incurred in adapting to a better, but worse-fitting interconnect.

Part II

Implementation

Chapter 4

Generic Hardware

For scientific progress to occur, it is not enough to criticize why the state of the art is lacking and subsequently propose a seemingly superior solution; One has to actually demonstrate this superiority. We have spent quite some pages to criticize the current options for device communication while being purposefully vague about implementation details of how generic Queue tunneling can actually be achieved. Such an achievement rests on two legs, RTL description and simulation and subsequent successful synthesis in either an FPGA or application-specific integrated circuit (ASIC). Designing hardware comes with the perhaps peculiar caveat that not everything that should be in theory possible is actually possible in practice which as we noted before often impedes scientific progress. There is a plethora of reasons why that holds true and mundane problems like budgetary and/or timing constraints during development or tool immaturity are surely a contributing factor to the reputation of hardware designers being overly conservative in relying on proven designs instead of innovation.

In particular, we observe a lack of the equivalent of what is called *generic programming* in the software world, a design philosophy where the RTL description is so deeply parameterized that a particular parametrization can result in drastically more different circuitry than one observes by changing e.g the width or depth of a FIFO. As we will see however, this is a necessary feature of our design because there is otherwise no way to construct an efficient yet generic Queue tunnel[1]. When we look at the properties of Queues described in Chapter 2, most of them correspond to straightforward well known implementations. The FIFO interface is a very popular archetype and is widely used in all kinds of module interfaces and of course also in de-

[1]We have pointed out before how PCIe achieves generic tunneling by having fixed headers for packet type and address space which are big enough for any application yet are often wasteful because not all functionality is used in a particular design

vice interconnects. Really, the only intriguing implementation question is posed in Figure 2.6: *How can we write an RTL description of some number of queues which can differ both in their widths and security level such that their data can be tunneled via a single queue and still be distinguishable at the receiving end?*

4.1 Sum Types

4.1.1 A software example

Let us for a moment step back and answer this question from a more abstract perspective that programming languages can give us, particularly when they strive for clarity above implementation. FIFO objects are present in most programming languages, we point to the `std::queue` in C++ as an obvious example. They can be used to transport and buffer messages between agents in different parts of a program. Generic programming in C++ allows us to create a `std::queue` of any type we like which gives us freedom of choosing any fitting data structure to serve as an atomic message between agents and then simply declare a `std::queue` on it.

```
// example message declaration
struct message {int a; double d;};
// declare a queue of messages
std::queue<message> q;
```

Now let us assume that we actually have two different message types that we want to transport via a single Queue:

```
// declare message type A
struct messageA {int a; double d;};
// declare message type B
struct messageB {char a; float d; long c};
// declare a queue of messages A and B
// what do we template on?
std::queue<???> tunnel;
```

It is obvious that we need a new type in the above code snippet that is derived from both `messageA` and `messageB` in the above example. This type must satisfy two properties: It must be big enough to contain either `messageA` or `messageB` and it must contain additional information that allows us to find out which type it is currently holding. These derived types are called *Sum Types* or tagged unions and are dual to the more commonly known *Product Types* such as structs and tuples. Sum Types were added in C++17 under the name of `std::variant<>`, which lets us complete the above code snippet:

```
// declare message type A
struct messageA {int a; double d;};
// declare message type B
struct messageB {char a; float d; long c};
// declare a queue of messages A and B
std::queue<sdt::variant<messageA, messageB>> tunnel;
```

This code gives us a Queue that contains a sequence of messages which can each be either `messageA` or `messageB`. Trying to add to the Queue a type that is neither `messageA` nor `messageB` will fail to compile which improves code correctness. C++ Variants can be queried about the type they are currently containing via the `std::visit` free function which then provides the splitting mechanism that allows us to de-multiplex the messages into individual Queues after tunneling.

4.1.2 HDL implementation

The question now becomes whether it is possible to create the same kind of Sum Type abstraction as e.g `std::variant<>` in a synthesizeable HDL. We will see that this is indeed possible if we are willing to sacrifice some type safety. Let us begin with a SystemVerilog example:

```
// declare message type A
typedef struct packed
{
    logic[1:0] a;
    logic[3:0] b;
} messageA_t;
// declare message type B
typedef struct packed
{
    logic[3:0] a;
    messageA_t b;
    logic c
} messageB_t;
```

Recall that the only base type of any synthesizeable HDL is `logic` which represents a single bit and hence either a single FlipFlop or wire in hardware. Any higher order types like arrays, tuples or structs are simply for human readability and do very rarely bear any real correspondence to the resulting circuit after synthesis. Although HDLs like SystemVerilog have types like `int` or `byte` these are nothing more than aliases for `logic[31:0]` and `logic[7:0]` respectively. Ultimately, any synthesizeable HDL data type has a direct correspondence to a bit vector of the appropriate size which is why the following can be always done:

```
// declare messages of type A
messageA_t messageA0, messageA1;
// declare plain bitvectors of the same size as messageA_t
logic [$bits(messageA_t)-1:0] messageA_plain0,
                               messageA_plain1;

assign messageA0 = messageA_plain0;
assign messageA_plain1 = messageA1;
```

We can freely assign both a bit vector to a packed struct of arbitrary field layout and vice versa as long as the total number of bits matches in both objects. There is no fundamental reason that can prevent us from doing so because the synthesized logic can only consist of binary functions and FlipFlops which have no further structure; Any attempt to prevent the designer from doing the above is the result of higher order reasoning about the design which is for example what code linters try to do. This is ultimately the reason why FIFO modules or interfaces only have a `WIDTH` parameter and are very rarely templated on types, although newer versions of SystemVerilog offer this feature; While the `data_{in,out}` port signifies the amount of bits that are accessed or stored at a time, their higher-level structure is irrelevant to the implementation and is merely syntax sugar. All of this is to say that it suffices being able to create Sum Types of bit arrays of variable lengths because that is the only property that is still present after synthesis.

As noted previously, a sum type needs to be able to hold any of the types it is derived on together with information on what it currently holds. A suitable representation is therefore a struct with the fields `data` and `tag`. To be able to construct sum types out of an arbitrary amount of parent types we can use a list that contains the sizes of the individual parents.

```
// declare a parameter list that holds the sizes of
both messageA and messageB
localparam int typelistAB [2] = {
                                  $bits(messageA_t),
                                  $bits(messageB_t)
                                 };
```

Since we are ultimately dealing with bit arrays of varying length, the `data` field is again a bit array with the greatest length of the parent types. The `tag` field will then simply contain the index of the parent type in the type list. Thus, we can now finally write down a SystemVerilog implementation of the sum type for `messageA_t` and `messageA_t` as follows:

```
typedef struct packed
{
    logic [$clog2($size(typelistAB))-1:0] tag;
    logic [$max(typelistAB)-1:0] data;
} sumAB_t;
```

Because we derived the sum type from a list it also inherits some very useful properties which we list here.

Slicing and Concatenation Two sum types can be merged to form a new flattened sum type. The type list of the resulting sum type is simply the concatenation of its parents.

As an example, assume that `sumAB_t` represents the two message types that a single client which might be the master side of a bus adapter as described in Figure 2.10 speaks. We can merge two of these clients into a single Queue by forming a concatenated sum type like so:

```
localparam int
    twoclientsAB [2*$size(typelistAB)] = '{2{typelistAB}};
```

After deriving the sum type struct out of the type list in the usual manner shown above, we now have a framework of creating Queues that can fit an arbitrary number of clients with an arbitrary number of message types. The reverse process, namely creating two sum type out of one is also directly equivalent to slicing the original type list into two children lists. Simple logic can then be built that assigns the data into the two children and also appropriately offsets the tag fields for easy and type safe de-multiplexing.

Scalar operations We can also just as easily modify sum types by doing element-wise operations on the individual elements in the underlying type list. The most useful of these is element-wise addition which corresponds to extending the parent types with an extra field.

4.1.3 Sum Type Queues

Because the sum type is again simply a collection of bits we can now trivially construct a FIFO interface for it using the template introduced in Listing 1:

```
fifo_if
#(
    .width($size(sumAB_t))
) fifoAB (
    .wrclk(),
    .rdclk()
);
```

Example pushes into the FIFO may look like this:

```
//write messageA_t into fifo
assign fifo.wrdata = '{ '0, '{2'b01, 4'ha}};
assign fifo.push = ~fifo.full;
```

Similarly, the de-multiplexing of the FIFO output could look like the following snippet:

```
messageA_t messageA;
messageB_t messageB;
logic [$size(typelistAB)-1:0] messages_valid;
sumAB_t message_read;
assign message_read = fifo.rdata;
assign messageA = message_read.data[$bits(messageA_t)-1:0];
assign messageB = message_read.data[$bits(messageB_t)-1:0];
always_comb
begin
    messages_valid = '0;
    if (~fifo.empty) begin
        case (message_read.tag)
            0: messages_valid[0] = 1'b1;
            1: messages_valid[1] = 1'b1;
        endcase
    end
end
```

Listing 5

The above de-multiplexing logic translates into the circuit shown below:

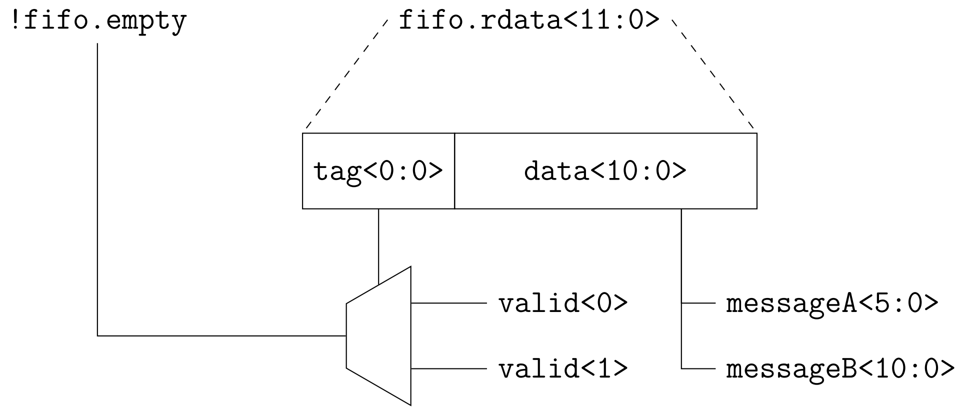


Figure 4.1

Comparing the HDL code in Listing 5 with the RTL circuit in Figure 4.1 we can discuss some of the benefits of generic hardware. On one hand, the RTL code manages to express the de-multiplexing of a sum type into the individual parent types without loss of generality. Changing the parent types will correctly propagate through the design after synthesis which is owed mostly to the fact that a SystemVerilog `struct packed` keyword provides us with automatic conversions to and from bit vectors[2]. Also, in case the sum type is modified to contain more parent types, adapting the de-multiplexing is as simple as adding a statement that casts the data into some new parent type as well as an additional branch in the case statement to generate the respective `valid` signal. It is even possible to create a de-multiplexer that will automatically create additional branches using `generate` loops, as long as it is parameterized not on the sum type but rather the underlying type list since it contains all the necessary information.

All of this is achieved without sacrificing hardware efficiency; The synthesized RTL circuit looks exactly as expected with no resources wasted. Recall as a contrast the user interface of a PCIe interconnect. As we have discussed in Section 3.3, it provides fields to distinguish between the payload data which can be used to implement the `tag` field of a sum type. However, both the data type field as well as the payload data itself have a fixed width. If the actually required tag is smaller than the data type field, resources are wasted and bits are unnecessarily transported wasting bandwidth. In the rather unlikely case that the tag field is too small some additional encoding must be employed to provide the missing information. As we also discussed,

[2]Languages like VHDL which tend to frown upon such liberal features require a bit more work in that for every type one must also additionally specify the casting functions to and from bit vectors.

the fixed width of the payload data in PCIe can result in an inefficient bit usage of the `data` field of the sum type depending on the parents with potentially some additional encoding logic required.
This leads us to a more general issue; We have now created a hardware domain where modules communicate with each other via arbitrary messages that are passed through FIFO interfaces with a structured way to combine and split messages where necessary. However, This simple view hinges on the requirement that each FIFO `push` or `pop` transaction produces exactly one message that can be either a singleton type or some sum type that can be processed further. As long as we are staying on-chip this is a manageable albeit somewhat cumbersome restriction. Virtually any synthesis tool flow provides us with FIFOs with parameterizable data widths of a wide range that can in principle accommodate almost any use case. To be useful in the context of chip interconnects, where as we have seen we often do not have the choice to freely parameterize the user interface, we must find a way to generically serialize sum types into a stream of words with a wide range of widths.

Chapter 5

The Universal Translator

We present here an implementation of a generic RTL module pair for the encoding and decoding of any sum type into words of any width. Colloquially, the names UT `sender` and `receiver` have been established for them to emphasize their truly awesome parameterizeability. Looking again to the software world, the UT implements functionality similar to `boost::serialize()` which can translate any C++ object into a byte stream and vice versa. This comparison is however not completely apt because the UT must be able to deal with blocking as well as non-blocking stream interfaces as well as arbitrary data widths and not only bytes.

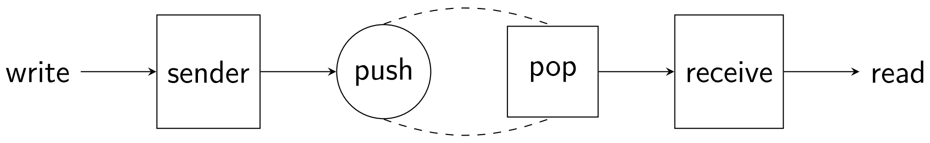


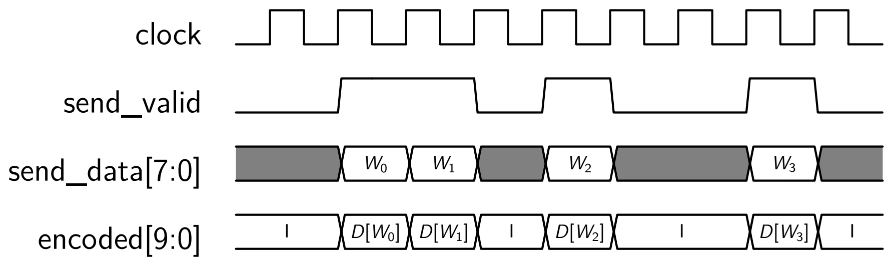
Figure 5.1: UT `sender` connecting to a `receiver` via a Queue. `write` and `read` are arbitrary sum types as described in Section 4.1, the width of the Queue is a free parameter.

5.1 Encoding scheme

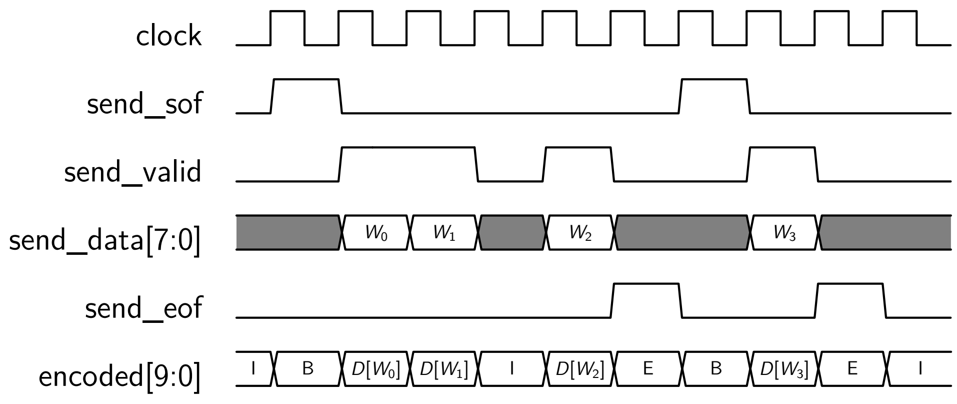
There are indeed many ways to encode information into a stream of symbols. One such way is for example the 8b10b encoding which we have discussed earlier. There, 8 bit data words are encoded into 10 bit symbols or code words. Disregarding the nice electrical properties that the scrambling provides and the error detection, the encoding also gives us a structure that we can use to meaningfully encode data. This is because the 8b10b encoding also defines several special symbols which can be used to denote the `BEGIN (B)`, `END (E)` and `IDLE (I)` phases of a transaction together with `DATA (D[])`

symbols that denote payload. We can then begin to define datagrams which get encoded as a string of such characters.

As an illustration, we show a timing diagram of a blocking 8b10b encoder that translates user bytes into code words.



In this example, we have no way to distinguish whether a higher level object is constituted of the sequence $[W_0, W_1]$ or just the single byte $[W_0]$. Because this encoding scheme only cares about the integrity of the individual code words, any information that lets the user group several bytes into a larger data structure must be implemented using the payload. Alternatively, we can use the aforementioned `BEGIN` and `END` characters to introduce framing into the coding scheme. We can extend the user `send` interface to include start-of-frame (SOF) and end-of-frame (EOF) ports that signal framing information to the encoder.



This interface allows the sender to clearly communicate which data words form a data structure using the `BEGIN` and `END` symbols. It is even possible to insert `IDLE` symbols during a frame transition to indicate a pause in transmission.

Note however, that there isn't any reason to keep the 8b10b encoding for the data words during a frame from an encoding point of view since an individual symbol doesn't really carry useful meaning anymore by itself. In fact, the entire scheme seems unnecessarily wasteful in encoded bits since we are expending 3 code words of 10 bit each to transmit 8 bit of payload in the worst case.[1] Often, these overhead concerns are brushed away by the observation that in the limit of long frames the bit efficiency converges to 80%. Still, this puts the burden on the user to make sure to create frames that are big enough which often requires one or several additional protocol layers that provide the user with a convenient interface while ensuring optimal packing and data alignment at the lower encoding level. These layers must be keenly aware of the underlying encoding and are almost always required to be altered or replaced entirely if the interface changes either at the user or `PHY` layer. And, as always, we are left with the problem that the `PHY` must be able to coherently transmit the 10 bit code words which can otherwise mean further bit efficiency losses and additional logic if, for example, the `PHY` uses an analog bit encoding that results in 3 bit symbols.

All of this is to say that the 8b10b encoding serves a very specific purpose and works best when used in certain applications where the payload is usually large and byte-aligned and the serializer is well-equipped to transmit 10 bit sized code words. We are however able to learn several lessons for our quest to find a more generic encoding:

- **Framing** The encoding scheme must be able to encode payload in such a way that a single transaction can be longer than a single encoded symbol and still be discernible at the receiver.
- **Commas** The encoding scheme must have a way to distinguish payload from idle transactions on the link that can be discarded by the receiver.

Furthermore, the encoding scheme should not concern itself with "analog" features like scrambling or link initialization but rather only focus on efficiently implementing the above points. The encoded words can then still be scrambled in a subsequent step at the PHY if the need arises.

It is useful to look at the minimum required bits when trying to transmit a single message that is n bits in size. The datagram will simply need to be $n+1$

[1]There is however some nuance here: Due to the scrambling and the increased code space these symbols are *unique* and can therefore be recovered from an unstructured data stream without any further information. We acquiesce that this is indeed a very useful property to have in a link since it is a straightforward method to synchronize sender and receiver so that data transport can occur. Nevertheless, we posit that a universal encoding scheme must surrender this property because the underlying Queue may have its own method for initialization and synchronization which we want to make usage of. We will discuss how to deal with link synchronization without unique comma characters at a later time.

bits long, because we will need to distinguish payload data from idle commas. The flattened struct `struct packed {logic c; logic [n-1:0] data}` will have the following bit layout:

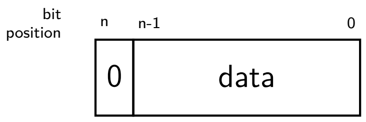


Figure 5.2

Let us assume that the datagram will need to be serialized into words of x bits. The natural thing to do would be to introduce padding to align our datagram of length $n+1$ bits to be divisible by x which makes its total length $L = lcm(n+1, x)$. For comma characters we create a word of length x where the first bit is set[2]. In cases where $n+1 < x$ we simply pad until the datagram fits in a single code word.

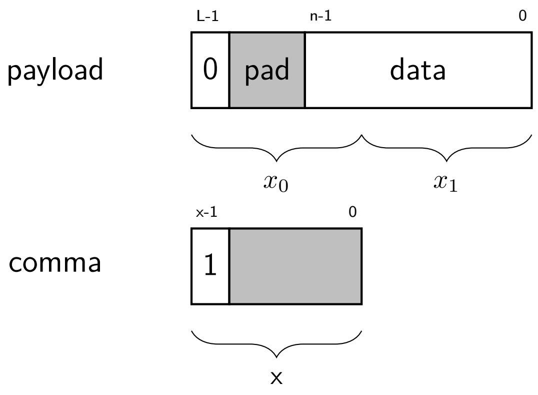


Figure 5.3: Padding example with $L = 2x$. The x_i denote the serializer words, lower subscript is encoded first.

This is the typical approach taken to gearbox data of some width to be serializable into another together with idle commas because it is both easy to describe in HDL code and synthesizes very well into small and fast circuits. Notice the crucial role the padding plays in this encoding: It is on one hand completely determined by the design parameters n and x, but also ensures that the resulting serialization circuit is simple at the cost of bit efficiency. Usually, if one is interested into optimizing the bit efficiency, changes are made not to the encoder itself but rather in the connecting modules at the

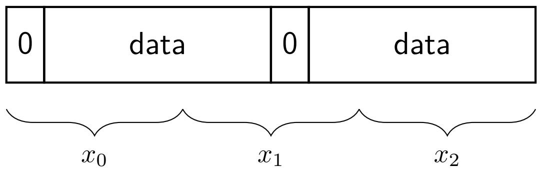
[2]This is not wasteful because there wasn't any data that could have been sent instead anyway

client or the serializer side, or both. If it is possible to adjust n, i.e, adding or cutting useful payload bits per message or adjusting the size of the actual physical layer x, the size of the padding can be reduced or done away with completely. However, as we have stressed several times already, this violates the separation of concerns (SoC) principle during the design process and is one of the reasons why large designs are so difficult to get right. Very often the size of x cannot be chosen freely except sometimes in integer multiples, recall for example the inherent byte alignment of PCIe with possible user side widths of usually 32 or 64 bits. This usually presents the user with a tough choice to either segment the messages so that the necessary padding is small or instead suffer significant bit efficiency losses. It also means that once that trade-off has been made in a design any changes to either x or n are met with strong reservations which can stunt design development if for example useful user extensions are discarded or a switch of serializer technology is burdened with user interface changes to maintain bit efficiency.

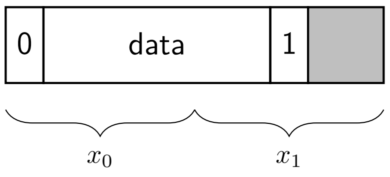
To address this issue we can introduce *variable length commas*.

Definition 5.1.1. Comma If a comma bit is encountered during decoding in a serialized word X, the rest of the word is a comma.

For illustration, let us look at how the serialization of the datagram shown in Figure 5.2 into words of x bits would have to look like without padding.



In this example two user datagrams can fit in three serializer words with 100% bit efficiency. However, this case is not guaranteed to always happen because there might not be two user datagrams available back to back, so we must insert a comma character to mark the rest of the word as invalid if there is only one word available:



We will call commas that need to be inserted within a serializer word *EOF commas* as opposed to *idle commas* which fill a complete serializer word as

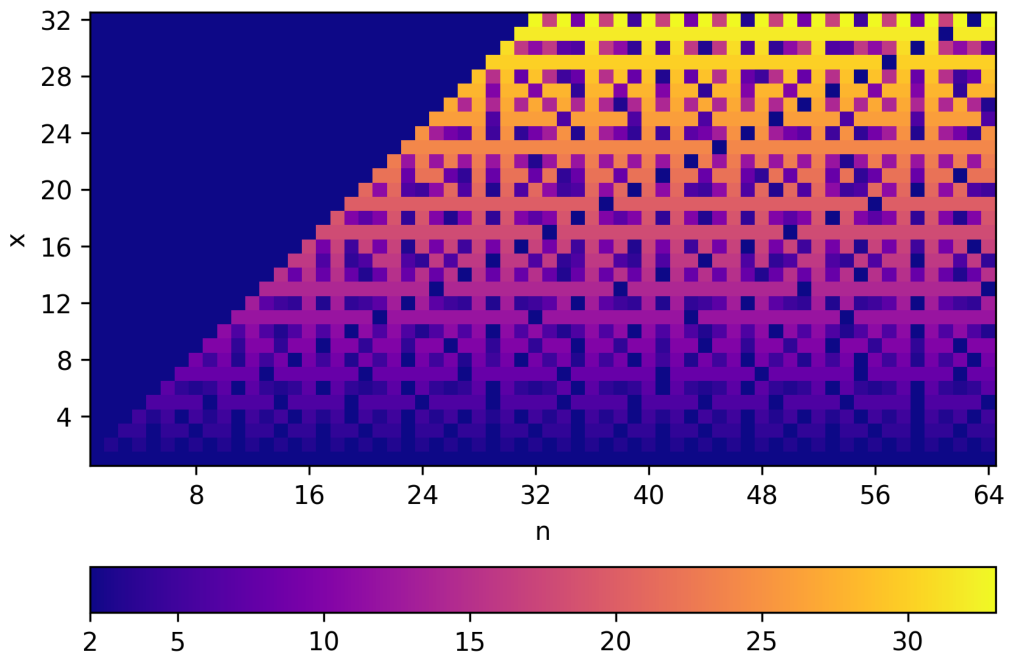


Figure 5.4: Algorithmic complexity of encoding n bit payload into x bit wide code words when using EOF commas and no padding.

shown in Figure 5.3. Notice how there are cases where only a single bit is left over and must be marked as a EOF comma, which is why we cannot use more than a single bit to distinguish between commas and payload. This encoding ensures optimal bit efficiency regardless of the choice of n and x, but is likely to synthesize into very large circuits due to the combinatorial complexity. $N = \frac{lcm(n+1,x)}{n+1}$ consecutive messages of n bit payload are needed to be aligned in x bit wide serializer words without the need of EOF commas. It follows that in all other cases where fewer messages are sent there needs to be a distinct EOF comma. This means that there are N distinct positions a data word must be shifted within the serializer word depending on the state of the encoder. The decision tree of the corresponding state machine therefore consists of $N + 1$ states[3] each having a choice to either insert a shifted datagram into the current serializer word or write a comma and flush the word.

Note the pattern in Figure 5.4: If $n + 1$ and x are co-prime, the required number of states grows linearly with the serializer width x. However, there are always neighboring combinations that produce small decision trees. We can reach these parameters by re-introducing padding into the encoding

[3]Including the idle state

scheme. This time however, the amount of padding is controlled by an additional free parameter C that now represents a smooth trade-off between bit efficiency and encoding complexity. C generates a padding p such that $(n + 1 + p) == 0 \bmod C$.

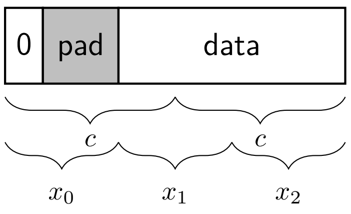


Figure 5.5: Padding example with $C = \frac{3}{2}x$. The datagram is now aligned such that it can fit into 3 code symbols without remainder.

Changing C effectively shifts the point (n, x) in Figure 5.4 to the right which can now be used to hit a low complexity point at minimal bit efficiency penalty. Even more importantly, it can both be used to freely adjust n or x without exploding the encoding complexity or in the opposite, improving bit efficiency of the encoding without the need to adjust the interfaces.

5.1.1 Encoding sum types

When encoding sum types, the enormous opportunity presents itself to be aware what the size of the current payload is and use that information to emit fewer code words if possible. We extend the encoding scheme to contain the tag information of the sum type immediately after the comma bit and insert the padding between the tag and the payload. We also support the edge case of zero width payload, where the payload is omitted and only the corresponding tag is serialized.

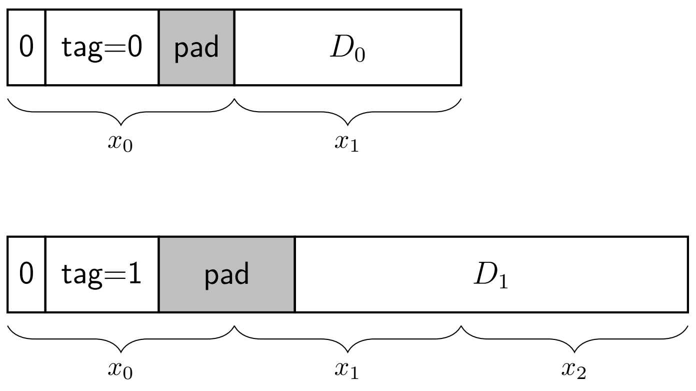


Figure 5.6: Sum type encoding example. The header, consisting of the `tag` field together with the comma bit has a fixed width, but the padding changes depending on the tag. Here, data D_0 with tag 0 can fit into two code words x_i, while D_1 has a larger size and needs three code words for encoding.

As before, the size of the padding depends on the common alignment parameter C. It is now even more crucial because the algorithmic complexity now scales with the co-primality of all the individual member type sizes n_i. A common and rather efficient choice of C would be to force alignment to x of all the sum type members. The trivial and potentially very wasteful choice would be to set C to be the least common multiple of all the n_i as well as x. This results in simple logic because all the members will get encoded exactly the same number of code words regardless of their size.

5.1.2 CRC

A convenient feature of the UT encoding scheme is the option to selectively append CRC data to a serialized datagram. For each member of a sum type we can decide at compile-time whether a CRC should be appended. In the current version we force certain alignment on datagrams with a CRC. The datagram must always start at a code word boundary which possibly forces EOF commas in the previous code word. The padding of the datagram must be such that it is aligned to x while also keeping the alignment to C that is mandatory for all datagrams. The CRC is then appended directly after the last code word and also begins at a code word boundary due to the previously mentioned alignment. It is itself also aligned to x which forces some restriction on the available polynomials. For example, one can only choose the CRC-8-Dallas/Maxim polynomial which has rank 8 in the case where x is either 1,2,4 or 8.

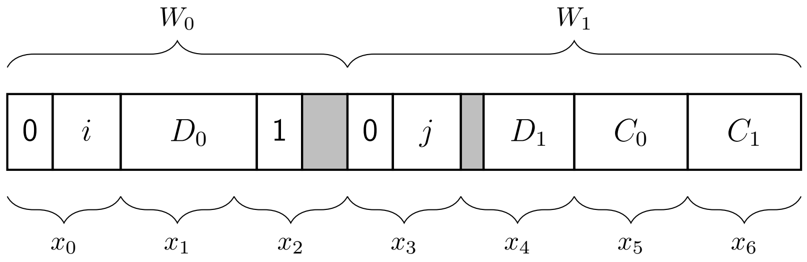


Figure 5.7: Example datagram layout for back-to-back transmissions of an unsecured datagram $W_0 = (i, D_0)$ which is encoded in the code words x_0 through x_2, directly followed by a secured datagram $W_1 = (j, D_1)$ encoded in $[x_3, x_4]$ with appended CRC in code words $[x_5, x_6]$. An EOF comma is inserted at the end of W_0 because the encoding of W_1 must begin at a code word boundary. The CRC is only computed on the two code words $[x_3, x_4]$ and streamed out immediately after.

Usually, the CRC is used in conjunction with rather large payload frames like the 1500 Byte Ethernet packets to minimize the overhead. The downside of this approach is that this immediately necessitates large sending and receiving buffers and also negatively impacts latency because we need to block the decoded data until the CRC check has been passed for the whole frame. The UT encoding scheme employs the CRC as an optional feature that can be used to selectively upgrade the security of certain datagrams while not affecting the bit efficiency of datagrams the user chose not to secure. There are many cases where data does not actually need to be secured as long as the link is stable enough that most of the transmissions are not corrupted. Furthermore, it might be the case that the used serializer actually does some framing of its own internally, in which case it is very useful to easily disable the UT CRC altogether, again without any changes to the interfaces.

Now that we have introduced the encoding scheme, we will discuss its HDL implementation.

5.2 UT sender

The UT `sender` is the encoder module of the UT encoding scheme. It is currently implemented in Very High Speed Integrated Circuit Hardware Description Language (VHDL) due to a better support of functions that can operate on unconstrained arrays, as well as stronger type checking than SystemVerilog. Nevertheless, most modern synthesis tools have good support for cross-instantiation of VHDL modules in SystemVerilog and vice versa as long as certain requirements on the interfaces are fulfilled. The UT `sender` is no exception and thus can be—and in fact has been—synthesized in both

VHDL as well as SystemVerilog environments. Let us first quote the complete module declaration and then move through the individual interfaces and parameters. Some of the types and methods we will encounter are imported from an accompanying utility library; We will quote their definition where necessary.

```
entity ut_send is
  generic(
    TYPELIST       : ut_entry_arr_t;
    PHY_WIDTH      : positive;
    CRC_POLY       : pos_arr_t;
    COMMON_DIV     : positive := 1
  );
 port(
   clock          : in std_logic;

   reset          : in std_logic;

   phy_data       : out std_logic_vector(
                      PHY_WIDTH - 1 downto 0
                    );
   phy_next       : in std_logic;

   has_data       : out std_logic;

   link_valid     : in std_logic;
   link_data      : in std_logic_vector(
                      max(
                         1,max(get_widths(TYPELIST))
                         ) - 1 downto 0
                    );
   link_data_idx  : in integer range
                      0 to
                      get_widths(TYPELIST)'high
                    ;
   link_next      : out std_logic
   );
end;
```

5.2.1 Client interface

The definition of the `ut_entry_arr_t` is as follows:

```
type ut_entry_t is record
       -- size of the type in bits
       width       : natural;
       -- if true, append CRC
       secured     : boolean;
   end record;

type ut_entry_arr_t is array(natural range <>) of ut_entry_t;
```

As we have promised, there are no inherent restrictions neither on the sizes of the individual types nor the length of the type list, the usability of the module is bound by the synthesis process only. Using the `TYPELIST` parameter, we define the client interface consisting of the ports `link_valid`, `link_data`, `link_data_idx`, `link_next`. Note also that `TYPELIST` does not have a default value assigned which causes compiler errors if not set during instantiation. This avoids bugs because there is no useful default value for `TYPELIST` so that the UT could still be used in an arbitrary context. Together they represent the blocking `push` interface of a Queue that accepts sum types as payload consisting of the tuple `(link_idx, link_data)`. The somewhat unwieldy calculation of the size of `link_data` is needed in case all entries in `TYPELIST` have size 0, in which case `link_data` becomes 1 bit wide but will never carry any information. The declaration of `link_data_idx` as a ranged integer is useful because it offers simulation and compile time constraints on the accessed values[4] yet is still compatible with SystemVerilog instantiations.

5.2.2 PHY interface

The PHY interface is the side of the UT `sender` that emits the code words as described in Section 5.1. Three parameters govern it:

`PHY_WIDTH` This sets the width of the code words and is equivalent to the parameter x we introduced in Section 5.1. It can be chosen freely, although

[4]An example is the case where the type list has 5 entries. Synthesis will infer a 4 bit wide port, yet the ranged `integer` declaration can check that the values `[5,6,7]` are not written in simulation.

in practice values larger than 64 are rarely encountered for reasons we will discuss later.

`COMMON_DIV` This parameter was introduced as C in Section 5.1 and is used to calculate the required padding for the individual entries. As discussed earlier, it is a very important parameter because it governs the trade-off between hardware complexity and bit efficiency in a completely independent manner from both `PHY_WIDTH` and `TYPELIST` parameters. It also has a useful default value at 1 which means that no padding is inserted unless forced by the CRC. This also gives the upper bound on the circuit area for a particular instance.

`CRC_POLY` The CRC polynomial can be set using this parameter in form of a list of positive integers. For example, the CRC-8-Dallas/Maxim polynomial $x^8+x^5+x^4+1$ can be selected using the value `(8,5,4)` omitting the $x^0 == 1$ term by convention. It does not have a default value to avoid unintended behavior but is of course not used in cases where no entry in `TYPELIST` has the `secured` flag set.

The PHY interface is formed with the ports `has_data, phy_data` and `phy_next`. While it may resemble a blocking interface like the client interface, the PHY interface is *optionally blocking*. This means that asserting `phy_next` will always produce legal data at the `phy_data` port regardless of the state of the `has_data` port which signifies whether the UT `sender` is currently in the process of datagram transmission or there is data available at the client interface. This is of course only possible due to idle commas that can be sent in cases where there is no datagram transmission in progress or the client interface has no data available. The reason for this somewhat ambivalent interface choice is that the UT `sender` must be compatible with both blocking and non-blocking PHYs.

As we discussed earlier, the PHY does not necessarily have a concept of an idle state by itself, especially in the more simple incarnations like raw serializers without encoding. These PHYs can simply perform their initialization routine and then start asserting `phy_next` at the appropriate symbol rate that they can process. The UT `sender` then ensures encoding synchronization by transmitting idle commas until payload is available at the client interface.

On the other hand, more complex interconnects like PCIe do have their own mechanism to insert idle commas during transition and can thus offer a blocking interface to the UT which can then be used together with the `has_-`

data port to only transmit code words that carry payload. The blocking UT PHY interface can also be used to encode data into memory buffers of some arbitrary alignment which avoids filling it with superfluous commas.

5.2.3 Derived constants

Several parameters like various offsets are computed at compile time and can be used at runtime by the control logic of the module. One such parameter is HEADER_LENGTH which determines the length of the header at the beginning of a datagram:

```
constant HEADER_LENGTH : positive
    := clog2(TYPELIST'length) + 1;
```

Another important parameter is the PADS list that contains the individual padding offsets between each datagram. Its declaration is

```
constant PADS : nat_arr_t(TYPELIST'range)
    := calc_pads(
            TYPELIST,
            COMMON_DIV,
            PHY_WIDTH,
            HEADER_LENGTH);
```

together with the function calc_pads() defined as

```
function calc_pads(
  typelist : ut_entry_arr_t;
  common_div, phy_width, header_len : positive
  ) return nat_arr_t is
   --return value is a list of naturals
   --with the same size as the typelist
   variable pads : nat_arr_t(typelist'range)
      := (others => 0);
begin
  for i in pads'range loop
    --pad everything to be at least one PHY_WIDTH long
    if header_len + typelist(i).width < phy_width then
      pads(i) := phy_width - header_len - typelist(i).width;
    end if;
    --additionally pad secured lengths to be PHY_WIDTH aligned
    if typelist(i).secured then
      pads(i) := pads(i) +
                  pad_to(
                      header_len + pads(i) + typelist(i).width,
                      phy_width);
    --otherwise align to COMMON_DIV
    else
      pads(i) := pads(i) +
                  pad_to(
                      header_len + pads(i) + typelist(i).width,
                      common_div);

    end if;
  end loop;
  return pads;
end function;
```

and an auxiliary function `pad_to(len, align_to)` that calculates the offset needed to make `len` a multiple of `align_to`:

```
function pad_to(len : natural; align_to : positive)
 return natural is begin
  return ( ( align_to - ( len mod align_to ) ) mod align_to );
end function;
```

The `PADS` parameter now finally allows us to find the longest datagram in the encoding:

```
constant WORK_LEN : natural
    := HEADER_LENGTH + max(get_widths(TYPELIST) + PADS);
```

with the `+` operator being overloaded to allow element wise addition of the arrays `get_widthsTYPELIST` and `PADS`. We show these parameters and their calculations partly because we observe a certain reservation within the hardware design community to use complex compile time calculations, which can drastically limit the achievable complexity of designs, for fear of introducing too much complexity for a module to be verifiable. Another common hindrance is the lacking tool support for complex expressions that are still recognized as being synthesizeable because they are compile time only. Instead, the trend seems to go towards code generation where high level languages are used to emit simplified verilog code that is guaranteed to be synthesizeable by the usual tools. We take the opportunity here to demonstrate how complex parametrizations can be achieved in plain HDLs like VHDL[5] and still be synthesizeable with the usual tools as we will see later.

5.2.4 Data path

The UT encoding process can be split up into several stages that the data passed into the user interface has to traverse before it appears at the PHY interface. Due to the polymorphic nature of the UT these stages can result in a wide range of RTL circuits, hence we will attempt to describe them in a general manner and note the influence of certain parameters on the individual stages.

`shift` stage The `shift` stage is, as the name suggests, a shift register that shifts its contents by `PHY_WIDTH` bit thereby generating both the code words `phy_data` as well as the input for the CRC module. Its size is the same as the largest datagram in the encoding scheme depending on the type list and the `COMMON_DIV` parameter, which we will call `WORK_LEN`.

`mask` stage The `mask` stage is a purely combinatorial circuit that can load individual regions of the `shift` stage. A new datagram is only loaded if the

[5]And to a somewhat lesser degree SystemVerilog if one is unwilling to use macros as they are not type safe.

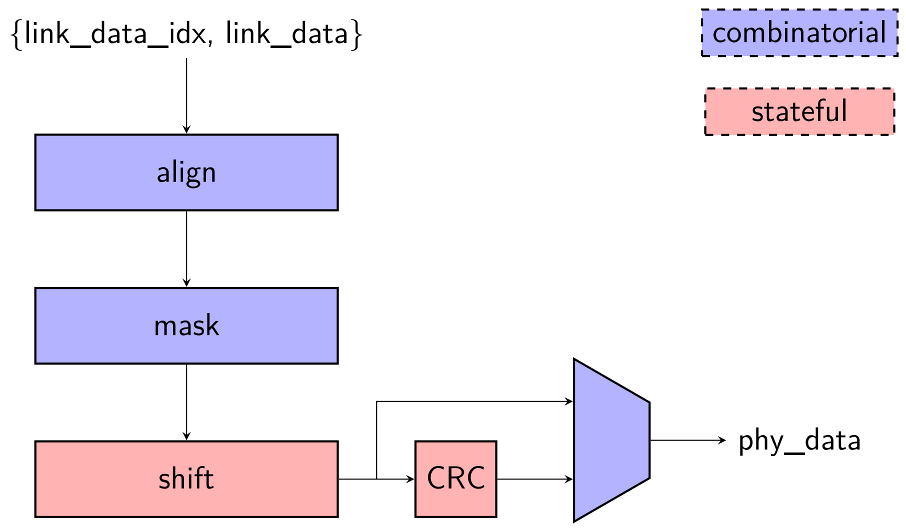


Figure 5.8: Block diagram of the UT `sender` module data path. Stateful stages like shift registers have internal state as opposed to combinatorial stages which do not. Regardless, it is always possible to insert delay stages between two processing stages to improve the critical path of the circuit although they are omitted in this diagram.

`shift` stage has less than a full `PHY_WIDTH` word in it left. Its computational complexity is rather low because there are only as many different masks as there are EOF commas. At the worst possible case there are `PHY_WIDTH - 1` of them if all datagram lengths are co-prime with each other and `PHY_WIDTH` itself.

`align` stage The `align` stage creates the datagrams out of the tuple `link_-data_idx` and `link_data` from the user interface. Depending on `link_-data_idx` and the current state of the `shift` stage the inputs must be shifted not only an absolute number of bits but also relative to each other to create the datagrams as specified by the UT encoding scheme. That shift is however static and does not require any internal state in the `align` stage, so it can be—and currently is—implemented in purely combinatorial logic. This is currently by far the most expensive part of the UT `sender` because it depends on `PHY_WIDTH`, `COMMON_DIV` as well as both the length of the type list and its maximum size. Barrel shifters can be used here as they enable shifts of any bits, but this approach is not without peril since barrel shifters are usually highly optimized dedicated circuits and are only available for certain sizes. We freely admit that the current implementation leaves much room for improvement when it comes to area and maximum frequency since we focused mainly on the feasibility and correctness of the design, and it shows most glaringly in the implementation of the `align` stage.

CRC stage The CRC stage consumes data in PHY_WIDTH chunks and subsequently shifts out the CRC it calculated on the data since last reset at its output where it can be switched from the code words themselves by an output multiplexer. This explains the forced alignment of all datagrams that require a CRC on PHY_WIDTH because otherwise the CRC might be calculated not only on the datagram itself but possibly on parts of the previous or subsequent datagrams.

5.2.5 Control path

The stages shown in Figure 5.8 are controlled by additional logic that is often called the control path of a module. It also controls the interfaces by asserting link_next to accept new data from the user, as well as producing new phy_data when phy_next is asserted. Again, the polymorphic nature of the UT encoding presents certain challenges because we have to devise a finite-state machine (FSM) with a variable number of states depending on the parametrization.

There are two main states the UT sender needs to track, how much data is left in the shift stage as well as the state of the CRC stage. For the shift stage we use a pointer approach and instantiate an integer workidx that points to the first unoccupied bit position in the shift register work. Their VHDL declaration are as follows:

```
signal work is std_logic_vector(WORK_LEN-1 downto 0);
signal workidx is integer range -1 to WORK_LEN - 1;
```

The workidx register is used both to track if there is enough space in work for a new datagram as well as where new data needs to be loaded within the shift stage. In VHDL we can define a generic procedure write declared as follows:

```
procedure write(
            work : inout std_logic_vector;
            idx : in natural;
            payload : in std_logic_vector
            );
```

Listing 6

It allows us to write some bit vector `payload` into a bit vector `work` starting from the bit position `idx`. Effectively, the `align` and `mask` stages accomplish the `write` of a new datagram into the `work` register at various bit positions `workidx`. The algorithmic complexity of the UT encoding and thus the required circuit area is mostly dependent on how many different `write()` calls must be supported, i.e, how large are the various `payload` vectors and at which different `workidx` positions must they be written. This is also why the `COMMON_DIV` parameter is so helpful to contain the area usage because it homogenizes both the length of the various `payload` datagrams as well as their loading locations thereby reducing the logic tree.

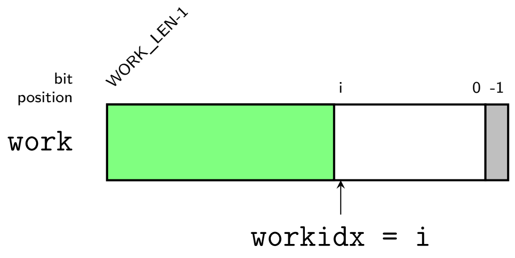


Figure 5.9: Diagram of the UT `sender shift` stage. Showed is the `work` register partially loaded with payload from its high bit position `WORK_LEN - 1` until bit position `i+1` marked in green. The control path register `workidx` contains the address of the first unoccupied bit position in `work i`. The edge values `workidx == WORK_LEN-1` and `workidx == -1` mean that the `work` register is empty or full respectively.

The CRC stage is managed by a simple state machine that tracks whether the current datagram has to be checksummed and then activates the CRC stream out by switching the multiplexer towards `phy_data` (see Figure 5.8). The required state registers are

```
type crc_state_t is (idle, active, draining);
subtype crc_cnt_t is integer range 0 to POLY_SIZE/PHY_WIDTH - 1;
```

where `POLY_SIZE` is the degree of the selected CRC polynomial. The `crc_cnt` register is needed during the `draining` stage to track the amount of `PHY_WORD` units that are being streamed out by the CRC stage. The CRC stage itself is essentially an instance of the `DW_crc_s` module from the Synopsys DesignWare® library (Syn). While we would prefer to use an open implementation we are not aware of any such modules that are silicon proven and support various polynomials. Nonetheless, we encourage the community to develop such a design so that it may be used in a future version of the UT[6].

5.3 UT receiver

This module performs the inverse operation to the UT `sender` introduced above; It consumes a stream of `PHY_WORD` sized code words and parses them into valid datagrams while discarding comma characters. In an attempt to avoid repeating ourselves, we will only discuss here the particularities of the `receiver` since the strong symmetry to the `receiver` is reflected not only in the functionality but also in the structure and features. Nevertheless we will also provide a full module declaration first:

[6]Another reason why we opted for a closed intellectual property (IP) is a currently unanswered question whether the CRC even should be the preferred method to secure UT datagrams. As we discussed previously there are many ways to ensure datagram integrity and the CRC mechanism may ultimately prove a sub-optimal solution. As long as this topic has not been finally addressed we feel justified to pick an off-the-shelf solution to accelerate implementation

```
entity ut_recv_base is
  generic(
    TYPELIST       : ut_entry_arr_t;
    CRC_POLY       : pos_arr_t;
    PHY_WIDTH      : positive;
    COMMON_DIV     : positive := 1;
  );
  port(
    clock          : in std_logic;
    reset          : in std_logic;
    phy_valid      : in std_logic;
    phy_data       : in std_logic_vector(
                         PHY_WIDTH - 1 downto 0);
    link_valid     : out std_logic;
    link_next      : in std_logic;
    link_drop      : out std_logic;
    link_data      : out std_logic_vector(
                        max(
                            1,
                            max(get_widths(TYPELIST))
                            ) - 1 downto 0);
    link_data_idx  : out integer
                        range 0 to get_widths(TYPELIST)'high;
    crc_failed     : out std_logic;
    decoding_err   : out std_logic
  );
```

Recall our discussion of the PHY interface of the UT `sender` in Section 5.2.2; We have emphasized that the `sender` must be able to create sensible `phy_data` regardless of the assertion pattern of `phy_next` because it is otherwise not universally usable with all possible serialization techniques such as described in Chapter 3. The inverse problem for the UT `receiver` is that it cannot anticipate the PHY input pattern in all cases and still be generic. At the same time, losing code words because of bottlenecks will immediately disrupt the decoding process which would require a wasteful link re-initialization. The most important design requirement for the UT `receiver` is thus that *it must never be a bottleneck* between the PHY and the user. The `receiver` must thus be able to process code words with a throughput of at most `PHY_WIDTH` bits per `clock` cycle, but also any rate or pattern below that maximum.

This also means that the receiver is *weakly blocking* at the user interface; It is able to hold an assembled datagram and assert link_valid until link_next acknowledgment, but will discard it if new phy_data is available before link_next was asserted as it is more important to preserve the parsing synchronicity than losing a single datagram.

Furthermore, since the UT receiver may be getting the phy_data from an unsecured Queue, there is always the possibility that code words were corrupted, are missing or reordered, which almost surely will disrupt the decoding process. As we have discussed earlier, the UT encoding scheme lacks uniquely identifiable commas that can be detected in an unstructured code word stream. Instead, the UT module rely on a synchronized state that is implicitly modified every time a code word is sent or received. If that state is de-synchronized because the receiver receives the wrong code word sequence it may very well happen that an incorrect header is decoded either from the wrong code words or from a corrupted code word which can lead to various scenarios.

The receiver may decode a header it recognizes and start collecting additional code words to assemble the datagram since the tag inside the header tells it how long the datagram must be. Then, in case that particular tag was secured via a CRC there is a very high probability that the check will fail so the receiver can discard the datagram, pulse crc_failed to notify the user, and start parsing anew. If however the tag was not secured it will present potentially completely wrong data—both in the tag as well as the payload—to the user. Recall however, that SUMTYPE is not required to have a power-of-two size but the tag is encoded within $clog2($size(SUMTYPE)) bits. If the receiver attempts to decode the wrong bits into a tag it might decode a header that does not correspond to a valid tag of the sum type with no associated type length. This is an illegal state that the UT encoding cannot recover from by itself, so the only option for the receiver is to assert the decoding_err port and stop further parsing.

The following timing diagrams show the various scenarios that may occur and what the expected behavior is. We omit all payload ports for the sake of clarity, and focus instead on the control flow of the interfaces.

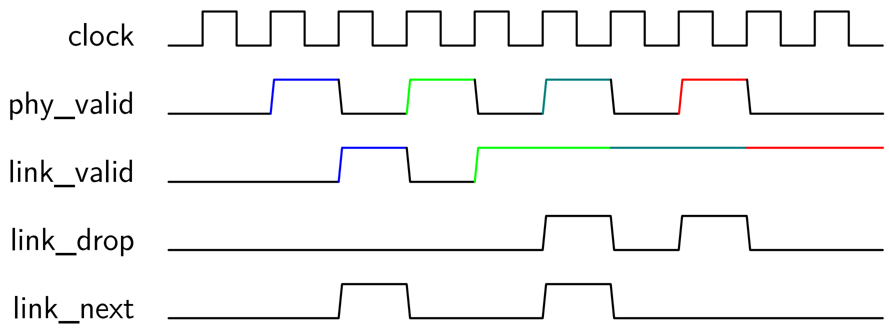


Figure 5.10: Example transactions demonstrating the blocking interface of the UT `receiver`. Assume an encoding where every code word contains exactly one datagram. `link_valid` is asserted delayed by one cycle with respect to `phy_valid` to announce new user data. Matching `phy_valid` and `link_valid` colors indicate which datagram the `link_valid` refers to. The first transaction is the fast case; `link_next` is asserted in the same clock as `link_valid`. The second transaction is the slow case; `link_next` is asserted one clock later than `link_valid`. Since there is already new `phy_data` available, as indicated by the assertion of `phy_valid`, `link_drop` is asserted to signal that this is the last chance for the user to capture the second transaction. The third datagram was lost because `link_next` was not asserted quickly enough. Unless new `phy_data` is arriving, the `receiver` will hold a valid datagram until acknowledged by the user via `link_next`.

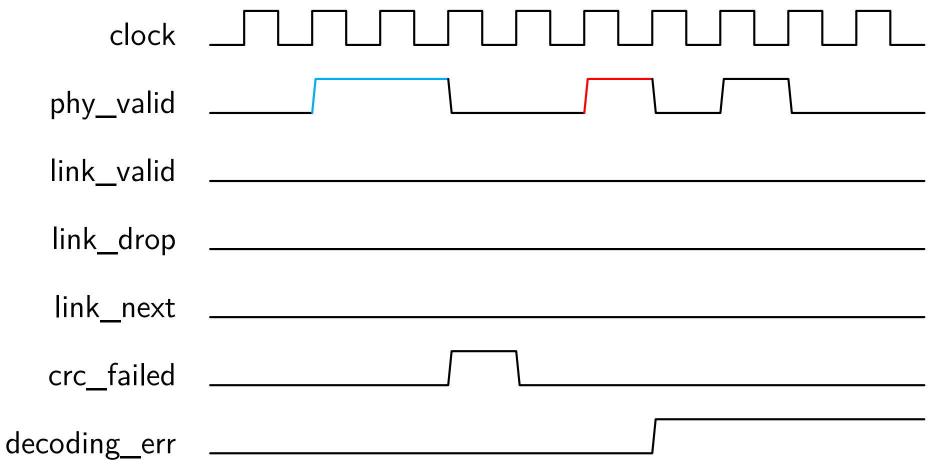


Figure 5.11: Examples of processing corrupt code word streams. The first transaction was decoded as a legal datagram, albeit with a broken CRC. Note that the `receiver` cannot determine whether the header was correct and there was a bit flip somewhere in the payload or if instead a wrong header was decoded into a legal tag but non-matching CRC. The second transaction could not be decoded into a valid header, so the `receiver` simply raises the `decoding_err` flag until reset. Any subsequent incoming code words are ignored.

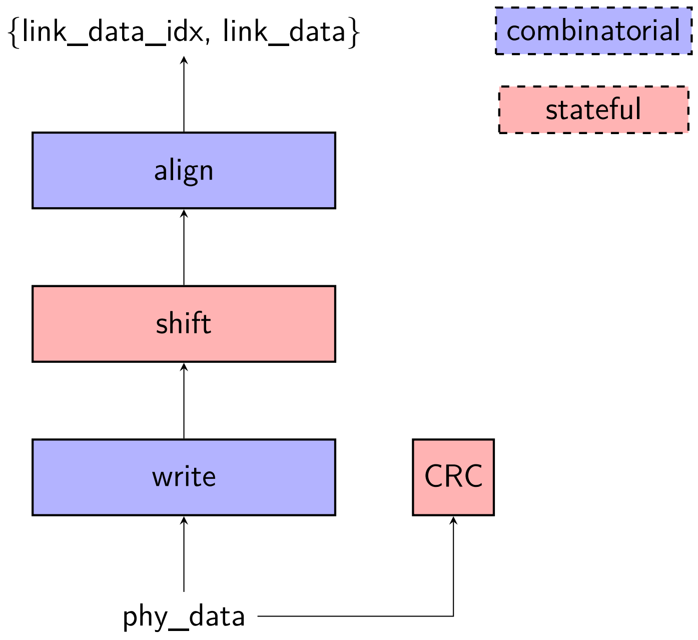


Figure 5.12: Block diagram of the UT `receiver` module data path. The CRC module is only used for data checks and has no data path relevant outputs.

5.3.1 Data path

The data path is naturally reminiscent of the UT `sender`. The `shift` stage is where the datagram is assembled, and is a register large enough to accommodate the amount of code words needed to construct the largest datagram. The `work` register of the UT `receiver` has the following declaration:

```
signal work : std_logic_vector(
                WORK_LENGTH + pad_to(WORK_LENGTH, PHY_WIDTH)
                - 1 downto 0);
```

We re-use the `WORK_LENGTH` parameter from the UT `sender` as well as the `pad_to()` function which were both introduced in Section 5.2.3. It is loaded via the `write` stage that can set any contiguous `PHY_WIDTH` bit within the `work` register with the incoming `phy_data`. The datagram is left-aligned such that bit `work'high` coincides with the Most Significant Bit (MSB) of the header. Using a counter `workidx` to store the current load state of the `work` register as described in Figure 5.9 as well as the `work()` procedure defined in Listing 6, the `write` stage performs an access of the form `write(work, workidx, phy_data)`. Once enough data has been collected, the `align` stage extracts the payload and tag out of the datagram and moves them to the output ports. In case the datagram was not aligned a check must be performed to determine whether the rest of the code word contained an EOF comma or the beginning of a new datagram. Should there be valid data left over, a shift is performed that aligns it to the left. At most `PHY_WIDTH-1` bits must be copied from various bit positions within `work` to realize this data shift, which subsequently also can mis-align the `write()` position during code word reception. All in all, the `receiver` can turn into a full-blown barrel shifter depending on the `TYPE_LIST` and the `COMMON_DIV` parameters, so great care should be taken in evaluating its operating characteristics.

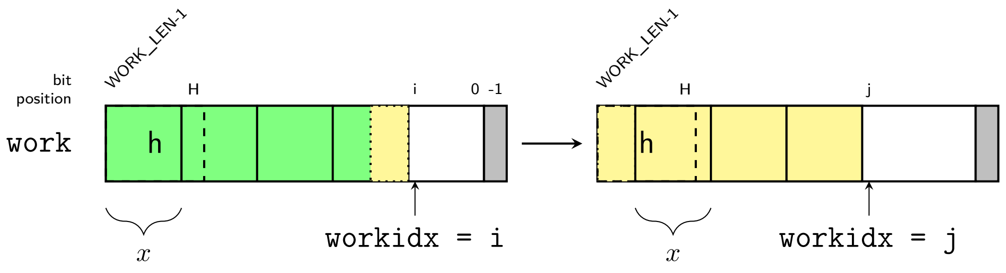


Figure 5.13: Example of unaligned datagram decoding. Shown is the `work` register partially loaded with code words x. Bit positions `[WORK_LEN-1:H]` comprise the header and are used to decode the datagram. The header can be larger than a single code word. Code words are appended to the right starting at the first unoccupied position denoted by the `workidx` which is then subsequently moved. The first datagram marked in green is smaller than four full code words and the last code word already carries a part of the second datagram marked in yellow. After the first datagram has been assembled and moved to the user output, the overhanging part of the second transaction is copied to bit position `WORK_LEN-1` of the `work` register. Three more full code words are needed to complete the datagram.

5.3.2 Control path

There are three principal components that determine the state of the UT `receiver`: the fill state of the `work` register, the CRC state as well as the state of the header, i.e, the value of the first `HEADER_LENGTH` bits of `work`.

The `workidx`, like in the UT `sender`, is a pointer towards the first unoccupied bit position within the `work` register. It serves both to track the occupancy of the `work` register as well as the position that a new code word can be written to. When `work` is empty, i.e, the condition `workidx == WORK_-LEN-1` holds, the next code word contains either the beginning of an aligned datagram, or an idle comma which can be recognized by its MSB being set. In this case the `workidx` does not move which discards the idle comma.

The `header` field are the left most `HEADER_LEN` bits of the `work` register. The header is valid only after the `workidx` has moved past it and is used to determine whether it contains a legal value as well as the length of the datagram.

Code words are continuously fed into the `crc` stage (see Figure 5.12) upon reception. Its only output is the `crc_ok` flag that is raised only[7] if all the code words of a datagram including the CRC have been passed into it. Since all secured datagrams must be aligned to `PHY_WIDTH`, the `receiver` can simply re-set the `crc` stage whenever an aligned code word is received. If the `header` indicates that the datagram carries a CRC, the `crc` state, which is simply a counter, is used to prevent writes of the code words containing the CRC. A secured datagram is only presented at the user output if, and only if, the value of `workidx` matches the datagram length decoded via the `header`, the `crc` counter indicates a completed reception, and `crc_ok` is raised.

5.4 Synthesis Example

To demonstrate the flexibility that the UT codec modules provide, we will discuss the serialization of a specific real-world sum type. It contains the types required for the communication with the current-generation neuromorphic hardware of the Electronic Vision(s) Group at the Heidelberg University, the High Input Count Analog Neural Network with HAGEN Extensions (HICANN-X) chip, which we will discuss in more detail in the next chapter. In particular, we will discuss the sum type in the to-chip direction, which we will denote by $\overrightarrow{S_x}$. For now, it is sufficient to understand that this benchmark is not artificial but rather an example of the decisions a design team may make when offered a new paradigm where universality and clarity are paramount.

$\overrightarrow{S_x}$ is a sum of thirteen types which were mostly chosen independently from the underlying serialization but instead mainly represent the data units the individual modules within the HICANN-X natively consume. The sizes of the individual types are not fixed, but rather derived from parameters and are thus dependent on the particular incarnation of the chip. In the current versions, the sizes of the types within $\overrightarrow{S_x}$ are in bits: `[24, 48, 72, 41, 40, 72, 40, 72, 40, 64, 8, 1, 0]`. The entries are not unique because two different types can have the same size, and vice versa, since if the same type belongs to different Queues it must also have multiple entries to distinguish it via the tag.

[7]Disregarding false positives that are mostly dependent on the size of the CRC polynomial

One detail we will omit for clarity in this discussion is the CRC scheme for $\overrightarrow{S_x}$. As we previously noted, the UT encoding scheme allows us to selectively secure individual types within a sum type, and this is utilized for $\overrightarrow{S_x}$ as indeed not all of these types are secured for HICANN-X. However, the choice of the CRC polynomial does depend on a specific `PHY_WIDTH` parameter and is furthermore dependent on the implementation, which we previously acknowledged to be commercially available IP. We will instead focus on the gearboxing and encoding part of the UT itself, and will thus synthesize modules that serialize $\overrightarrow{S_x}$ for various values of `PHY_WIDTH` and `COMMON_DIV` respectively but do not apply a CRC anywhere[8].

5.4.1 Experiment Setup

We synthesize a UT codec pair consisting of each an `sender` and `receiver` module using the Synopsys™ Design Compiler® Graphical tool in the version `2018.06-SP3 for linux64` for the `tsmc65lp` technology node. We choose an ASIC synthesis in favor of an FPGA implementation because the architecture of an FPGA will obfuscate the effect we want to demonstrate, namely how the choice of `(PHY_WIDTH, COMMON_DIV)` affects the resulting area, because there is now the additional effect how well the design fits in the particular look-up table (LUT) network. We synthesize for a rather low clock frequency of 100 MHz which however still yields code word rates up to several hundred Mega Bytes for large values of `PHY_WIDTH`. We choose `PHY_WIDTH = {1, 8, 32, 64}`, since as we noted before, these are the common widths of serializer user interfaces. For each `PHY_WIDTH`, we sweep `COMMON_DIV` from 1 to C_{max}, which is calculated as follows:

```
def Cmax(S, phy_width):
    #calculate length of header
    #log2(S) + 1 comma bit
    header_length = len(S - 1).bit_length() + 1
    return _align(header_length + max(S), phy_width)
```

For each `PHY_WIDTH` and sum type S, C_{max} calculates the padding such that all the datagrams are of the same length and also a multiple of `PHY_-WIDTH`. This results in the simplest encoding logic but also large unused data fields that are only used for padding (see Figure 5.14).

[8]Note that we may forego using a CRC at the link level in a future version of HICANN-X simply because the PHY might be word secure by itself.

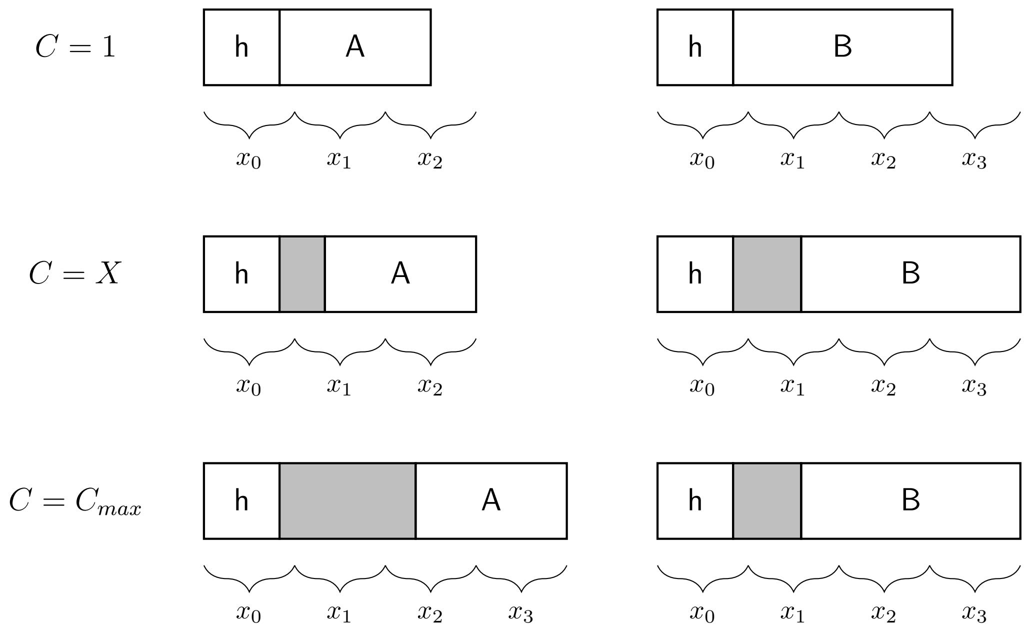


Figure 5.14: Various alignments `COMMON_DIV` $= C$ of two datagrams for a certain `PHY_-WIDTH` size X. The case $C = 1$ preserves the individual sizes of the types and has the highest bit efficiency for the UT encoding scheme because no padding is used. When $C = X$, padding is added such that every datagram is aligned to `PHY_WIDTH`, which eliminates the need of EOF commas and greatly reduces the logic complexity of the codec. When $C = C_{max}$, padding is such that all datagrams are the size of the largest datagram in the case of $C = X$. This produces the simplest logic but results in the lowest bit efficiency as entire code words may contain only padding data.

As we have mentioned previously, the `COMMON_DIV` parameter represents a trade-off between logic complexity and bit efficiency. The average bit efficiency for S depends on the average pad size for a particular tuple `COMMON_DIV, PHY_WIDTH` as well as the length of the header. It can be calculated via the function `calc_biteff(S, phy_width, common_div)`, declared as follows:

```
# returns the datagram sizes
# for a sum type S
def calc_datagrams(S, phy_width, common_div):
    headerlen = (len(S)-1).bit_length() + 1
    retval = []
    for width in S
        val = width + headerlen
        # evey datagram must be at least
        # phy_width bits large
        if val < phy_width:
            val = phy_width
        retval.append(_align(val,common_div))
    return retval

# average bit efficiency for S
# returns an averaged a_i/d_i
# where a_i is the individual size in S
# and d_i is the datagram size for a_i
# depending on phy_width, common_div
def calc_biteff(S, phy_width, common_div):
    return sum(\
      # element wise division
      np.divide(\
        S, calc_datagrams(S, phywidth, common_div))\
      )/len(S)
```

In practice, the bit efficiency will also depend on the actual transmission pattern, i.e the subset of $\overrightarrow{S_x}$ that is used the most during operation. However, it will still follow the general trend of the full average, i.e an—albeit somewhat artificial—scenario, where all of $\overrightarrow{S_x}$ is used evenly.

A related measure is the amount of code words needed to transmit a type in $\overrightarrow{S_x}$ for a particular parametrization `COMMON_DIV, PHY_WIDTH`. It is a measure of encoding latency, since the UT codec can at most consume or produce one code word per clock cycle. We can again construct an average latency for $\overrightarrow{S_x}$ via `avg_latency(S, phy_width, common_div)`, declared as follows:

```
def calc_wordeff(widths, cd, phy_width):
    # average element-wise division
    # of calc_datagrams(widths, cd, phywidth)/phy_width
    return sum(\
            np.divide(\
                calc_datagrams(widths, cd, phywidth),\
                phywidth)\
                )/len(widths)
```

5.4.2 Results

Figure 5.15 shows the results of the synthesis sweep together with the bit efficiency and code word sizes. As we expected, the choice of `COMMON_DIV` has a significant impact on the synthesized chip area. This is due to the varying combinatorial complexity which is proportional to the number of different sized datagrams as well as EOF commas as discussed previously in Section 5.1.

The bit efficiency predictably shows a general trend downwards, since the pads of the datagrams tend to grow with a larger `COMMON_DIV`. It jumps upwards whenever `COMMON_DIV` divides into one or several datagrams which then reduces padding. As intended, it is possible to very favorably trade chip area for bit efficiency for any `PHY_WIDTH` X.

To show a particular example, the HICANN-X employs UT codecs with `PHY_WIDTH = 8`. In the first version, a `COMMON_DIV` of 1 was chosen for maximum throughput. Later versions of HICANN-X increased that parameter to 8 which sacrificed about 6% average bit efficiency and latency for over 30% area decrease (see Listing 7).

Version	`COMMON_DIV`	Area [μm^2]	avg. bit eff. [%]	avg. latency
1	1	6855.48	75	5.69
2	8	4671.72	71	6

Listing 7: UT codec differences between HICANN-X version 1 and 2.

Still, the most attractive feature of the UT codec remains the fact that these trade-offs are possible while remaining agnostic to the user interface. There is no need to artificially change type sizes within $\overrightarrow{S_x}$ to try and improve alignment, and it is also possible to encode sum types into any code word sizes without affecting the user interface. The UT encoding achieves true separation of concerns for a link layer which we so sorely missed in contemporary architectures.

5.5 Conclusion

After introducing the UT encoding scheme and the corresponding RTL implementation of the codec, we now have a way to tunnel arbitrary sum types S through a tunneling Queue Q of any width. If Q is stream secure, the UT codec will simply facilitate the gearboxing of the datagrams as well as provide flow control via commas regardless if the Queue is blocking by itself. Should Q however be unsecured, the UT codec can selectively upgrade any entries of S to be word secure via CRC that are calculated and appended during the encoding process. The receiver provides a link status interface that can notify the user that corrupt data has been received which then can prompt her to re-initialize Q in case of unrecoverable decoding errors. A free alignment parameter can be used to trade bit efficiency for chip area without the need to modify either S or Q.

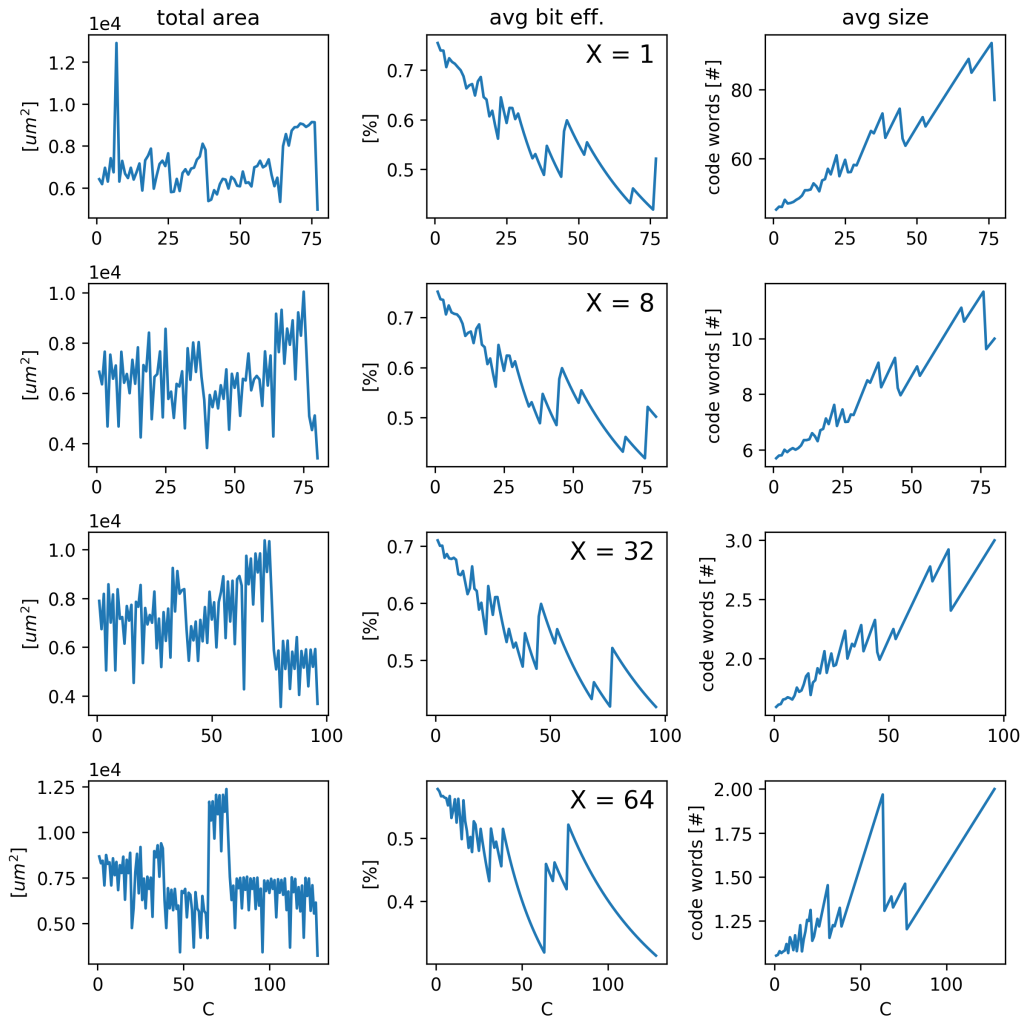


Figure 5.15: Synthesis sweep for the UT codec of $\overrightarrow{S_x}$ for various parameters `(PHY_WIDTH, COMMON_DIV) =` (X, C). Figures in the same row share the same value for X and the same range for $C = [1, C_{max}(\overrightarrow{S_x}, X)]$ The leftmost column shows the total synthesized areas of the encoder and decoder. Center column shows the averaged bit efficiency as a function of C. Right most column shows the average code word length for $\overrightarrow{S_x}$.

Chapter 6

Stream secure Queues

Whereas the UT codec gives us word secured Queues, we are still missing the highly desirable stream security feature for Queue tunneling. Following the spirit of the UT, a potential solution should not only be parameterizable on any sum type, but also provide parameters that allow us to tune the protocol to a wide range of application scenarios.

6.1 ARQ revisited

The Automatic Repeat-Request (ARQ) sliding window protocol is successfully used in the Electronic Vision(s) group in several projects since at least 2008 (Karasenko, 2011, 2014; Philipp, 2008). It employs backward error correction by storing data in a replay buffer and implementing a sliding window protocol using sequence numbers (SEQs) as additional meta data. It is flexible enough to be used as a transport layer both for Host communication as well as in custom chip-to-chip links (Karasenko, 2014). The latter incarnation implements a *tinypacket* version of the ARQ that appends a SEQ to the individual words pushed into it instead of collecting several words into larger packets that carry a single SEQ. This structure already looks promising, since it not only provides the FIFO interfaces of a parameterizable Queue, but is also very easy to tunnel through another Queue of an appropriate width because individual user words are also individual TLPs. In case the tunneling Queue is at least word secured this immediately creates a stream secure Queue.

A slightly awkward property of the ARQ, and indeed any error correction mechanism, is the need of a back-channel over which the receiver transmits acknowledgment numbers (ACKs) to the sender upon successful reception. This back-channel has far lower bandwidth requirements, because it only transports ACK information back to the sender, which is at least two ACKs per transmit window. The *tinypacket* ARQ implementation within the BrainScaleS neuromorphic hardware (BSS) system symmetrized this by *piggy-backing* the ACKs on top of normal payload datagrams and adding a header bit that signified whether the payload is valid. While this was a reasonable design choice because one typically wants a duplex connection between chips anyway[1], it also creates potentially wasteful serializations because the data and SEQ fields are transmitted even if they are not valid. Even if one devises a serialization technique that partially decodes the TLP and only serializes the actual data that is needed, this link layer stands little hope to be in any way generic.

```
typedef struct packed
{
    logic [DATA_WIDTH-1:0] data;
    logic                  seqv;
    logic [SEQ_WIDTH -1:0] seq,
                           ack;
} tinypacket_t;
```

Listing 8: Parameterized TLP declaration of the `tinypacket_duplex` ARQ module. `DATA_WIDTH` is the size of the FIFO interfaces at the client side, `SEQ_WIDTH` is dependent on the Bandwidth-Delay Product (BDP) of the interconnect. The `ack` field is piggy-backed from the back-channel. A `seqv` bit is needed to distinguish between valid payload and ACK-only packets since ACKs are always valid.

6.2 Dynamic timeouts

The latest significant feature introduced to the ARQ implementation are dynamic timeouts. Two timers are active during operation: The SEQ timeout in the master, and the ACK timeout in the corresponding target.

SEQ timeout Also called the resend timeout, is the time the ARQ master waits until the first outstanding SEQ is re-sent. If it is too short, the master will re-sent data that was never lost but whose ACKs simply have not reached the master yet thereby congesting the link. If it is too large, the master will not start re-sending lost data quickly enough which wastes bandwidth.

[1]The two data channels are otherwise logically completely independent

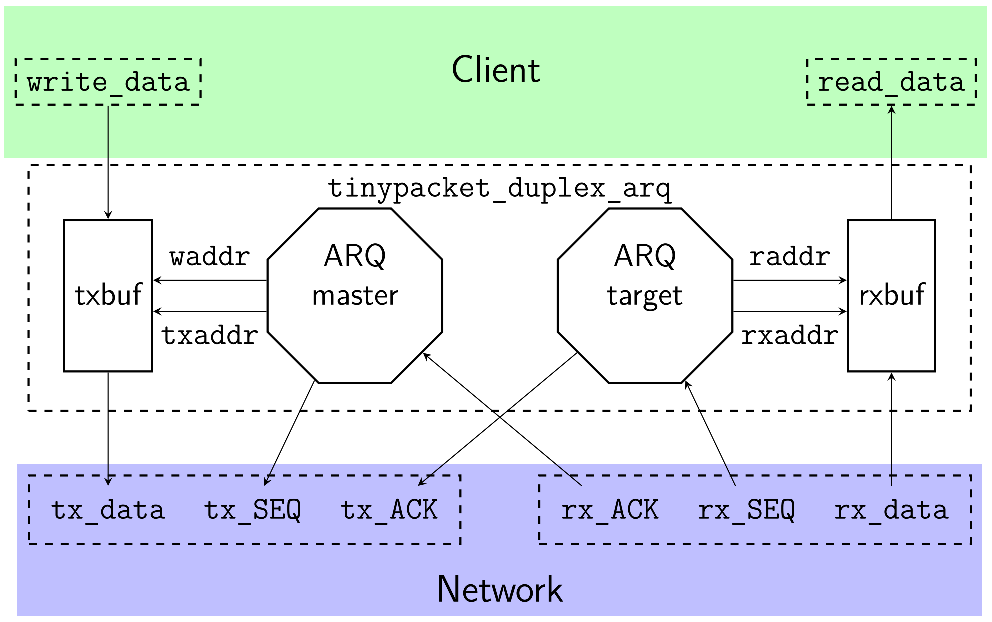


Figure 6.1: Block diagram of the `tinypacket_duplex` ARQ module. It is connected via blocking FIFO-like interfaces to the Client and Network layers (handshake ports omitted for simplicity). Every word at the Client layer is treated as an atomic network packet by the ARQ and transported on the Network layer with additional SEQ and ACK fields used for data recovery.

ACK timeout This is the time that the ARQ target waits after determining that an ACK needs to be sent to further accumulate ACKs before triggering the transmission of an ACK-only packet. To maintain throughput, the ACK timeout should be such that the master receives at least the ACKs for half of its window when it is just about done with sending the full window. Again, if the timeout is to short, the target will waste bandwidth in the back-channel by congesting the link with unnecessary ACKs.

Both of these timeouts used to be static, i.e they could be manually set by the user but did not change afterwards. This posed a problem since the optimal value of the timeouts is heavily dependent on the state of the link at runtime. The link could be congested, thereby increasing the delay and decreasing the throughput, or the client at the receiving end point could be blocking which causes a stall of the protocol.

These issues were addressed by Gaëtan Delétoille (Deletoille, 2016), who replaced both static timeouts with dynamic versions that probe the link state and set the timeout counters accordingly.

For the resend timeout, the master now maintains running averages of measured round-trip times (RTTs) and ACK frequency and sets the timeout to be a weighted sum of the latest values. We also now employ a variant of the *Slow Start* mechanism described in (Allman et al., 2009): Each time a packet loss is detected, the current window is halved until new ACKs start flowing in where the window is subsequently reset to the maximum value.

As for the ACK timeout, the target now maintains a running average of the incoming packet rate to get an estimation of the current link throughput at the receiving end. It then sets the timeout to be

```
timeout_cnt <= WINDOW_SIZE/MAX_ACKS_PER_WINDOW*estimation;
```

using the parameter `MAX_ACKS_PER_WINDOW`, which rather self-explanatory determines the target ACK rate at runtime conditions.

6.3 Sum type ARQ

The concept of sum types very naturally carries over to the ARQ *tinypacket* implementation. Since the dynamic ACK timings now make sure that the throughput is optimized at runtime with as few ACKs as possible, the inclusion of a mandatory ACK field in the network datagram seems to be rather inefficient. Instead, we can create a sum type that either carries user data or an ACK.

Going even further, we can also introduce sum types at the client layer simply by reinterpreting the flat bit arrays `write_data, read_data` as packed structs containing the tag and data of the sum type as described in Section 4.1. A sum type S at the client layer now generates a derived sum type S'(SEQ_WIDTH), where every entry in S is now appended by a SEQ field of size `SEQ_WIDTH`. In a duplex configuration, S' is then further merged with an ACK-only type that carries the back-channel ACKs for the other direction.

This gives us now a generic building block for tunneling arbitrary sum types, and therefore an arbitrary amount of arbitrarily sized Queues, with stream security as long as the network side of the ARQ module is at least word secure. This obviously fits very well together with the UT modules and now gives us the core functionality of a generic link layer which can be used to transport any protocol using any serializer.

To demonstrate this generic behavior, we will now proceed with the discussion of the HICANN-X communication infrastructure.

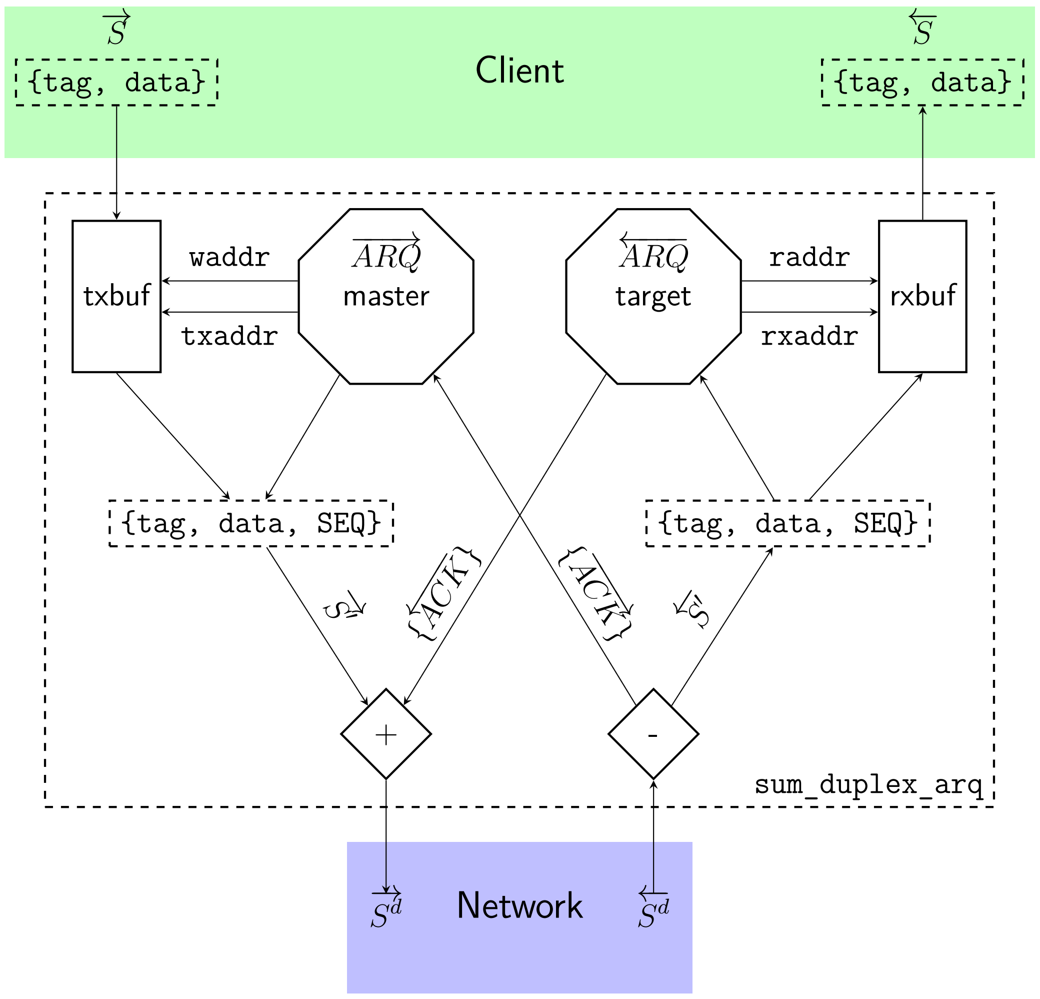


Figure 6.2: Block diagram of the `sum_duplex` ARQ module. The client and network layer ports now represent sum types instead of raw bit strings. The client sum types $\overrightarrow{S}$ and $\overleftarrow{S}$ can be entirely independent, as well as the parametrization of the two ARQ channels. The outgoing duplex network sum type $\overrightarrow{S^d}$ is a sum of $\overrightarrow{S'}$ and the back-channel type containing the $\overleftarrow{ACK}$, the inverse is true for the incoming network sum type $\overleftarrow{S^d}$.

Part III

HICANN-X

Chapter 7

Overview

The HICANN-X is the latest incarnation of the accelerated mixed-signal neuromorphic devices build by the Electronic Vision(s) group at the Heidelberg University. It is part of the BrainScaleS-2 neuromorphic hardware (BSS-2) family of these devices, manufactured in the 65 nm TSMC process and its most advanced member to date. The BSS-2 devices pair asynchronous analog continuous-time neuron circuits connected by a programmable synapse memory crossbar with a Plasticity Processing Unit (PPU) (Friedmann et al., 2013) that can access them to employ various plasticity algorithms, but also perform general-purpose tasks like device configuration and monitoring as needed. The HICANN-X embeds two PPUs, each controlling two quadrants each consisting of 128 neuron circuits and 128×256 synapses containing the weight storage and correlation measurement circuitry.

Substantial literature has been published on the principles of the BSS approach (Brüderle et al., 2011; Schemmel et al., 2010; Schmitt et al., 2017), and in particular the BSS-2 architecture (Aamir et al., 2017; Billaudelle et al., 2019; Friedmann et al., 2017; Schemmel et al., 2017) as well as their predecessor called Spikey (Pfeil et al., 2013). Furthermore, a growing number of manuscripts is available on the HICANN-X, both from an experimental as well as architectural perspective which we will make heavy use of when referring to specific parts of the device that are already documented. We will focus mainly on the particular I/O requirements these devices exhibit and our way to address them.

7.1 Continuous-time computing

Broadly speaking, the von Neumann bottleneck is addressed in two ways by conventional von Neumann architectures;

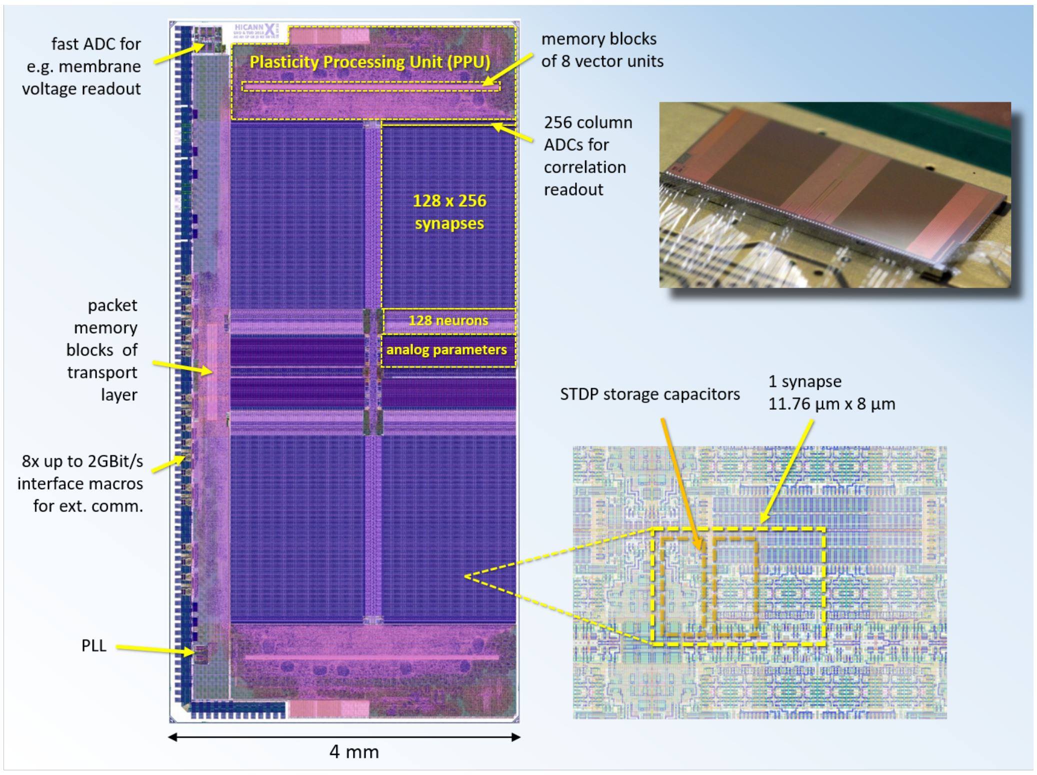


Figure 7.1: The HICANN-X. With kind permission from Andreas Grübl.

Data locality Also called caching is the technique to use small but fast local memories and use various algorithms to fetch data from the large but slow main storage, ideally just before that data is actually needed. This technique relies on the fact that all modern computers can freeze their state while waiting for data, even if the latency jumps due to cache misses.

Packet Serialization The various data consumers in a device are usually connected over a Bus that can span both on- and off-chip nodes. To avoid clogging the bus with high request rates which can easily overwhelm the arbitration logic, virtually all modern bus systems are designed to operate on large chunks of data that are moved in bursts between nodes. When moving off-chip, packets are used to minimize overhead on the link and thus maximize data throughput. For example, a typical cache line size in processors is 64 Bytes, even when the processor itself is scalar and in-order, thus consuming much less than that per clock cycle. Similarly, PCIe typically implements maximum TLP sizes of 4 KiB which is the typical page size in memory management schemes. As a consequence, modern bus networks and chip interconnects support rather low transaction frequencies relative to their data frequency.

However, devices like the HICANN-X struggle to fit into this design space. Because the neuron circuits are analog and asynchronous, their stimulation with spike events must be precisely controlled. This can be done by synchronizing the input interface—not the neurons themselves—with the user and then inject events via timed release into the device. While latency is somewhat less of a concern, the jitter, i.e the latency variation, is much more so, since the timing of a spike is the only information it carries and thus must be precise with respect to the intrinsic neuron time scales. Furthermore, the maximum link transaction rate determines the minimal temporal distance between events, which is also an important metric for the system. There is also no use for flow control with event data, since events that must be delayed due to congestion can usually be dropped entirely due to the abnormally high jitter they may have incurred during transport. Finally, as long as link error rates stay reasonably low, we can even do away with a guarantee for data integrity if it suits us, because corrupt event data may result in the wrong neuron being stimulated once in a while which introduces additional noise and does not pose a problem in most cases.

These are all properties of an unsecured Queue with a moderate word size where individual words contain the tuple `{address, timestamp}`, where `address` contains the target synapse address and `timestamp` contains the intended time of delivery. In the from-chip direction, the `address` field contains the source neuron and the `timestamp` field the time of recording at the interface. Note that the size of the `address` field is a matter of the ANN block size for a particular architecture, but the `timestamp` field size is dependent on the link latency which it needs to be able to compensate for. *Separation of concerns* dictates that these fields must be able to be sized independently from each other while maintaining bit efficiency which directly influences the achievable transaction rate. The method of synchronization is irrelevant and can be as simple as a RTT measurement in case both link partners share the same clock source as is the case with the HICANN-X.

7.2 Configuration

The HICANN-X has a large configuration space that needs to be set before running an experiment. This includes the PPU memories that need to be initialized as well as the configuration of various neuron parameters, synapse

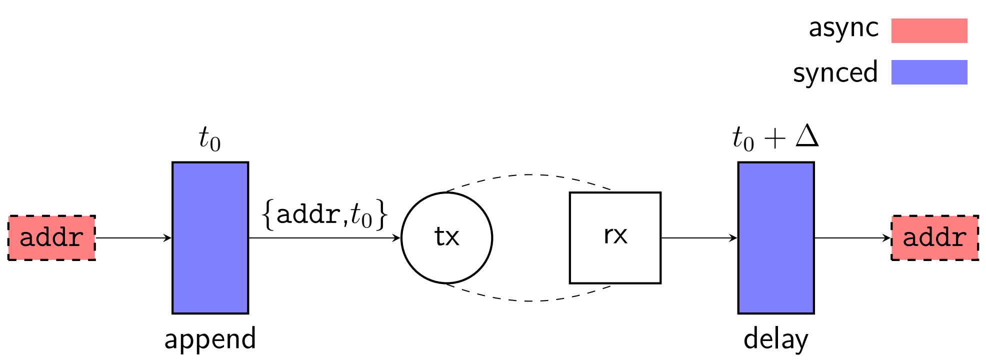


Figure 7.2: Tunneling of real-time data via a Queue. An event with an address label `addr` is appended at the `append` stage with a time stamp at time t_0 when it is pushed into the Queue. After appearing at the `rx` side of the Queue, it is held back at the `delay` stage until at least $t_0 + \Delta$ has been reached where it gets released. The `append` and `delay` stages must be synced via some implementation-dependent mechanism. The choice of Δ is crucial, as it depends on the delay and jitter of the tunneling Queue. See (Rettig, 2019) and (Schmidt, 2017) for more details.

crossbars and weights. Due to the accelerated neuron dynamics and thus relatively low execution times, this setup phase can be a significant proportion of the total runtime and thus limit the achievable experiment rates. This configuration data is loaded via the main on-chip bus, which is an implementation of the OCP bus specification called *Omnibus* developed by Simon Friedmann (Friedmann, 2013).

The main HICANN-X Omnibus (HX-Om) tree is multi-master, multi-slave interconnect clocked at 125 MHz with 32 bits address space and 32 bits payload size. It has a burst size of one, i.e, a single transaction contains only a single data word for reads or writes. The maximum transaction rate in the HX-Om is determined both by the RTT to a particular node as well as the maximum allowed in-flight transactions within the sub-tree.

The masters acting on the HX-Om are:

JTAG This master is tunneled via JTAG from the host FPGA as a side channel for debugging and startup purposes. It is unused during normal operation and has a very low bandwidth.

Highspeed This is the main entry point for off-chip configuration into the HICANN-X. It should be fast enough to support the maximum transaction rates in the bus without bottlenecking.

PPUs top and bottom While the main task of the PPUs is modifying synapse weights in their ANN block via a dedicated vector unit, they can also be used to configure the chip, as we have mentioned previously, and can thus act as a master on the HX-Om. In software, these transactions are memory mapped to a high bit offset.

For each master, the ordering of the transactions is guaranteed, i.e all responses return in the same order as the corresponding requests even if several masters are active at the same time. However, there is no concept of locking the bus for exclusive usage which can result in data hazards if more than one master is operating on the same address range. This is in practice not a problem since there are currently no use cases where two masters need to access the HX-Om at the same time, but it is a good reason to make sure that the JTAG master is used as little as possible and ideally not at all during operation.

The main slave subtrees, which are accessed most during the setup phase, are

PPUs top and bottom These nodes are each further subdivided to access the static random-access memory (SRAM) main memory of the processors as well as the respective synapse memory.

ANNcore This node provides access to the neuron parameters in the four quadrants as well as the configuration of the event merger that can route events between quadrants and off-chip.

7.3 PPU memory interface

The PPUs have a rather small local memory of 16 KiB for running programs, which can somewhat limit their usage in certain use cases. The HICANN-X does not have a dedicated external memory interface such as double-data rate synchronous dynamic random-access memory (DDR-SDRAM), but it is connected to a host FPGA via high-speed serial links that does have ample storage both in on-board SRAM cells and external DDR-SDRAM chips. An external memory interface for the PPU has been developed by Christian Pehle that allows it to access memory on the host FPGA. There are two logical channels per PPU:

Data The PPU uses an Omnibus variant that acts on 64 bit wide data with a burst length of two to store and read data from external memory. The number of in-flight transactions is determined by the prefetch size the memory controller of the PPU can handle, which is currently at most 128 B. Because Omnibus supports byte enables, data can be accessed in byte granularity, although this is mostly useful for writes as the whole 128 bit per transaction are still transported.

Instructions The PPU also has a separate channel that can request instructions from the external memory. The PPU issues 32 bit wide addresses, which return 32 bit wide instructions per request. The in-flight parameter is determined by the cache eviction policy and is currently set at 16 instructions.

Both the PPU memory interfaces and the HX-Om access must be stream secure because they are not designed to work with out-of-order or malformed transactions. They are also asymmetric both in data width and direction, since the HICANN-X acts as the master on the memory interface but is a slave for configuration. Still, both protocols can be split into Queues in the same manner as described in Section 2.2. For the HX-Om and PPU data interfaces three Queues are needed, two in the master direction containing data and commands respectively and a single Queue in the slave direction containing the slave responses. The PPU instruction interface only needs a single queue each in master and slave directions because it is a read-only channel.

Chapter 8

PHY

The HICANN-X employs the 65 nm version of the LVDS serializers used for the BSS chips (Scholze et al., 2012). In contrast to the interconnect described in (Scholze et al., 2012) we use only the raw serializer full-custom macros without the link layer designed for the BSS system. It establishes a byte aligned full-duplex link using one differential data pin and one differential clock pin per direction. Data is transmitted in a DDR scheme, which gives the parallel side to serial side clock ratio at four-to-one. Because the link is capable of data rates of $2\,\mathrm{Gbit\,s^{-1}}$ at a serial clock frequency of 1 GHz, the parallel side of the PHY is clocked with up to 250 MHz. We will however limit ourselves to $1\,\mathrm{Gbit\,s^{-1}}$ at 500 MHz serial clock frequency, since the Kintex-7 host FPGA can only support LVDS DDR bitrates up to about $1.2\,\mathrm{Gbit\,s^{-1}}$ at the GPIO pins (Sawyer, 2018). The PHY implements a link training command during which the link partners transmit a known data pattern and simultaneously adjust programmable delays trying to detect the training pattern at the receiving end. After both PHYs found alignment the control is then passed to the user interface which can then send and receive bytes.

The PHY does not support flow control and will serialize whatever data is at the `tx_par_data` port during a `par_clk` period. Conversely, every `par_clk` period a byte is present at the `rx_data_par` port if `rx_par_valid` is raised.

Besides the full-custom ASIC implementation, there also exists an FPGA implementation for Xilinx devices that uses the `ISERDESE2` and `OSERDESE2` primitives for DDR serialization and `IDELAY` programmable delays for the link startup phase (Xil, 2018).

```
interface phy_if();
parameter int PHY_WIDTH = 8;

logic tx_next,
      rx_valid,
      start_link;
logic [PHY_WIDTH-1:0] rx_data,
              tx_data;

modport client (
    input tx_next,
          rx_valid,
          rx_data,
    output tx_data,
           start_link
);

modport phy (
    output tx_next,
           rx_valid,
           rx_data,
    input tx_data,
          start_link
);
endinterface
```

Listing 9: SystemVerilog interface encapsulating the client side of the PHY. It is synchronous to the client byte clock `par_clk` that is divided by four from the serializer bit clock ser_clock.

This serializer is very well suited to be used in conjunction with a UT codec. Using the UT enhances the PHY from a non-blocking unsecured Queue to a blocking tunnel capable of transporting any number of Queues with varying security levels.

8.1 Link initialization

The PHY start-up procedure is very simple, yet reliable. Each of the link partners can be configured to act as the `init-master` or `init-slave`. To initialize the link, both `start_link` ports are raised one after the other start-

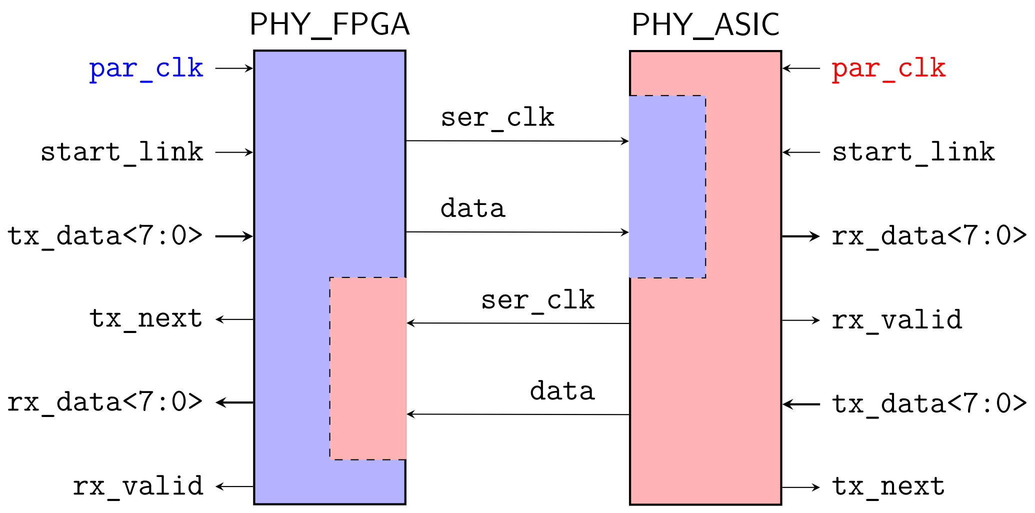


Figure 8.1: Block schematic of a duplex serializer-deserializer (SerDes) module pair. Each partner sends its serial `data` synchronous to the `ser_clk`, whose period is a quarter of the corresponding `par_clk` that clocks the user interface. Because the two `par_clk` clock domains should have the same period to avoid over- or underruns, but are not necessarily phase aligned, some clock domain crossing circuitry is needed to transport the received bytes to the user side.

ing with the `init-master` side. After some time ($\mathcal{O}(\mu s)$) the link training is completed and both link partners will raise their `tx_next` and `rx_valid` ports to begin client data transfers starting with the `init-slave`.

To avoid synchronization issues it is prudent to flush the link with commas for some time of the order of the RTT after link startup. This can be achieved via valves at the client side of the UT modules that are released after link training is completed. The PHYs can be re-trained by pulsing `start_link` on both sides low, which subsequently puts them into the training state and de-asserts `rx_valid` and `tx_next` until alignment is found again. The decision when the link needs to be re-trained must be made by the client, since the serializers do not have a method to track the link health state.

8.2 UT Link Checking

As we discussed previously, the UT decoder is able to detect link errors by checking the CRC and validating the datagram headers. We can use this to implement an automatic link health check mechanism that continuously monitors the data moving through the PHY and restarts the link training

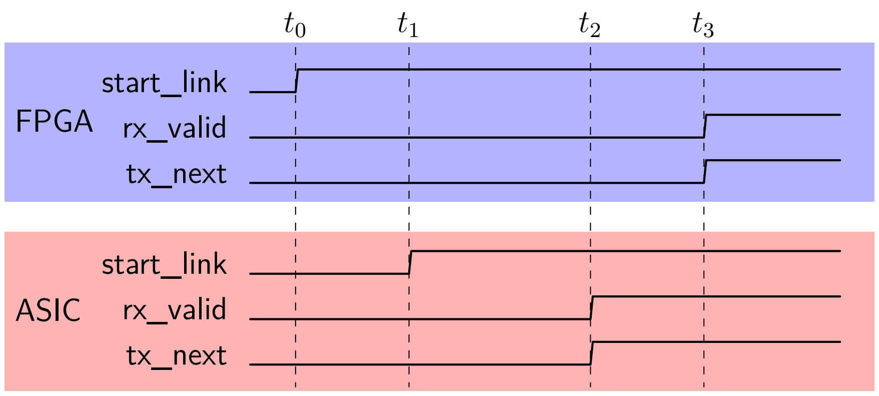


Figure 8.2: Timing diagram of the PHY start-up procedure. At t_0 the FPGA side raises its **`start_link`** flag as the **`init-master`**. After the ASIC side raises its **`start_link`** flag at t_1, both link partners are sending the training pattern and sweep the data eye via programmable delays. At t_2 the ASIC side raises its client side data accept flags after successfully completing the training. The FPGA side follows suit at t_3. Times are not to scale in this figure. Simulations indicate $t_3 - t_1 \approx 8.5\,\mu\text{s}$ for the complete training phase as well as $t_3 - t_2 = 88\,\text{ns}$, which is the client latency in the **`init-slave`**$\rightarrow$**`init-master`** direction for the $1\,\text{Gbit}\,\text{s}^{-1}$ bit rate.

routine as needed. The link error detection is further improved by imposing a minimum rate requirement for CRC carrying data, i.e the receiver also makes sure that a checksummed word is successfully decoded within a certain maximum time frame. In situations where the link is only carrying idle commas or unsecured datagrams, artificial CRC traffic should be generated and filtered at the decoder since it does not carry any information besides the CRC itself.

The module `ut_duplex` wraps a UT codec pair together with the link check functionality and can be used to connect to a PHY such as the one we have available for the HICANN-X. We will again first provide the VHDL module declaration:

```
entity ut_duplex is
  generic(
    WRITE_TYPELIST      : ut_entry_arr_t;
    READ_TYPELIST       : ut_entry_arr_t;
    CRC_POLY            : pos_arr_t;
    COMMON_DIV          : positive := 1;
    PHY_WIDTH           : integer range 2 to 64
  );
  port(
    clock               : in std_logic;
    reset               : in std_logic;
    link_enable         : in std_logic;
    link_check_period   : in integer;
    crc_err_cnt         : out integer;
    decode_err          : out std_logic;
    write_valid         : in std_logic;
    write_data          : in std_logic_vector(
                            max(get_widths(WRITE_TYPELIST))
                            - 1 downto 0);
    write_idx           : in integer range
                            0 to WRITE_TYPELIST'high;
    write_next          : out std_logic;
    read_valid          : out std_logic;
    read_data           : out std_logic_vector(
                            max(get_widths(READ_TYPELIST))
                            - 1 downto 0);
    read_idx            : out integer range
                            0 to READ_TYPELIST'high;
    start_link          : out std_logic;
    tx_data             : out std_logic_vector(
                            PHY_WIDTH-1 downto 0);
    tx_next             : in std_logic;
    rx_valid            : in std_logic;
    rx_data             : in std_logic_vector(
                            PHY_WIDTH-1 downto 0)
    );
end;
```

Again, we see that the user sum types $\overrightarrow{S}$ and $\overleftarrow{S}$ encoded by the `WRITE_-`

TYPELIST and READ_TYPELIST respectively can be entirely independent from each other. Internally, $\overrightarrow{S_l}$ and $\overleftarrow{S_l}$ are derived by appending a link check type:

```
constant LINK_CHK_ENTRY  : ut_entry_t :=  (0, true);

constant SEND_TYPELIST   : ut_entry_arr_t
                             (0 to WRITE_TYPELIST'length)
                             := WRITE_TYPELIST & LINK_CHK_ENTRY;

constant RECV_TYPELIST   : ut_entry_arr_t
                             (0 to READ_TYPELIST'length)
                             := READ_TYPELIST & LINK_CHK_ENTRY;
```

As we noted earlier, this type does not have a payload since its only purpose is to create intermittent CRC traffic during long idle or unsecured transmissions. Figure 8.3 shows a block schematic of the ut_duplex module.

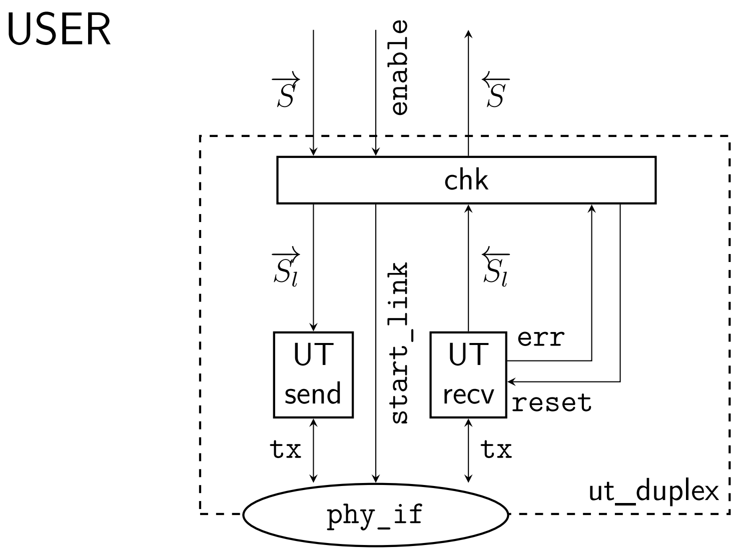


Figure 8.3: Block schematic of a duplex UT module pair including a PHY link checking mechanism.

On the sender side, the link checker chk counts the cycles since last push of a secured word. If a maximum amount of cycles is reached it temporarily blocks the client interface and injects the LINK_CHK_ENTRY of the SEND_-TYPELIST instead. On the recv side, any decoded LINK_CHK_ENTRY words are filtered from the stream and not passed to the client. Simultaneously, the period between two successfully decoded and CRC secured words is tracked. Link re-training is initialized if either one of the following conditions is met:

- Two consecutive datagrams with CRC errors are reported.
- The UT `receiver` cannot validate the datagram header.
- It has been too long since the last successful CRC reception.

Interestingly, this procedure is enough to trigger a re-training of the PHY even if there were only errors in one direction. If one of the link partners issued the re-training routine, its PHY will cease to accept data from the UT `sender` and start transmitting the training pattern instead. Since the PHY on the other side is not yet aware of anything abnormal, it will pass this training pattern to the UT `receiver` which practically guarantees that one of the previously mentioned error criteria is soon met because it is effectively trying to decode garbage data. This will then trigger the re-training routine on this side of the PHY as well until both link partners are re-aligned again.

The link training will not complete in case there is a bit error on the clock line during the alignment phase, since that effectively shifts the data a full clock cycle which currently cannot be compensated for. While this behavior can be observed in simulation by corrupting the clock pin, the probability for such event is very low at the usual bit error rates of $< 10^{-9}$. The default link check frequency is currently set at 2048 ns, or 256 link layer clock cycles.

8.3 Channel bonding

The data requirements of the HICANN-X can easily saturate a single 1 Gbit s^{-1} PHY. Increasing the throughput by using several PHYs could be a feasible solution to this problem. This *channel bonding* is usually implemented such that the link layer sees a single PHY interface of `N*PHY_WIDTH` bits using `N` PHYs. The main benefit is that there is no hardware duplication on any higher layer and data ordering is not affected since single DLLPs are serialized over several PHYs simultaneously.

There is however significant complexity added at the PHY layer to implement channel bonding which is why it is usually only seen on high-end interconnects like PCIe or Intel® QuickPath inteconnect (QPI) (Corporation, 2009). The lanes must train not only their own data delays but also align their de-serialized data to the others. Furthermore, several fall-back possibilities should exist in case not all PHYs are connected or were not able to complete their training for any reason. Finally, these schemes usually admit only a few possible lane amounts, e.g, $\mathtt{N} \in \{1, 2, 4, 8, 12, 16, 32\}$ for PCIe or $\mathtt{N} \in \{5, 10, 20\}$ in the case of QPI.

In contrast, we implement channel bonding above the link layer. Since the link layer operates on sum types of small- to moderate sizes that represent individual Queue words, it is very easy to distribute them between various `ut_duplex` modules, each connected to its own PHY. All that is needed in the off-chip direction is a switching layer that distributes data from n inputs of the same sum type to m outputs each connecting to a `ut_duplex` write port in a load distributing manner. In the to-chip direction, the distribution logic connecting the links to the clients must also contain some routing mechanism that determines which client should receive a particular data word. Various such schemes are possible, and their impact on overall performance can be quite significant. The current implementation of UT-to-client routing is reminiscent of the Token Ring topology (IEEE),(Bux, 1989). Data is committed into the router where it moves past all outputs until a client accepts it. See (Kanzleiter, 2018) for a detailed report on the link data distribution for the HICANN-X.

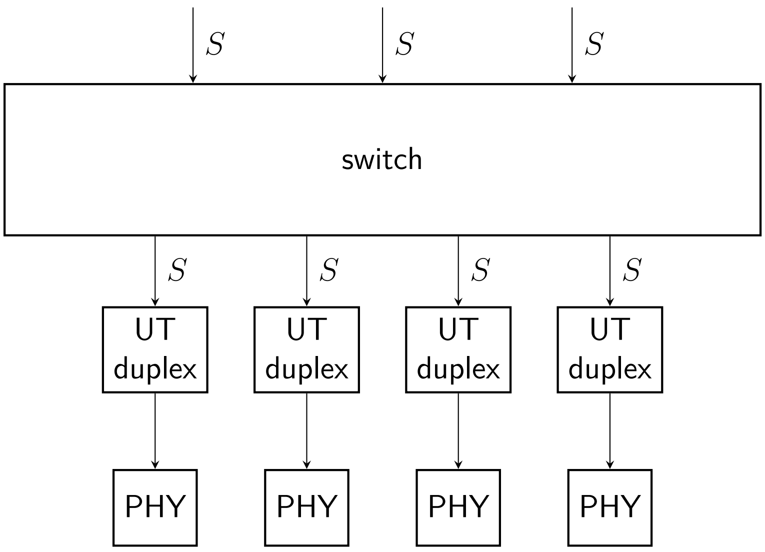


Figure 8.4: Block schematic of link layer-based channel bonding. Shown is the off-chip data flow from 3 clients through a load balancing switch that distributes the individual words to 4 `ut_duplex` modules each connecting to an individual PHY.

There are several disadvantages to channel bonding above the link layer. Firstly, and somewhat less importantly, duplicating the link layer, i.e, the `ut_duplex` modules imposes a significant area cost on the scheme. While channel bonding at the PHY layer would also require some additional alignment logic it would not depend on the encoding complexity of the link layer, which can be significant as we have previously discussed.

More troublesome however is the link data distribution problem outlined earlier, especially in the to-chip direction which often requires some directional routing as opposed to simple pressure-based load balancing in off-chip direction. Not only can the area cost become significant, there is also a very real risk of reordering the link data words during distribution or routing. This does not concern any clients that were stream-secured via a ARQ `sum_duplex` module that would connect to the link layer via the switch, because the SEQ fields are used to guarantee ordering on the client side of the ARQ modules. Queues that are not stream-secured will likely experience some reordering depending on the architecture and link activity. This is however a manageable drawback for Queues carrying real-time events, since they carry time stamps and are also often tolerant against reordering on small timescales compared to the neuron dynamics. We point again to (Schmidt, 2017), (Rettig, 2019) and (Kanzleiter, 2018) for studies that investigated the feasibility of this approach for event data in neuromorphic hardware.

While the overall throughput increases, latency tends to stagnate or increase when using several PHYs. This is due to the fact that each data word is serialized independently from others within its respective UT node. Thus, while the total amount of data words that are simultaneously serialized increases with the number of PHYs, the individual latency stays the same, which puts it at a disadvantage compared to PCIe where latency indeed improves with the number of lanes. Additional latency can arise from the link distribution circuitry and is strongly dependent on the individual interconnect requirements.

There are however significant advantages as well, most important of which is simplicity. We have increased throughput just by duplicating some modules and employing an additional distribution layer. This can obviously be done in a generic manner using **`generate`** statements and introducing a free parameter `NUM_PHYS`. It also greatly aids the separation of concerns, since this parameter can be chosen such that it fits into the available area and/or pin count while studying marginal gains using benchmark simulation sweeps.

Additionally, each of the PHYs operates entirely self-contained which gives the scheme great flexibility. There are no additional alignment issues due to the number of lanes that for example PCIe experiences which limit it to only certain combinations of PHYs. It is also trivial to shut down any subset of available PHYs without issues besides possibly dropping any in-flight datagrams traversing the affected lanes. Even mixing different PHYs is feasible which can for instance greatly aid reliability of the design, since slow but simple serializers like universal asynchronous receiver-transmitter (UART) (Fang and Chen, 2011) can be used as an additional fall-back PHY in case the main high-speed connection breaks down. This would avoid side-channel issues because the data words would still arrive at the same Queue interfaces, and ordering can be again guaranteed if the Queue was tunneled through an ARQ session.

8.3.1 Clocking

As we mentioned, the PHY does not employ clock-data recovery methods, but instead has a dedicated clock pin per direction. We can save on redundant clock pins by only propagating a single clock pin per direction regardless of `NUM_PHY`. This scheme distinguishes a single PHY as the `clk_master` which receives the `rx_clk` pin and is responsible to distribute it locally to the other PHYs. The link training routine needs no modifications since each PHY aligns its data pins individually, hence any phase shifts of the capture clocks between PHYs are automatically compensated for. A similar technique is used in QPI, which combines 20 serial data pins with a dedicated clock pin. One drawback in this scheme is a slight loss of redundancy, since the `clk_master` PHY must always be functioning and online to generate and propagate the clocks.

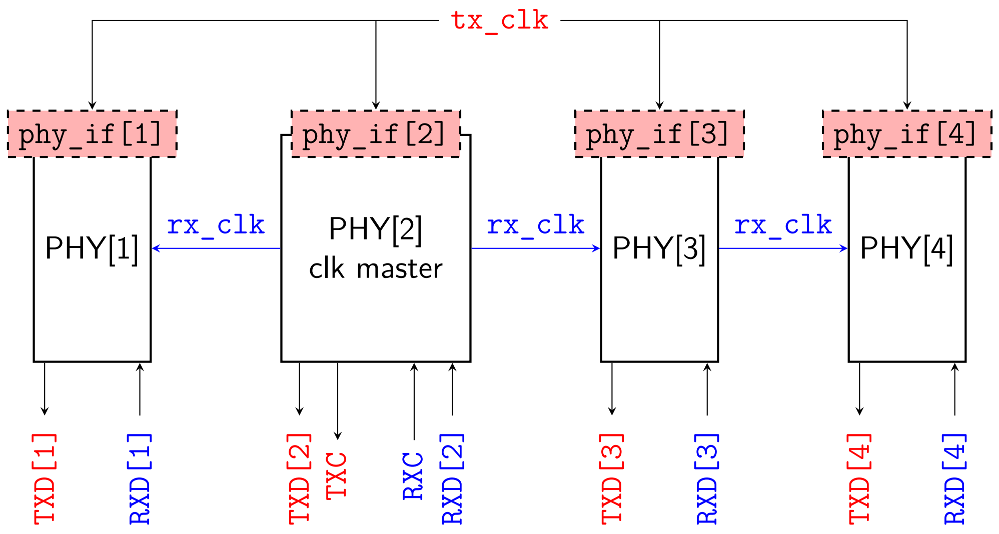


Figure 8.5: Clock distribution in the HICANN-X channel bonded PHY scheme. The client interface `phy_if[i]` of each `PHY[i]` and the outgoing LVDS data (`TXD[i]`) and clock (`TXC`) pins are synchronous to the internal clock `tx_clk`. Incoming LVDS data pins `RXD[i]` are synchronous to the incoming clock at the `RXC` LVDS pin. The clock master PHY is in the middle of the array (in this example PHY[2]), and is the source of two feed-forward `rx_clk` chains to its left and right which reach all other PHYs that use it to align their respective serial data.

Chapter 9

HICANN-X communication infrastructure

The high-speed host communication interface of the HICANN-X is also called Layer 2 Communication (L2) for historical reasons to distinguish it from the Layer 1 communication that connects neighboring chips for event transmission. Its purpose is to tunnel the various interfaces of the core logic in the HICANN-X as described in the previous chapter to a host FPGA to perform experiments. The tunneling is bi-directional, not only because protocols like OCP or ARQ require it, but because the HICANN-X acts like a master and slave simultaneously on different channels including external events.

As serializers, we employ 8 instances of the PHY discussed in the previous chapter which are combined using the channel bonding method introduced there. They provide an aggregated bandwidth of $8\,\mathrm{Gbit\,s^{-1}}$ per direction and can be individually brought online using a JTAG side channel. Consequently, the `ut_duplex` modules run at the PHY client clock frequency of 125 MHz and have the parameters `PHY_WIDTH = 8`, `COMMON_DIV = 1` and `CRC_POLY = [16, 13, 11, 10, 9, 8, 4, 2]`, which is the 16 bit wide CRC-16-Chakravarty polynomial (Chakravarty, 2001), and particularly suitable for smaller payloads of 64 bit or smaller.

We will begin by translating these client interfaces into Queues that are present at the client side of the L2.

9.1 Downstream

The to-chip or downstream direction transports the various Queues into the HICANN-X. There are four unsecured event interfaces, each containing a 14 bit address label, that connect to an event merger that routes the events within the ANN core logic. Two secured Queues are needed to tunnel the request side of the HX-Om, and another two Queues are needed per PPU which contain the instruction and data responses from the memory interface. Recall that all of these Queues are tunneled entirely independently from each other, as no ordering is guaranteed between any of them. Any synchronization happens at the consumer side within the core logic, as for example is the case for the two HX-Om Queues.

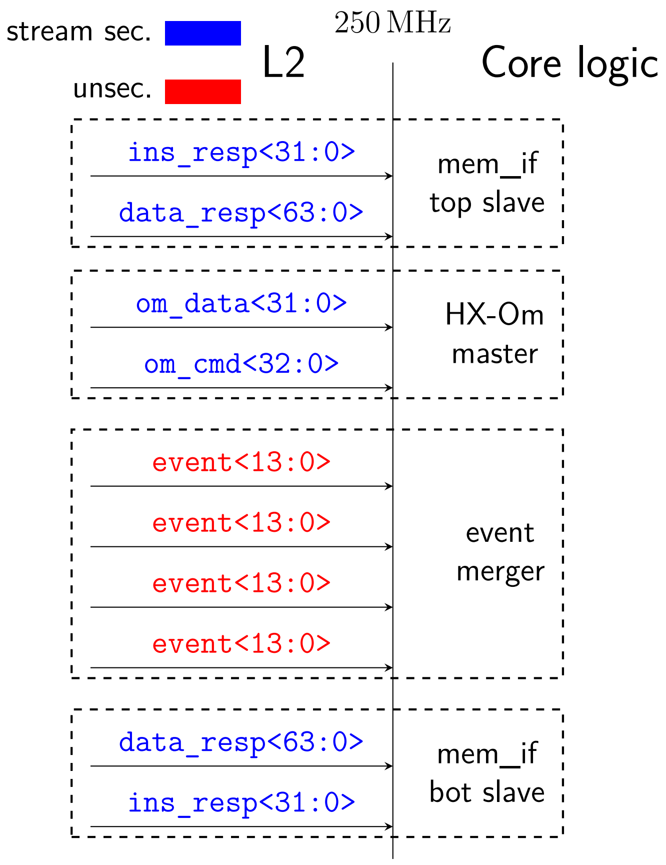


Figure 9.1: Downstream Client interfaces at the L2-Core logic boundary.

None of the types shown in Figure 9.1 are transported as-is, but are instead passed through some auxiliary logic.

ARQ data All Queues that require stream security form the sum type $\overrightarrow{S}$ which is the client side read sum type for an ARQ `sum_duplex` module. As we explained in Chapter 6, the network sum type $\overrightarrow{S^d}$ is derived from $\overrightarrow{S}$ using `SEQ_SIZE = 8` for both directions. $\overrightarrow{S^d}$ is forwarded from the 8 `ut_duplex` modules to the `sum_duplex` via a token-ring-style router. Splitting of $\overrightarrow{S}$ into the individual client side Queues is done via a multiplexer that gets data from the `sum_duplex` read port and routes the payload to the appropriate client via the `tag` field.

Events Event data is transported through the links with an additional time stamp field attached, denoted by $\overrightarrow{S^e}$. It then enters an l2 event router module developed by Alexander Schmidt (Schmidt, 2017) where events are delayed and routed to one of the 4 l2 event interfaces.

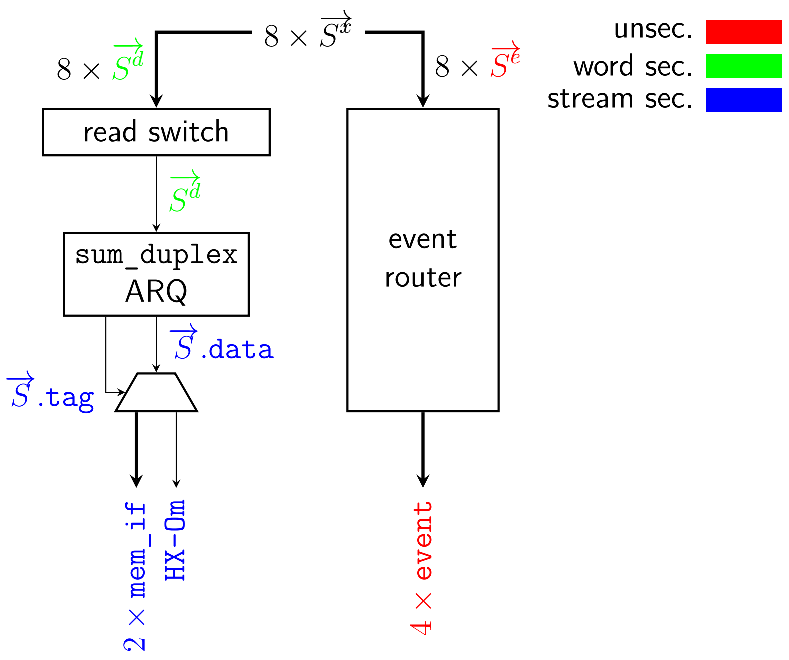


Figure 9.2: Block schematic of the l2 downstream modules. The from PHY side consists of 8 sum type interfaces $\overrightarrow{S^x}$ that are first split into the timestamped and unsecured event data $\overrightarrow{S^e}$ and word secured ARQ data $\overrightarrow{S^d}$. The ARQ data is routed to a `sum_duplex` module that constructs the stream secure sum type $\overrightarrow{S}$ which is then split into the client side Queues. $\overrightarrow{S^e}$ is passed through an event router that strips the time stamp after synchronization and routes the address to the unsecured client side event Queues.

9.2 Upstream

The from-chip or upstream direction of the L2 at the client level is dual to the downstream direction. The two PPU Queue collections now consist of the instruction request addresses as well as the data command and payload Queues since these belong to the master side of the memory interface. Conversely, there is now only one Queue containing the slave response data from the HX-Om. Besides these stream secured Queues, there are again four unsecured event Queues connecting to the event merger that are used to transport events off-chip.

M-ADC Additionally, another unsecured Queue is used to transport membrane analog-to-digital converter (M-ADC) samples. This analog-to-digital converter (ADC) can be configured via HX-Om and is used to monitor analog membrane voltages of selected neuron circuits. We choose an unsecured Queue here to attain a high throughput on the link because the M-ADC can produce data at rather high rates. However, because the samples are independent from each other, it is not critical to guarantee transport of all samples as missing or corrupt data would simply make the membrane trace noisy.

Figure 9.3 shows the upstream client Queues and their respective sizes. The internal structure of the L2 is dual to the downstream direction.

ARQ data All stream secured Queues are first collected into the write port of the `sum_duplex` ARQ module where they form the sum type $\overleftarrow{S}$. The network side of the `sum_duplex` inserts the derived sum type $\overleftarrow{S^d}$ into a load balancing switch that distributes the data to the available links.

Events Both neuron events and M-ADC data first pass through a time stamping module within the L2 that attaches a field containing time information on reception. The timestamped events $\overleftarrow{S^e}$ are then mixed together with $\overleftarrow{S^d}$ to form $\overleftarrow{S^x}$ at the link layer which is then serialized by the UT modules.

The host FPGA side of the L2 is entirely complementary to the chip side. On the client side, it contains the same Queue endpoints, but simply switches the `push` with the `pop` side of each Queue and vice versa. On the PHY side, it still consists of eight sum type interfaces per direction, each connecting to a single `ut_duplex` instance. Here, the sum types are swapped, since the to-chip sum type $\overleftarrow{S}$ on the HICANN-X is now the off-chip sum type for the FPGA.

9.3 Testing and Evaluation

It is precisely the highly generic nature of our interconnect framework, which the HICANN-X L2 is an incarnation of, that greatly simplifies the testing which in turn gives a high confidence of a bug-free design. Unit testing the

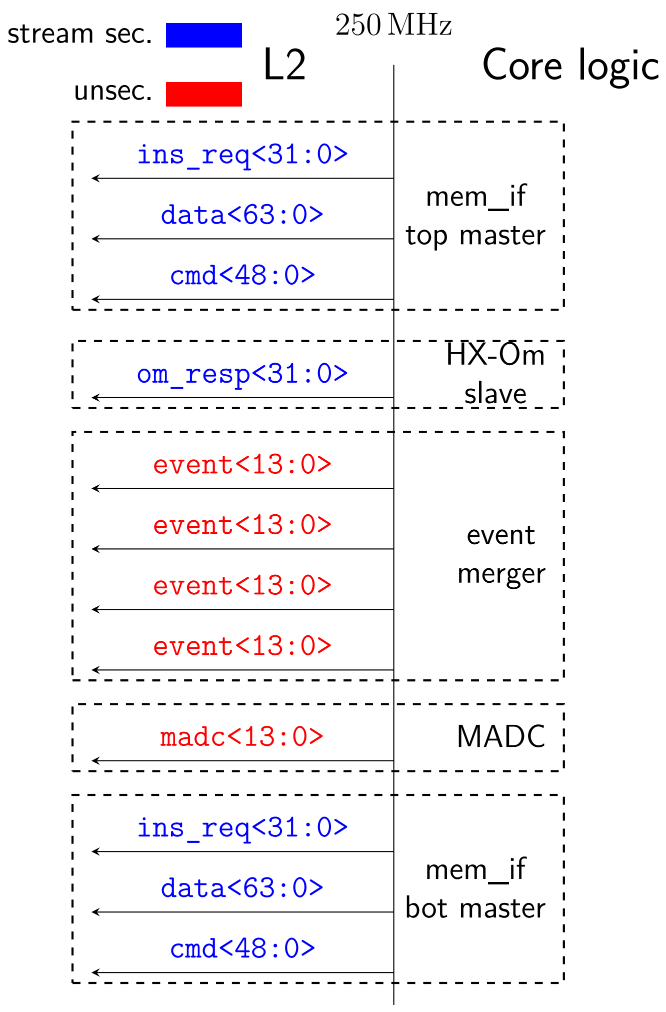


Figure 9.3: Upstream Client interfaces at the L2-Core logic boundary.

UT codec by first mocking a simple PHY and then using the HICANN-X serializer ensures that the link layer works correctly for any client sum type parametrization with simple tests that pass random payload to the codec and check for equality at the receiving side. Randomly corrupting the serial pins of the PHY then ensures that the link checking mechanism works correctly.

The `sum_duplex` ARQ modules also can be stand-alone unit tested by connecting two of them together with randomly picked client sum types and passing a random sequence of payload data while randomly dropping network words. Attaching a `sum_duplex` module as a client to the PHY and `ut_duplex` modules with corruption gives us a unit test that makes sure we can securely tunnel any sum type even if the network is unreliable. Finally, the switch and event router modules have been independently verified

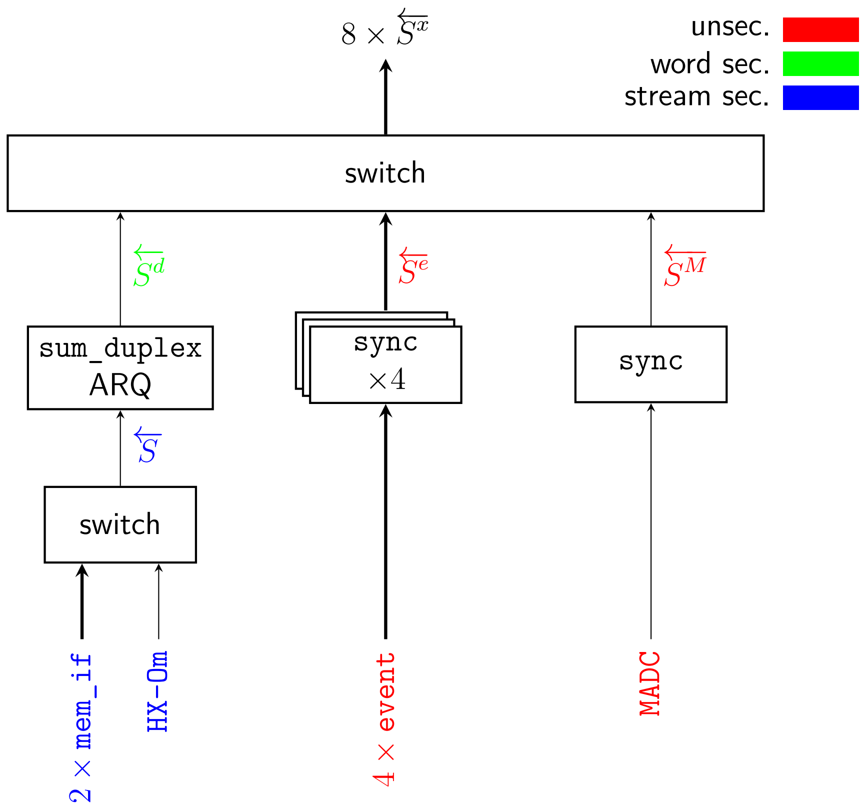


Figure 9.4: Block schematic of the l2 upstream modules.

in (Kanzleiter, 2018) and (Schmidt, 2017) respectively.

Having gained a high confidence in the building blocks of the L2, we can move on to integration tests. Because of the clear separation of the HICANN-X into core logic consisting of complex interfaces and the L2 which translates them into Queues and then tunnels them, integration testing is simply done by running the test suite for the core logic interfaces through the L2.

After successful manufacturing in the beginning of 2019, extensive stress tests were conducted in hardware by using the same integration tests as for simulation. We were pleased to observe that the L2 seems to performs exactly to specification as far as the testing can tell. No unexplained systematic data losses are observed during transport, in particular, all stream secured Queues are in fact so, with neither data corruption nor reordering issues. The HICANN-X is in productive usage since the summer of 2019 and already has sparked active usage within the Electronic Vision(s) group with some first experimental results published in (Göltz et al., 2019) and (Vision, s)

We will note here some more technical characteristics for the L2 not published elsewhere.

9.3.1 Events

One asymmetry between the host FPGA and the HICANN-X itself is the way events are handled. On the HICANN-X we have four independent event ports that can send and accept event data from the L2. At 250 MHz, the event Queues on the HICANN-X support a total bandwidth of 1 GHz event rate, which translates to 14 Gbit s^{-1} of net event throughput in both directions. Since the total available PHY bandwidth is slightly shy of 8 Gbit s^{-1} at the link layer due to the link check mechanism, it is not enough to transport the full event throughput even if there were no further overheads. The timestamped link layer events have the form

```
typedef struct packed{
        logic [13:0] addr;
        logic [1:0] channel;
        logic [7:0] timestamp;
} timestamped_event_t;
```

where `channel` denotes the origin and the source L2 event port respectively. Due to the `COMMON_DIV` layer and the total size of the sum type being `len(`$\overrightarrow{S^{x}}$`) = 17` and `len(`$\overrightarrow{S^{x}}$`) = 13`, this results in UT datagrams slightly smaller than 4 bytes. Thus, the theoretical event throughput maximum should be about 250 MHz.

However, the current host FPGA design has only a single port for events, both from and to the HICANN-X each operating at the FPGA L2 frequency of 125 MHz. Since the FPGA can only process a single word per cycle at most, this would effectively bottleneck the event throughput to 125 MHz. To counteract this, we have introduced a simple, yet generic event packing scheme to increase the event throughput at the FPGA side at constant frequency. The sum types $\overrightarrow{S^{e}}$ and $\overleftarrow{S^{e}}$ which contain link layer events have three entries each with the type sizes being

```
int packed_events[3]
 = {
      $bits{timestamped_event_t},
      2*$bits{timestamped_event_t},
      3*$bits{timestamped_event_t}
    };
```

This allows us to aggregate events into double or triple events using a simple shift register and transport them in parallel to the link layer which will then only serialize the valid number of events since it is type aware. This way we can now send and receive events at the FPGA L2 client side of more than 125 MHz because we can process a variable number of events per cycle. We point again to (Rettig, 2019) for a detailed explanation of the mechanism, and the overall performance characteristics of the HICANN-X L2 event transport mechanism.

The studies done in (Rettig, 2019) showed how sensitive the event traffic is towards the channel bonding and the distribution circuitry associated with it at the link layer. Experiments showed lossless data streams only up to about 125 MHz in event loopback tests which is only about half of the aggregated link layer maximum. Higher data rates experienced significant drops, which could be localized to the off-chip direction, both within the chip as well as FPGA L2 layers. Marco Rettig proposed in (Rettig, 2019) a mechanism to improve the drop rates on the FPGA which were subsequently implemented.

By exploiting the nature of the event packing we were able to insert compressors in the event routing logic towards the client side on the FPGA that collapsed subsequent events into larger packed ones in case the client layer that records the off-chip events is stalling. This pushes the lossless event off-chip traffic to about 200 MHz until the routing layer starts experiencing congestion and subsequent drops. This situation dramatically improves if one uses the switch proposed by Lea Kanzleiter in (Kanzleiter, 2018) for the distribution of L2 data towards the links, as well as investing more resources into the link-to-client event routing on the FPGA. The to-chip throughput indeed shows lossless event traffic up to the theoretical maximum of 250 MHz in simulation with the expectation that this also holds in hardware.

9.3.2 ARQ

Looking at the stream secured Queues tunneled via the L2 ARQ modules, both to- and from-HICANN-X, we encounter a slightly different picture. In

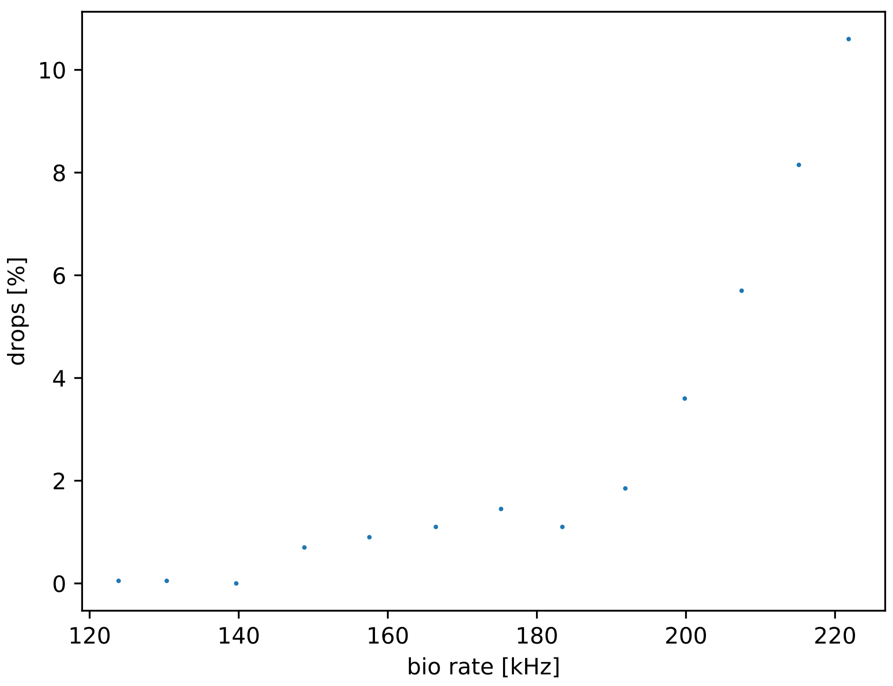


Figure 9.5: Event loopback hardware experiments on HICANN-X. 2000 spikes of varying rates are tunneled through the L2 starting at the host FPGA. They then enter the event merger logic on the HICANN-X where they are immediately looped back towards the FPGA. Pictured is the drop rate as the ratio between sent and received events at the biological time scale which is a factor of 1000 slower than wall clock time.

all cases, the client throughput is limited by the client itself rather than the interconnect. This is little surprising since the L2 was designed to handle the high event throughput in the first place. Still, there are some interesting details worth noting.

The discussion of ARQ performance inevitably involves the investigation of the BDP within the interconnect.

The transmission delay of the data is variable, since it depends on the size of the type being transmitted. This is naturally due to the fact that the UT is type aware and thus takes a varying amount of cycles to serialize data. Since the sizes of the stream secured client Queues range from 32 to 64 bit this incurs some non-negligible delay variation depending on the transmission profile. Additionally, there is total overhead of four byte per word, two for the CRC, and one for the `tag` and `seq` fields each. This puts the payload efficiency at between 50 to 66% depending on the word with serialization delays of 8 to 12 cycles, or 64 to 96 ns at 1 Gbit s^{-1} PHY bandwidth.

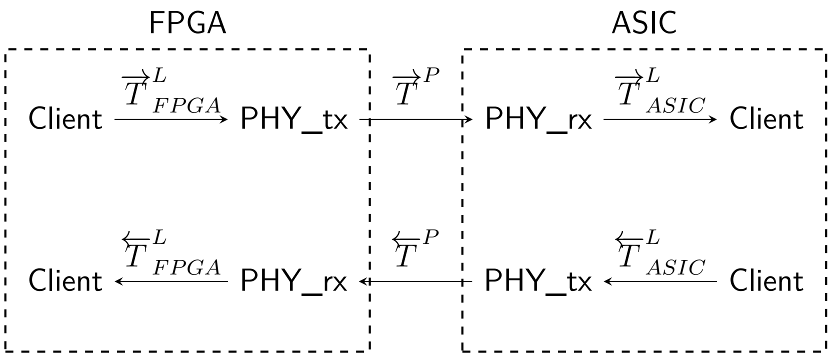

FPGA
ASIC
Client
$\overrightarrow{T}^{L}_{FPGA}$
PHY_tx
$\overrightarrow{T}^{P}$
PHY_rx
$\overrightarrow{T}^{L}_{ASIC}$
Client
Client
$\overleftarrow{T}^{L}_{FPGA}$
PHY_rx
$\overleftarrow{T}^{P}$
PHY_tx
$\overleftarrow{T}^{L}_{ASIC}$
Client

Figure 9.6: Delay diagram for the L2 ARQ traffic.

Figure 9.6 shows the various delays involved in the transport of stream secured data to and from the HICANN-X.

$\overrightarrow{T}^{L}_{FPGA}$ is the delay starting from the ARQ client write side until the input of the PHY client side on the FPGA. It does not depend on the size of the word because the transport happens on a per-word basis within a sum type. However, there is some variation due to the link data distribution scheme since it takes longer to reach PHYs that are further away.

$\overrightarrow{T}_{P}$ is the PHY transport delay in the to-chip direction. It is separate from the serialization delay mentioned previously and acts as an offset for the total serialization time between the PHY parallel interfaces.

$\overrightarrow{T}^{L}_{ASIC}$ is then the time from the link layer to the ARQ rx network side. It again varies because the individual link layers have a different routing delay.

$\overleftarrow{T}^{L}_{ASIC}$, $\overleftarrow{T}_{P}$, $\overleftarrow{T}^{L}_{FPGA}$ are the analogous transport delays for the off-chip ARQ data transport. Listing 9.1 shows the values for all of the delays in both directions.

	[ns]
$\overrightarrow{T}^{L}_{FPGA}$	64-120
$\overrightarrow{T}_{P}$	88
$\overrightarrow{T}^{L}_{ASIC}$	52-208

	[ns]
$\overleftarrow{T}^{L}_{ASIC}$	36-60
$\overleftarrow{T}_{P}$	80
$\overleftarrow{T}^{L}_{FPGA}$	88-120

Table 9.1: Ranges for delays to and from HICANN-X for ARQ data.

In total, the delays show a rather high spread between 268 ns to 512 ns in the to-chip direction, and 268 ns to 356 ns in the off-chip direction depending on the word size and PHY utilization.

Calculating the bandwidth is more straightforward, as it is just the aggregated $8\,\mathrm{Gbit\,s^{-1}}$ of the PHYs per direction. In contrast to the unsecured traffic there is no reduced bandwidth due to link check traffic since all the ARQ data has a CRC and thus automatically triggers the link check mechanism.

We can write the BDP as

$$D \times B = P \times W \tag{9.1}$$

with the delay D, bandwidth B, packet size P and window size W. Since everything else is now fixed, we can find the optimal window size for the ARQ, i.e the size of the replay buffer to be able to saturate the SerDes connection. While P is fixed by the client Queues, it also has a certain range. To be able to saturate the PHYs using any data make-up, we assume the worst case delays and smallest available payloads of 32 bit (plus 32 bit overhead):

$$\begin{aligned} 512\,\mathrm{ns} \times 8\,\mathrm{Gbit\,s^{-1}} &= 64\,\mathrm{bit} \times \overrightarrow{W} && (9.2)\\ \Rightarrow \quad \overrightarrow{W} &= \frac{512\,\mathrm{bit}}{64\,\mathrm{bit}} \times 8 = 64 && (9.3) \end{aligned}$$

$$\begin{aligned} 356\,\mathrm{ns} \times 8\,\mathrm{Gbit\,s^{-1}} &= 64\,\mathrm{bit} \times \overleftarrow{W} && (9.4)\\ \Rightarrow \quad \overleftarrow{W} &= \frac{356\,\mathrm{bit}}{64\,\mathrm{bit}} \times 8 = 44 && (9.5) \end{aligned}$$

Since the current ARQ implementation requires W to be a power of two we set both the $\overrightarrow{W}$ and $\overleftarrow{W}$ parameters of the `sum_duplex` ARQ modules at 64 words.

We can test the ARQ performance independently from the client Queues by measuring the bandwidth of an internal loopback Queue that connects the to-chip and from-chip sides of the L2. It is 59 bit wide a value chosen such that no further padding is required at the link layer which represents optimal bit efficiency for the UT encoding. This `perftest` testing method is identical to methods used for BSS chips (Debus, 2015): An FSM can be programmed to generate loopback data with increasing payload on the FPGA at full throughput for some duration. Looped data is checked for increasing payload as well to verify stream security. Packet counters are used to determine the macroscopic loopback bandwidth for the ARQ.

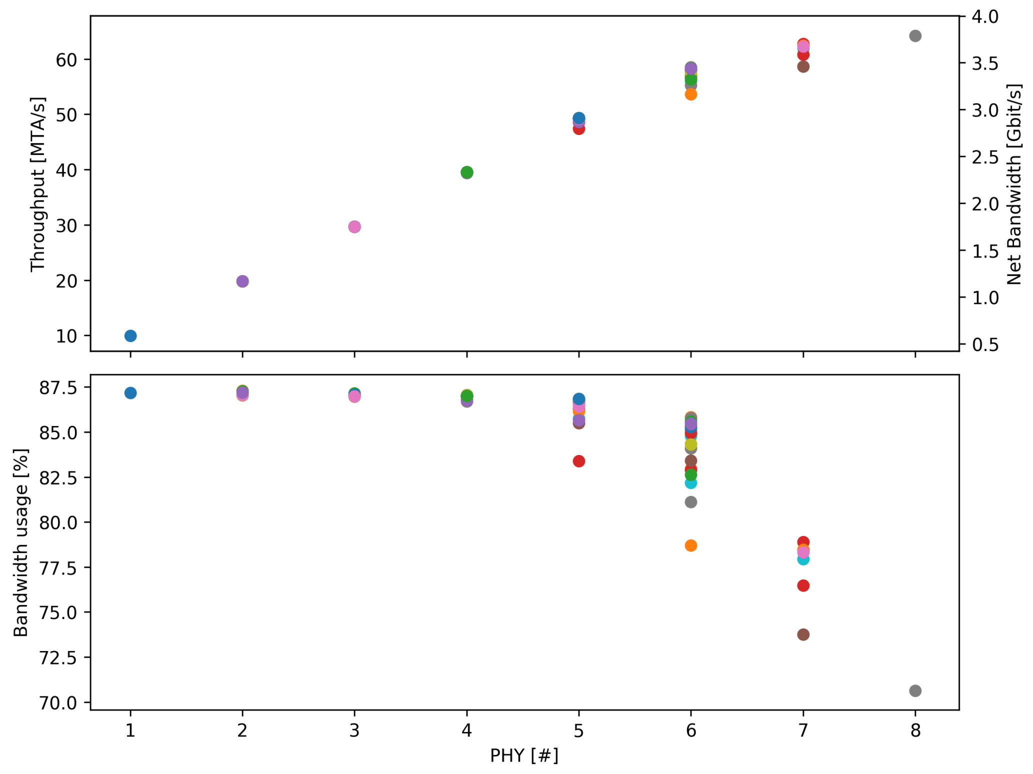


Figure 9.7: ARQ loopback throughput sweeps for various PHY constellations.

Figure 9.7 shows the results of hardware `perftest` experiments. There are 8 PHYs in total, but as we mentioned in Chapter 8 the middle `clk_master` PHY must be always enabled which results in 128 combinations that are grouped by the total number of active PHYs. The top figure shows that we can achieve throughput of over 60 MTA/s when using all PHYs, or slightly below $4\,\mathrm{Gbit\,s^{-1}}$ net bandwidth excluding overhead, which is about half of the total available bandwidth. We can however see that the marginal gains per PHY diminish after using 4 PHYs, which is shown more clearly in the bottom figure where the gross ARQ bandwidth including overhead is shown as a percentage of the total available bandwidth. Not only does the Bandwidth usage drop from around 88% to 71%, the spread between the various configurations is also quite substantial. All these effects are due to the lacking link distribution scheme which incur significant delay fluctuations. We expect significant improvements in future chip versions, where the switching scheme presented in (Kanzleiter, 2018) will be used.

Nonetheless, the `perftest` experiments only show a synthetic performance that is not influenced by the client layers. When measuring the PPU memory interface as well as HX-Om throughput, we find that they all fail to saturate the available ARQ bandwidth with the notable exception of PPU data write transactions which can indeed saturate the off-chip ARQ bandwidth.

For the HX-Om it is because the bus fabric supports only four transactions in flight in the latest HICANN-X incarnation to save on pipelining logic. The HX-Om transactions complete on average in 10 cycles, which, at the HX-Om operating frequency of 125 MHz and 65 bit per write transaction only requires

$$B = \frac{4}{10} \times 65\,\text{bit} \times 125\,\text{MHz} = 3.25\,\text{Gbit}\,\text{s}^{-1} \tag{9.6}$$

which can be comfortably achieved using 5 or more PHYs. A pre-fetching scheme compensates for the ARQ tunneling delay by allowing 48 transactions in flight across the L2 tunnel between FPGA and HICANN-X.

In contrast, the PPU memory interface is not well equipped to deal with a tunneling RTT approaching 1 µs because its in-flight window is too small. For instance, the longest burst that the PPU data interface can handle is the width of the PPU *vector register* (Friedmann, 2013) which is 128 B. Committing the content of the vector register into the external memory can be done in a fire-and-forget fashion because the ARQ guarantees transport and no acknowledgment is needed apart from the usual blocking interface towards the ARQ. This 128 B burst is serialized into 16 64 bit wide words that are tunneled through a stream secured Queue to the FPGA. At 250 MHz HICANN-X L2 operating frequency, this creates traffic of

$$B = \frac{64\,\text{bit}}{4\,\text{ns}} = 16\,\text{Gbit}\,\text{s}^{-1} \tag{9.7}$$

over a 64 ns time frame which is vastly more than the sustained ARQ throughput. Thus, the PPU can saturate the L2 if it can issue bursts at a faster rate than the L2 is able to tunnel them, which strongly depends on the workload scenario and the PPU frequency itself.

Loading data from external memory is however far slower because the PPU can only issue reads of the vector register size and then has to block until the data arrives to issue a subsequent read burst. Simulations indicate RTTs of about 950 ns at the PPU vector register for external loads, which limits the bandwidth to $\frac{1024\,\text{bit}}{950\,\text{ns}} = 1.08\,\text{Gbit}\,\text{s}^{-1}$ This can be comfortably supported by the L2, even if both PPUs perform external data loads at the same time.

External instruction fetches suffer from a similar problem, where the in-flight data size is 16 instructions of 32 bit. This is only half of the in-flight data packet for the memory interface, and thus also only requires only about $500\,\text{Mbit}\,\text{s}^{-1}$ per PPU due to the almost identical RTT. Due to instruction caching there is however rarely the need to fetch large amounts of instructions at a time, and increasing the fetch size will almost certainly increase the probability of premature eviction within the cache in typical workloads which would hurt the performance (Smith, 1987)

Note how in both the memory load and instruction fetch scenarios, the performance issues are still separated from the implementation of the L2. We can try to improve performance by decreasing the RTT, or we can instantiate a bigger cache and fetch larger amounts of data at a time, or both. The individual design decisions influence each other across module boundaries, but do not enforce hard architectural dependencies, but instead admitting parametrizable solutions that can be tuned in the design exploration phase.

Chapter 10

Conclusion

We conclude this work by summarizing the key points of the generic approach we have taken to implement a scalable chip-to-chip interconnect for the HICANN-X. Introducing the concept of *sum types* to HDL design gives us the ability to reason about communication protocols as a collection of Queues that are processed and tunneled between endpoints. We also have a very natural way to increase the transport security of a Queue by composing it with another derived Queue containing metadata such as sequence numbers or CRC, while still being entirely generic.

The UT modules enable the tunneling of arbitrary sum types through a plethora of possible serializer implementations, offering not only the encoding and gearboxing logic, but also the possibility to provide a blocking client interface via commas and adding checksums to any member type, thereby guaranteeing word security if needed. A free alignment parameter aids the *separation of concerns* by adjusting the encoding scheme while keeping both the client and serializer interface stable which represents a trade-off between chip real-estate and bit efficiency of the encoding. A *link checking* mechanism can be used to monitor the serializer health and re-trigger the available link training mechanism on both ends without side-band information.

The sum type approach also provides a convenient way of channel-bonding serializers; even pairing several different architectures is possible due to the high parametrization of the UT codec. The logic complexity to align several serializers in parallel to realize channel bonding at PHY level is replaced by the much better understood mechanism of packet distribution and routing at the link layer. A hybrid approach is also possible by channel bonding several serializers at the PHY layer and then also bonding several of these again at the link layer. The flexibility as well as the largely independent design choices benefit the development process to efficiently utilize available resources while staying within a general framework.

The precise control of the data transport through all layers enable the creation of low-jitter interconnects that transport small packets at high transaction rates, which is a crucial feature for asynchronous computers that cannot stall for data. These devices, which stream data while continuously evolving, will often look to combine stream-secure low-frequency configuration traffic with high-frequency event data that should be as bit-efficient as possible, even at the cost of spurious data losses. Examples include the Heidelberg neuromorphic devices, but also various detector chips, found for instance in high-energy physics such as the Monopix chips (Caicedo et al., 2019) as well as the medical field (Ritzer et al., 2020). To date, no unifying concept was established for the stream interconnect due to the large variation in underlying serializers and other device constraints. Works like (Gabrielli, 2014) describe the readout of digital pixel arrays only up to the PHY layer, without guidance on data framing and other encoding issues. Raw streams, while very popular in these devices, cannot be a reliable solution because they all suffer from usability issues, and even encoding schemes like scrambling codes (e.g, 8b10b) are merely a step in the right direction but do not provide a comprehensive scheme that fits many scenarios.

The interconnect approach we propose is also beneficial for traditional von-Neumann computers, although fast and finely tuned technologies like PCIe are already available. Open-source hardware is gaining traction, with the RiscV (Asanović and Patterson, 2014) architecture at the helm, with the goal of providing cheap or even free alternatives to conventionally available microprocessors. RiscV uses TileLink (til) as its bus architecture, which facilitates data transfers between clients in a similar manner as our –admittedly somewhat less sophisticated– solution, HX-Om bus. While bridges for existing off-chip interconnects such as PCIe, Ethernet, UART and others are available, they are all tailor-made for that specific technology and often lack more advanced features like stream security. Using our approach, a generic TileLink bridge can be developed that can be connected to almost any kind of serializers just as we have demonstrated for the HICANN-X L2. The design team can then make the serializer design process completely orthogonal to the rest of the device, which greatly simplifies the development process. Furthermore, the evolution of the bridge can be concentrated on a single design instead of spreading development effort across supporting various bridges with different requirements.

List of Acronyms

ACK	acknowledgment number	75
ADC	analog-to-digital converter	104
ANN	artificial neural network	3
ARQ	Automatic Repeat-Request	75
ASIC	application-specific integrated circuit	33
AXI	Advanced eXtensible Interface	15
BDP	Bandwidth-Delay Product	76
BEC	backward error correction	12
BSS-2	BrainScaleS-2 neuromorphic hardware	83
BSS	BrainScaleS neuromorphic hardware	76
CDR	clock-data recovery	24
CMOS	complementary metal-oxide-semiconductor	3

CRC	cyclic redundancy check	14
DDR-SDRAM	double-data rate synchronous dynamic random-access memory	87
DDR	double-data rate	
DLLP	data link layer packet	25
EOF	end-of-frame	44
FEC	forward error correction	12
FIFO	first-in-first-out buffer	7
FPGA	field programmable gate array	5
FSM	finite-state machine	59
GPIO	general purpose In/Out	21
HDL	hardware description language	5
HICANN-X	High Input Count Analog Neural Network with HAGEN Extensions	67
HX-Om	HICANN-X Omnibus	86
I2Q	interface to queues	15
IP	intellectual property	61

JTAG Joint Test Action Group 23

L2 Layer 2 Communication 101

LUT look-up table 68

LVDS low voltage differential signaling . . . .

M-ADC membrane analog-to-digital converter 104

MGT multi-gigabit transceiver 24

MSB Most Significant Bit 65

MTTF mean time to failure 10

MTTSC mean time to stream corruption 11

OCP Open Core protocol 15

P2P point-to-point . . . .

PCIe Peripheral Component Interconnect Express 25

PLL phase-locked loop 24

PPU Plasticity Processing Unit 83

Q2I queues to interface 15

QPI Intel® QuickPath inteconnect	95
QoS quality of service	22
RTL register-transfer-level	
RTT round-trip time	78
SATA Serial AT Attachment	27
SEQ sequence number	75
SOF start-of-frame	44
SPI Serial Peripheral Interface	23
SRAM static random-access memory	87
SerDes serializer-deserializer	91
SoC separation of concerns	47
TLP transaction layer packet	25
UART universal asynchronous receiver-transmitter	98
USB Universal Serial Bus	27
UT Universal Translator	5

VHDL Very High Speed Integrated Circuit Hardware Description Language 51

Bibliography

Tilelink. URL `https://bar.eecs.berkeley.edu/projects/tilelink.html`.

S. A. Aamir, P. Müller, L. Kriener, G. Kiene, J. Schemmel, and K. Meier. From lif to adex neuron models: Accelerated analog 65 nm cmos implementation. In *IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 1–4. IEEE, October 2017.

M. Allman, V. Paxson, ICSI, and E. Blanton. RFC 5681: Tcp congestion control, 2009. URL `https://tools.ietf.org/html/rfc5681`.

K. Asanović and D. A. Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

S. Billaudelle, Y. Stradmann, K. Schreiber, B. Cramer, A. Baumbach, D. Dold, J. Göltz, A. F. Kungl, T. C. Wunderlich, A. Hartel, et al. Versatile emulation of spiking neural networks on an accelerated neuromorphic substrate. *arXiv preprint arXiv:1912.12980*, 2019.

D. Brüderle, M. A. Petrovici, B. Vogginger, M. Ehrlich, T. Pfeil, S. Millner, A. Grübl, K. Wendt, E. Müller, M.-O. Schwartz, D. de Oliveira, S. Jeltsch, J. Fieres, M. Schilling, P. Müller, O. Breitwieser, V. Petkov, L. Muller, A. Davison, P. Krishnamurthy, J. Kremkow, M. Lundqvist, E. Muller, J. Partzsch, S. Scholze, L. Zühl, C. Mayr, A. Destexhe, M. Diesmann, T. Potjans, A. Lansner, R. Schüffny, J. Schemmel, and K. Meier. A comprehensive workflow for general-purpose neural modeling with highly configurable neuromorphic hardware systems. *Biological Cybernetics*, 104: 263–296, 2011. ISSN 0340-1200. URL `http://dx.doi.org/10.1007/s00422-011-0435-9`.

W. Bux. Token-ring local-area networks and their performance. *Proceedings of the IEEE*, 77(2):238–256, Feb 1989. ISSN 1558-2256. doi: 10.1109/5.18625.

I. Caicedo, M. Barbero, P. Barrillon, I. Berdalovic, S. Bhat, C. Bespin, P. Breugnon, R. Cardella, Z. Chen, Y. Degerli, and et al. The monopix chips: depleted monolithic active pixel sensors with a column-drain read-out architecture for the atlas inner tracker upgrade. *Journal of Instrumentation*, 14(06):C06006–C06006, Jun 2019. ISSN 1748-0221. doi: 10.1088/1748-0221/14/06/c06006. URL `http://dx.doi.org/10.1088/1748-0221/14/06/C06006`.

T. Chakravarty. Performance of cyclic redundancy codes for embedded networks. Master's thesis, 2001.

I. Corporation. *An Introduction to the Intel® QuickPath Interconnect*. 2009.

J. Debus. *Configuration performance testing of the Hicann v4*, 2015.

G. Deletoille. Arpe report: Data transmission protocol for neuromorphic hardware, 2016.

Y. Fang and X. Chen. Design and simulation of uart serial communication module based on vhdl. In *2011 3rd International Workshop on Intelligent Systems and Applications*, pages 1–4, May 2011. doi: 10.1109/ISA.2011.5873448.

S. Friedmann. *A New Approach to Learning in Neuromorphic Hardware*. PhD thesis, Ruprecht-Karls-Universität Heidelberg, 2013.

S. Friedmann, N. Frémaux, J. Schemmel, W. Gerstner, and K. Meier. Reward-based learning under hardware constraints â€" using a RISC processor embedded in a neuromorphic substrate. *Frontiers in Neuroscience*, 7:160, 2013. ISSN 1662-453X. doi: 10.3389/fnins.2013.00160. URL `http://journal.frontiersin.org/article/10.3389/fnins.2013.00160`.

S. Friedmann, J. Schemmel, A. Grübl, A. Hartel, M. Hock, and K. Meier. Demonstrating hybrid learning in a flexible neuromorphic hardware system. *IEEE Transactions on Biomedical Circuits and Systems*, 11(1):128–142, 2017. ISSN 1932-4545. doi: 10.1109/TBCAS.2016.2579164.

S. Furber. Large-scale neuromorphic computing systems. *Journal of Neural Engineering*, 13(5):051001, aug 2016. doi: 10.1088/1741-2560/13/5/051001. URL `https://doi.org/10.1088%2F1741-2560%2F13%2F5%2F051001`.

A. Gabrielli. Fast readout architectures for large arrays of digital pixels: examples and applications. 2014.

P. S. I. Group. *PCI Express Base Specification, Revision 3.0*, Nov 2010.

J. Göltz, A. Baumbach, S. Billaudelle, O. Breitwieser, D. Dold, L. Kriener, A. F. Kungl, W. Senn, J. Schemmel, K. Meier, and M. A. Petrovici. Fast and deep neuromorphic learning with time-to-first-spike coding, 2019.

IEEE. IEEE Token Ring standards. URL `http://www.ieee802.org/5/`.

G. Indiveri, B. Linares-Barranco, T. J. Hamilton, A. van Schaik, R. Etienne-Cummings, T. Delbruck, S.-C. Liu, P. Dudek, P. Häfliger, S. Renaud, J. Schemmel, G. Cauwenberghs, J. Arthur, K. Hynna, F. Folowosele, S. Saighi, T. Serrano-Gotarredona, J. Wijekoon, Y. Wang, and K. Boahen. Neuromorphic silicon neuron circuits. *Frontiers in Neuroscience*, 5(0), 2011. ISSN 1662-453X. doi: 10.3389/fnins.2011.00073. URL `http://www.frontiersin.org/Journal/Abstract.aspx?s=755&name=neuromorphicengineering&ART_DOI=10.3389/fnins.2011.00073`.

L. Jian-Liang and Y. Hong-Sen. *Decoding the Mechanisms of Antikythera Astronomical Device*. Springer, 2016. ISBN 9783662484456.

L. Kanzleiter. A parametrizable switch for neuromorphic hardware. Bachelorarbeit, Universität Heidelberg, 2018.

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

V. Karasenko. A communication infrastructure for a neuromorphic system. Master's thesis (English), University of Heidelberg, 2014.

C. Mead. Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10): 1629–1636, 1990.

T. Pfeil, A. Grübl, S. Jeltsch, E. Müller, P. Müller, M. A. Petrovici, M. Schmuker, D. Brüderle, J. Schemmel, and K. Meier. Six networks on a universal neuromorphic computing substrate. *Frontiers in Neuroscience*, 7:11, 2013. ISSN 1662-453X. doi: 10.3389/fnins.2013.00011. URL `http://www.frontiersin.org/neuromorphic_engineering/10.3389/fnins.2013.00011/abstract`.

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

M. Rettig. Characterizing the event interface of the hicann-x, 2019.

C. Ritzer, R. Becker, A. Buck, V. Commichau, J. Debus, L. Djambazov, A. Eleftheriou, J. Fischer, P. Fischer, M. Ito, P. Khateri, W. Lustermann, M. Ritzert, U. Röser, M. Rudin, I. Sacco, C. Tsoumpas, G. Warnock, M. Wyss, A. Zagozdzinska-Bochenek, B. Weber, and G. Dissertori. Initial characterisation of the SAFIR prototype PET-MR scanner. *IEEE Transactions on Radiation and Plasma Medical Sciences*, pages 1–1, 2020. ISSN 2469-7303. doi: 10.1109/TRPMS.2020.2980072.

N. Sawyer. *LVDS Source Synchronous 7:1 Serialization and Deserialization Using Clock Multiplication.* Xilinx, Inc, 2018.

J. Schemmel, D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1947–1950, 2010.

J. Schemmel, L. Kriener, P. Müller, and K. Meier. An accelerated analog neuromorphic hardware system emulating NMDA-and calcium-based nonlinear dendrites. *arXiv preprint arXiv:1703.07286*, 2017.

A. Schmidt. Design und charakterisierung einer routing-schnittstelle für neuromorphe hardware, 2017.

S. Schmitt, J. Klaehn, G. Bellec, A. Grübl, M. Guettler, A. Hartel, S. Hartmann, D. H. de Oliveira, K. Husmann, V. Karasenko, M. Kleider, C. Koke, C. Mauch, E. Müller, P. Müller, J. Partzsch, M. A. Petrovici, S. Schiefer, S. Scholze, B. Vogginger, R. A. Legenstein, W. Maass, C. Mayr, J. Schemmel, and K. Meier. Neuromorphic hardware in the loop: Training a deep spiking network on the brainscales wafer-scale system. *CoRR*, abs/1703.01909, 2017. URL `http://arxiv.org/abs/1703.01909`.

S. Scholze, H. Eisenreich, S. Höppner, G. Ellguth, S. Henker, M. Ander, S. Hänzsche, J. Partzsch, C. Mayr, and R. Schüffny. A 32gbit/s communication soc for a waferscale neuromorphic system. *Integration*, 45(1): 61 – 75, 2012. ISSN 0167-9260. doi: https://doi.org/10.1016/j.vlsi.2011.05.003. URL `http://www.sciencedirect.com/science/article/pii/S0167926011000538`.

K. Schouhamer Immink. A survey of codes for optical disk recording. *Selected Areas in Communications, IEEE Journal on*, 19:756 – 764, 05 2001. doi: 10.1109/49.920183.

A. J. Smith. Design of cpu cache memories. Technical Report UCB/CSD-87-357, EECS Department, University of California, Berkeley, Jun 1987. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/1987/5288.html`.

C. Smith. How i learned to stop worrying and love the bombe: Machine research and development and bletchley park. *History of Science*, 52(2): 200–222, 2014. doi: 10.1177/0073275314529861. URL `https://doi.org/10.1177/0073275314529861`.

D. Swade. *The difference engine : Charles Babbage and the quest to build the first computer*. Penguin Books, 2002.

DesignWare Library - Datapath and Building Block IP. Synopsys, Inc., 700 East Middlefield Road, Mountain View, CA 94043.

C. S. Thakur, J. L. Molin, G. Cauwenberghs, G. Indiveri, K. Kumar, N. Qiao, J. Schemmel, R. Wang, E. Chicca, J. Olson Hasler, et al. Large-scale neuromorphic spiking array processors: A quest to mimic the brain. *Frontiers in neuroscience*, 12:891, 2018.

E. Vision(s). Hbp sga2 sp9 kpi and student numbers, 2020.

J. von Neumann. First draft of a report on the edvac. *IEEE Ann. Hist. Comput.*, 15(4):27–75, Oct. 1993. ISSN 1058-6180. doi: 10.1109/85.238389. URL `https://doi.org/10.1109/85.238389`.

A. X. Widmer and P. A. Franaszek. A dc-balanced, partitioned-block, 8b/10b transmission code. *IBM Journal of research and development*, 27(5):440–451, 1983.

7 Series FPGAs SelectIO Resources. Xilinx, Inc., 2018. URL `https://www.xilinx.com/support/documentation/user_guides/ug471_7Series_SelectIO.pdf`.

Acknowledgments

Firstly, I would like to thank **Johannes Schemmel**, who unwaveringly took on the burden KM left behind and is pressing to continue the Vision(s) that connect us all.

Secondly, my gratitude goes to Profs. **Brüning**, **Fischer** and **Salmhofer**, who all swiftly agreed on very short notice to be part of my examination committee and also took the time to discuss my work on several occasions.

Thirdly, credit goes to the entire **Electronic Vision(s)** group, with whom I spent most of the past ten years partly working, but mostly bbq'ing, watching NeuroVision(s), having beer tastings for science and arguing about nothing. Every thesis that is completed within this group is in a sense always a team effort of the entire Vision(s), and my own time here was certainly no different. Still, special thanks must be extended to the following people:

Andreas Grübl, for never losing his cool and teaching me when the right time is to do things Grübl-style.

Eric Müller, for ALWAYS losing his cool when trying to convince people that the only way is *rischdisch mache*.

Yannick Stradmann, **Mitja Kleider**, and **Sebastian Billaudelle** for being the core commissioning team of the HICANN-X and setting up so much of the amazing experiment and testing environment that benefits us all so greatly. Extra special mention goes to **Philipp Spilger**, who defines the term 'above and beyond'.

My students **Gaëtan Delétoille**, **Jan Debus**, **Alexander Schmidt**, **Lea Kanzleiter** and **Marco Rettig**, who eagerly partook in my own small vision and rose to the challenge to contribute their best work as part of something greater than a single Bachelor's project. If you are missing some detail in this thesis, you will surely find it in one of theirs.

Lastly, I want to thank **Elisabeth Wintersteller** and **Mihai Petrovici** for. . . quite a bit of stuff actually. Thank you.