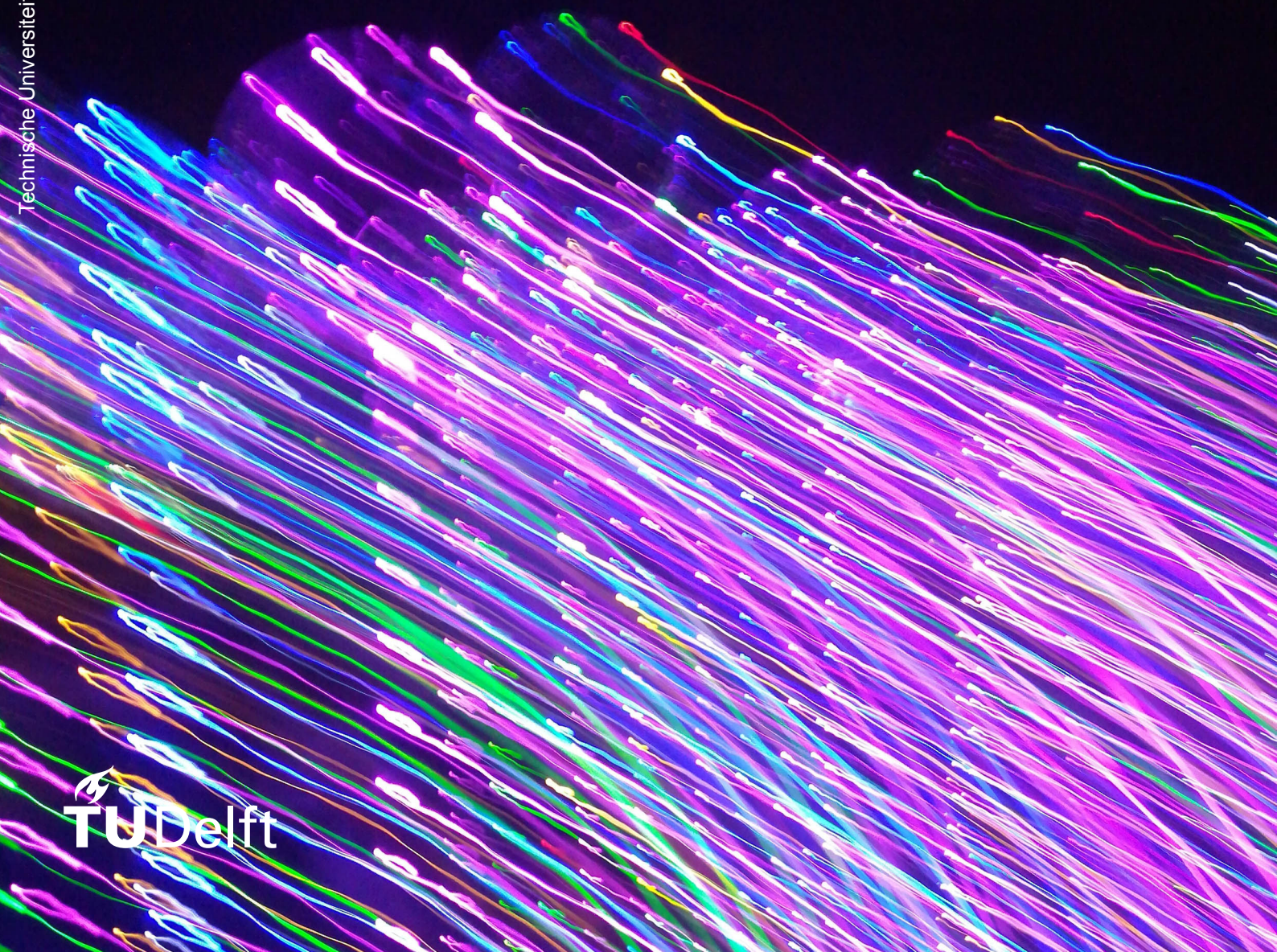
Tydi-Chisel

Collaborative and Interface-Driven Data-Streaming Accelerator Design

Casper Cromjongh



Technische Universiteit Delft

TU Delft

Tydi-Chisel

Collaborative and Interface-Driven Data-Streaming Accelerator Design

by

Casper Cromjongh

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday October 12, 2023 at 13:00.

Student number:	4494156	
Project duration:	September, 2022 – October 12, 2023	
Thesis committee:	Prof. Peter Hofstee,	TU Delft, IBM, supervisor
	Dr. Casper B. Poulsen,	TU Delft
	Dr. Zaid Al-Ars,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

In spite of progress on hardware design languages, the design of high-performance hardware accelerators forces many design decisions specializing the interfaces of these accelerators in ways that complicate the understanding of the design and hinder modularity and collaboration. In response to this challenge, Tydi has been presented as an open specification for streaming dataflow designs in digital circuits, allowing designers to express how composite and variable-length data structures are transferred over streams using clear, data-centric types. Earlier efforts in providing an implementation framework for Tydi managed to generate VHDL boilerplate code for Tydi interfaces, but offered limited design value over custom solutions due to VHDL's low abstraction level. In contrast, Chisel, with its high level of abstraction and customizability offers a suitable platform to implement Tydi-based components.

In this thesis, the Tydi-Chisel library is presented along with an A-to-Z design-process description for data-streaming accelerators. A stream-interface solution is presented that offers both compatibility with Tydi in traditional HDLs and maximum utility within Chisel through two intercompatible representations. In addition, design complexity is reduced through novel utilities like stream-complexity conversion, developed to alleviate interface specification mismatches between components. Using the presented toolchain and library, the amount of code required to specify Tydi interfaces for representative use-cases can be reduced several times compared to a Verilog description, while offering increased utility.

Tydi-Chisel aims to simplify the design of data-streaming accelerators through the integration of the Tydi interface standard in Chisel, along with helper components, syntax sugar, and verification tools. In combination Chisel and Tydi help bridge the hardware-software divide, making solo-design and collaboration between designers easier.

Preface

Before starting this project, I did not know the first thing about Tydi, or Chisel, and had only basic knowledge about practical computation hardware design. The concept of enabling better hardware accelerator design through better communication between components and people intrigued me, however. The practical use and absolute real world potential that Tydi, and as I came to realize the combination with Chisel, can provide has been a strong motivator for me in this project.

I got interested in this topic by the thesis opportunities presented at the end of Dr. Zaid Al-Ars' "Supercomputing for Big Data" course, which I recommend any to take if they have an interest in this field. I would like to thank Prof. dr. Peter Hofstee for helping me find a direction to take the research in and continued support and ideas, somehow always finding a time for a meeting. Peter and Zaid are both great supervisors.

This project can be described as the third generation of toolchain development projects, so there was a lot to learn about previous work in Tydi and Tydi itself. Therefore I must thank Matthijs Reukers, Yongding Tian, Jeroen van Straten, and others for answering all my Tydi-related questions and provide corrections. It was great to see Yongding's enthusiasm for Tydi-lang being re-lit by our collaboration on this project.

I can recommend anyone to learn Chisel and take an interest in Tydi. Finally, I hope the reader will experience some of the inspiration and intrigue I have felt myself.

Casper Cromjongh
Delft, September 2023

Contents

1 Introduction

Parts of the contents of this thesis also appear in “Enabling Collaborative and Interface-Driven Data-Streaming Accelerator Design with Tydi-Chisel”, a paper that was submitted to NorCAS 2023 [7].

1.1. Context: General

The deceleration in performance gain of CPUs signals the advent of the post-Moore’s Law era [44, 12]. Nonetheless, computational demands, especially from fields like machine learning and big data, continue to escalate rapidly. To meet these growing needs, there has been a pivot towards heterogeneous computing platforms that include GPUs and FPGAs. Yet, the process of algorithm implementation on FPGAs tends to be more prolonged and intricate compared to that on GPUs. While there are several frameworks, like HLS (High-Level Synthesis), OpenCL [37], and HLS4ML [24], designed to streamline FPGA development, challenges remain. These challenges are amplified in the big data domain, where developers can typically write a few lines of SQL to execute a query, whereas translating the same query to FPGA requires thousands of lines of hardware description code. Sampson [33] accentuates this disparity, as depicted in Figure 1.1, subsequently advocating for a transition from Hardware Description Language (HDL) to Accelerator Design Language (ADL).

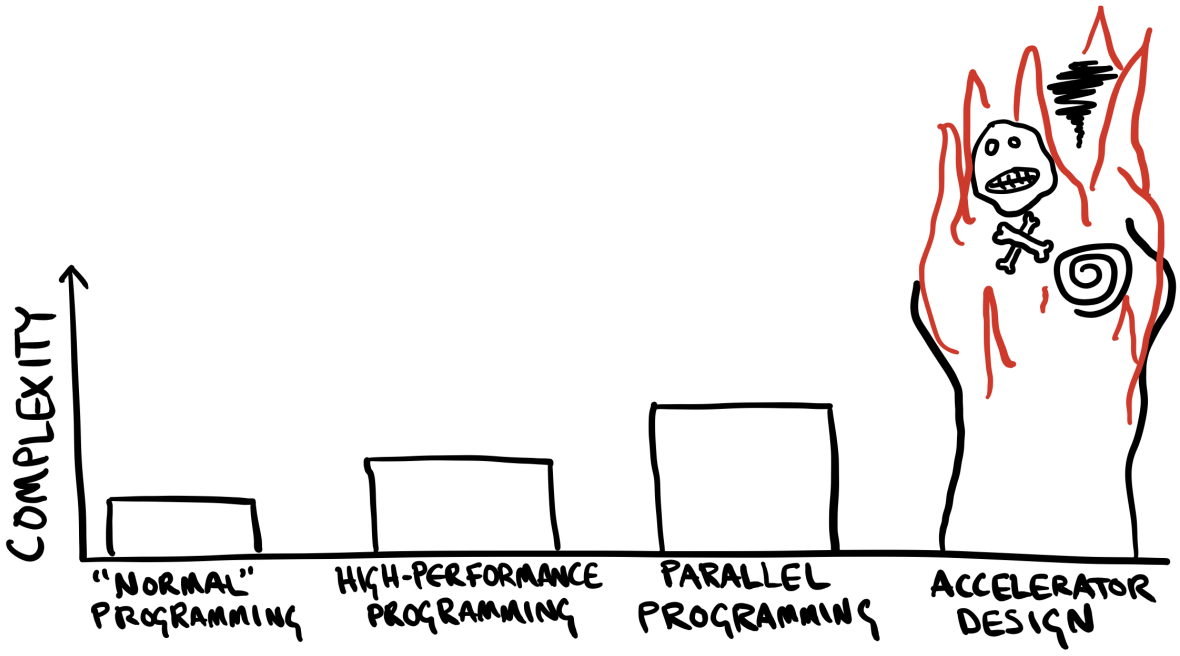


Figure 1.1: The difficulty of accelerator design [33]

1.2. Context: Tydi

A central challenge in designing data-streaming accelerators pertains to the transfer of structured and dynamically-sized data between components in a flexible manner. Peltenburg's observations provide insight into this conundrum:

"We have explored active (open-source) hardware frameworks, including classical HDLs and contemporary ones (Cλash, Chisel, and Spatial). All these HDLs support compound types that map onto bit-vectors (e.g., VHDL's record, Chisel's Bundle,etc.), and statically sized aggregate types, but lack inherit support for dynamically sized aggregate types mapped onto streamspace. This is unsurprising; the type systems of these frameworks reason only about space, but not about stream transfers—the latter being typically left to the designer—as the goal is to describe hardware just above the register-transfer level.

In libraries of some of the languages, abstractions for streaming dataflow designs are provided, e.g., Chisel's `DecoupledIO`. The abstractions move toward the level we envision when composing designs out of streams and streamlets, but only abstract the handshake mechanism for otherwise completely user-defined signals, lacking inherent support for throughput scaling of streams that is available in AXI/Avalon." [28, pp. 123].

To combat the issues faced by the authors, the idea for Tydi (Typed Dataflow Interface) was erected. Figure 1.2 aims to illustrate the difference by showing a metaphorical representation of a raw data stream, handshaked stream, and Tydi stream. A handshaked stream already allows the transfer of data in packets when the sender has valid data available and the receiver, like a conveyor belt in a supermarket where the transfer stops until the cashier is ready. These handshaked packets can even be formed with a structured bundle inside. However, in real-world scenarios, data that needs to be transferred often cannot be represented by a flat data structure, since the data is, or contains, (a) multidimensional list(s). Tydi offers a straightforward solution for these more intricate data-structures, and can be seen a salable set of conveyor belts, transferring packets with dimensionality data attached.

Without a common interface standard like Tydi, designers are often left designing their own communication protocol. While in simple cases this is often trivial, during development and optimization complexity generally rises. As the complexity rises, communication solutions will become more specific and divergent. When adapting IP or working between projects, this creates a lot of unnecessary overhead in specification and conversion. Debugging and interpreting the communication flow easily becomes very hard, quickly leading to the feeling expressed by Figure 1.1. Standards and tooling can help alleviate hardship in design choices, implementation effort, and debugging & interpretation. Tydi aims to be a standard that can offer this. In this thesis, we show how to create a Tydi-based communication flow specification for complex structured data and how Chisel is a suitable implementation platform using Tydi-Chisel.

Tydi development started with the introduction of the original specification by Peltenburg et al. in [28] . With it, a basic tool-chain set-up was created. Multiple follow-up projects ensued using or working on the Tydi ecosystem, listed in detail in Chapter 2. A second round of tool-chain projects [31, 41] developed a more complete and usable design pipeline. The Tydi-lang front-end allowed users to describe their data-structures, data-flow and components. In turn, the resulting Tydi intermediate representation could be used to acquire VHDL interface and boilerplate code to start implementation development. While this technically allowed hardware designers to develop Tydi-using components, establishing a presence for Tydi was still difficult. The value Tydi could add in an established ecosystem is clear. IP with Tydi interface specification could easily be adopted by other designers and teams. However, without an established ecosystem, Tydi's utility is still limited. While the available tooling was functional, creating a design with Tydi components was overall not much simpler than creating one's own project-specific solution. When awaiting the advantage of an ecosystem, the incentive for adoption should therefore come from ease of implementation for single projects. As mentioned, the VHDL boilerplate provided by the previous toolchain saves work getting started writing an implementation, but does not inherently make creation of the implementation much easier. VHDL, often parodied with the description "Verbose Hardware Description Language" has a very low abstraction level, comparable to assembly for software. As previously articulated, successful wider deployment of hardware accelerators can only be achieved by offering a higher abstraction level. Tydi-lang already provides a high-level way to describe the dataflow in a system in a way that is close to the software domain. For Tydi's success and achieving low effort hardware accelerator design, implementation of Tydi-using components must become simpler.

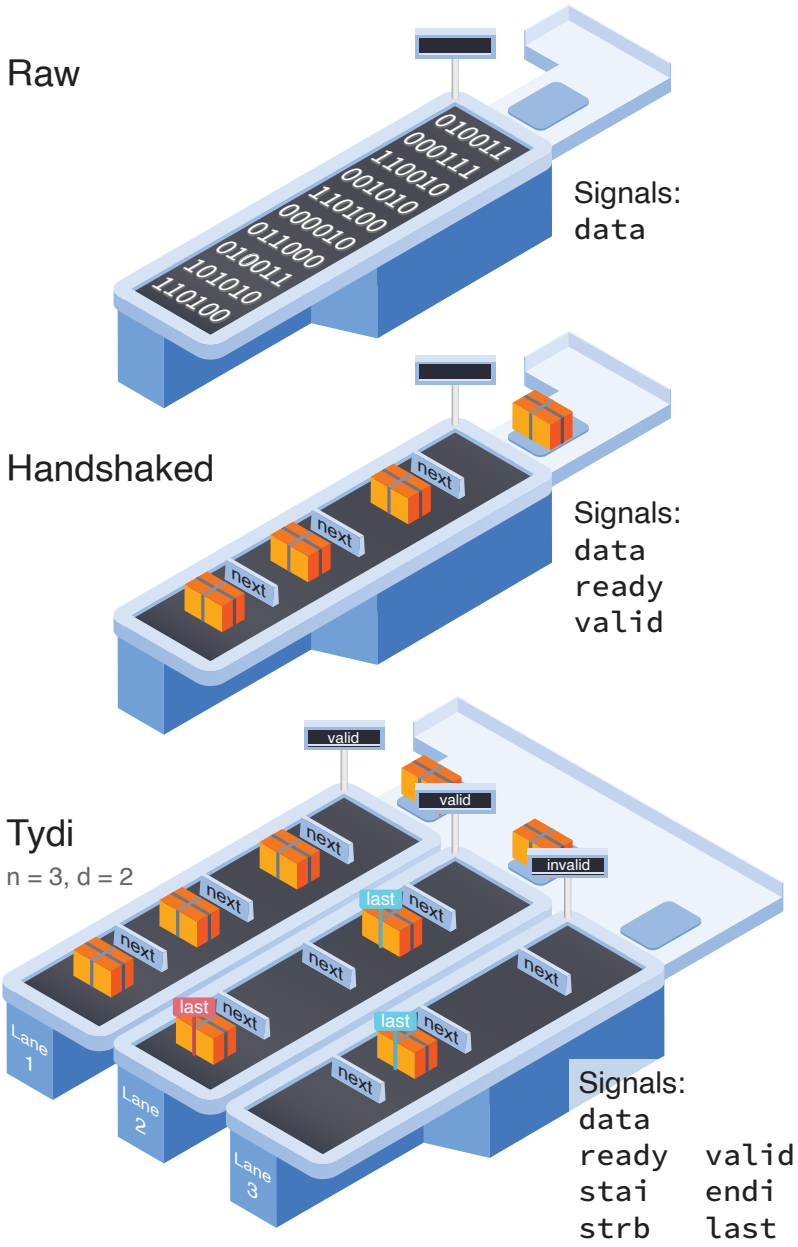


Figure 1.2: A comparison of stream types represented as checkout conveyor belts.
The Tydi stream has $n = 3$ lanes and a dimensionality of $d = 2$.
`stai`, `endi`, and `strb` relate to data-lane validity. `last` transfers dimensionality information.

Chisel has emerged as a promising way to achieve this transition. Chisel [5] aims to lower design complexity by providing designers more powerful design tools. These tools empower designers to craft highly parameterized generator components, seamlessly manipulate intricate signal-aggregates, and utilize high-level programming paradigms. Since Chisel is tailored for broad-spectrum hardware design, however, an opportunity exists to further refine the design process through a domain-specific strategy, particularly for data-streaming accelerators.

1.3. Problem statement & research questions

The narrative thus far presents two critical challenges impeding the widespread adoption and efficacy of Tydi. First, despite the strides made in rendering the Tydi ecosystem more accessible and complete, establishing Tydi's presence remains a challenging task. The potential value within an established ecosystem is evident - IP components with Tydi interfaces and communication specifications, ranging from low-level utilities to entire query-processors, could easily be adopted various designers and teams. Yet, in the absence of such an ecosystem, the potential utility of Tydi remains limited. Existing tools have not significantly simplified the process of designing of Tydi component implementations, making it comparable in complexity to creating a project-specific solution. While the VHDL boilerplate provided by the previous toolchain aids the initiation of implementation development, it does not substantially reduce the laboriousness inherent in creating the implementation. The low abstraction level of classic HDLs can deter wider utilization of hardware accelerators - an eventuality dependent on a higher abstraction level offering.

The solution potentially lies in the Chisel language, which emerged as a promising means to lower design complexity and equip designers with more potent design tools. An opportunity thus lies in integrating Tydi into Chisel. Chisel's broad-spectrum hardware design orientation leaves a gap for enhancing the design process via a domain-specific approach, especially for data-streaming accelerators. Accordingly, the main research question motivating this study is: how can the Chisel language be used within the Tydi ecosystem, simplifying the process of implementing Tydi-using components, thereby reducing data-streaming accelerators design complexity? To answer this main question, several more focused questions can be identified:

- What role can Chisel fulfill in existing and future Tydi-based toolchains?
- How can Tydi-concepts be effectively integrated in Chisel?
- What additional utilities are required to lower design-complexity?
- How can developed functionality be verified?

1.4. Contributions

This project makes several notable contributions to the fields of hardware description and data-streaming communication in relation to Tydi and Chisel. First, an overview is provided of Tydi specification, toolchains and projects up till now. This includes an updated stream description diagram, displaying stream signal layout, inclusion conditions, and role at different stream complexities.

Second, a library is created to enable and make accessible the employment of Tydi concepts in Chisel. This library allows intuitive declaration of all Tydi element types and stream interfaces in Chisel. Tydi streams can be expressed in intercompatible standard and detailed representations. The standardized representation, following the Tydi standard, ensures compatibility with external components. This compatibility furthers the scope of communication with components crafted outside of Chisel. The detailed representation featured in the library gives improved clarity and usability within components and tests. Nested streams are completely supported. For streams, several helper functions are included for recurring signal use-cases. A detailed analysis of implementation methods of Tydi elements and the stream representations is included.

Additionally, the library facilitates the creation of Tydi Modules. It provides a base class, with possibility for extension, alongside multiple utilities for common use-cases, similar to a standard library. Notable among these utilities are a stream duplicator, an "interleave" component that splits a multi-lane stream into multiple single-lane streams for easy processing, and importantly, a stream complexity converter component that can be deployed between components to convert any incoming stream to the lowest

source complexity, ultimately reducing design efforts in source and sink components. The project further contributes a syntax for chaining stream-processing components, presenting advantages akin to method-chaining in software.

For reciprocity with Tydi-lang, a Tydi-lang-2-Chisel transpiler is devised to convert Tydi types, interfaces, and components described in Tydi-lang-2 to Chisel code, thereby utilizing the Tydi-Chisel library. A reverse-transpiler is also created to share types and components created with Tydi-Chisel in the form of Tydi-lang-2 code. In this process, limitations of information acquisition from the Chisel library were encountered. These limitations, and opportunities, are discussed.

An all-inclusive design-pipeline for data-streaming hardware accelerators is outlined through the use of available tooling, accompanied by a simple illustrative example. This design-methodology demonstrates the applicability and efficacy of an interface-driven and dataflow oriented workflow. A more advanced example utilizing the majority of the developed tools and utilities follows the pipeline. The advanced example strongly illustrates the composibility of systems when working with Tydi-enabled components, strengthening Tydi's claims about added value. For a real-world scenario example, an industry-standard TPC-H query is examined and its top level components described together with streams and data-types. An analysis of the code writing effort saved by using the pipeline is supplied for all examples.

Last, testing utilities are developed, featuring a Chiseltest test-driver for Tydi stream interfaces, which encompasses helper functions for creating Chisel data literals and offers user-friendly enqueue and dequeue functions. Verification methods for aforementioned library utilities are described, which can be used as inspiration for test development for other components.

1.5. Document outline

This thesis consists of several thematic chapters. For easy overview, every chapter starts with a short summary in cursive text.

- In Chapter 2 a background of the work presented in this thesis and related work is provided. Background information is supplied on the Tydi specification, previous toolchain projects, and Tydi-using projects. Related work on alternative HDLs, streaming design, and accelerator design is supplied.
- Chapter 3 offers a short narrative on the previous toolchain and search for enhancement opportunities and use of Chisel's capabilities, together with an analysis of the usability and scope of the Tydi specification.
- Tydi-Chisel's main library contents are covered in Chapter 4. Descriptions and motivations are given for the implementation of Tydi elements, streams, and components. General utility components that can be used for common use-cases are listed and exampled. Particularly the stream complexity converter, a drop-in-between component to bridge stream complexity issues between source and sink.
- Interaction between Chisel and Tydi-lang is explored in Chapter 5, where the process of forward and backward conversion is described, together with limitations of Tydi-lang and dynamic system introspection in Chisel.
- In Chapter 6, a complete Tydi-driven design workflow is explored with several example use-cases. Examples include a minimum complexity processing pipeline design, an advanced, higher throughput design using several Tydi-Chisel utilities, and the macro components of the TPC-H 19 benchmark query.
- Chapter 7 describes testing utilities included in Tydi-Chisel as well as verification methods and results of the developed utilities and components.
- Finally, Chapter 8 summarizes the whole document and provides an overview of future work.

2 Background

In this background chapter, a short overview is given of technologies used in this project and related work. A synopsis of the Tydi specification and its history is provided. Chisel is placed in a context of emerging open source hardware description languages. Fletcher is presented as a related project to achieve acquisition and deposition of data from and to a host computer. Finally, related work on interface design and acceleration of data-streaming processing is discussed.

2.1. Tydi

2.1.1. Specification

The Tydi specification was first introduced in [28]. This initial version defines a methodology for representing composite, dynamically-sized data structures along with the physical-level streaming protocol. Later, a refined version of Tydi specification was released [6]. Based on this refined version, Reukers et al. developed an intermediate representation tailored for hardware circuit design using the Tydi framework, accompanied by a compiler for VHDL translation [31, 32]. In co-operation, a "front-end" language called Tydi-lang [41, 42] was developed to more naturally express Tydi concepts with a higher abstraction level. These projects laid the foundation of practical Tydi implementation and corresponding terminology that were not yet featured or fully developed in [28, 6]. The terms utilized within the Tydi intermediate representation and Tydi-lang and their meanings are summarized in Table 2.1. Section 3.1 provides additional details about the roles and interplay of the Tydi-IR and Tydi-lang projects. For a broader perspective, a comparative study between Tydi and prevalent protocols such as AXI and Avalon can be consulted in Table 4 of [28].

The basic data types used in Tydi intermediate representation are $Null$, $Bits$, $Group$, $Union$. The $Stream$ is a wrapper of basic data types, adding extra streaming properties, such as $complexity$,

Term	Type	Software equivalent	Chisel equivalent	Meaning
Null	Tydi logical type	Null	Bits(0)	Empty data, a stream of Null type will be optimized out.
Bits	Tydi logical type	Any primary data type	Any non-aggregate Data object	Represents data that requires x hardware bits to represent.
Group	Tydi logical type	Struct	Bundle	A tuple of several other logical types. Total hardware bit would
Union	Tydi logical type	Union	Bundle & tag	An union of several other logical types. Total hardware bit would
Stream	Tydi logical type	Bus	–	Represents a stream of a Tydi logical type. The stream can also
Streamlet	Tydi hardware element	Interface	Trait with IO definitions	Represents the port map of a component. This term is almost
Impl	Tydi hardware element	Class with functionality	Module	"impl" is the abbreviation of "implementation, representing

Table 2.1: Tydi terms and corresponding meaning

dimension, and *throughput*.

- **Complexity**: denotes the intricacy of the physical protocol. The present Tydi specification delineates eight distinct complexity levels, ranging from 1 to 8. A lower complexity value implies a more straightforward protocol, yet correspondingly, the component may necessitate increased complexity to guarantee data availability. Figure 2.1 visually represents the protocol of complexities at levels 1 and 8. Table 2.2 specifies constraints of streams for different complexity levels, while Figure 2.2 shows the difference in signal layout and usage for a stream at different complexities along with example transfers. It is noteworthy that a source port with a lower complexity is able to connect with a sink port of a higher complexity.

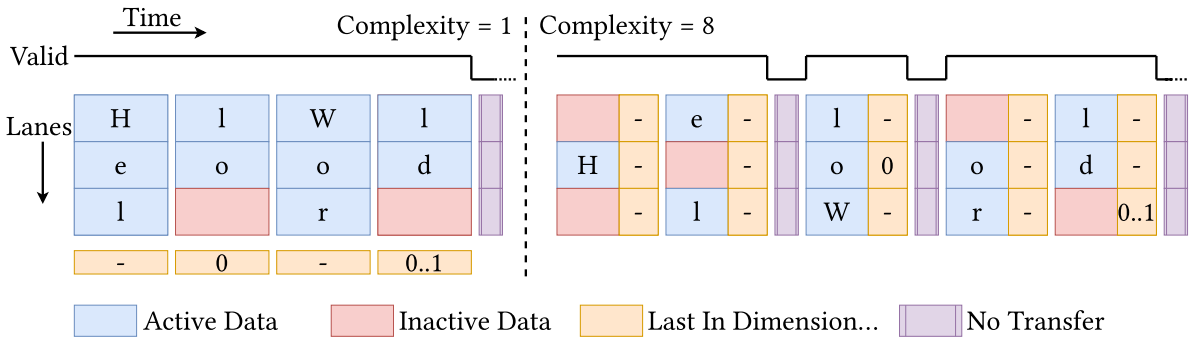


Figure 2.1: The stream complexity property [31]

- **Dimension**: indicates the dimension of data. Consider the representation of the phrase “she is a dolphin” in terms of data dimensions as it transits between components. Conceptually, this phrase can be parsed as a 2-dimensional array: `[ [s,h,e], [i,s], [a], [d,o,l,p,h,i,n] ]`. Given that each character requires 8 bits for representation, the appropriate streaming type for this data structure would be designated as `Stream(Bit(8), dimension=2)`.
- **Throughput**: indicates the designed throughput. Referring back to the streaming sentence example, if the throughput is specifically designed to be 3, then the total data lane would be 24 bits (8 bits per character multiplied by 3).

Tydi’s design flexibility promotes teamwork in engineering, enabling one group to concentrate on the source component and another on the sink. This adaptability in design also means components can be easily used in different setups without needing extra steps like manual protocol conversion.

2.1.2. Tydi-using projects

Several projects have emerged that utilize Tydi-related methodologies. Among these, Tydi-JSON [10] is a collection of Tydi-interfacing hardware components that can be used to create a JSON parser, written in VHDL. Building upon the foundations laid by [10] and [31], JSON-TIL [11] examines a provided JSON reference input, subsequently generating the requisite Tydi-IR (TIL) and VHDL files. This process facilitates the creation of a comprehensive JSON parser tailored to the specific JSON schema in question. Additionally, the VHDL-regex match generator [36] incorporates Tydi interfaces. This initiative enables the generation of hardware blueprints for regular expression matchers that operate on UTF-8-encoded strings.

2.2. Chisel

The call for higher abstraction levels in hardware design as described in the introduction resulted in the rise of high-level synthesis tools by major vendors and creation of a multitude of open source alternative hardware description languages [5, 19, 23, 20, 25, 18, 9, 21, 17, 4] with different paradigms and abstraction levels. Of these languages and frameworks, Chisel [5] (Constructing Hardware In a Scala Embedded Language), is one of the oldest and most established, while still undergoing active development.

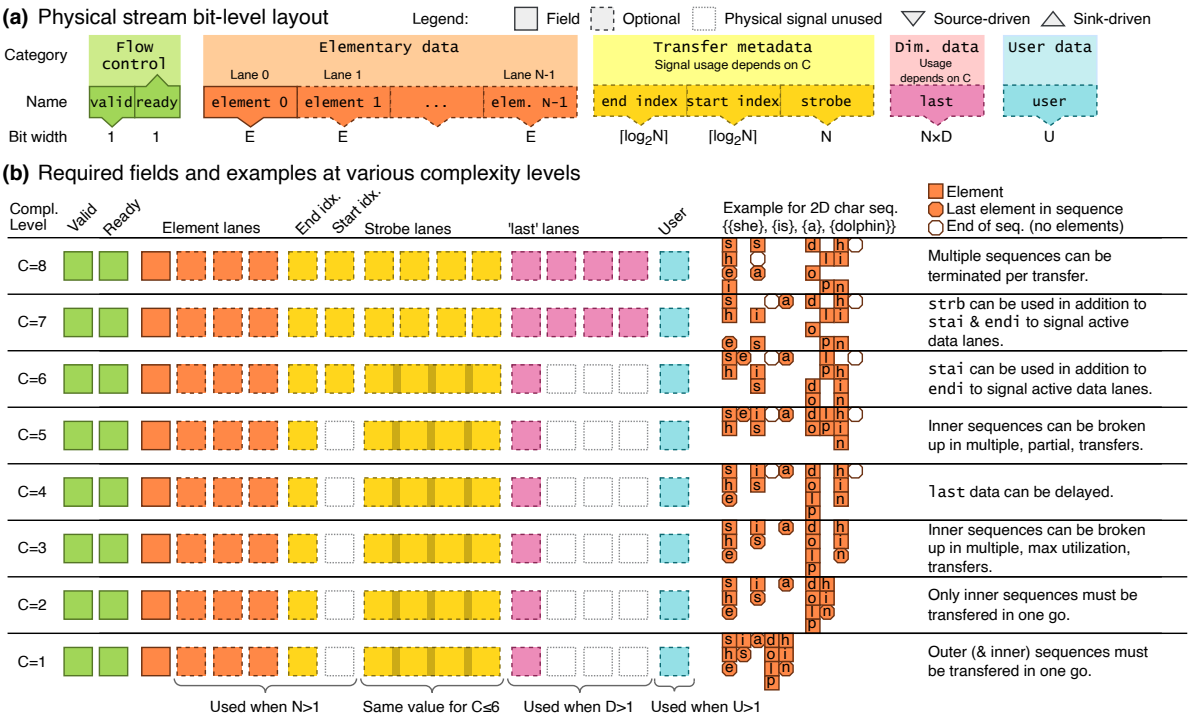


Figure 2.2: Tydi stream complexities diagram based on Figure 3 of [28], updated with info from [6]

C	Description
< 8	Only one sequence can be terminated per transfer.
< 7	The indices of the active data lanes can be described with a simple range.
< 6	The range of active data lanes must always start with lane zero.
< 5	All lanes must be active for all but the last transfer of the innermost sequence.
< 4	The last flag cannot be postponed until after the transfer of the last element.
< 3	Innermost sequences must be transferred in consecutive cycles.
< 2	Whole outermost instances must be transferred in consecutive cycles.

Table 2.2: Stream complexity limitations [6]

Chisel is an open-source hardware construction language developed to facilitate the design of highly parameterized hardware components. Traditional HDLs primarily focus on the structures and interconnections of hardware components. Chisel allows designers to leverage Scala's built-in features such as high-level abstraction and type inference features to describe components more efficiently. This allows for the creation of sophisticated hardware modules with reduced development effort. Importantly, designs written in Chisel are ultimately translated to low-level Verilog code, ensuring compatibility with existing digital design flows.

2.3. Fletcher

The Fletcher project [27, 26] was developed to facilitate the delivery of in-memory Apache Arrow data to hardware accelerators. To achieve this, Fletcher offers an automated toolset capable of generating VHDL components directly from data schemas. Complementarily, it provides a software framework tailored for efficient data delivery to these generated components. At its core, Fletcher serves as a comprehensive framework, designed to bridge FPGA accelerators with software tools and frameworks that employ Apache Arrow. However, despite Fletcher's capabilities in generating components for memory data access, the challenge of designing the processing circuits on FPGAs remains. This is an application domain where Tydi proves relevant, especially since in-memory data structures tend to be

both complex and dynamic.

2.4. Related work

The field of stream processing has been the subject of extensive research across varied contexts. This research trajectory has culminated in the development of multiple languages and frameworks for software-oriented stream design [3, 39, 8, 15] on multi-threaded CPUs and GPUs. Moreover, specific studies have focused on the intricacies of FPGA-oriented streaming [13, 34]. Notably, these studies primarily address data transfer at the bit stream level, often neglecting the complexity and dynamic nature of the data from software side. Beyond the specific research domains mentioned, [38] proposed a holistic language, meticulously crafted for universal streaming logic.

Simultaneously, the evolving landscape of hardware design workflows has given rise to innovative languages and representations [35, 16], although their objectives diverge from those of Tydi. Efforts have been made for their seamless integration with existing languages & frameworks to simplify the design process [22]. In response to the challenges posed by component interface compatibility, several industry standards have been established [1, 2, 14, 29]. However, these existing works focus on the hardware signals rather than an effective representation of complex data, which is addressed by Tydi.

The introduction mentions Chisel's `DecoupledIO` to create handshaked connections and reasons why it is insufficient for complex communication flows. With the *dsptools* library, Chisel also features a `DspBlock` component. A `DspBlock` implements signal-processing functionality with an interchangeable interface (TileLink, AXI4, APB, AHB, …). While this also offers more flexible implementation in a project, it is implementation centered, whereas Tydi is interface centered. Tydi therefore leads to interface-driven design, a successful concept in software development.

3

Chisel's place in the Tydi toolchain

In this chapter, a brief overview is given of process evaluation of the toolchain after previous projects, the search for options in utilizing Chisel, and a small review of the completeness of the Tydi specification. The IR output for Tydi-lang was changed from Tydi-IR to a JSON format for easier expansion of tools. Chisel is found to be capable of implementing Tydi concepts from a low to high level. Tydi's specification is found to be targeted towards low-abstraction HDLs and low-level transfer, missing descriptions or guides for communication as a whole.

3.1. Previous toolchain structure & motivation for a new toolchain

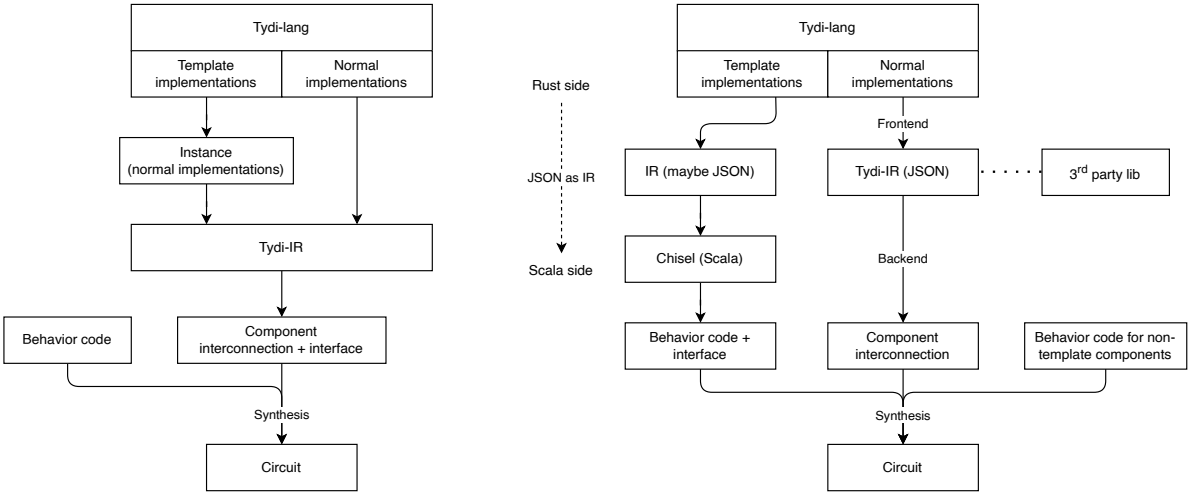
Initially no-one in the research group was very familiar with Chisel. Chisel appeared to be a promising research direction with its powerful hardware generation system. The exact extend of possibilities of the role that Chisel could fulfill with respect to Tydi and the role that Tydi could fulfill with respect to Chisel were unclear. The toolchain resulting from Tydi-lang and Tydi-IR [41, 31] is depicted in Figure 3.1a. Tydi-lang would be used to describe logical types (elements), streamlets, and implementations. The templates could either be 'normal' implementations, or templates for implementations that take type parameters. The Tydi-lang code would get get compiled and expanded to Tydi-IR, or `til` code. This `til` code would, through compilation, generate component interconnects and interfaces, and link corresponding behavior files in the output.

After the projects, it became clear that tooling with the custom intermediate representation was hard to further develop. While the domain specific grammar and syntax is suitable for specifying Tydi types and components, the need to parse and process this custom dialect restricted expansion to new languages and development of new tools. Tian proposed a new conceptual toolchain, shown in Figure 3.1b. As a more generally applicable intermediate representation, JSON was chosen. JSON can be used to describe all properties of objects, and is established as a portable data storage format. A new "back-end" could then parse the JSON description of the Tydi elements and components and convert it to a chosen output format. In fact, this idea was implemented in this work, the process of which is described in Section 5.1.

3.2. Exploration of a role for Chisel

As a first iteration and for lack of information about Chisel's exact capacity, the new toolchain concept included Chisel in a similar way as VHDL was included in the previous toolchain. As can be seen, a perceived possibility was to use Chisel to implement parameterized template components. Chisel's ability to describe parameterized component generators and Scala's advanced typing abilities seemed like a good fit.

Eventually, after exploration of stream implementation possibilities (see Section 4.2), it became clear that, thanks to polymorphic typing, little boilerplate code was necessary to use streams and build Tydi-enabled components. Types and interfaces could be described with similar verbosity to and ease as Tydi-lang. This meant Chisel could be used both as a general purpose implementation framework in combination with Tydi-lang and as a stand-alone tool to develop projects.



(a) Previous/legacy toolchain

(b) Ideas for renewed toolchain

Figure 3.1: Toolchain diagrams for brainstorming Chisel's role

3.3. Role & limitation of the Tydi spec

The Tydi standard was developed with at least compatibility in mind with traditional HDLs such as VHDL and Verilog. In that sense it is not very imaginative with stream representation and capabilities. The advantage of this is that it can likely be implemented in any HDL. The disadvantage is that extra effort is required to offer usability to the user while still retaining standard compatibility, as described in Subsection 4.2.2. Additionally, the Tydi specification is limited to data transfer specification, and does not prescribe any propositions for communication where multiple streams are involved. This is not surprising, projects can have wildly different use-cases and requirements. A component might function as a subordinate component in a processor with a request response interface. Or, a query must be executed with a lot of child streams for strings in the row data. Yet, as an advocated specification that helps transfer complex structured data, the lack of further (suggested) specification of communication can be seen as a limitation. This limitation and need for specification is briefly discussed in Subsection 6.1.3.

4

Tydi in Chisel

This chapter provides an overview of the implementation methods and choices for integration of Tydi concepts in Chisel. Implementation of Tydi's datatypes is outlined. Streams are implemented in a Tydi-standard compatible variant and a high-utility detailed representation, making use of Chisel's strong typing and aggregate signal support. Different base-classes are described for component implementation development. Various utility components that can be used in composing systems are outlined. Notably, the stream-complexity converter, a component that can take in a high complexity stream, buffer it, and output a stream at the lowest complexity. Implementation challenges, solutions, and design choices are given.

4.1. Tydi elements

Elements are an essential part of the Tydi specification. They are the building blocks of the data structures used for communication. Even the streams are elements, since they can be included in e.g. *Group* elements. This must be reflected in the implementation and usage of Tydi elements within Chisel. The following Subsection will give some information about the implementation method considerations. The next Subsection will give some details about the specific types.

4.1.1. Basic implementation

Every element must, from a type perspective, be recognizable as being a Tydi element. Therefore, every element must share a common super-class or trait. Finally, it must be usable within Wires and IO, for use in Subsection 4.2.3's detailed representation. For this project, a *trait* called `TydiEl` was created that extends Chisel's `Bundle` class. This is convenient for implementation of the *Group* and *Union* elements, since they have multiple sub-fields/elements.

The `Bundle` class includes several methods, such as `getElements` that returns a sequence of the bundle's elements. This functionality is essential for the connection of detailed and standard form streams (see Subsection 4.2.4). Behavior like `getElements` requires knowledge about the structure of the `Bundle` sub-class in the class itself. In interpreted languages, this is often easy. In Scala this is more difficult, but still possible through two types of *reflection*. Chisel handles this by employing a *compiler plug-in*. This plug-in, among other things, provides the information that `getElements` uses. By sub-classing `Bundle`, the burden of implementing this inspection is saved.

Bit elements only contain a single value. It would, therefore, be convenient to express this class as a subclass of the more fundamental `Data` and not `Bundle`. A problem here is that neither the ground types (such as `UInt`, `SInt`, `Bool`, ...), nor the `Data` class can be manipulated. The ground types cannot be extended either, for their classes are marked as `sealed`. It was therefore chosen to implement all elements as a sub-class of `Bundle`.

4.1.2. Types

Null

The *Null* element is meant to signify the absence of a value. Its implementation accordingly is a bundle without any fields that therefore can not have a value.

Bits
A *Bit* element is the lowest form of element in Tydi, signifying some ground datatype of a certain bit-width. As already explained in the previous Subsection, it must still be implemented as a sub-class of `Bundle`. Straightforwardly, it contains a single field called *value* of an unsigned int of specified width. One can override the `value` field or simply create a *Group* with a single `value` field.

Group
A *Group* represents a structure with multiple (Tydi) element fields. This directly mirrors what a bundle already does in Chisel.

Union
In Tydi, a *Union* is similar to a *Group*, except that only a single field carries significance at a time. Which field that is, is indicated by a `tag` field that has the active field index as value. It is therefore dissimilar to unions in, for example, C, where all fields share the same data. Instead, it bares more resemblance to a mux. Union behavior is clearly explained in [31], specifically Section 2.2 and Figure 2.3.

An accompanying `object` can be specified with value constants for user-friendly assignment to the tag field. Field indexes need then not be remembered. This methodology is shown in Figure 4.1. Generation of such an object is automated by the Tydi-lang-2-Chisel transpiler, see Section 5.1.

It would be convenient if a `Union`-extending class automatically determined the required `tag` field width from the union's elements. This is, regretfully, not possible. It is unsure if this is because, in Scala, a superclass constructor is called before the subclass constructor, or because a `getElements` in the superclass constructor call will not yield information about the subclass.

4.2. Streams

In Tydi-Chisel, two representations of Tydi streams exist. A standard representation, and a detailed representation. Since we concern ourselves with hardware design here, these are both variations on the *physical* stream description. The standard representation follows the Tydi specification [6] as close as possible. It is meant to be compatible with other HDLs such as VHDL and Verilog. This way Tydi-using components designed in Chisel can be used in projects using different HDLs. The detailed representation focuses on offering maximum usability in Chisel. Both representations were made to share a common base class (also a `TydiEl`), share the same signal names, and can be connected to each other at will.

4.2.1. Standard representation

Standard representation streams, are, in Tydi-Chisel, a direct implementation of the *PhysicalStream* as described in the specification. Parameters and signals are exactly the same. A physical stream is constructed in the spec as $PhysicalStream(E, N, D, C, U)$. The parameters of a physical stream are as follows:

- `E` Element type
- `N` Number of lanes
- `D` Dimensionality
- `C` Complexity
- `U` User signals

This is directly reflected in the instantiation of a standard stream in Chisel:
`new PhysicalStream(e, n, d, c, u)`. The number of lanes N is assigned directly, instead of working with the logical stream parameter throughput (t). A physical stream's signals are `ready`, `valid`, `data`, `stai`, `endi`, `strb`, `last`, and `user`. The standard representation is only meant for usage in IO ports and components that do not operate on the data of a stream itself. In other cases, the detailed representation can better be used. More information about interconnection and usage will follow in Subsection 4.2.4.

4.2.2. Why standard is not enough

The Tydi specification considers the `data` signal to be a least-index-first concatenation of the data lanes. The idea behind this order is that it is the most natural to work with. E.g., taking the first bit index will result in the first bit of the first element. A packing order of elements is not explicitly stated, but could be derived from the `fields` function described in [6, 2.4. Logical streams]. Whichever packing is considered, this concatenated `data` signal representation requires subword reads and writes from and to the signal. A subword is a range of bits in a bit-vector signal. In VHDL, one would do this by either reading or writing to `data(y downto x)`. Working with indexes like this is cumbersome and prone to error. The Tydi specification mentions alternative representations "to improve code readability in hardware definition languages supporting array and aggregate constructs (record, struct, ...)". It recommends only using it within components, keeping outside connections standard for compatibility, but the value of other representations is recognized.

Chisel's strong support for aggregate types, such as vectors and bundles, seems to offer better usability in hardware design. Working with bit-vectors to manipulate data does, therefore, seem like a poor strategy. In fact, the Chisel docs state: "Chisel3 *does not support subword assignment*. The reason for this is that subword assignment generally hints at a better abstraction with an aggregate/structured types, i.e., a `Bundle` or a `Vec`.". In Chisel, a subword can be *read* using `data(y, x)`, but not written. Clearly, working with structured types is not only a more usable option, but also the suggested way to work from within Chisel's philosophy.

4.2.3. Detailed representation

As described in the previous Subsections, the detailed stream representation goal is to offer maximum utility within the Chisel environment. This can be done by making use of bundles and vectors. Specifically, the detailed stream representation is implemented as a polymorphic class. The parameters for its construction are the same as for the standard representation, but their significance is different. Instead of working with just the `TydiEl` interface of the passed element and using the width, the `PhysicalStreamDetailed` class is type-parameterized based on the *element* data and the *user* data. Its signature is as follows:

```
class PhysicalStreamDetailed[Tel <:TydiEl, Tus <:Data](private val e: Tel, n: Int = 1, d: Int = 0, c: Int,
var r: Boolean = false, private val u: Tus = Null())extends PhysicalStreamBase(e, n, d, c, u)
```

The class still accepts the same parameter types, but by polymorphically parameterizing the class type, more information becomes available that can be used in declarations:

```
val data: Vec[Tel] = Output(Vec(n, e))
val user: Tus = Output(u)
```

The `user` and `data` signals get the type of (vector of) the exact datatype that is passed. In other words, if a `Group` is passed as the element argument of the stream, the `data` property becomes a signal of a vector of this group type, instead of a bit-vector.

Listing 4.1 gives an example of a nested stream, similar to the timestamped message of [28, Figure 2.b], but with a *Union* instead of just a character. This is a good test-case for nested stream syntax and usability. Because the `data` and `user` signals now have structured types, they can be used for direct reads and assignments. This type information is also available to IDEs, offering auto-complete and live type-checking support. See Figure 4.1 for an example situation working with the detailed representation of Listing 4.1. The stream's `el` method is just a shortcut for accessing the first data lane, `data(0)`.

Listing 4.1: Example nested timestamped message example

```
class NestedBundle extends Union(2) {
  val a: UInt = UInt(8.W)
  val b: Bool = Bool()
}

object NestedBundleChoices {
  val a: UInt = 0.U(1.W)
  val b: UInt = 1.U(1.W)
}

class TimestampedMessageBundle extends Group {
```

```
  private val charWidth: Width = 8.W
  val time: UInt = UInt(64.W)
  val nested: NestedBundle = new NestedBundle
  val message = new PhysicalStreamDetailed(BitsEl(charWidth), n = 3, d = 1, c = 8)
}

stream: PhysicalStreamDetailed[TimestampedMessageBundle, Null] = PhysicalStreamDetailed(new
    timestampedMessageBundle, 1, c = 8)
```

```
// → Assign values to logical stream group elements directly
stream.el.time := System.currentTimeMillis().U
stream.el.message.data(0).value := 'H'.U
stream.el.message.data(1).value := 'e'.U
stream.el.message.data(2).value := 'l'.U
stream.el.nested.a := 5.U
stream.el.nested.b := true.B
stream.el.nested.tag := NestedBundleChoices.b  // Using object value to set
```

```
tag                                   UInt
b                                     Bool
a                                     UInt
getStreamElements   Seq[PhysicalStreamDetailed[_, _]]
```

Figure 4.1: IDE assistance example

The detailed physical stream class contains utility methods to quickly produce a standard form stream from the detailed instance. Since the parameters are the same, this is straightforward. This method can also directly connect the two streams. How this is done is explained in Subsection 4.2.4. The reverse cannot be done, since the standard representation only has knowledge about the `TydiEl` type and can therefore not properly parameterize the detailed representation.

Limitations of nested streams

A drawback of the detailed representation implementation as elements inside other elements presents itself when working with multiple lanes. Every lane will contain the complete element structure, including nested streams. This breaks with the Tydi standard, where a nested stream in the hierarchy is only exposed once. The nested streams of lanes other than the first can be marked `DontCare`, however, and be left unused. In the FIRRTL to Verilog compilation step, unused signals will be trimmed.

Additionally, though the stream signals of the standard representation should be compatible with the Tydi specification, the naming convention is not yet followed. A strategy to achieve this probably involves name generation by the Tydi-lang-2-Chisel tool and use of one of Chisel's methods to control signal names[1]. Signal names generated by the `til` tool [30] deviate from standard representation slightly as well, as VHDL by default does not support two consecutive underscore characters. Some work to standardize across tools is, therefore, still required.

Failed implementation method

The first attempt to create a detailed physical stream was based on a `Module`. The module had the IO associated with a detailed physical stream, and an IO port for the standard representation. The advantage of working this way was that all signals between the detailed representation and the standard representation could automatically be connected upon instantiation of the module class. There are three problems with this approach. The first is that, since the class is based on `Module` and not `Bundle`, the detailed representation does not have the same base-class as the element-manipulating elements. This is just an inconvenience that could probably be worked around with common traits. The second is that nested streams are not accessible and so unusable. This is due to the fact that a nested detailed stream would be a module *within* the outer stream's module. Signals that are not a module's IO are inaccessible from outside the module. Since the inner module's IO are just signals in the outer module and not IO, the signals cannot be used. Third, this model assumes that one is always working with a detailed stream and a standard stream equivalent, connected together. This offers little flexibility. This implementation method was therefore discarded.

[1] See the naming cookbook: https://www.chisel-lang.org/chisel3/docs/cookbooks/naming

Limitations of anonymous bundles
From a syntactical perspective, it should not matter if a class is defined as named class, or declared as anonymous class where it is used. Due to a limitation in the Chisel compilation framework, anonymous classes cannot reliably be used in Tydi-Chisel. Listing 4.2, using an anonymous class, will therefore not work well. Subsection 4.1.1 already talks about the enumeration of bundle and therefore `TydiEl` elements. This plug-in functionality only seems to work for named classes, however. This is not a problem when referencing the bundle in code, but functionality that uses the element enumeration will not work as it should. Connection of bundles is an example of this. More details about this process will can be found in the next Subsection.

Listing 4.2: Using an anonymous bundle for the nested group

```
class TimestampedMessageBundle extends Group {
  private val charWidth: Width = 8.W
  val time: UInt = UInt(64.W)
  // It seems anonymous classes don't work well
  val nested: Group = new Group {
    val a: UInt = UInt(8.W)
    val b: Bool = Bool()
  }
  val message = new PhysicalStreamDetailed(BitsEl(charWidth), n = 3, d = 1, c = 8)
}
```

4.2.4. Connection & syntax

The standard and detailed stream representation are both vital to creating hardware components that are both easy to write and compatible with other languages and projects. To be able to use both, it is essential that the signal information can easily be exchanged between formats. Because of the difficulties mentioned in Subsection 4.2.2, this is not trivial. Subwords cannot be written to and the order of items is reversed in an array (RTL) relative to a bit-vector (LTR).

Four situations can be identified for connecting streams:

1. Standard to standard
2. Standard to detailed
3. Detailed to standard
4. Detailed to detailed

Situations 1 and 4 are trivial, since for compatible streams, the signals and structures will be exactly the same and source and sink can directly be connected to each other. Situations 2 and 3 involve the outlined conversion difficulties. Although their solutions are not the same, both methods start with obtaining a deterministic packing order of the fields. To remind the reader, every Tydi stream consists of a tree of data elements and streams, called element manipulating and stream manipulating types. For the packing of the data in a bit-vector, we need all the elements that are not streams. Indeed, sub-streams in the hierarchy should get their own (standard) physical streams. To achieve this, a depth-first recursive function is defined that gets the element's child elements and does a `flatMap` operation on the result of all `TydiEl` elements that are not streams. Listing 4.3 shows this function's implementation. This process is similar to the $fields$ method in [6].

Listing 4.3: Recursive acquisition of all element fields

```
/** Gets data elements without streams. I.e. filters out any `Element`s that are also streams */
def getDataElements: Seq[Data] = getElements.filter(x => x match {
  case x: TydiEl => !x.isStream
  case _ => true
})

/** Recursive way of getting only the data elements of the stream. */
def getDataElementsRec: Seq[Data] = {
  val els = getDataElements
  val mapped = els.flatMap(x => x match {
    case x: TydiEl => x.getDataElementsRec
    case x: Bundle => x.getElements
    case _ => x :: Nil
```

```
    })
    mapped
  }
```

With this list of data-fields in hand, signal connection can take place. In situation 2, standard to detailed, all fields must be set to the correct subword in the `data` bit-vector. Listing 4.4 shows this process' implementation. Every lane will contain the collection of fields given by the `getDataElements` functions. Then, because of the RTL order of data in the bit-vector, the data fields list is reversed and iterated over using a `fold` operation. The starting value is the bit-index of the start of the lane. Then, every data field gets its value assigned based on the current index and the field's width, adding the width to the start value for the next field. This process is repeated for the `user` data fields, but then without the different lanes. `this` refers to the detailed stream and `bundle` to the standard stream.

Listing 4.4: Connection of bit-vector subwords to data fields

```
// Connect data bitvector back to bundle
for ((dataLane, i) <- this.data.zipWithIndex) {
  val dataWidth = bundle.elWidth
  dataLane.getDataElementsRec.reverse.foldLeft(i*dataWidth)((j, dataField) => {
    val width = dataField.getWidth
    // .asTypeOf cast is necessary to prevent incompatible type errors
    dataField := bundle.data(j + width - 1, j).asTypeOf(dataField)
    j + width
  })
}
// Connect user bitvector back to bundle
this.getUserElements.foldLeft(0)((i, userField) => {
  val width = userField.getWidth
  userField := bundle.user(i + width - 1, i)
  i + width
})
```

For situation 3, detailed to standard, the subword assignment problem is encountered. The way around this is to not assign parts of the bit-vector signal at all, but to create a new signal and assign the signal. This signal is then constructed purely by concatenation of data field signals. Care needs to be taken to maintain the right order of signals, but the code, shown in Listing 4.5 is otherwise quite simple. `this` refers to the standard stream and `bundle` to the detailed stream.

Listing 4.5: Concatenation of data fields to signal and assignment to bit-vector

```
def getDataConcat: UInt = data.map(_.getDataConcat).reduce((a, b) => Cat(b, a))
def getUserConcat: UInt = user.asUInt

...
this.data := bundle.getDataConcat
this.user := bundle.getUserConcat
```

To verify that the packing by these connection methods is done correctly, a Tydi compliance test was created. Details can be found in Subsection 7.2.1.

Direction of streams

In Tydi, streams can have a *forward* or a *reverse* direction. In HDLs, generally, there is an *input* and an *output*. In Chisel, if no direction is specified, an IO signal is considered to be an output. Specific directions can be indicated by wrapping a data type in `Input` or `Output` before turning them into IO with the `IO` call. Bundles, luckily, can have different directions for sub-signals. This is not only useful for the reverse-direction `ready` signal, but also for nested streams with reverse direction in the detailed representation. When a bundle is wrapped in `Input` or `Output`, the internal directions are overwritten. Instead, the `Flipped` function can be used to reverse the direction of all nested signals. Unfortunately, there is no way to *acquire* the direction of a signal in code. This limitation is further discussed in Subsection 5.2.2. This makes it slightly harder to ensure all signals are registered in the correct direction, and users have to rely more on Chisel's low-level errors. In the future, better error detection and messaging could be implemented. When a stream is set to *reverse* direction, it will execute a flip on all its child streams. Flipping can only be done *before* registering the signals as IO. When using the signals as wire instead of IO, the situation becomes even more uncertain, for wires must be connected at two ends, and so have no explicit direction. Therefore, it is up to users to clearly

denote their signal directions. Assistive error handling could assist users in cases where a low-level error occurs.

Syntax
Chisel knows a few connect operators that are used for connecting signals together. The documentation mainly mentions `:=`, known as the mono or “strong” connect operator, and `<>`, known as the bulk connect operator. Other available options are `:#=`, the mono connect operator, `:<>=`, the bi-connect operator, and `:<=`, the aligned connect operator, and `:>=`, the flipped connect operator. These all have different semantics in how they connect sub-signals in aggregate signal types. For simple signals, `:=` is normally always used.

When a stream is connected to a similar representation, their bundles will be the same (with mirrored direction if IO ports), and the `:<>=` operator can be used to connect them. Connecting different representations is more intricate, however, as outlined previously. Since streams are considered as one signal or communication channel, it is advantageous to use a single connect operator to connect all sub-signals correctly. The strong connect operator (`:=`) was chosen for this purpose. By sticking to this notation, it is expected that it will feel intuitive to existing Chisel developers and be easy to understand for new ones. These operators are defined as methods on the stream classes and can be used as operators by Scala’s infix notation. A connect operator method is defined for every situation in the list earlier this Section. Listing 4.4 and Listing 4.5 contain snippets from these connect functions.

4.3. Tydi components

With elements and streams covered, components can be designed. Referring back to Table 2.1, Chisel’s equivalent of an implementation is a module. This Section will describe library methods and classes related to writing and working with modules in a Tydi context.

4.3.1. Module bases

In principle, writing a module and using Tydi streams is enough to call something a Tydi component. To offer some additional functionality and abstraction, some base classes were created.

Tydi module
Modules come in three flavors in Chisel: `Module`, `RawModule`, and `BlackBox`, all being variants of `BaseModule`. General purpose components will use `Module`. When one or more non-default clock and reset signals are required, `RawModule` can be used. `BlackBox` modules are used for interfacing with external component and are not emitted to Verilog.

For most circumstances, a `Module` will be used. Accordingly, the base module for Tydi components, called `TydiModule`, is based on Chisel’s `Module`. Though in part writing components based on `TydiModule` is done for conveying semantic meaning, `TydiModule` also offers some functionality. This is mainly related to the reverse transpilation, discussed in Section 5.2. The class defines some overloaded methods to e.g. keep track of the registered nested `Module`s, get the ports, etc. In the end, basing all components on `TydiModule` allows for future expansion of base functionality.

Processor base
The processor base is a class that was originally developed for the multiprocessor utility of Subsection 4.4.3. Essentially, it is an abstraction to save boilerplate code when manually defining components. That is, not generated from Tydi-code (see Section 5.1. Its constructor allows creation of a Tydi component with an input and output stream of specified type. Standard streams, `in` and `out`, are exposed as IO. Detailed streams of the specified type are set-up and connected to both the standard representation IO streams and each other. By default, the component thus creates pass-through. Connections can be overwritten. This makes it possible to very simply define a filter component that controls the strobe lanes based on some criteria. In fact, this would only require a single line in simple cases. A “simple” version is also available that does not include detailed streams. This is relevant for components that just describe stream connections for nested modules and don’t include logic themselves. Both versions are based on an abstract signal definition class (`SubProcessorSignalDef`), that defines the presence of the `in` and `out` streams. This is considered to be a standard interface for simple, single-stream components. This interface is, at the moment, required for the pipeline notation of next subsection.

4.3.2. Syntax sugar & overloading

When writing a component's implementation, first defining a module instance, and then connecting its input and output streams is rather verbose and makes it difficult to visualize the dataflow. In software, a popular paradigm is "method-chaining". In method chaining, each method acts on the output of the previous method, forming a chain. Method-chaining is used by many big-data frameworks because of its convenience and conciseness. Therefore, a similar pipeline notation was developed to more naturally formulate a data-stream processing pipeline and provide a better overview of what is happening to a data-stream. This notation is shown in Listing 4.6. The `processWith` method instantiates the module it gets passed, connects the input stream of the module to the referenced output stream, and returns the module's output stream for further chaining. At the moment, this only works with modules adhering to the `SubProcessorSignalDef` interface, with one input and one output stream. This could later be extended for more advanced configurations. The `convert` method connects a stream complexity converter (as discussed next Section) with a specified buffer size similar to the `processWith` method.

Listing 4.6: Example data-streaming processing pipeline with pipeline notation

```
class PipelineExamplePlus extends PipelineExamplePlus_interface {
  out := in.processWith(new MultiNonNegativeFilter)
           .convert(bufferSize)
           .processWith(new MultiReducer(n))
}
```

4.4. Utilities

To advance the goal of making Tydi-based data-streaming hardware design easier, several utilities can be defined. These utilities fulfill the role of "standard library". Their intent is to take away design-effort in often occurring use-cases. Since the exact use-cases cannot be predicted, these utilities should be as general as possible. The utilities are consequently often (polymorphically) parameterized to be made corresponding to the user's situation. This Section will describe several utilities.

4.4.1. Complexity converter

One of the more important and complicated developed utilities is the stream complexity converter, or complexity converter for short. The idea behind this component is that it can be inserted between components with incompatible sink and source complexities but compatible data-layout. In Tydi, compatible sinks and sources can be connected if the complexities $C_{\text{sink}} \geq C_{\text{source}}$. Designing a sink component requires more effort to take in a high-complexity stream, whereas designing a source component outputting a high-complexity stream requires less effort. The complexity converter's role becomes apparent. By creating a universal component that can convert a high-complexity source stream to a low-complexity sink stream, design efforts can be saved on both ends, while retaining correct operation. Section 4.5 will detail the design of this component.

4.4.2. Stream duplicator

The stream duplicator is a standard component that was already introduced in [41] to split one output stream into two or more output streams, each going to another component. Generally, an output can be connected to multiple inputs, but not the other way around. Since the Tydi signal definition (Subsection 2.1.1) includes the `ready` signal in reverse direction, being an input for an output stream, one cannot simply connect all wires together. Receiving components have independent sinks with `ready` signals. A transfer must logically not happen before all components are ready to receive data. The ready pulse is therefore only sent to the source when all sinks are ready. Another strategy would be to transfer data to a sink as soon as it is ready and then set `valid` low.

4.4.3. Interleave/multiprocessor

The interleave/multiprocessor component is a component that offers easy workload parallelization for multi-lane streams carrying independent data elements. In essence, the component is an extension of the functionality that the stream duplicator offers. The idea is that a stream with N lanes is split into N separate streams with a single lane. Every lane is connected to a processing element (called processor here) that operates on the data from that stream. While the splitting of data, strobe, and dimensional

information is trivial, special care must be taken in connecting and driving the `ready` and `valid` signals. The first step is similar to the way the stream duplicator creates multiple streams, except in this case the streams carry independent data. A similar reverse process must be repeated after the processing elements. The processors should not receive a ready pulse until all processors have valid output ready. Only then can all information be sent as a single transfer at the interleave output. Essentially the streams are synced at the processor inputs and outputs. A framework- and vendor-agnostic stream synchronization VHDL component has been developed at the ABS group before[2].

Integration of the multiprocessor has been kept as simple as possible. The module generator class takes arguments about the stream datatypes, number of lanes, number of dimensions, and most importantly, a module Definition of a processor. It is used in the advanced example presented in Subsection 6.2.2. This processor should extend the Processor Base of Subsection 4.3.1. In combination with the pipeline syntax of Subsection 4.3.2, an interleave operation can be inserted in a stream in a single line.

4.5. Complexity converter implementation

4.5.1. Desired functionality

As mentioned, the complexity converter's job is to take in a stream with high complexity, and to output the same data in a stream of low complexity. It should function such that it can be inserted between two components without either of the source or sink noticing the difference from being connected to respectively a high complexity sink and a low complexity source.

More technically, the component take in the data and dimensionality information at C_{source}, buffer and compress it, and output the information again at the required C_{sink}. Table 2.2 denotes the constraints for stream behavior at different complexities.

The mentioned buffering of information is evident when looking at Figure 2.2. Higher complexity streams are not required to send the data in one continuous cascade. Instead they can use the `stai` ($C \geq 6$), `endi` ($C \geq 5$), and `strb` ($C \geq 7$) signals to turn individual lanes off. At $C \geq 3$ `valid` can go low in the middle of a sequence, effectively pausing the input stream. In general it can be said that the high complexity input signal *can* be more fragmented and the low complexity output *must* be less fragmented as dedicated by the complexity rules.

For complexities $C \geq 4$, the `last` flag can be postponed, i.e. sent after the element data. For $C < 4$ this is not allowed, which means the complexity converter component should be able to *re-align* the dimensionality information with the element data. This is one of the harder problems, and its solution is discussed in Subsection 4.5.3. But first, the general implementation.

4.5.2. Implementation information

The idea behind the stream complexity conversion is quite simple. The design requires some intricate operations that are tricky to implement. Two implementation paths seem plausible. One is making multiple converter layers that each reduce the stream complexity one or a few steps, another is making a design that does complete reduction from the highest complexity (8 right now), to the lowest (1). The advantage of the first approach is that layers could be combined to form the converter necessary for the situation. Different layers will likely contain duplicate (and so superfluous) buffering, and more hardware must be designed. The advantage of the second approach is that only a single component must be designed, and tested, which subsequently is able to handle any situation. Efficiency will depend on the stream complexity conversion that is required. It was decided to go for the single-component option for a focused and complete resolution.

Operation can be split up in three stages. Input processing, buffer, and output. Central are the buffer registers: the data register, last register, and empty register. Each buffer is a vector of registers of length n, the buffer size. The buffer should be able to hold the longest full sequence. A good value for n will, therefore, depend on project-specific requirements.

- *Input processing* consists of re-aligning and reducing optional delayed dimensional data, concluding when an empty element/sequence occurs, and computing where all element data should go. Order of the elements is also important here.

[2] https://github.com/abs-tudelft/vhlib/blob/master/stream/StreamSync.vhd

- For the *buffering* it is important that the data is coherently stored, in the right order. New data is coming in, and stored data is going out when the sequence is ready. Input and output are uncoupled, so this can happen in the same clock-cycle! Care should, therefore, be taken in getting all the indexes right.
- *Output generation* consists of the logic to form a valid $C = 1$ stream.

Details of the implementation will now be described. Figure 4.2 shows a visual explanation of the complexity converter in case converting of a 2D character stream, in the middle of sending `[["Hello", "World"], ["Tydi", "is", "nice"], [""], []]`. The explanations will work using this example. Note that the data does not actually contain indicators for the *start* of a sequence. The displayed `{` characters are just for illustration purposes.

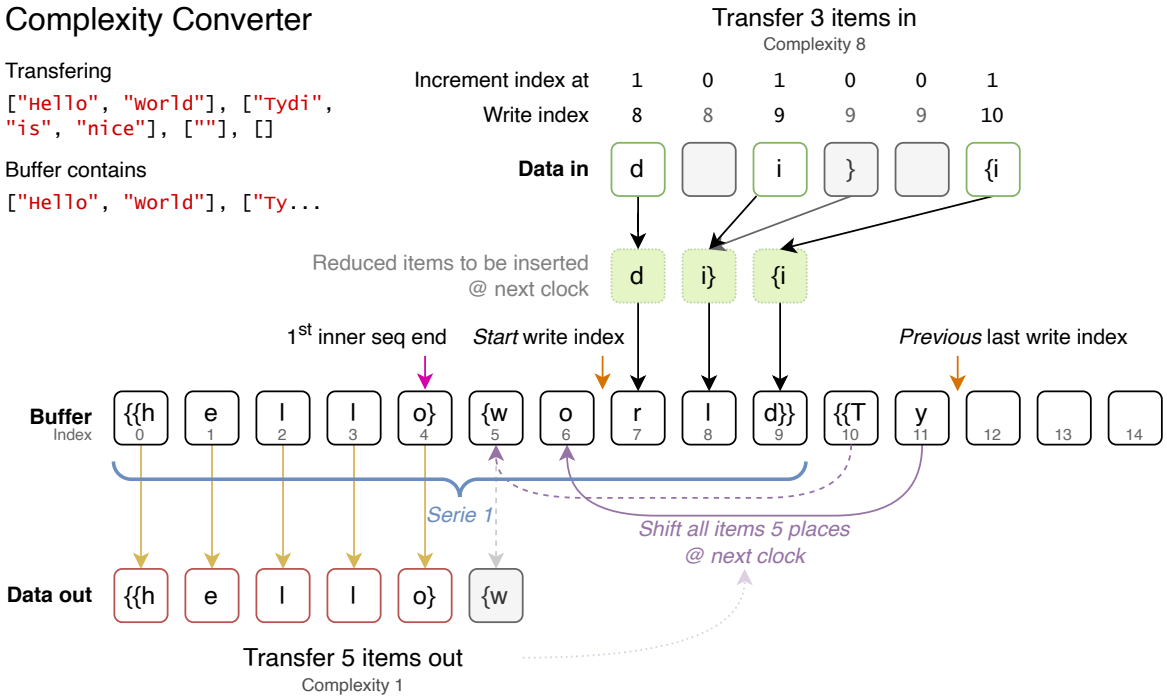


Figure 4.2: Visual explanation of complexity converter operation

Write index basis

Calculation of the correct write indexes is one of the harder parts of the design. As already noted, data can come in and go out simultaneously. Clearly, if no data is coming in, the base writing index, w, stays the same and is equal to the top index i. If data is going out, then $w = i_{\text{next}} = i - n_{\text{out}}$. If data is coming in (and not going out), then $w = i_{\text{next}}$ will be $i_{\text{next}} = i + n_{\text{in}}$. More concretely, the last valid element's write index + 1. This relationship is given by Equation 4.1. Write indexes are denoted by w and the base write index by w_0. The individual write indexes $w[j]$ are given by Equation 4.3 later.

$$i_{\text{next}} = w[n_{\text{lanes}} - 1] + 1 \tag{4.1}$$

$$w_0 = i - n_{\text{out}} \tag{4.2}$$

Figure 4.2 shows the top index, marked by previous last write index at $i = 12$, which would be w if no data was going out, and the start write index as actual w at 7.

Dimensionality information

At stream complexities of $C \geq 4$, dimensionality information can be delayed with respect to the last data element sent. Even more tricky, empty sequences can be formed based on `last` values. Retrieval of correct dimensionality information and detection of empty elements is a problem on its own and is discussed in the next subsection (4.5.3).

Individual write indexes & empty elements

After the possible empty sequences have been identified by the last sequence processor, the individual write indexes can be computed. In general, writing starts at w. The n data lanes are iterated over and examined. When a lane either has valid data, *or* indicates an empty element/sequence through the last sequence processor, the index is incremented. This is shown in Figure 4.2 at *increment index at* and *write index*. An element is marked as *empty* when the last sequence processor indicates a new sequence but there is *no* valid data. Finally, data is written to the buffers for lanes where `incrementIndexAt` is high to the lane's corresponding write index, given by Equation 4.3. The `last` buffer register uses a slight variation of this approach, as indicated by Subsection 4.5.3.

$$w[j] = w_0 + PopCount(incrementIndexAt[j:0]) - 1 \quad \text{for } 0 \leq j \leq n_{\text{lanes}} \tag{4.3}$$

In this equation, *PopCount* is a function that counts the number of 1's in the signal. Alternatively, one could formulate it this way:

$$\begin{aligned} w[j] &= w[j-1] + incrementIndexAt[j] \quad \text{for } 0 \leq j \leq n_{\text{lanes}} \\ w[0] &= w_0 + incrementIndexAt[0] - 1 \end{aligned} \tag{4.4}$$

In the example of Figure 4.2, three elements are transferred in the first, third, and sixth lane. This results in `incrementIndexAt` to be, in reversed binary notation for clarity, 101001, and $w_0 = 8$ so $w = [8, 8, 9, 9, 9, 10]$. 'd' is written to index 8, 'i' #1 to index 9 and 'i' #2 to index 10. The difference in standard notation order of a bit-vector (least item right) and a sequence (least item left) is something to watch out for here.

Counting sequences

For reasons illustrated in output stream, the number of full sequences must be kept track of. A full sequence is indicated by an element which has the most significant `last` bit set. Multiple full sequences can come in per transfer at $C = 8$, but only one can go out for $C \leq 7$. One can either count all the MSB's present in the `last` buffer, or keep a register with the count. The count is incremented with the MSB's in the `last` lanes and decremented by the presence of an MSB in the `last` value of the last item of an outgoing transfer. In Figure 4.2, a single full-sequence is present. The 'd' at index 9, marks the end of this sequence. In this situation, that character was received in the previous transfer.

Output stream

As mentioned, the output should be formatted as a valid $C = 1$ stream. For $C = 1$, the output should be written in one continuous set of transfers. Accordingly, outputting can only start when it is know that at least one entire sequence is stored. Else, the risk of running out of data exists, a pause must be inserted, and the protocol at $C = 1$ is broken. Additionally, not more than a single inner sequence can be sent at a time, so the length of an inner sequence must be found (if shorter than the number of output lanes). Logically, this is done by finding the index of the first lane with an asserted LSB `last` bit. In Figure 4.2, the inner sequence "hello" has 5 letters/elements, so 5 elements are transferred and the buffer shifts 5 places to the left.

Next to sequence management, the usual signals must correctly be set. `stai` is driven to 0. Relevant are the other signals: `ready`, `valid`, `data`, `endi`, `last`, and `strb`. `strb` is used to indicate empty signals. The `last` bits must be acquired from the buffer index of the last valid element. This index is the same integer that is driven to `endi`. As before, care must be taken with the order of elements in the `data` signal.

4.5.3. Dimensionality information reduction

In the context of the complexity converter implementation, a crucial aspect is the reduction of dimensionality information within the data stream to cross from $C \geq 4$ to $C < 4$. This Subsection delves into the intricate task of dimensionality information reduction. The algorithm for this reduction and hardware design for this challenging problem are explored in detail. Figure 4.2 does show an example of `last` compression for the third and fourth lanes.

What is a new sequence?
The first question that must be addressed when attempting to solve the problem of dimensionality reduction is to know what we are trying to achieve. Globally, at $C = 1$, this is to output a stream without gaps in the data for a whole outer sequence, each inner sequence in its own transfer, or transfers, depending on length (see Figure 2.2). The central question is therefore, how do we identify a sequence? In a Tydi stream, *a sequence runs from the sequence starting point, to the item before the start of a next sequence*. While this seems like a nonsensical definition at first, it lays at the core of how one can detect sequences. The reason for this is the possibility of delayed `last` signals and empty sequences.

In the simplest situation, a sequence is terminated by the most significant `last` bit, signaling an end of the previous sequence. Any sequence after it, empty or not, will start at the next element. When the MSB `last` bit has not been asserted yet and another transfer takes place, there are a few options of things that can happen:

- a new element follows
  - if a `last` bit was asserted, a new sequence starts
  - if no `last` bit was asserted, the previous sequence continues
- `last` information follows without element data
  - additional dimensions can be closed off, still relating to the old sequence
  - a new, empty sequence can be started

The first situation category is trivial and does not require any special handling. It is the second situation category that illustrates the dimensionality reduction problem. Looking at these situations it becomes apparent that we have two tasks. The first is to keep adding the `last` information to the last element (either with data or empty), this is the reduction. The second is detecting when a new (empty) sequence starts based on the `last` information alone. It is here that the loop becomes apparent. A new sequence depends on the `last` information of the previous element, but to compile this information we must know where the next (empty) sequence starts!

$$i_{\text{prev}LSB} \geq i_{\text{new}LSB} \tag{4.5}$$

Breaking the loop
When viewing the reduction problem from a sequential/software perspective, executing the task is really easy. One simply starts with the first element, decide if a new element should start based on the current reduction value and `last` information, create the new reduction value, and move on to the next element. If one stores which index the previous element started, or store the element itself, one can write the reduced `last` information to the element index in your buffer when a new element starts. This process is visualized in Figure 4.3.

If we only worked with single-lane streams for the complexity converter, life would indeed be this simple. A few state registers and you are done. Because taking in a single lane is much simpler, at $n = 1$, the converter has limited use. The problem thus becomes executing the reduction task for multiple lanes at the same time. A possible way to do this is by pipelining the solution. In each stage, one would figure out the index of the next element, do the reduction, and pass on the index as starting point for the next stage. There would be as many stages as lanes, even though there might be fewer elements. At the end all information would be available for writing to the buffer. Creating such a design is possible. Designing the dataflow in such a component is hard and a lot of registers are required to store the state, however. Luckily, a simpler solution is available, which will be detailed next.

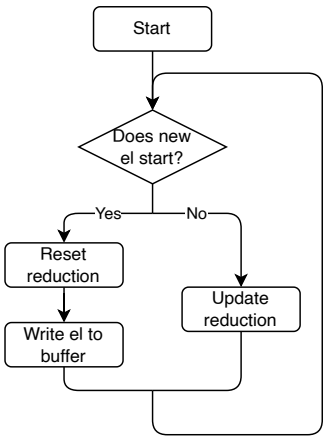


Figure 4.3: Sequential processing of incoming elements

Combinational solution

As it turns out, the problem can be simplified by separating the creation of `last` reduction & new sequence information and getting the information to the right place. For the first part, the design shown in Figure 4.5, called the “last sequence processor” was conceived. This design essentially works like a basic carry-propagation adder. Each processor element can be compared to a full adder, with a `reduced` in (analogous to carry in), two inputs, an output (whether a new sequence starts based on the `last` values), and a `reduced` out (analogous to carry out). Figure 4.4 shows the functionality of such a processor element. Note that the lane validity and lane `last` values are not shown here, even though they are used in the decision process.

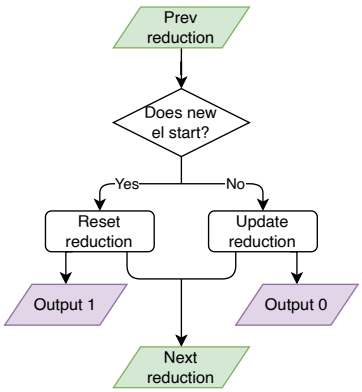


Figure 4.4: Processing of lane information

The adder-like structure essentially acts as a sequential solution, but without a clock. Each processor element takes decisions based on the previous element. Because the design is very simple, it should be fast enough for any situation except for very large lane counts and high clock speeds. In such situations, a more personalized approach is more fitting anyway.

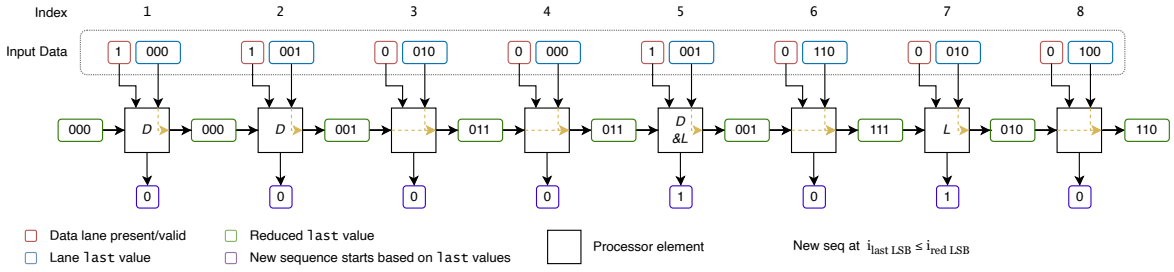


Figure 4.5: Last sequence processor component schematic

Example details

Figure 4.5 shows an input sequence of 8 lanes, where several possible situations are outlined. A processor element contains the letter D when a new element starts based on the presence of element data. It contains an L when a new element starts based on `last` values. Lane 1 is an average situation with data and no last information. Lanes 2, 3, and 4 together form the second element with delayed last information for the second dimension. At lane 5, a new element is denoted by data presence, but also by `last` information; a dimension is closed that was already closed by the previous element. This would result in an empty sequence if there had not been element data. At lane 7, such an empty sequence is started since there is no element data. Summarizing, an empty element arises when a new sequence starts based on `last` values. As can be seen from the figure, the final reduced `last` value is available at the index before the next element. This is important for the next section, where the data is aligned and written to the buffer.

Matching element information

After executing the reduction the following information is available:

- Flags for which lanes start new elements based on `last` values
- Reduced `last` values for each element from the start of the element to before the next one.

The remaining problem consists of getting the correct reduced `last` values corresponding to the item. Two solutions are available to solve this problem. One option is to propagate the element data with the (reduced) `last` data in the sequence processor. Another is to use “look back” indexing. The main difference between the two strategies is how they handle trans-transfer delayed dimensionality info. When a transfer does not start with a new element but with (additional) `last` assertions, this information must be added to the last item in the buffer. This behavior is native to the look back indexing. For the data-propagation strategy, one must keep the last element in memory, i.e. not commit it, until new sequence starts. In general, in both cases, the system must wait with outputting the sequence until either an outer sequence ends, or a new inner sequence starts. After all, additional related `last` bits could still follow. At $C = 1$, an outer sequence end is always awaited before output of that sequence can start.

In the end, the look back indexing was chosen since it offers better separation of concerns. Listing 4.7 shows calculation of the write indexes and writing the data to memory. The look back indexing is done where `lastReg` is written. For each valid element, the `last` information of the element before it in the buffer is set to the reduced `last` value of the index before this element of the last sequence processor. Since the last processor works from left to right, this is the index that will contain all `last` information for this element. Notice that if the data is a continuous stream of elements, the previous buffer element gets written to the previous `last` value in a 1:1 fashion. For completeness, the final reduced `last` value is also written to the last element in the buffer. This is required for situations where the outer index ends at the last element lane and no element follows after to write the required `last` information back.

Listing 4.7: Write index calculation and writing data to buffer in `ComplexityConverter`

```
// Calculate & set write indexes
for ((indexWire, i) <- writeIndexes.zipWithIndex) {
  // Count which index this lane should get
  // The strobe bit adds 1 for each item, which is why we can remove 1 here, or we would not fill
       the first slot.
  indexWire := writeIndexBase + relativeIndexes(i) - 1.U
  // Empty is if the a new sequence is assigned by last bits, but the lane is not valid
  val isEmpty: Bool = lastSeqProcessor.outCheck(i) && !in.laneValidity(i)
  val isValid = incrementIndexAt(i) && in.valid
  when (isValid) {
    dataReg(indexWire) := lanesIn(i)
    // It should get the reduced lasts of the lane *before* the *next* valid item
    // If the first item has data we don't care about the previous last value anymore.
    // If it does not, it will get set at i=1
    if (i > 0) {
      when(indexWire > 0.U) {
        lastReg(indexWire - 1.U) := lastSeqProcessor.reducedLasts(i - 1)
      }
    }
    emptyReg(indexWire) := isEmpty
  }
}
// Fix for the "looking back" way of setting last signals.
lastReg(writeIndexes.last) := lastSeqProcessor.reducedLasts.last
```

5

Working with Tydi-lang

Within this chapter, the reciprocity of Chisel and Tydi-lang code is explored. A code generation tool for producing Chisel code from Tydi-lang 2 output is described. Data format, pre-processing, and code generation procedure are explained. A reverse code generation strategy from Chisel to Tydi-lang is also introduced. In this process, several limitations of the Chisel library are encountered and work-arounds explored.

Tydi-lang was designed as a domain-specific language to effectively describe Tydi elements, streams, streamlets, and, from a connection perspective, implementations. Hence, the role of a Tydi-lang description of a system in the Tydi ecosystem is that of a blueprint. This blueprint in part serves as a specification of the system's communication structures (elements) and channels (streams & ports). The blueprint can further be used to create boilerplate code for implementations.

Subsection 2.1.1 outlined the first Tydi-lang project that generated Tydi intermediate representation language (TIL) code and the Tydi-IR project that is able to generate VHDL component boilerplate code [41, 31]. Tian [40] developed Tydi-lang 2 with several syntax and compiler improvements and new output format. Since Tydi-lang 2 supercedes Tydi-lang 1, yet shares most grammar, Tydi-lang code in this document refers to Tydi-lang 2 compliant code. As mentioned in Section 3.1 JSON was chosen as a favorable output format for Tydi-lang 2, for it is widely used and parsing is straightforward in almost every popular programming language. This should lead to easier development of code generation tools such as [31], in turn lightening the burden of adoption into ecosystems.

5.1. Code generation from Tydi-lang to Chisel

In this section, the strategy and details of Chisel code generation from a Tydi-lang specification is outlined. Output interpretation, information processing, code generation, and project considerations are discussed. Take a look at Table 2.1 for the relation between Tydi concepts and Chisel concepts.

5.1.1. Format

When Tydi-lang code is compiled, a JSON document is produced, as already explained in the chapter's introduction. The JSON document contains objects, also known as dictionaries or maps, that describe the properties of all elements, streamlets, and implementations. The three top-level keys are therefore `logic_types`, `streamlets`, and `implementations`. In each of these category objects, objects describing the properties of the entities described in Tydi-lang reside under unique keys. Listing 5.1 shows an example JSON object under `logic_types` describing a stream. The stream and user types are specified as references to the keys of the datatypes for the element and user fields respectively. Every logic type object has a `type` property that can be either `Ref`, `Stream`, `Bit`, `Group`, or `Null`. Importantly, emitted logic type information includes documentation string, if specified. This documentation is later rendered in Chisel code as well, improving clarity.

Listing 5.1: Example JSON output snippet

```
    "generated_0_36_JI5PTYzg_24": {
```

```
      "type": "Stream",
      "value": {
        "stream_type": {
          "type": "Ref",
          "value": "package_pack0__NumberGroup"
        },
        "dimension": 1,
        "user_type": {
          "type": "Ref",
          "value": "7RVI_15__user_type_generated_0_36_JI5PTYzg_24"
        },
        "throughput": 1.0,
        "synchronicity": "Sync",
        "complexity": 1,
        "direction": "Forward",
        "keep": false
      }
    },
```


5.1.2. Code generation strategy

To perform the actual task of Chisel code generation from a Tydi-lang JSON document a program was written in Python. Python was chosen for its simplicity, flexibility, and the authors pre-existing knowledge of it and relevant libraries. The program works in three phases.

1. Parsing the JSON document containing the Tydi-lang entity descriptions
2. Pre-processing the information
3. Generating Chisel code based on templates with `Jinja2`

Parsing JSON in Python can be done with the built-in JSON library and does not merit further explanation. `Jinja2` was selected as templating engine for its extensive functionality, including logic and operations in the template, and simple operation. While it could be attempted to generate code directly from the JSON object, placing necessary logic in the templates, this is not very efficient or legible form a programming perspective. Therefore, some pre-processing is done on the information first. The next section will describe this process.

5.1.3. Information pre-processing

The pre-processing function executes several tasks to transform the raw data in such a way that it is easier to use in the templating environment. The following transformation tasks are included:

- Extract entity names and package information from object key
  - Clean names are available for *streamlets*, *implementations*, *groups*, *unions*, group & union *elements*, *ports*, and *connections*.
  - Other entities have, at the moment of writing, automatically generated names. This is not a big issue, since these are types that are used by streams, groups, or unions. A user working with the generated interfaces and components will thus not be exposed to these names.
- Replace references in the data structure by the objects they are referencing
  - Notably for groups, union & group elements, ports, and connections
  - In several places, the JSON file features references to references
- Extract names for streamlet *ports* and implementation *connections*
- For every streamlet's port, find the stream's child streams.
  - This information is required to specify all connections as Tydi's standard representation child streams are exposed as parallel streams, whereas Tydi-lang logically only specifies the top-level stream that is required.
- Filter out duplicate entities

5.1.4. Output

After the information is processed it is ready for translation into code by evaluating the templates. As already mentioned, `Jinja2` is the framework used to do this. The templates files contain templates of the Scala code for the Tydi-Chisel datatypes, interfaces, and components. All the *Bit* elements are grouped in an object called “MyTypes”, where they are defined as a parenthesisless function that returns the ground datatype. Tydi and Tydi-lang explicitly do not express the datatype that a *Bit* element describes, leaving it to the target HDL to fill this in. After code generation, all *Bit* elements are thus emitted as Chisel’s `UInt` type. This types is meant to be replaced by the desired ground type by the user. An `assert` line is in place that checks if the (user) specified type has the correct bit-width. All streams are rendered as a subclass of `PhysicalStreamDetailed` with the specified *e*, *n*, *c*, *r*, and *u* parameters. The *d* parameter is constructed as an integer-cast of the floating point *t* property. Streamlets are emitted as interface classes extending the `TydiModule` class. All ports specified in the streamlet are defined and initialized here. Implementations are classes that extend the relevant interface class. The class contains the modules analogous to the implementation’s inner instances and port connections. As mentioned, item documentation is propagated to Chisel output code where available.

5.2. Code generation from Chisel to Tydi-lang

The example in Chapter 6 assumes a situation where a complete reference implementation or blueprint is already available. In reality, it often happens that specifications change during a project, or influencing factors are overlooked at the start. To facilitate a more design-cycle like workflow, a “reverse”-transpiler was created. This functionality allows generating Tydi-lang code from a Chisel definition of a Tydi *Element* or *TydiModule*, including its dependencies. This simplifies making changes to the Tydi-lang spec, or generate a first draft spec when converting existing projects. This role is also shown in Figure 6.1.

5.2.1. Strategy

A complete Tydi-lang description of a system consists of *implementation*s of *streamlet*s using *stream*s using *element*s (that can contain other streams/elements). This top-to-bottom dependency is analogously reflected in Chisel of Module implementations of Module interface classes using streams using (a hierarchy of) `TydiEl` elements. Generation of Tydi-lang code is done by traversing this dependency hierarchy in a depth-first manner.

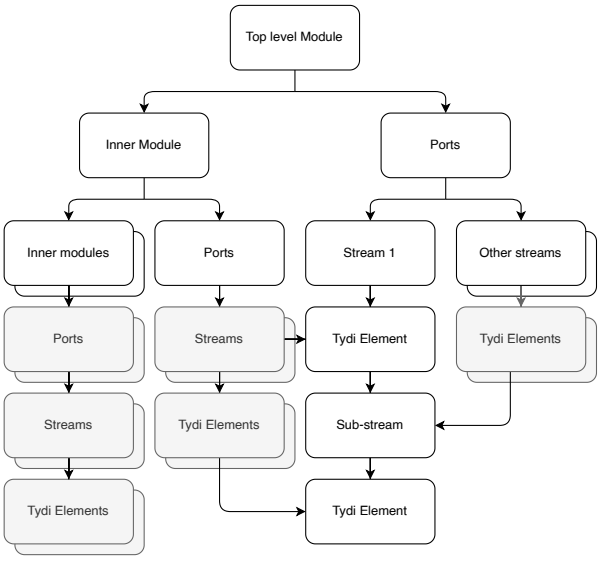


Figure 5.1: Example dependency graph of a system using Tydi components

Figure 5.1, similar to [41, Figure 4.2], shows an example of a dependency graph for a system. In the depth-first approach, the Tydi elements on the left bottom would get transpiled first. The figure features some elements and streams that get referred to by multiple others. This will often happen in a real

design, especially for the ground types. To make sure every element will only get emitted as Tydi-code once and for efficiency, every element generates a deterministic unique fingerprint based on its type. When a Module's or element's transpile function is called, first the depending item's transpile function is called, and then the Tydi-lang code is generated. This Tydi-code is added to a `map`, with the fingerprint strings as keys and code strings as values. Finally, this map can be joined into a string with newline separators and the Tydi-lang code for the system is obtained. Please note that while Figure 5.1 shows a top level module as starting point, one can start anywhere in the graph, also in streams or elements.

5.2.2. Implementation notes & Chisel lib limitations

Every class that must be able to generate Tydi-lang code must be able to do three things: generate a unique fingerprint for the class (or parametric instance), generate Tydi-lang code equivalent of the Chisel code for this class, and call the transpile functions of all the items this class depends on. This is implemented as a *trait* defining three functions that extending classes must implement: `fingerprint`, `tydiCode`, and `transpile`. This is enough for TydiModules and TydiElements (which include streams). The ground types are all subclasses of `Data`, however. To transpile these types to `Bit` types, an implicit class implementing the transpile trait is required to extend `Data`'s interface. Additionally, the method implementations of the implicit class include upcasting functionality for cases where the transpile function is called from a place the items are type-wise processed as `Data` sub-classes to still execute the correct functions.

Acquiring a module's sub-modules and ports to traverse the dependency tree is a challenge in Chisel. A lot of functionality and data is locked behind a `private[chisel3]` modifier. This restricts a function's usage to within the Chisel library only, even for sub-classes. This limits the functionality that can be achieved by the code generation function. Thankfully, the ports of a module can be acquired with the `getModulePorts` function, but this is only because, by other use, it cannot be private-restricted like other functions. Specifically, a comment above the function says "`getPorts` unfortunately already used for tester compatibility". The *direction* of the port signals cannot be extracted, however, even though the information is available in the port's instance. Because of this, a guess is made based on the port's name containing either "in" or "out". Wire connection information is not available.

There seems to be no direct internal functionality to get a Module's sub-modules. It is also likely that a method supplying this information would be library-private as well. This can still be achieved by overloading the `Module` method that instantiates modules. The `TydiModule` class contains a `Module` method that instantiates the Module by calling the Chisel `Module` function and adding that instance to a list of the parent module instance. This same mechanism could not successfully be applied for Chisel's experimental `Instance` method to instantiate parameterized Module Definitions, for the original module's information is not exposed outside the Chisel library either.

This lock-down of information is not inexplicable. The authors of Chisel probably did not foresee use-cases of examination of Modules and IO, while meddling with this internal data could hamper the functionality and correctness of hardware generation.

Using a few work-arounds, an almost complete reverse-transpilation can be achieved. Port direction and connection information is unavailable and can thus not be emitted. A complete description could be acquired when starting from the FIRRTL output of a circuit, which contains the missing information. This requires parsing the FIRRTL dialect, however, and is considered not worth the effort by the author. Theoretically, in the event that Tydi would be adopted into the Chisel base library, these workarounds would not be necessary. This is unlikely however, for even libraries that are considered official are developed as external packages, outside the `chisel3` package. In the future, the Chisel library could release more information to allow easy system introspection.

5.2.3. Tydi-lang limitations

Most of the structure from Chisel systems can be translated to Tydi-lang. There are a few limitations, however.

- `in` and `out` are direction keywords in Tydi-lang, and can therefore not be used as stream names. This reportedly is a technical limitation of the parsing framework. The workaround is to rename streams from `in` to `std_in` and from `out` to `std_out`. The forward transpiler renames these back.
- No low-level types can be specified for `Bit` elements. This is a design decision of Tydi. This

information can therefore only be emitted as comments (see next point).

- Not all elements have support for doc-strings yet in Tydi-lang at the time of writing. Comments can be added, but these are not propagated back were forward transpilation to be executed.
- Some items still have auto generated names, making the structure harder to decipher.
- Use of streams in Tydi-lang creates a new copy in the JSON output for every use of the stream. This is superfluous of course.

6

Streaming hardware design using Tydi and Chisel

This chapter features a systematic design-strategy a.k.a. cookbook, along with several illustrative examples. In the design strategy all steps are included one could need when developing a hardware accelerator from a reference software implementation. The examples demonstrate these design steps and reveal the added value of building systems with Tydi-interfaced components through composition.

6.1. Steps

This section describes how Tydi, Tydi-Chisel, and Chisel can be used to develop a hardware design that operates on a streaming dataflow. This design flow is illustrated with examples in Section 6.2.

In this scenario, desired is a hardware design for a stream-processing problem that already has a software implementation. A step-by-step pipeline going from a software specification to a hardware design would look like this:

1. Idea / software definition
2. Write interface specifications in Tydi-lang code
3. Describe additional communication specification
4. Transpile with Tydi-lang-2 & Tydi-lang-2-Chisel
5. Write component functionality in Chisel with generated interfaces
6. Test with testing utilities
7. Synthesize using vendor tools

These steps and relevant toolchain projects are depicted in Figure 6.1.

6.1.1. Software definition

It is likely that many projects will start with a software implementation. Either because the software was already developed before deciding on the design of a hardware accelerator, as a means of quick development to make design choices, or simply as a reference for correct operation. Whatever the reason may be, the software implementation serves as a good starting point for hardware development. It already contains all data-structures, data-flow, and operations that the hardware must also contain.

6.1.2. Tydi-lang specification

As explained before, Tydi-lang was designed to close the gap between software and hardware design. It is therefore the next step in our design pipeline. Identified data-structures must be converted to Tydi *elements*. Operations must be decomposed into components, their communication signatures described in *streamlets* with ports of streams of the described elements. After streamlets, data-flow must

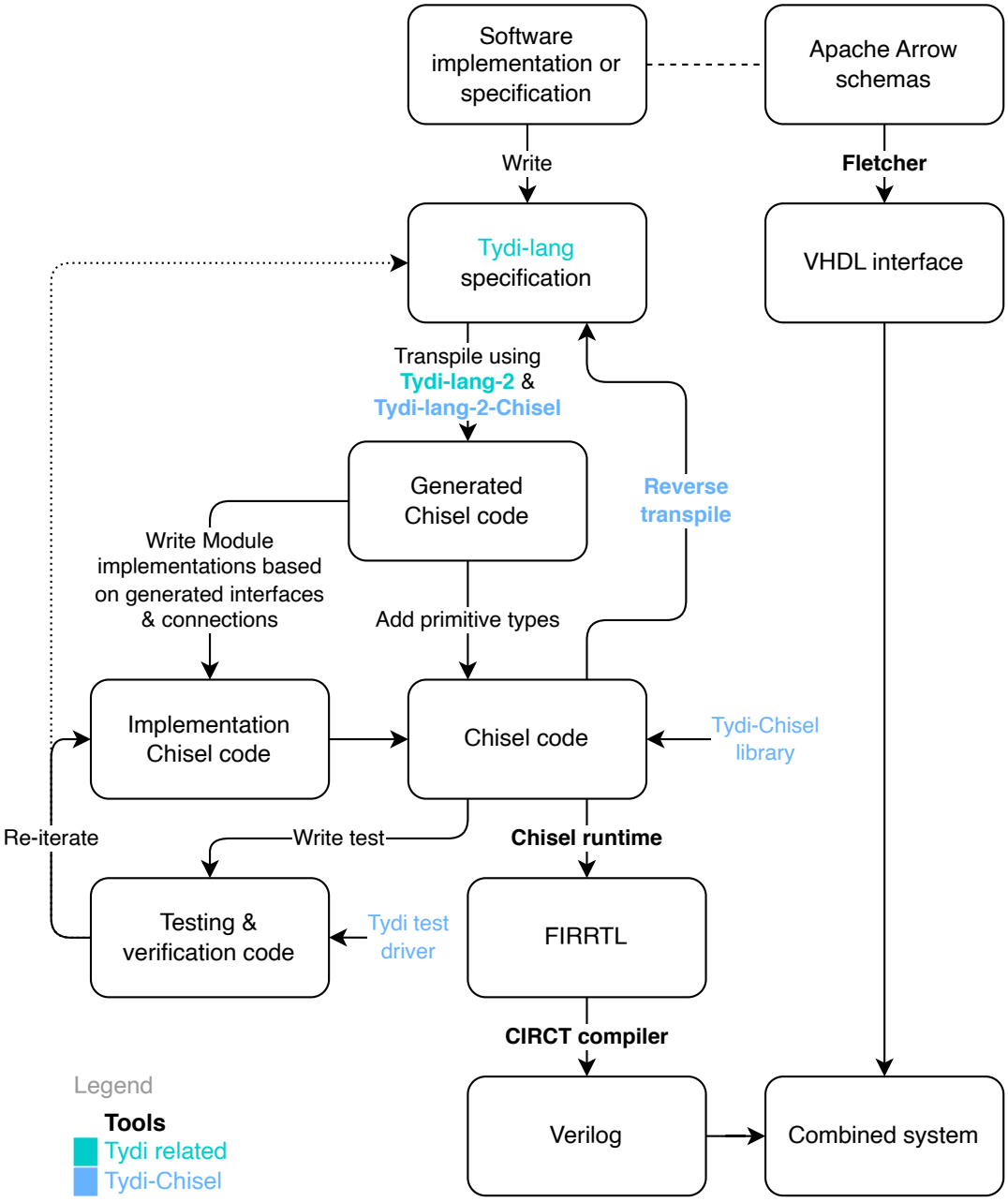


Figure 6.1: Tydi toolchain components

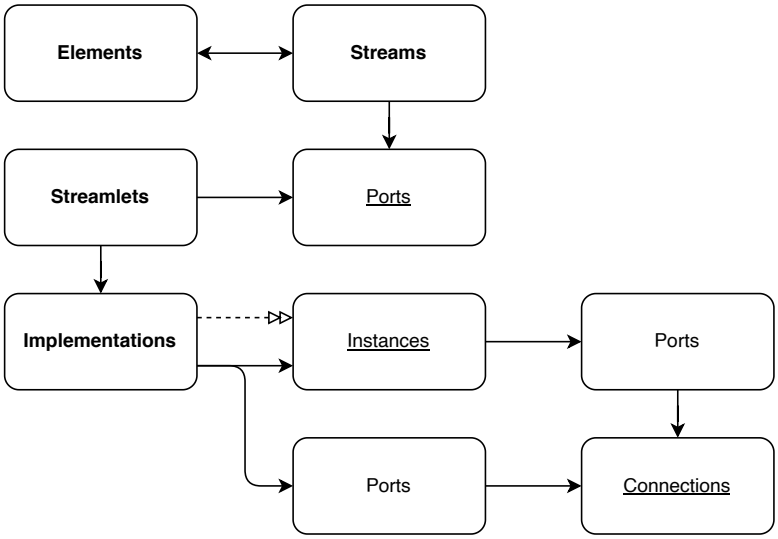


Figure 6.2: Tydi-lang specification contents. **Bold** items are top-level defined. Underlined items are specified within other items. Instances of nested implementations are declared inside implementations.

be described in terms of connections specified in *implementations* between implementation *instance* ports. Nested streams. Implementations can either purely consist of instances and connections, or contain logic themselves. Implementation definitions that require such practical implementations are left empty. Figure 6.2 shows the contents of a Tydi-lang specification.

It is here that the interface driven design takes place. A well-defined interface specification allows abstraction. Hardware designers can then purely focus on their own components, relying on the Tydi standard and interface specification for their components to work with others. With only data structure and port assignments, the specification is not yet complete, however. For that some extra detailing is required.

6.1.3. Communication specification

The Tydi specification establishes a standard for communicating data over streams. It also specifies how a data structure can be formed by combining different data elements through nesting. It does not specify how communication between components should take place, just like the internet protocol does not specify how TCP packets should be sent. In other words, low-level data transfer in specified, not data-packet communication. The difference is not immediately visible when looking at a single stream transferring a sequence. Instead, the difference is notable when working with multiple streams. In Tydi, streams can be nested to describe their hierarchical semantic relation. In the standard representation, however, these streams end up as parallel streams. Inherently, this is also what they are, separate communication channels to transfer data related to the parent stream. Tydi specifies a basic order-of-operations and allowed dependencies mechanism to prevent deadlocks, but otherwise leaves coherency up to the user.

6.1.4. Transpile code

Once data-structures, components, data-flow, and communication protocol have been created, work can proceed towards HDL code. The first step is to obtain Chisel boilerplate code from the Tydi-lang specification code. This is done by first running `Tydi-lang-2`, obtaining a json description that is used to generate the Chisel code with `Tydi-lang-2-Chisel`. This process is explained in detail in Section 5.1.

The output Chisel code will contain the transformed *Element* datatypes, interface specifications (from streamlets), and implementation skeletons. As Subsection 5.1.4 explains, after code generation, the correct primitive types must be substituted for the `UInt` placeholders.

6.1.5. Implementation code

After finishing the generated code with primary types, implementations for Modules must be written. Interface classes with port registrations following the *streamlet* definitions are available to build on top of. In addition, implementation stubs extending the interface classes are available, and will include any specified nested modules and stream connections. The application-specific logic can be added to the these stubs. This is the hardest part of the design process. The tools and utilities outlined in Chapter 4 can be used to assist in design. Writing correct implementations will likely remain difficult, however, especially when a lot of streams are involved in a component.

6.1.6. Test

When the implementation has been designed for a component, the subsequent step is to verify its functionality. Section 7.1 describes available tools and utilities to aid in Tydi component testing. Likely, a design cycle is now entered where implementation, or even an earlier design step, is altered based on simulation results.

6.1.7. Finishing the system

Depending on the scenario, different steps may remain to finish the system. After testing the design might be done. If the component is destined as IP for other projects, a design that is tested and documented with Tydi-lang and communication specification is ready to be exported. For complete system designs, more steps are likely to be required. Acquisition and deposition of the data streams, for example. Fletcher can play a role here, by generating VHDL interfaces for FPGAs to acquire data from Apache Arrow RecordBatches on a PC through some accelerator interface. Once in the system, the data can be transferred to the Tydi-using components. Potential for future work here includes the automatic generation of Tydi interfaces and components by Fletcher. When all necessary hardware design files are available, mapping and synthesis can be executed with vendor tools to deliver the hardware.

6.2. Examples/proof of concepts

6.2.1. Simple number pipeline

To illustrate the aforementioned pipeline, an example is provided. This example is purposefully kept simple and the reader must remember that the advantage of Tydi's ecosystem is greater with more complex projects. In this example, we take in a stream of numbers with timestamps attached. The structure is thus like `{time: unsigned Integer, value: Integer}`. This stream first gets filtered on $value \geq 0$ and then reduced to statistics: min value, max value, sum of values, and average, all unsigned integers since there are no negative values anymore. The structure of this processing pipeline is displayed in Figure 6.3.

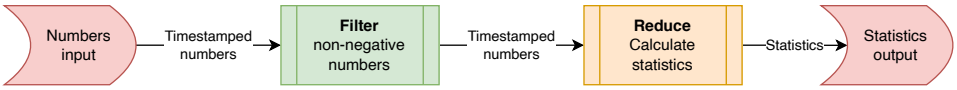


Figure 6.3: Number pipeline structure

Software definition

As a reference of both functionality and difficulty of implementation, a software definition is first created. The task is not hard, and can be executed in Apache Spark in just a few lines of code, as shown in Listing 6.1. Here, the method chaining syntax is used that was already mentioned in Subsection 4.3.2.

Listing 6.1: Example Spark code

```
df.filter(col("value") >= 0).groupBy("timestamp")
  .agg(
    min("value").as("min_value"),
    max("value").as("max_value"),
    sum("value").as("sum_value"),
    avg("value").as("avg_value")
  )
```

Tydi-lang specification

The first design step for our system is to create a Tydi-lang description of the data-structures, streamlets, and implementations. Listing 6.2 shows Tydi-lang code for our example. It includes two ground data-types, one for the signed and one for the unsigned integers. Two groups are included, `NumberGroup` and `Stats`, with respective streams. For simplicity, a throughput of 1 is chosen. According to Figure 6.3, streamlets and implementations are defined for the non-negative filter, statistics reduction module, and top level module. The implementation of the top level module includes instances of the filter and reducer modules and connections of their streams. The implementations of the filter and reducer modules themselves are empty, since their logic will be defined later. For order, the code is split up in two files, one for the logic types and one for the streamlets and implementations.

Listing 6.2: Example Tydi-lang source code

```
#### package pack0;

UInt_64_t = Bit(64); // UInt<64>
SInt_64_t = Bit(64); // SInt<64>

Group NumberGroup {
    value: SInt_64_t;
    time: UInt_64_t;
}

Group Stats {
    average: UInt_64_t;
    sum: UInt_64_t;
    max: UInt_64_t;
    min: UInt_64_t;
}

NumberGroup_stream = Stream(NumberGroup, t=1.0, d=1, c=1);
Stats_stream = Stream(Stats, t=1.0, d=1, c=1);

#### package pack1;
use pack0;

streamlet NonNegativeFilter_interface {
    std_out : pack0.NumberGroup_stream out;
    std_in : pack0.NumberGroup_stream in;
}

impl NonNegativeFilter of NonNegativeFilter_interface {}

streamlet Reducer_interface {
    std_out : pack0.Stats_stream out;
    std_in : pack0.NumberGroup_stream in;
}

impl Reducer of Reducer_interface {}

streamlet PipelineExample_interface {
    std_out : pack0.Stats_stream out;
    std_in : pack0.NumberGroup_stream in;
}

impl PipelineExample of PipelineExample_interface {
    instance filter(NonNegativeFilter);
    instance reducer(Reducer);
    filter.std_out => reducer.std_in;
    reducer.std_out => self.std_out;
    self.std_in => filter.std_in;
}
```

Communication specification

Since all components have only one input stream and one output stream without nested streams, no further communication specification need be made. For bigger projects, this step is more important, as previously explained.

Transpile code

A snippet of the generated Chisel code is given in Listing 6.3. The code shows the transformed *Element* datatypes, interface specifications (from streamlets), and implementation skeletons. Since Tydi focusses on the structure of the data, not about the primitive data-types, after code generation, the

correct primitive types must be substituted for the `UInt` placeholders. The code includes an `assert` to check if the used datatype adheres to the specified bit-width. After finishing the specification with primary types, implementations for Modules must be written, following the *streamlet* `_interface` definitions. An example implementation of the filter function is given in Listing 6.4 where the data-lanes of filtered items are turned off. The cost of this simple implementation is that the output complexity is raised to $C = 7$; The next component must do work to re-align the items when the sequence is required.

Listing 6.3: Chisel output code from Tydi-lang transpilation

```
object MyTypes {
    /** Bit(64) type, defined in pack0 */
    def generated_0_7_AudkORtF_29 = UInt(64.W)
    assert(this.generated_0_7_AudkORtF_29.getWidth == 64)

    /** Bit(64) type, defined in pack0 */
    def generated_0_7_CTh3cRpJ_27 = UInt(64.W)
    assert(this.generated_0_7_CTh3cRpJ_27.getWidth == 64)
}

/** Group element, defined in pack0. */
class NumberGroup extends Group {
    val time = MyTypes.generated_0_7_CTh3cRpJ_27
    val value = MyTypes.generated_0_7_AudkORtF_29
}

/** Group element, defined in pack0. */
class Stats extends Group {
    val average = MyTypes.generated_0_7_CTh3cRpJ_27
    val max = MyTypes.generated_0_7_CTh3cRpJ_27
    val min = MyTypes.generated_0_7_CTh3cRpJ_27
    val sum = MyTypes.generated_0_7_CTh3cRpJ_27
}

/** Stream, defined in pack0. */
class Generated_114_139_TSuzlpzQ_3 extends PhysicalStreamDetailed(e=new NumberGroup, n=1, d=1, c=1, r=
     false, u=Null())

object Generated_114_139_TSuzlpzQ_3 {
    def apply(): Generated_114_139_TSuzlpzQ_3 = Wire(new Generated_114_139_TSuzlpzQ_3())
}

...

/** Streamlet, defined in pack1. */
class NonNegativeFilter_interface extends TydiModule {
    /** Stream of [[in]] with input direction. */
    val inStream = Generated_165_190_BDoT0FmX_5().flip
    /** IO of [[inStream]] with input direction. */
    val in = inStream.toPhysical
    /** Stream of [[out]] with output direction. */
    val outStream = Generated_114_139_TSuzlpzQ_3()
    /** IO of [[outStream]] with output direction. */
    val out = outStream.toPhysical
}

/** Streamlet, defined in pack1. */
class PipelineExample_interface extends TydiModule {
    // Docstrings left out
    val inStream = Generated_607_632_HlxXfYFN_13().flip
    val in = inStream.toPhysical
    val outStream = Generated_562_581_8ln94DFm_11()
    val out = outStream.toPhysical
}

...

/** Implementation, defined in pack1. */
class NonNegativeFilter extends NonNegativeFilter_interface {}

/** Implementation, defined in pack1. */
class PipelineExample extends PipelineExample_interface {
    // Modules
    val filter = Module(new NonNegativeFilter)
    val reducer = Module(new Reducer)

    // Connections
    reducer.in := filter.out
    out := reducer.out
    filter.in := in
}
```

Implementation code
An example implementation of the filter function is given in Listing 6.4 where the data-lanes of filtered items are turned off. The cost of this simple implementation is that the output complexity is raised to $C = 7$; The next component must do work to re-align the items when the sequence is required.

Listing 6.4: Example implementation of multi-lane capable filter

```
class NonNegativeFilter extends NonNegativeFilter_interface {
  outStream := inStream
  for ((dataLane, strbLane) <- inStream.data.zip(outStream.strbVec)) {
    strbLane := dataLane.value >= 0.S && inStream.valid
  }
}
```

Test
Verification of the pipeline modules was done bottom up. First, the `NonNegativeFilter` module was tested, then the `Reducer`, and finally the top level module. First, a sequential test with some manual values was done for the top level module. Afterwards, an automatic randomized test was created where 100 values were enqueued and the dequeued values checked against computed values.

6.2.2. Advanced number pipeline

As a demonstration of Tydi-Chisel's utilities and the modularity of well-specified Tydi components, an advanced version of the number pipeline was developed. In this version, the number of lanes is increased. In a project one might want to do this to increase throughput. The design is parameterized, so the exact number of lanes does not matter, but $n = 4$ is chosen for this example. From the original number pipeline, filter and reducer components are available that are both single lane. These components must be adapted or replaced to accept multiple lanes. Since the number elements can be independently filtered, the filter component can be put inside an interleave component. Each filter then processes a single lane and the interleave infrastructure routes the signals such that, from the outside, the block acts like a multi-lane filter. Specifically, the filter modules operate on the strobe signal. This raises the stream complexity to $C \geq 7$. For the reduction step, the elements are not independent, so a new multi-lane version must be designed. For this example, we specify an input complexity of $C = 3$. At this complexity, stop bits are included in the same transfer as elements and the number of items can be derived from `endi`. The output stream complexity of the multi-lane filter block is now higher than the specified input stream complexity of the statistics reducer. To solve this, the stream complexity converter can be inserted between the components! The buffer size of the complexity converter must be chosen to be at least as big as the longest (filtered) sequence. The resulting system structure is shown in Figure 6.4 and an example of data flowing through the component in Figure 6.5.

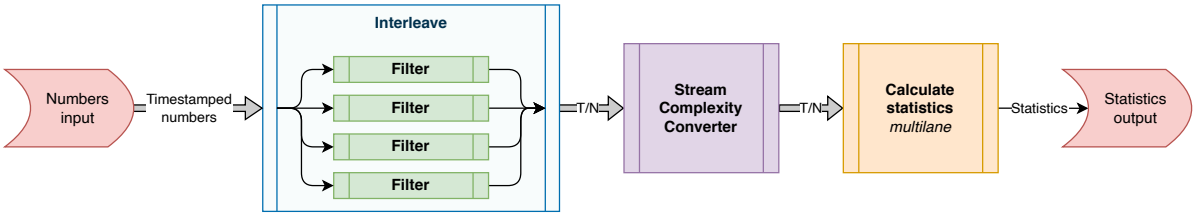


Figure 6.4: Advanced number pipeline structure (T/N stands for timestamped numbers)

The code for composition of this module is shown in Listing 6.5. Assuming the multi-lane statistics reduction component and filter are available IP components, a designer wishing to implement the pipeline of Figure 6.4 would effectively only need to write the few displayed lines of code. The whole design is parametric in the number of lanes and buffer size of the complexity converter.

Listing 6.5: Compositioning of advanced pipeline in Chisel

```
class MultiNonNegativeFilter extends class MultiNonNegativeFilter extends MultiProcessorGeneral(
  Definition(new NonNegativeFilter), 4, new NumberGroup, new NumberGroup, d=1
)
```

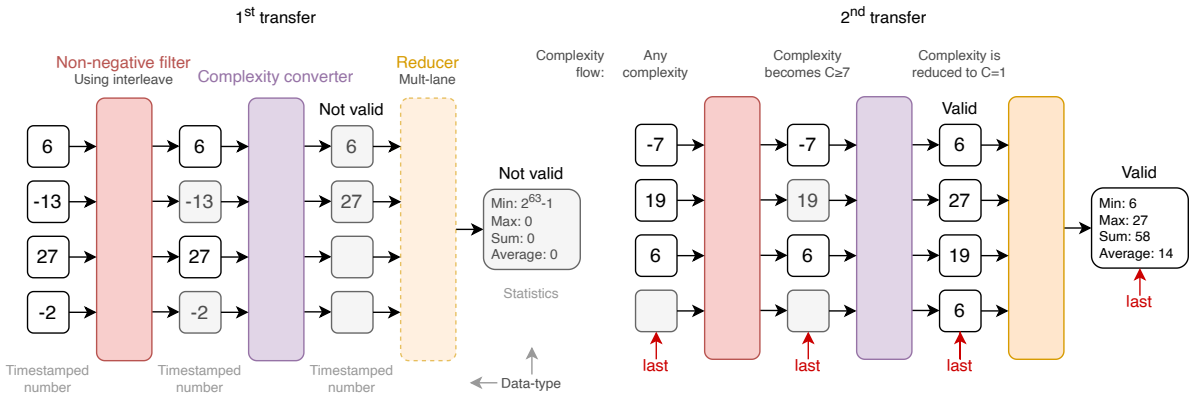


Figure 6.5: Flow of data through the the advanced number pipeline

```
class PipelinePlusModule(n: Int = 4, bufferSize: Int = 50)  extends SimpleProcessorBase(
  new NumberGroup, new Stats, nIn = n, nOut = 1, cIn = 7, cOut = 1, dIn = 1, dOut = 1) {
  out := in.processWith(new MultiNonNegativeFilter)
           .convert(bufferSize)
           .processWith(new MultiReducer(n))
}
```

6.2.3. TPC-H 19 – interfaces and implementation stubs

The Transaction Processing Performance Council benchmark H [43] (TPC-H for short), is an industry standard for testing query execution performance. It includes 22 queries of varying difficulty, providing an excellent reference for real-world use-cases that one might want to build a hardware accelerator for. Tian extensively used TPC-H queries to illustrate Tydi-lang 1's capabilities [41]. As an experiment and demonstration of capabilities for a real world scenario, a proof of concept implementation of the interfaces of the macro-components of the TPC-H 19 query and their connections was created, as depicted by Figure 6.6. Specifically, all the TPC-H data-types and the TPC-H 19 query data-types, streamlets, and implementation stubs were considered. Created code samples are given in Appendix B. Listing B.1 gives the Tydi code for all TPC-H tables and corresponding streams, followed by the TPC-H 19 specific data types and streams together with streamlet descriptions and implementation stubs. The top level component includes stream connections between its three nested instances: the table join, filter, and reducer. Any internal functionality of these components is outside the scope of this investigation and could be a project upon itself.

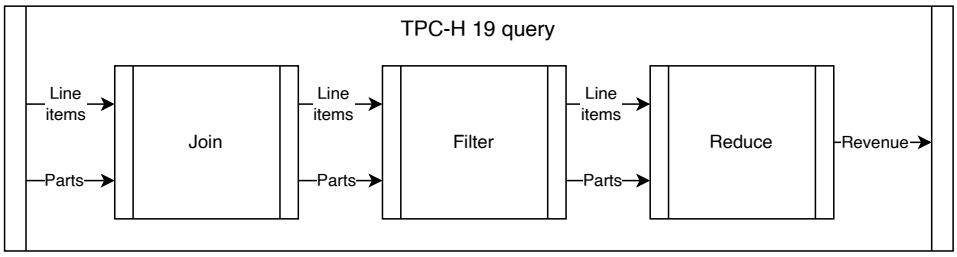


Figure 6.6: Developed TPC-H 19 data-flow

For this example, non-nullable `Text` and `String` types are implemented as a 1D character stream. Nullable `Text` is implemented as a *Union* of the respective character stream and *Null*. The Scala/Chisel file with all the TPC-H types and streams is very long and therefore not bundled in this document. Listing B.2 gives the generated Chisel code TPCH-19 specific data-types, streams, and interface classes. Listing B.3 shows the implementation stubs for the components, the generated top level component that includes stream connections, and a main class that provides the number of code-lines for next section's analysis. This code shows a disadvantage of usage of the standard representation streams. In this case, but likely in general, every string in the data-structure is a nested stream that is exposed as a parallel stream. All these streams must be connected in order to transfer all data. This results in

	Tydi-lang	Generated Chisel boilerplate	Manual Chisel	Verilog output – unoptimized	Verilog output – debug optimization
Simple number pipeline	45	107	27	113	60
Advanced number pipeline	45(+8)	107	33	25259	381
TPCH 19 interfaces	80	357	-	1587	487

Table 6.1: Lines of code for different representations; see text for explanation

a lot of boilerplate code and reduces visual clarity. On the other hand, since the code is automatically generated from a cleaner description in Tydi-lang, it should not require further inspection or alteration.

6.3. Investigation of saved effort

It is difficult to give a general heuristic about effort saved through the contributions of this paper. A lot of the value provided by this methodology will manifest itself through the lack of research required in custom communication solutions and effort saved through better tooling. A general statistic that is often included in code transformation and generation research is lines of code of input and output. An attempt was made to analyze the the lines of code of various representations of the three systems discussed in the previous section. Numbers are provided for the Tydi-lang description, generated Chisel boilerplate, manual creation in Chisel using Chapter 4's utilities, and (System)Verilog output, both in unoptimized and debug-optimized mode for a representation of expression in low-abstraction HDLs. Production optimized Verilog code is not included because empty components and signals are trimmed, resulting in the code being unsuitable to write further implementation code for.

The first system is the simple number pipeline. The Tydi-lang code for this example was given earlier in Listing 6.2, counting 47 lines. The generated, deduplicated, Chisel boilerplate code, with -to be developed- implementations, comes in at 107 lines including docstring comments. An efficient manual implementation using Tydi-Chisel's base classes and syntax in a clever way, can express the same circuit in 27 lines, excluding imports, however. Unoptimized and debug-optimized Verilog outputs are 113 and 60 lines respectively. For a simple system, the number of lines used is thus in a same order of magnitude.

For the second system, the advanced number pipeline, the Tydi-lang code is almost exactly the same as the first. Only the number of lanes have changed from this perspective. If the complexity converter were included as a streamlet and instance, this would add approximately 8 lines. Similarly, the generated Chisel boilerplate code is the same, just with more lanes. To remind the reader, the boilerplate must still be extended with implementation code. In the manual Chisel code, partially shown in Listing 6.5[1], the interleave and stream complexity converter utilities are used, logically resulting in (much) more Verilog output. While this can be seen as an unfair comparison, since implementation code is included in the Verilog output that is also dependent on the specified buffer size, a lot of design effort could be saved by use of these utilities.

Finally, the example of the TPC-H 19 query is the biggest. Although the full Tydi-lang description in Listing B.1 is 155 lines, the relevant code for TCPH-19 is only 80. This part results 357 lines of Chisel boilerplate. A manual implementation was not attempted here. Unoptimized and debug-optimized Verilog outputs are 1587 and 487 lines respectively for this system.

The results of this comparison show that the lines of code saved by use of tooling varies by design complexity, use of utilities, and re-use of data-types and interfaces. Comparing to debug-optimized Verilog, in the simplest case, an improvement of 25-55% on code size is made, depending on method of implementation. The second example mainly shows how use of utilities and available IP blocks can save a lot of design effort. For a big project like the TPC-H 19 query, the debug-optimized Verilog is six times as large as the original Tydi-lang specification! This will be mainly due to the number of nested streams for strings and the re-use of elements and streamlets. As mentioned at the start of this section and throughout this document, the added value of using Tydi and the tools and utilities of this document

[1] Listing A.3 in Appendix A shows the code of the advanced example *with* implementation. For the comparison, the `MultiReducer` and `NonNegativeFilter` implementations were left empty

are encountered at various stages of development and cannot simply be expressed in lines of code of expressed boilerplate. That being said, next to easier implementation, the suggested methods still show a reduction in lines to write, up to several times the code's size.

7

Testing & verification

The contents of this chapter revolve around the development tools to simplify component testing and subsequent verification of the developed framework utilities and components. A Chiseltest test driver is developed and the potential for further development analyzed. Various verification strategies for the library functionality are sketched. The stream complexity converter is tested in detail.

Testing and verification clearly is an important field in both software and hardware design. A good test will guarantee that a component will function in any prescribed condition. This chapter describes utilities to make testing and debugging easier, and verification of components that were developed for this project.

7.1. Testing & debugging utilities

When designing data-streaming components that require asynchronous structured data communication, components are designed and expected to fulfill some high-level function. In the simplest case, a component might be fed with a string or a series of numbers. In more complicated cases, a top-level component might be responsible for processing whole data-sets with multiple tables. When working with high-level communication and processing like this, it is undesirable, even on the verge of hopeless, to inspect and test a component by looking at simple signal waveforms. It is clear that additional utilities are required to help designers work with Tydi streams.

7.1.1. Desired functionality

It is easy to think of all kinds of high-level test and debug utilities that would be convenient to use. Tools must be built-up from the ground, or on top of other tools, however. Speculating about very high-level tools is, therefore, not a good strategy. Instead, it can be investigated what the most laborsome component of working with current tools is, create new tooling, and iterate. Chisel's core testing utility is *Chiseltest*. This framework allows assigning values to signals, reading signal values, placing assertions on them, and clock stepping. This tooling's basic functionality is suitable for simple combinational circuits, final state machines, or more even complicated input-output circuits, but is inadequate for dataflow components. A Tydi stream is an aggregate signal consisting of multiple sub-signals. The total meaning of the data is made up of all sub-signal values together. A more fitting way to set and check the signal values would, therefore, be to address the Tydi stream signal as a whole instead of the sub-signals.

A stream can have multiple data lanes, so it should be easy to distribute data onto the lanes and set other relevant data such as the lane `last` flags and strobe. A way to do this in a systematic and accessible way is thus necessary. Subsection 7.1.2 will introduce the concept of a test driver to do this and explicate the implementation's functionality.

Difficulty in conceiving the best test tooling arises based on the fact that different modules have different optimal testing methods. During this master thesis project, most of the tests involved verifying functionality of lower-level utility components. In these situations, exact placement and timing of data and dimensionality information matters to check the behavior of the component. For these situations,

a low-level abstraction is fitting that exposes a lot of information and allows for great control. In other cases, when testing a top level module, the focus will be more on the data flowing in and out, and less on how this data is e.g. distributed over lanes. If anything, this would ideally be done automatically by the software according to a specified stream-complexity. A high-level abstraction is more fitting accordingly. Finally, each project will have specific requirements with respect to data layout and communication. It is likely that specific utility functions will still be required to fulfill functions that a general tooling solution cannot fulfill. In the end, it is therefore best to create flexible tools that can easily be modified or built upon.

7.1.2. Testing driver

It was established in the previous section that, when testing high-level, data-driven circuits, it is undesirable to `poke` and `peek` individual wires at set times. Instead, a more asynchronous approach of enqueueing data on the input streams and waiting for and checking the validity of the data that is dequeued at the output stream(s) is a more functional approach.

The Chiseltest package offers an example of a testing driver for `DecoupledIO` signals, Chisel's handshaked IO solution that was already mentioned in Chapter 2. This test driver allows specifying data to be enqueued to or dequeued from a `DecoupledIO` signal, abstracting away the need for manual interaction with the `ready` and `valid` signals. This concept can be extended to Tydi streams to enqueue and dequeue data on stream signals. So, to aid designers with writing these functional tests for Tydi-interface using components, a test driver was developed for Tydi stream signals based on the `DecoupledIO` test driver from the Chiseltest package.

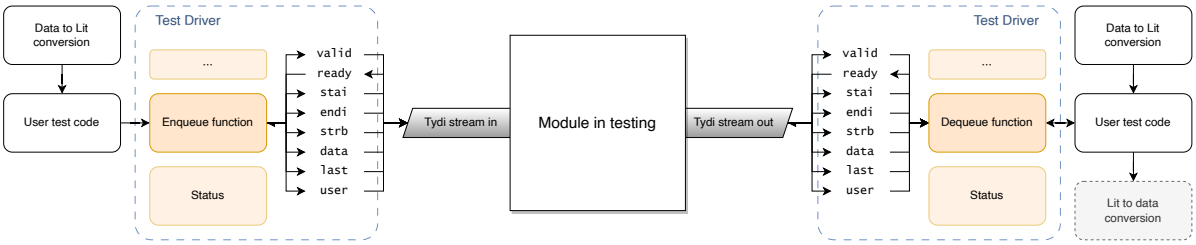


Figure 7.1: Testing set-up with module, streams, and testing driver

7.1.3. Test wrapper

Creation of data literals

One of the more challenging parts about setting, getting, and checking data to and from signals is the creation of data literals. A `poke` or `expect` call requires such a literal value to respectively set the signal value to or compare the signal value to. For ground types such as `UInt` and `SInt`, this is straightforward. Different fabrication methods exist, such as `int` or `String` conversion: `-8.S`, `"b101101".U`. For aggregate types, such as bundles and vectors, experimental fabrication methods are available (added through implicit classes, just like the ground methods). A bundle literal can then, for example, be created with `myBundleInstance.Lit(_.a -> -8.S, _.b -> "b101101".U)`. A vector literal can be created in two ways. One option is direction construction: `Vec.Lit(1.U, 2.U)`, creating a contiguous sequence of elements with width of the widest element. Another is again instance based: `myVecInstance.Lit(0 -> 1.U, 2 -> 2.U)`, where a sequence is created of `myVecInstance`'s type with elements at specified indexes. When for a bundle not all elements are specified, `pokePartial` can be used to only poke the specified sub-signals without throwing an error about incompleteness.

In most cases, an instance of the `Vec` or `Bundle` instance must be available to be able to call these literal creation methods. On top of this, the instance is not allowed to be a hardware registered instance such as a wire. One can therefore not simply state `c.in.data.Lit(0 -> c.in.el.Lit(_.a -> ...), ...)` because both of these are hardware. Creating the required instances is somewhat verbose and keeping them around is inconvenient. The test driver therefore includes some methods to more easily create the hardware literals. These include `elLit` to create a literal for a stream's data element type, and `dataLit` to create a literal for a stream's data lane vector. These test driver methods can be combined with a user's own functions to easily create the required literals for testing from an

intuitive source representation. A valid way to create a full data literal using the driver would then be `c.in.dataLit(0 -> c.in.elLit(_.a -> ...), ...)`.

Enqueueing and dequeueing

The `DecoupledIO` test driver's enqueue functions are rather simple. A ready stream is either awaited or expected, depending on variant, valid is asserted and signal data assigned, followed by a clock step. The `enqueue` method waits for the sink to be ready, whereas the `enqueueNow` method expects the sink to be ready the moment it is called (it asserts this). Finally, an `enqueueSeq` method is available that takes a sequence of data literals and calls `enqueue` for each item in the sequence. Mirrored `dequeue`, `dequeueNow`, and `dequeueSeq` methods are available for sources. All methods have their functionality wrapped in a `timescope`, such that signals manipulated inside the call return to their previous value when the call is finished. This prevents issues like accidentally leaving a flag set or data driven on a signal.

For Tydi streams, a few adaptations must be made. First of all, a stream's state does not only consist of the handshake and data bits, but also the other sub-signals. On the other hand, often their value is optional, see Figure 2.2. The Tydi stream driver `enqueue` and `enqueueNow` methods therefore contain `Option` arguments for the non-data signals. When these arguments are not specified, the signals are not perturbed. If they are, the signals get set to their specified values. The same principle applies for the `dequeue` and `dequeueNow` methods.

The `enqueueSeq` and `dequeueSeq` are a bit different. When checking a sequence, it is likely undesirable to specify all signal values for every time-step that the stream is ready/valid. What's more, the data sequence can be multi-dimensional with an arbitrary number of dimensions. Since Scala is a statically typed language, all types must be known at compile time. One cannot dynamically nest the source type in an unknown number of sequences. Lastly, optimal behavior differs between complexities $C < 8$ and $C = 8$, where multiple inner sequences can be sent per transfer. This makes writing a general purpose sequence-processing function quite difficult. This is therefore considered future work. Fortunately, with the available enqueue and dequeue methods, writing code that transmits or receives a use-case specific sequence should be straightforward.

State printing

Since a Tydi stream has 8 signals and can have multiple lanes, it is hard to get a grasp of the overall state that is conveyed at a given time. Therefore, a state printing function was developed that attempts to portray the information in a comprehensible and structured way. Listing 7.1 and Listing 7.2 show examples of a state print of respectively the output and input stream of the complexity converter in the "Hello World" test of Subsection 7.2.2. In the header, the name of the signal, signal direction, and clock tick count are displayed (if available). Then follow `stai`, `endi`, `strb`, and `last`. Subsequently, lane-specific information is printed. This example's streams have six lanes and a complexity of 8 for the input and 1 for the output. Since the signals behave differently at different stream complexity levels (see e.g. Table 2.2, Figure 2.2), the relevant information is also different for different C. In the example of the $C = 1$ output, only the last lane of the `last` signal is relevant and, therefore, displayed. The strobe is also used to turn off all lanes at once, so its value is displayed as `true` or `false`. For the $C = 8$ input, all last values are relevant and displayed, both in the header and per lane. A lane's validity is controlled by the start index, end index, and the relevant strobe bit. To make debugging this easier, lane activeness is displayed together with its reason components. Finally, the data is printed per lane. The data layout designers will use for their stream elements is arbitrary and cannot be predicted. Therefore, by default, the data literal is printed. For this example, the letter 'n' will be rendered as `BitsEl(value=UInt<8>(110))`. While this is a technically correct representation of the value, it is not very useful to a designer. Therefore, a data renderer function can be supplied to the state printing method to convert the data literal to a string format that is useful to the designer. In this case the function is `def charRenderer(c: BitsEl): String = s"'${c.value.litValue.toChar}'"` or `_.value.litValue.toChar` as a lambda function.

Listing 7.1: State printing example of out stream

```
State of "out" ↑ @ clk-step 6:
valid ↑: ✓          ready ↓: ✓
stai ≥: 0           endi ≤: 3
strb: true (001111)
```

```
last: 11
Lanes:
0   data: 'n'
    active: ✓       (strb=✓; stai=✓; endi=✓;)
1   data: 'i'
    active: ✓       (strb=✓; stai=✓; endi=✓;)
2   data: 'c'
    active: ✓       (strb=✓; stai=✓; endi=✓;)
3   data: 'e'
    active: ✓       (strb=✓; stai=✓; endi=✓;)
4   data: ' '
    active: ×       (strb=×; stai=✓; endi=×;)
5   data: ' '
    active: ×       (strb=×; stai=✓; endi=×;)
```

Listing 7.2: State printing example of in stream

```
State of "in" ↓ (unable to get clock):
valid ↓: ✓          ready ↑: ✓
stai ≥: 0           endi ≤: 5
strb: 000011
last: 00|00|01|10|11|10
Lanes:
0   data: 'c'
    last: 00
    active: ✓       (strb=✓; stai=✓; endi=✓;)
1   data: 'e'
    last: 00
    active: ✓       (strb=✓; stai=✓; endi=✓;)
2   data: ' '
    last: 01
    active: ×       (strb=×; stai=✓; endi=✓;)
3   data: ' '
    last: 10
    active: ×       (strb=×; stai=✓; endi=✓;)
4   data: ' '
    last: 11
    active: ×       (strb=×; stai=✓; endi=✓;)
5   data: ' '
    last: 10
    active: ×       (strb=×; stai=✓; endi=✓;)
```

Other utilities

Finally, some functions are defined to more nicely print data literals. Primarily, three general functions have been created. The first creates a binary representation string from a number value or literal with appropriate zero-padding. The second transforms a vector literal to a string of comma separated item literal values. The third does the same but prints the items in binary using the first function.

7.1.4. Future work

Indulging in speculating about useful tooling, several ideas for future work on tooling can be identified. One of the bigger problems with the currently available tooling is that creating the data literals required to `poke` or `expect`, and conversion back to a well-readable format is cumbersome. As explained, in order to `poke` a wire, a literal value must be supplied. These literals are still required when using the higher-level enqueue and dequeue functions. While the literal-creation methods of the test driver make this easier, wrapping user code is still often necessary. To lower test design efforts, mechanisms to ease literal creation could be developed.

Another opportunity for progress was mentioned in the section about enqueueing and dequeueing. There is no standard way or mechanism to enqueue or dequeue a data sequence to or from a stream. Figure 7.2 shows a variation of Figure 7.1 where the test driver abstracts away more functionality so designers have to focus less on writing utilities for testing. In a software framework, one would load in a data-frame or -table, feed it to the program, and check the output. Despite the described tooling advancements, verification of hardware components is still far off from the ease of software verification.

Further, the functionality offered by the stream state printing could be extended as a plug-in for waveform viewing applications or other visualizations. The better the state, data, and communication of a component can be outlined, the easier it is to grasp what is going on in a component, identify issues, and solve problems.

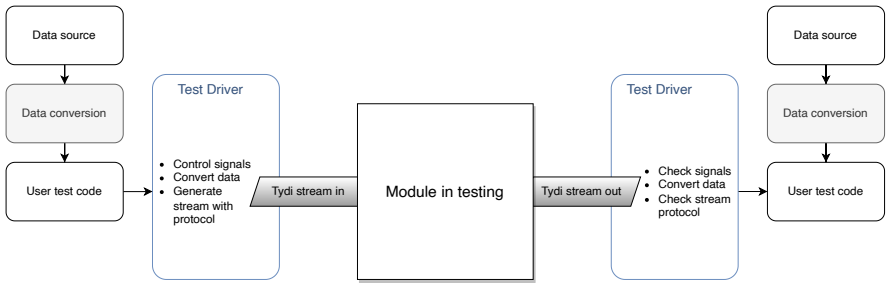


Figure 7.2: Envisioned testing set-up with module, streams, and testing driver

Finally, opportunities lie in stream protocol conformation verification. This holds for both source and sink streams. When enqueueing data to a sink, edge cases of a specified stream complexity should be investigated to guarantee compliance of the component. Similarly, the output stream of a component should be examined to verify standard conformity. Tooling to automatically permutate the input signals to explore edge cases and check output stream compliance can help hardware designers to certify their IP as stable and ready for use.

7.2. Framework and component testing

7.2.1. Tydi compliance

Section 4.2 describes the two stream representations that Tydi-Chisel offers, and the intricacies of connecting/converting between the two. Because Tydi prescribes a special ordering for the data, care should be taken to follow this order, and keep data intact between connections. To ensure this, a Tydi compliance test was created. The module in testing (MIT) is a pass-through module that has a detailed stream input, connected to a standard stream in the middle, connected to a detailed stream output. Data is `poke`d at the `data` and `last` signals of the input. Data direction and integrity is thereafter verified at the middle and output stream. For proper verification of aspects, the streams consist of two lanes and the datatype is a `Group` with two elements with byte values. The data is set to, in binary notation, `{00000000, 01010101}, {11111111, 10101010}` and the last lanes to `00000000, 11111111`. Following Tydi standard, this will result in a `data` signal value of `0xAAFF5500` and a `last` signal of `0xFF00`. Some convenience type conversion methods are also tested for correctness.

7.2.2. Complexity converter

The stream complexity converter, highlighted in Section 4.5, was not conceived and designed in one go, but rather in phases. In each phase, more tests were developed or expanded to verify added functionality. In the end, five tests were developed, verifying different different aspects of the component at different levels.

1. A sequential low level internal and external signal test with one data lane
2. A sequential low level internal and external signal test with two data lanes
3. A sequential enqueue and dequeue test using testing driver
4. Functional parallel 2D string test – “she is a dolphin”
5. Functional parallel 2D string test – “Hello World”

Low level tests

Tests 1 and 2 were created in the first phase of the module’s development. The complexity converter is only concerned with the elements themselves, and not with the element contents. It therefore operates on standard stream representation. For these basic tests, standard representation streams are therefore used. Consequently, for both cases, data is poked manually onto the stream IO wires. Some data is first enqueued, and various indexes and counters are checked along the way to ensure correct internal state. Checked signals include the current write or top index, number of series stored, number of items ready to transfer out, an number of items that are transferring out. Then, dequeueing is

initiated and the same signals checked. The purpose of these tests was to ensure the correctness of these basic signals to rule them out as sources of error for further development and to detect errors if changes were made to the component. This did happen later in development in conjunction with the functional tests. Test 2 added verification when varying the number of supplied or withdrawn elements using e.g. strobe.

Test driver test
The 'test driver test' was developed in part to test and develop the test driver and in part test the component in a more use-case appropriate data-driven way. To do this, the component interface used should be a detailed stream, not a standard stream. Therefore, a wrapper component was created for testing that simply exposes a detailed stream that is connected to the complexity converter's standard stream for both in- and output. The detailed streams can be manipulated by the testing driver. This test driver test is still sequential and still verifies some low level signals, similar test 1 and 2. A single data lane is used.

"She is a dolphin" test
This is the first test that is set-up as a high-level functionality test. Transfers are enqueued and dequeued in parallel functions to undo the forced synchronicity of sequential tests. This test is again executed with the detailed-stream-exposing wrapper and test driver. The test follows the transfer of `{{she}, {is}, {a}, {dolphin}}`, as depicted in [28, Figure 3.b], modernized in Figure 2.2. Aspects tested include the re-alignment of elements, output formatting, closing multiple sequences per transfer, and delayed `last` info. For this test, a stream layout of 4 lanes was chosen with a simple 8 bit `BitsEl` data element, capable of containing ASCII characters. Table 7.1 displays the transfers to the input and output of the MIT. A _ character means that there is no data in the respective lane, and the respective strobe bit will be low. `last` bits are asserted at word ends, except for the last character, for which the `last` flags are delayed to transfer 7. Important to note is that output transfer #1 will not happen at the same time as input transfer #1, but rather after input transfer #7, where the sequence is closed off. The table shows how the garbled input is correctly transformed to neat continuous output transfers, as is expected.

Transfer #	Input	Output
1	shei	she
2	-	is
3	s_a_	a
4	d_ol	dolph
5	ph__	in
6	__in	
7	`last` indicator	

Table 7.1: Transfers of "she is a dolphin" test

"Hello World" test
The Hello World test is the most complete system test of the complexity converter component. The set-up for this test is the same as the previous test, with wrapper component, driver, and data-type. The only difference is the number of lanes, which is 6 in this case. The testing case is taken from [6, last signal description], where a detailed representation of the input transfer can be found as well. This test tests parallel input and output, delayed `last` flags & re-alignment, and empty sequences. These aspects cover the hardest scenarios that the component will face when used in real designs. Since input and output is, for the first time, parallel, the data indexing for and movement in the buffer is verified.

The visual explanation of Figure 4.2 displays the state of the complexity converter at transfer 3, but with different input data to offer more insight in the re-alignment of data and `last`-reduction. The data that is eventually transferred is `["Hello", "World"], ["Tydi", "is", "nice"], [""], []`. Table 7.2 shows what data is in which transfer for input and output. A technical an semantic

Transfer #	Input	Input semantic	Output	Output semantic
1	H e l l o W 00 00 00 01 00 00	[“Hello”“W	Hello 01	[“Hello”
2	o r l d T y 00 00 00 11 00 00	orld”][“Ty	World 11	“World”]
3	d i i s n i 00 01 00 01 00 00	di”“is”“ni	Tydi 01	[“Tydi”
4	c e _ _ _ _ 00 00 01 10 11 10	ce”][“”][]	is 01	“is”
5			nice 11	“nice”]
6			_ 11	[“”]
7			_ 10	[]

Table 7.2: Transfers of “Hello World” test

representation is included. The first line gives the characters that are sent, where again a _ indicates an inactive data-lane. The second line gives the `last` signal value.

In Listing 7.3 and Listing 7.4, the code for enqueueing and dequeueing & checking the data is shown. Some utility functions are used for the construction of required vector literals from strings and bundle literals from characters.

Listing 7.3: “Hello World” test, enqueue side

```
// Send HelloW
c.in.enqueueNow(vecLitFromString("HelloW"),
    last = Some(Vec.Lit("b00".U(2.W), "b00".U, "b00".U, "b00".U, "b01".U, "b00".U)),
    strb = Some(bRev("111111")))

// Send orldTy
c.in.enqueueNow(vecLitFromString("orldTy"),
    last = Some(Vec.Lit("b00".U(2.W), "b00".U, "b00".U, "b11".U, "b00".U, "b00".U)),
    strb = Some(bRev("111111")))

// Send diisni
c.in.enqueueNow(vecLitFromString("diisni"),
    last = Some(Vec.Lit("b00".U(2.W), "b01".U, "b00".U, "b01".U, "b00".U, "b00".U)),
    strb = Some(bRev("111111")))

// Send ce____
c.in.enqueueNow(vecLitFromString("ce"),
    last = Some(Vec.Lit("b00".U(2.W), "b00".U, "b01".U, "b10".U, "b11".U, "b10".U)),
    strb = Some(bRev("110000")))
```

Listing 7.4: “Hello World” test, dequeue side

```
c.out.waitForValid()
c.out.ready.poke(true)

c.out.data.expectPartial(vecLitFromString("Hello"))
c.out.last.last.expect("b01".U)
c.out.endi.expect(4.U)

c.clock.step(1)
c.out.data.expectPartial(vecLitFromString("World"))
c.out.last.last.expect("b11".U)
c.out.endi.expect(4.U)

c.clock.step(1)
c.out.data.expectPartial(vecLitFromString("Tydi"))
c.out.last.last.expect("b01".U)
```

```
c.out.endi.expect(3.U)

c.clock.step(1)
c.out.data.expectPartial(vecLitFromString("is"))
c.out.last.last.expect("b01".U)
c.out.endi.expect(1.U)

c.clock.step(1)
c.out.data.expectPartial(vecLitFromString("nice"))
c.out.last.last.expect("b11".U)
c.out.endi.expect(3.U)

c.clock.step(1)
// Transfer should be done now.
println("\n-- Transfer out 6")
c.out.last.last.expect("b11".U)
c.out.strb.expect(0.U)
c.out.endi.expect(0.U)

c.clock.step(1)
c.out.last.last.expect("b10".U)
c.out.strb.expect(0.U)
c.out.endi.expect(0.U)
```

8

Conclusions and recommendations

8.1. Conclusions

The research question encompassing this work was:
how can the Chisel language be used within the Tydi ecosystem, simplifying the process of implementing Tydi-using components, thereby reducing data-streaming accelerators design complexity?

There were several aspects and challenges to achieving this.

- First was: what role can Chisel fulfill in existing and future Tydi-based toolchains?
  In this project, Chisel is shown to be a suitable platform for writing component implementations for systems defined in Tydi-lang, but also to be an effective platform for full development on its own using Tydi-Chisel's utilities.
- Second, how can Tydi-concepts be effectively integrated in Chisel?
  Through intricate and polymorphically typed implementations, Tydi concepts were successfully integrated in Chisel. Tydi-elements are based on Chisel's `Bundle` class for natural integration. Tydi-streams are available in Tydi-Chisel in two variants, offering maximum utility to hardware designers and compatibility with external projects. Advanced concepts like nested streams are fully supported.
- Third, what additional utilities are required to lower design-complexity?
  Several general purpose utilities were developed in various categories that allow designers to save development effort. Component implementation effort was reduced by supplying module base classes to automate initialization. Working with streams was simplified by providing detailed stream interfaces with excellent IDE typing support. Compositing a system from components is facilitated by the interleave and stream complexity converter components and module chaining syntax.
- Finally, how can developed functionality be verified?
  The answer to this question is twofold. To start, testing utilities to aid Tydi-enabled component verification were developed. These testing utilities were subsequently used to verify functionality of developed utilities and components.

In answer to the main question and in summary, Tydi-Chisel is presented as a library with tools and utilities to effectively integrate, implement, and test Tydi concepts and components in Chisel. Next, using two-way code generation, a comprehensive accelerator design pipeline is introduced. In various examples, the power of system composability when working with Tydi-components is demonstrated.

Tydi's standard and specification abilities allow software and hardware designers to work together better in an interface-driven approach. It also allows hardware designers to avoid the pitfalls of designing or working with custom dataflow communication solutions. Thanks to the work in this thesis, together with previous projects, tools are available for specification of dataflows in the design, creating hardware design boilerplate code from this specification, utilities for writing the implementations, testing,

and generating software-hardware interfaces for communication trough Apache Arrow. In the future, Tydi-related tooling can be expanded to aid developers in various stages of accelerator development.

Eventually the author envisions an ecosystem of IP components with Tydi interface specifications. Designers working on a data-streaming hardware design project could then use these IP components, needing to concern themselves only the data communication specification, which is easy to implement, and not the component's implementation, avoiding implementation-dependent design.

8.2. Recommendations

The work presented in this thesis can serve as a foundation for future development of Tydi or otherwise related tools. Additionally, the project itself has several areas for enhancement.

- Similar frameworks to Tydi-Chisel could be created for other advancing HDLs. Code generation for new platforms could easily be implemented by adapting the Tydi-lang-2-Chisel code and templates.
- Additional library utilities can be developed to aid hardware designers. Interesting targets might be components that execute common operations required for executing queries, such as string matching.
- Enhanced tooling could be developed for testing and verification, both within Chisel and in waveform utilities.
  - For Tydi-Chisel specifically, an easier way to create data literals from data sources and en-/dequeue those.
  - Stream protocol compliance verification.
- Signal naming is as of yet inconsistent between tools and the Tydi standard.
- The Fletcher project could be expanded to directly generate hardware components with Tydi interfaces to enable easy data exchange with computers.
- The Chisel framework could be adapted to release more information about the generated system for introspection.
- Tydi-lang's functionality can be expanded and limitations (discussed in Subsection 5.2.3) reduced.
- Detection of signal direction and error handling of stream connection methods could be improved.

A

Number pipeline code

Listing A.1: Number pipeline in Chisel

```
package pipeline

import tydi_lib._
import chisel3._
import chisel3.internal.firrtl.Width
import chisel3.util.Counter
import chiseltest.RawTester.test
import circt.stage.ChiselStage.{emitCHIRRTL, emitSystemVerilog}

/** Basic data-types used in groups etc. */
trait PipelineTypes {
  val dataWidth: Width = 64.W
  def signedData: SInt = SInt(dataWidth)
  def unsingedData: UInt = UInt(dataWidth)
}

/** A basic Group element with a timestamp and a value. */
class NumberGroup extends Group with PipelineTypes {
  val time: UInt = UInt(64.W)
  val value: SInt = signedData
}

/** Statistics output group. */
class Stats extends Group with PipelineTypes {
  val min: UInt = unsingedData
  val max: UInt = unsingedData
  val sum: UInt = unsingedData
  val average: UInt = unsingedData
}

/** A module based on a stream-processing base with input and output streams of type `NumberGroup`.
 * Input and output streams are passthrough-connected by default so only meaningful signals are
    overridden.  */
class NonNegativeFilter extends SubProcessorBase(new NumberGroup, new NumberGroup) {
  outStream.strb := inStream.strb(0) && inStream.el.value >= 0.S
}

/** Another streaming module that calculates some statistics of the incoming stream. */
class Reducer extends SubProcessorBase(new NumberGroup, new Stats) with PipelineTypes {
  val maxVal: BigInt = BigInt(Long.MaxValue)  // Must work with BigInt or we get an overflow
  val cMin: UInt = RegInit(maxVal.U(dataWidth))
  val cMax: UInt = RegInit(0.U(dataWidth))
  val nValidSamples: Counter = Counter(Int.MaxValue)
  val nSamples: Counter = Counter(Int.MaxValue)
  val cSum: UInt = RegInit(0.U(dataWidth))

  inStream.ready := true.B
  outStream.valid := nSamples.value > 0.U

  when (inStream.valid) {
    val value = inStream.el.value.asUInt
    nSamples.inc()
    when (inStream.strb(0)) {
      cMin := cMin min value
      cMax := cMax max value
      cSum := cSum + value
```

```
      nValidSamples.inc()
    }
  }
  outStream.el.sum := cSum
  outStream.el.min := cMin
  outStream.el.max := cMax
  outStream.el.average := Mux(nValidSamples.value > 0.U, cSum/nValidSamples.value, 0.U)
}

/** Using the stream processing modules, connecting them manually. */
/*class PipelineExampleModule extends TydiModule {
  private val numberGroup = new NumberGroup
  private val statsGroup = new Stats

  // Create and connect physical streams following standard with concatenated data bitvector
  val numsIn: PhysicalStream = IO(Flipped(PhysicalStream(numberGroup, 1, c = 7)))
  val statsOut: PhysicalStream = IO(PhysicalStream(statsGroup, 1, c = 7))

  val filter = Module(new NonNegativeFilter())
  filter.in := numsIn
  val reducer = Module(new Reducer())
  reducer.in := filter.out
  statsOut := reducer.out
}*/

/** Using the stream processing modules with chaining syntax.
 * SimpleProcessorBase is similar to SubProcessorBase but does not expose the detailed Stream content signals. */
class PipelineExampleModule extends SimpleProcessorBase(new NumberGroup, new Stats) {
  out := in.processWith(new NonNegativeFilter).processWith(new Reducer())
}

object PipelineExampleModule extends App {
  println("PipelineExample")

  test(new TopLevelModule()) { c =>
    println("Tydi-lang code of PipelineExample")
    println(c.tydiCode)
  }

  println("FIRRTL & Verilog of NonNegativeFilter")
  println(emitCHIRRTL(new NonNegativeFilter()))
  println(emitSystemVerilog(new NonNegativeFilter(), firtoolOpts = firNormalOpts))

  println("FIRRTL & Verilog of Reducer")
  println(emitCHIRRTL(new Reducer()))
  println(emitSystemVerilog(new Reducer(), firtoolOpts = firNormalOpts))

  println("FIRRTL & Verilog of PipelineExampleModule")
  println(emitCHIRRTL(new PipelineExampleModule()))
  println(emitSystemVerilog(new PipelineExampleModule(), firtoolOpts = firNormalOpts))

  println("Done")
}
```

Listing A.2: Test code for number pipeline

```
package TydiTesting

import chisel3._
import chiseltest._
import org.scalatest.flatspec.AnyFlatSpec
import tydi_lib._
import tydi_lib.testing.Conversions._
import pipeline._

class PipelineExampleTest extends AnyFlatSpec with ChiselScalatestTester {
  behavior of "PipelineExample"

  class NonNegativeFilterWrap extends TydiTestWrapper(new NonNegativeFilter, new NumberGroup, new NumberGroup)
  class ReducerWrap extends TydiProcessorTestWrapper(new Reducer)
  class PipelineWrap extends TydiTestWrapper(new TopLevelModule, new NumberGroup, new Stats)

  it should "filter negative values" in {
    test(new NonNegativeFilterWrap) { c =>
      // Initialize signals
      c.in.initSource().setSourceClock(c.clock)
      c.out.initSink().setSinkClock(c.clock)

      parallel(
        c.in.enqueueElNow(_.time -> 123976.U, _.value -> 6.S),
```

```
25          c.out.expectDequeueNow(_.time -> 123976.U, _.value -> 6.S),
26        )
27
28        parallel(
29          c.in.enqueueElNow(_.time -> 123976.U, _.value -> 0.S),
30          c.out.expectDequeueNow(_.time -> 123976.U, _.value -> 0.S),
31        )
32
33        parallel(
34          c.in.enqueueElNow(_.time -> 123976.U, _.value -> -7.S),
35          timescope {
36            c.out.ready.poke(true.B)
37            fork.withRegion(Monitor) {
38              c.out.strb.expect(0.U)
39            }.joinAndStep(c.clock)
40          }
41        )
42      }
43    }
44
45    it should "reduce" in {
46      test(new ReducerWrap) { c =>
47        // Initialize signals
48        c.in.initSource().setSourceClock(c.clock)
49        c.out.initSink().setSinkClock(c.clock)
50
51        c.in.enqueueElNow(_.time -> 123976.U, _.value -> 6.S)
52        println(c.out.printState())
53        c.out.expectDequeueNow(_.min -> 6.U, _.max -> 6.U, _.sum -> 6.U, _.average -> 6.U)
54
55        c.in.enqueueElNow(_.time -> 124718.U, _.value -> 12.S)
56        println(c.out.printState())
57        c.out.expectDequeueNow(_.min -> 6.U, _.max -> 12.U, _.sum -> 18.U, _.average -> 9.U)
58
59        c.in.enqueueElNow(_.time -> 129976.U, _.value -> 15.S)
60        println(c.out.printState())
61        c.out.expectDequeueNow(_.min -> 6.U, _.max -> 15.U, _.sum -> 33.U, _.average -> 11.U)
62      }
63    }
64
65    it should "process a sequence" in {
66      test(new PipelineWrap) { c =>
67        // Initialize signals
68        c.in.initSource().setSourceClock(c.clock)
69        c.out.initSink().setSinkClock(c.clock)
70
71        // Enqueue first value
72        c.in.enqueueElNow(_.time -> 123976.U, _.value -> 6.S)
73        println(c.out.printState())
74        c.out.expectDequeueNow(_.min -> 6.U, _.max -> 6.U, _.sum -> 6.U, _.average -> 6.U)
75
76        // Enqueue second value that should be filtered out, output remains constant
77        c.in.enqueueElNow(_.time -> 123976.U, _.value -> -6.S)
78        println(c.out.printState())
79        c.out.expectDequeueNow(_.min -> 6.U, _.max -> 6.U, _.sum -> 6.U, _.average -> 6.U)
80
81        // Enqueue second valid value
82        c.in.enqueueElNow(_.time -> 124718.U, _.value -> 12.S)
83        println(c.out.printState())
84        c.out.expectDequeueNow(_.min -> 6.U, _.max -> 12.U, _.sum -> 18.U, _.average -> 9.U)
85
86        // Enqueue second invalid value
87        c.in.enqueueElNow(_.time -> 124718.U, _.value -> -12.S)
88        println(c.out.printState())
89        c.out.expectDequeueNow(_.min -> 6.U, _.max -> 12.U, _.sum -> 18.U, _.average -> 9.U)
90
91        // Enqueue third value
92        c.in.enqueueElNow(_.time -> 129976.U, _.value -> 15.S)
93        println(c.out.printState())
94        c.out.expectDequeueNow(_.min -> 6.U, _.max -> 15.U, _.sum -> 33.U, _.average -> 11.U)
95      }
96    }
97
98    it should "process a sequence in parallel" in {
99      test(new PipelineWrap) { c =>
100       // Initialize signals
101       c.in.initSource().setSourceClock(c.clock)
102       c.out.initSink().setSinkClock(c.clock)
103
104       // define min and max values numbers are allowed to have
105       val rangeMin = BigInt(Long.MinValue)
106       val rangeMax = BigInt(Long.MaxValue)
107       val nNumbers = 100
108
```

```
109         // Generate list of random numbers
110         val nums = Seq.fill(nNumbers)(
111           Int.MinValue + BigInt(32, scala.util.Random)
112         )
113
114         // println(nums)
115
116         // Storage for statistics
117         case class StatsOb(count: BigInt = 0,
118                            min: BigInt = rangeMax,
119                            max: BigInt = 0,
120                            sum: BigInt = 0,
121                            average: BigInt = 0)
122
123         val initialStats = StatsOb()
124
125         // Calculate cumulative statistics
126         val statsSeq = nums.scanLeft(initialStats) { (s, num) =>
127           if (num >= 0) {
128             val newCount = s.count + 1
129             val newSum = s.sum + num
130             val newMin = s.min min num
131             val newMax = s.max max num
132             val newAverage = newSum / newCount
133
134             s.copy(count = newCount, min = newMin, max = newMax, sum = newSum, average = newAverage)
135           } else {
136             s
137           }
138         }.tail
139
140         // Test component
141         parallel(
142           {
143             for ((elem, i) <- nums.zipWithIndex) {
144               c.in.enqueueElNow(_.time -> i.U, _.value -> elem.S)
145             }
146           },
147           {
148             for ((elem, i) <- statsSeq.zipWithIndex) {
149               // println(s"$i: $elem")
150               c.out.expectDequeue(_.min -> elem.min.U, _.max -> elem.max.U, _.sum -> elem.sum.U, _.
                      average -> elem.average.U)
151             }
152           }
153         )
154       }
155     }
156   }
```

Listing A.3: Advanced number pipeline in Chisel

```
1   package pipeline
2
3   import tydi_lib._
4   import chisel3._
5   import chisel3.experimental.hierarchy.Definition
6   import chisel3.internal.firrtl.Width
7   import chisel3.util.Counter
8   import chiseltest.RawTester.test
9   import circt.stage.ChiselStage.{emitCHIRRTL, emitSystemVerilog}
10
11
12  // Operating at C=1
13  class MultiReducer(val n: Int) extends SubProcessorBase(new NumberGroup, new Stats, nIn=n, dIn = 1,
        dOut = 1) with PipelineTypes {
14    val maxVal: BigInt = BigInt(Long.MaxValue)  // Must work with BigInt or we get an overflow
15
16    val cMinReg: UInt = RegInit(maxVal.U(dataWidth))
17    val cMaxReg: UInt = RegInit(0.U(dataWidth))
18    val cSumReg: UInt = RegInit(0.U(dataWidth))
19    val nSamplesReg: UInt = RegInit(0.U(dataWidth))
20    val lastReg: Bool = RegInit(false.B)
21
22    private val incomingItems = inStream.endi + 1.U(n.W)
23    private val last: Bool = inStream.last(n-1)(0)
24    private val strb: Bool = inStream.strb(0)
25
26    lastReg := lastReg || last
27
28    // Reset everything after transfer
29    when (lastReg && outStream.ready) {
```

```
      cMinReg := maxVal.U(dataWidth)
      cMaxReg := 0.U(dataWidth)
      cSumReg := 0.U(dataWidth)
      nSamplesReg := 0.U
      lastReg := false.B
    }

    // Do work when we have a valid transfer
    when (inStream.valid && inStream.ready && strb) {
      nSamplesReg := nSamplesReg + incomingItems

      val values: Vec[UInt] = VecInit(inStream.data.zipWithIndex.map {
        case (el, i) => Mux(i.U <= inStream.endi, el.value.asUInt, 0.U)
      })

      cMaxReg := cMaxReg max values.reduceTree(_ max _)
      cSumReg := cSumReg + values.reduceTree(_ + _)

      cMinReg := cMinReg min VecInit(inStream.data.zipWithIndex.map {
        case (el, i) => Mux(i.U <= inStream.endi, el.value.asUInt, maxVal.U(dataWidth))
      }).reduceTree(_ min _)
    }

    inStream.ready := !lastReg
    outStream.valid := lastReg
    outStream.last(0) := lastReg
    outStream.el.sum := cSumReg
    outStream.el.min := cMinReg
    outStream.el.max := cMaxReg
    outStream.el.average := Mux(nSamplesReg > 0.U, cSumReg/nSamplesReg, 0.U)
    outStream.stai := 0.U
    outStream.endi := 1.U
    outStream.strb := 1.U
  }

  class MultiNonNegativeFilter extends class MultiNonNegativeFilter extends MultiProcessorGeneral(
    Definition(new NonNegativeFilter), 4, new NumberGroup, new NumberGroup, d=1
  )

  class PipelinePlusModule(n: Int = 4, bufferSize: Int = 50)  extends SimpleProcessorBase(
    new NumberGroup, new Stats, nIn = n, nOut = 1, cIn = 7, cOut = 1, dIn = 1, dOut = 1) {
    out := in.processWith(new MultiNonNegativeFilter)
             .convert(bufferSize)
             .processWith(new MultiReducer(n))
  }

  object PipelineExamplePlus extends App {
    println("Test123")

    test(new PipelinePlusModule()) { c =>
      println(c.tydiCode)
    }

    println(emitCHIRRTL(new MultiNonNegativeFilter()))
    println(emitSystemVerilog(new NonNegativeFilter(), firtoolOpts = firNormalOpts))

    println(emitCHIRRTL(new MultiReducer()))
    println(emitSystemVerilog(new Reducer(), firtoolOpts = firNormalOpts))

    println(emitCHIRRTL(new PipelinePlusModule()))

    // These lines generate the Verilog output
    println(emitSystemVerilog(new PipelinePlusModule(), firtoolOpts = firNormalOpts))

    println("Done")
  }
```

Listing A.4: Test code for advanced number pipeline

```
  import chisel3._
  import chisel3.experimental.BundleLiterals.AddBundleLiteralConstructor
  import chisel3.experimental.VecLiterals.{AddObjectLiteralConstructor, AddVecLiteralConstructor}
  import chiseltest._
  import org.scalatest.flatspec.AnyFlatSpec
  import pipeline._
  import tydi_lib._
  import tydi_lib.testing.Conversions._

  class PipelineExamplePlusTest extends AnyFlatSpec with ChiselScalatestTester {
    behavior of "PipelineExamplePlus"

    private val n: Int = 4
```

```
14
15    class NonNegativeFilterWrap extends TydiTestWrapper(new MultiNonNegativeFilter, new NumberGroup, new
          NumberGroup)
16    class ReducerWrap extends TydiProcessorTestWrapper(new MultiReducer(n))
17    class PipelineWrap extends TydiTestWrapper(new PipelinePlusModule, new NumberGroup, new Stats)
18
19    private val numberGroup = new NumberGroup
20
21    def vecLitFromSeq(s: Seq[BigInt]): Vec[NumberGroup] = {
22      val mapping = s.map(c => numberGroup.Lit(_.value -> c.S, _.time -> 0.U)).zipWithIndex.map(v => (v.
            _2, v._1))
23      Vec(n, numberGroup).Lit(mapping: _*)
24    }
25
26    def numRenderer(c: NumberGroup): String = {
27      s"${c.value.litValue} @ ${c.time.litValue}"
28    }
29
30    def statsRenderer(c: Stats): String = {
31      s"min: ${c.min.litValue}, max: ${c.max.litValue}, sum: ${c.sum.litValue}, av: ${c.average.litValue
            }"
32    }
33
34    it should "reduce multi-lane" in {
35      test(new ReducerWrap) { c =>
36        // Initialize signals
37        c.in.initSource().setSourceClock(c.clock)
38        c.out.initSink().setSinkClock(c.clock)
39
40        val t1 = vecLitFromSeq(Seq(3, 6, 9, 28))
41        val t1Last = Vec.Lit(0.U, 0.U, 0.U, 1.U)
42
43        c.in.enqueueNow(t1, endi = Some(2.U), last = Some(t1Last))
44        println(c.out.printState(statsRenderer))
45        c.clock.step()
46        println(c.out.printState(statsRenderer))
47        c.out.expectDequeueNow(_.min -> 3.U, _.max -> 9.U, _.sum -> 18.U, _.average -> 6.U)
48
49        println(c.out.printState(statsRenderer))
50
51        val t2 = vecLitFromSeq(Seq(18, 6, 9, 28))
52        val t2Last = Vec.Lit(0.U, 0.U, 0.U, 0.U)
53        val t3 = vecLitFromSeq(Seq(3, 10, 12, 0))
54        val t3Last = Vec.Lit(0.U, 0.U, 0.U, 1.U)
55
56        c.clock.step()
57        println(c.out.printState(statsRenderer))
58        c.in.enqueueNow(t2, endi = Some(3.U), last = Some(t2Last))
59        println(c.out.printState(statsRenderer))
60        c.out.expectInvalid()
61        c.in.enqueueNow(t3, endi = Some(3.U), last = Some(t3Last))
62        println(c.out.printState(statsRenderer))
63        c.clock.step()
64        println(c.out.printState(statsRenderer))
65        c.out.expectDequeueNow(_.min -> 0.U, _.max -> 28.U, _.sum -> 86.U, _.average -> 10.U)
66      }
67    }
68
69    case class StatsOb(count: BigInt = 0,
70                       min: BigInt = Long.MaxValue,
71                       max: BigInt = 0,
72                       sum: BigInt = 0,
73                       average: BigInt = 0)
74
75    def randomSeq(n: Int): Seq[BigInt] = {
76      Seq.fill(n)(
77        Int.MinValue + BigInt(32, scala.util.Random)
78      )
79    }
80
81    def processSeq(seq: Seq[BigInt]): StatsOb = {
82      val filtered = seq.filter(_ >= 0)
83      val sum = filtered.sum
84      StatsOb(
85        count = filtered.length,
86        min = filtered.min,
87        max = filtered.max,
88        sum = sum,
89        average = sum / filtered.size,
90      )
91    }
92
93    it should "process a sequence in parallel" in {
94      test(new PipelineWrap) { c =>
```

```
95        // Initialize signals
96        c.in.initSource().setSourceClock(c.clock)
97        c.out.initSink().setSinkClock(c.clock)
98
99        // Generate list of random numbers
100       val nNumbers = 50
101       val nums = randomSeq(nNumbers)
102       val stats = processSeq(nums)
103       val filtered = nums.filter(_ >= 0)
104
105       println(s"Number of filtered items: ${stats.count}")
106       println(s"Stats: $stats")
107
108       // Test component
109       parallel(
110         {
111           for (elems <- nums.grouped(4)) {
112             c.in.enqueueNow(vecLitFromSeq(elems), endi = Some((elems.length-1).U))
113           }
114           c.in.enqueueElNow(numberGroup.Lit(_.time -> 0.U, _.value -> -1000.S), last = Some(1.U))
115         },
116         {
117           c.out.waitForValid()
118           println(c.out.printState(statsRenderer))
119           c.out.expectDequeue(_.min -> stats.min.U, _.max -> stats.max.U, _.sum -> stats.sum.U, _.
                average -> stats.average.U)
120         }
121       )
122     }
123   }
124 }
```

B

TPCH 19

Listing B.1: Tydi-lang code for TPCH types and types, streamlets, and implementations for TPCH-19 query

```
package pack0;

// Character type
Char = Bit(8);
// Integer type
Integer = Bit(4*8);
// Double type
Real = Bit(8*8);

// Stream for non-nullable text
TextStream = Stream(Char, t=1.0, d=1);

# This Union specifies a nullable text #
Union OptionalText {
    text: TextStream;
    null: Null;
}

// Stream for nullable text
OptionalTextStream = Stream(OptionalText, t=1.0, d=0);

Group Region {
    R_RegionKey: Integer;
    R_Name: TextStream;
    R_Comment: OptionalTextStream;
}

Region_stream = Stream(Region, t=1.0, d=1);

// NATION TABLE
Group Nation {
    N_NationKey: Integer;
    N_Name: TextStream;
    N_RegionKey: Integer; // FOREIGN KEY REFERENCES Region
    N_Comment: OptionalTextStream;
}
Nation_stream = Stream(Nation, t=1.0, d=1);

// PART TABLE
Group Part {
    P_PartKey: Integer;
    P_Name: TextStream;
    P_Mfgr: TextStream;
    P_Brand: TextStream;
    P_Type: TextStream;
    P_Size: Integer;
    P_Container: TextStream;
    P_RetailPrice: Real;
    P_Comment: TextStream;
}
Part_stream = Stream(Part, t=1.0, d=1);

// SUPPLIER TABLE
Group Supplier {
    S_SuppKey: Integer;
    S_Name: TextStream;
```

```
    S_Address: TextStream;
    S_NationKey: Integer; // FOREIGN KEY REFERENCES Nation
    S_Phone: TextStream;
    S_AcctBal: Real;
    S_Comment: TextStream;
}
Supplier_stream = Stream(Supplier, t=1.0, d=1);

// PARTSUPP TABLE
Group Partsupp {
    PS_PartKey: Integer; // FOREIGN KEY REFERENCES Part
    PS_SuppKey: Integer; // FOREIGN KEY REFERENCES Supplier
    PS_AvailQty: Integer;
    PS_SupplyCost: Real;
    PS_Comment: TextStream;
}
Partsupp_stream = Stream(Partsupp, t=1.0, d=1);

// CUSTOMER TABLE
Group Customer {
    C_CustKey: Integer;
    C_Name: TextStream;
    C_Address: TextStream;
    C_NationKey: Integer; // FOREIGN KEY REFERENCES Nation
    C_Phone: TextStream;
    C_AcctBal: Real;
    C_MktSegment: TextStream;
    C_Comment: TextStream;
}
Customer_stream = Stream(Customer, t=1.0, d=1);

// ORDERS TABLE
Group Orders {
    O_OrderKey: Integer;
    O_CustKey: Integer; // FOREIGN KEY REFERENCES Customer
    O_OrderStatus: TextStream;
    O_TotalPrice: Real;
    O_OrderDate: TextStream;
    O_OrderPriority: TextStream;
    O_Clerk: TextStream;
    O_ShipPriority: Integer;
    O_Comment: TextStream;
}
Orders_stream = Stream(Orders, t=1.0, d=1);

// LINEITEM TABLE
Group LineItem {
    L_OrderKey: Integer; // FOREIGN KEY REFERENCES Orders
    L_PartKey: Integer; // FOREIGN KEY REFERENCES Partsupp
    L_SuppKey: Integer; // FOREIGN KEY REFERENCES Partsupp
    L_LineNumber: Integer;
    L_Quantity: Integer;
    L_ExtendedPrice: Real;
    L_Discount: Real;
    L_Tax: Real;
    L_ReturnFlag: TextStream;
    L_LineStatus: TextStream;
    L_ShipDate: TextStream;
    L_CommitDate: TextStream;
    L_ReceiptDate: TextStream;
    L_ShipInstruct: TextStream;
    L_ShipMode: TextStream;
    L_Comment: TextStream;
}
LineItem_stream = Stream(LineItem, t=1.0, d=1);

Revenue_stream = Stream(Real, t=1.0, d=1);

streamlet Tphc19_Top_interface {
    lineItemsIn: LineItem_stream in;
    partsIn: Part_stream in;
    revenueOut: Revenue_stream out;
}

streamlet Tphc19_LineItem_Part_Passthrough_interface {
    lineItemsIn: LineItem_stream in;
    partsIn: Part_stream in;
    lineItemsOut: LineItem_stream in;
    partsOut: Part_stream in;
}

impl Tphc19_Join of Tphc19_LineItem_Part_Passthrough_interface {}
impl Tphc19_Filter of Tphc19_LineItem_Part_Passthrough_interface {}
impl Tphc19_Reducer of Tphc19_Top_interface {}
```

```
impl Tphc19_Top of Tphc19_Top_interface {
    instance join(Tphc19_Join);
    instance filter(Tphc19_Filter);
    instance reducer(Tphc19_Reducer);

    self.lineItemsIn => join.lineItemsIn;
    self.partsIn => join.partsIn;
    join.lineItemsOut => filter.lineItemsIn;
    join.partsOut => filter.partsIn;
    filter.lineItemsIn => reducer.lineItemsIn;
    filter.partsIn => reducer.partsIn;
    reducer.revenueOut => self.revenueOut;
}
```

Listing B.2: TPCH-19 specific generated types and interfaces

```
package tpch
import chisel3._
import tpch._
import tydi_lib._

class Revenue extends Group {
  val value: UInt = MyTypes.real
}

/** Stream, defined in pack0. */
class RevenueStream extends PhysicalStreamDetailed(e=new Revenue, n=1, d=1, c=1, r=false, u=Null())

object RevenueStream {
  def apply(): RevenueStream = Wire(new RevenueStream())
}

/**
 * Streamlet, defined in pack0.
 */
class Tphc19_LineItem_Part_Passthrough_interface extends TydiModule {
  /** Stream of [[lineItemsIn]] with input direction. */
  val lineItemsInStream = LineItem_stream().flip
  /** IO of [[lineItemsInStream]] with input direction. */
  val lineItemsIn = lineItemsInStream.toPhysical
    val L_CommentIn = lineItemsInStream.el.L_Comment.toPhysical
    val L_CommitDateIn = lineItemsInStream.el.L_CommitDate.toPhysical
    val L_LineStatusIn = lineItemsInStream.el.L_LineStatus.toPhysical
    val L_ReceiptDateIn = lineItemsInStream.el.L_ReceiptDate.toPhysical
    val L_ReturnFlagIn = lineItemsInStream.el.L_ReturnFlag.toPhysical
    val L_ShipDateIn = lineItemsInStream.el.L_ShipDate.toPhysical
    val L_ShipInstructIn = lineItemsInStream.el.L_ShipInstruct.toPhysical
    val L_ShipModeIn = lineItemsInStream.el.L_ShipMode.toPhysical

  /** Stream of [[lineItemsOut]] with output direction. */
  val lineItemsOutStream = LineItem_stream()
  /** IO of [[lineItemsOutStream]] with output direction. */
  val lineItemsOut = lineItemsOutStream.toPhysical
    val L_CommentOut = lineItemsOutStream.el.L_Comment.toPhysical
    val L_CommitDateOut = lineItemsOutStream.el.L_CommitDate.toPhysical
    val L_LineStatusOut = lineItemsOutStream.el.L_LineStatus.toPhysical
    val L_ReceiptDateOut = lineItemsOutStream.el.L_ReceiptDate.toPhysical
    val L_ReturnFlagOut = lineItemsOutStream.el.L_ReturnFlag.toPhysical
    val L_ShipDateOut = lineItemsOutStream.el.L_ShipDate.toPhysical
    val L_ShipInstructOut = lineItemsOutStream.el.L_ShipInstruct.toPhysical
    val L_ShipModeOut = lineItemsOutStream.el.L_ShipMode.toPhysical

  /** Stream of [[partsIn]] with input direction. */
  val partsInStream = Part_stream().flip
  /** IO of [[partsInStream]] with input direction. */
  val partsIn = partsInStream.toPhysical
    val P_BrandIn = partsInStream.el.P_Brand.toPhysical
    val P_CommentIn = partsInStream.el.P_Comment.toPhysical
    val P_ContainerIn = partsInStream.el.P_Container.toPhysical
    val P_MfgrIn = partsInStream.el.P_Mfgr.toPhysical
    val P_NameIn = partsInStream.el.P_Name.toPhysical
    val P_TypeIn = partsInStream.el.P_Type.toPhysical

  /** Stream of [[partsOut]] with output direction. */
  val partsOutStream = Part_stream()
  /** IO of [[partsOutStream]] with output direction. */
  val partsOut = partsOutStream.toPhysical
    val P_BrandOut = partsOutStream.el.P_Brand.toPhysical
    val P_CommentOut = partsOutStream.el.P_Comment.toPhysical
    val P_ContainerOut = partsOutStream.el.P_Container.toPhysical
    val P_MfgrOut = partsOutStream.el.P_Mfgr.toPhysical
```

```
    val P_NameOut = partsOutStream.el.P_Name.toPhysical
    val P_TypeOut = partsOutStream.el.P_Type.toPhysical
}

/**
 * Streamlet, defined in pack0.
 */
class Tphc19_Top_interface extends TydiModule {
  /** Stream of [[lineItemsIn]] with input direction. */
  val lineItemsInStream = LineItem_stream().flip
  /** IO of [[lineItemsInStream]] with input direction. */
    val lineItemsIn = lineItemsInStream.toPhysical
    val L_CommentIn = lineItemsInStream.el.L_Comment.toPhysical
    val L_CommitDateIn = lineItemsInStream.el.L_CommitDate.toPhysical
    val L_LineStatusIn = lineItemsInStream.el.L_LineStatus.toPhysical
    val L_ReceiptDateIn = lineItemsInStream.el.L_ReceiptDate.toPhysical
    val L_ReturnFlagIn = lineItemsInStream.el.L_ReturnFlag.toPhysical
    val L_ShipDateIn = lineItemsInStream.el.L_ShipDate.toPhysical
    val L_ShipInstructIn = lineItemsInStream.el.L_ShipInstruct.toPhysical
    val L_ShipModeIn = lineItemsInStream.el.L_ShipMode.toPhysical

  /** Stream of [[partsIn]] with input direction. */
  val partsInStream = Part_stream().flip
  /** IO of [[partsInStream]] with input direction. */
  val partsIn = partsInStream.toPhysical
    val P_BrandIn = partsInStream.el.P_Brand.toPhysical
    val P_CommentIn = partsInStream.el.P_Comment.toPhysical
    val P_ContainerIn = partsInStream.el.P_Container.toPhysical
    val P_MfgrIn = partsInStream.el.P_Mfgr.toPhysical
    val P_NameIn = partsInStream.el.P_Name.toPhysical
    val P_TypeIn = partsInStream.el.P_Type.toPhysical

  /** Stream of [[revenueOut]] with output direction. */
  val revenueOutStream = RevenueStream()
  /** IO of [[revenueOutStream]] with output direction. */
  val revenueOut = revenueOutStream.toPhysical
}
```

Listing B.3: Unfilled implementation stubs and Verilog code generation; Top module is generated

```
package tpch.IO

import chisel3._
import circt.stage.ChiselStage.{emitCHIRRTL, emitSystemVerilog}
import tpch.{Tphc19_LineItem_Part_Passthrough_interface, Tphc19_Top_interface}
import tydi_lib._

/**
 * Implementation, defined in pack0.
 */
class Tphc19_Join extends Tphc19_LineItem_Part_Passthrough_interface {
  lineItemsInStream := DontCare
  partsInStream := DontCare
  lineItemsOutStream := lineItemsInStream
  partsOutStream := partsInStream
}

/**
 * Implementation, defined in pack0.
 */
class Tphc19_Filter extends Tphc19_LineItem_Part_Passthrough_interface {
  lineItemsInStream := DontCare
  partsInStream := DontCare
  lineItemsOutStream := lineItemsInStream
  partsOutStream := partsInStream
}

/**
 * Implementation, defined in pack0.
 */
class Tphc19_Reducer extends Tphc19_Top_interface {
  partsInStream := DontCare
  lineItemsInStream := DontCare
  revenueOutStream := DontCare
}

/**
 * Implementation, defined in pack0.
 */
class Tphc19_Top extends Tphc19_Top_interface {
  lineItemsInStream := DontCare
  partsInStream := DontCare
```

```
43    revenueOutStream := DontCare
44
45    // Modules
46    val join = Module(new Tphc19_Join)
47    val filter = Module(new Tphc19_Filter)
48    val reducer = Module(new Tphc19_Reducer)
49
50    // Connections
51    join.lineItemsIn := lineItemsIn
52    join.L_CommentIn := L_CommentIn
53    join.L_CommitDateIn := L_CommitDateIn
54    join.L_LineStatusIn := L_LineStatusIn
55    join.L_ReceiptDateIn := L_ReceiptDateIn
56    join.L_ReturnFlagIn := L_ReturnFlagIn
57    join.L_ShipDateIn := L_ShipDateIn
58    join.L_ShipInstructIn := L_ShipInstructIn
59    join.L_ShipModeIn := L_ShipModeIn
60    join.partsIn := partsIn
61    join.P_BrandIn := P_BrandIn
62    join.P_CommentIn := P_CommentIn
63    join.P_ContainerIn := P_ContainerIn
64    join.P_MfgrIn := P_MfgrIn
65    join.P_NameIn := P_NameIn
66    join.P_TypeIn := P_TypeIn
67
68    filter.lineItemsIn := join.lineItemsOut
69    filter.L_CommentIn := join.L_CommentOut
70    filter.L_CommitDateIn := join.L_CommitDateOut
71    filter.L_LineStatusIn := join.L_LineStatusOut
72    filter.L_ReceiptDateIn := join.L_ReceiptDateOut
73    filter.L_ReturnFlagIn := join.L_ReturnFlagOut
74    filter.L_ShipDateIn := join.L_ShipDateOut
75    filter.L_ShipInstructIn := join.L_ShipInstructOut
76    filter.L_ShipModeIn := join.L_ShipModeOut
77    filter.partsIn := join.partsOut
78    filter.P_BrandIn := join.P_BrandOut
79    filter.P_CommentIn := join.P_CommentOut
80    filter.P_ContainerIn := join.P_ContainerOut
81    filter.P_MfgrIn := join.P_MfgrOut
82    filter.P_NameIn := join.P_NameOut
83    filter.P_TypeIn := join.P_TypeOut
84
85    reducer.lineItemsIn := filter.lineItemsOut
86    reducer.L_CommentIn := filter.L_CommentOut
87    reducer.L_CommitDateIn := filter.L_CommitDateOut
88    reducer.L_LineStatusIn := filter.L_LineStatusOut
89    reducer.L_ReceiptDateIn := filter.L_ReceiptDateOut
90    reducer.L_ReturnFlagIn := filter.L_ReturnFlagOut
91    reducer.L_ShipDateIn := filter.L_ShipDateOut
92    reducer.L_ShipInstructIn := filter.L_ShipInstructOut
93    reducer.L_ShipModeIn := filter.L_ShipModeOut
94    reducer.partsIn := filter.partsOut
95    reducer.P_BrandIn := filter.P_BrandOut
96    reducer.P_CommentIn := filter.P_CommentOut
97    reducer.P_ContainerIn := filter.P_ContainerOut
98    reducer.P_MfgrIn := filter.P_MfgrOut
99    reducer.P_NameIn := filter.P_NameOut
100   reducer.P_TypeIn := filter.P_TypeOut
101
102   revenueOut := reducer.revenueOut
103 }
104
105 object Tpch19 extends App {
106   println("TPCH 19 implementation stubs")
107
108   private val FIRRTL: String = emitCHIRRTL(new Tphc19_Top())
109   println(s"FIRRTL code = ${FIRRTL.split('\n').length} lines")
110
111   // These lines generate the Verilog output
112   private val verilogCodeNormal: String = emitSystemVerilog(new Tphc19_Top(), firtoolOpts =
          firNormalOpts)
113   println(s"Verilog normal options = ${verilogCodeNormal.split('\n').length} lines")
114
115   private val verilogCodeUnoptimized: String = emitSystemVerilog(new Tphc19_Top(), firtoolOpts =
          firNoOptimizationOpts)
116   println(s"Verilog no optimization options = ${verilogCodeUnoptimized.split('\n').length} lines")
117
118   println("Done")
119 }
```

Bibliography

[1] Arm Limited. *AMBA® AXI Protocol Specification*. Mar. 2023. URL: https://developer.arm.com/documentation/ihi0022/j/ (visited on 08/31/2023).

[2] Arm Limited. *AMBA® AXI-Stream Protocol Specification*. Apr. 2021. URL: https://developer.arm.com/documentation/ihi0051/b (visited on 08/31/2023).

[3] Joshua Auerbach et al. "Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. New York, NY, USA: Association for Computing Machinery, Oct. 2010, pp. 89–108. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869469. URL: http://doi.org/10.1145/1869459.1869469 (visited on 04/29/2022).

[4] C. P. R. Baaij et al. "CλaSH: Structural Descriptions of Synchronous Hardware using Haskell". In: *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*. 13th EUROMICRO Conference on Digital System Design, DSD 2010: Architectures, Methods and Tools. IEEE, Sept. 2010, pp. 714–721. DOI: 10.1109/DSD.2010.21. URL: https://research.utwente.nl/en/publications/c%CE%BBash-structural-descriptions-of-synchronous-hardware-using-haske (visited on 09/28/2023).

[5] Jonathan Bachrach et al. "Chisel: Constructing hardware in a Scala embedded language". In: *DAC Design Automation Conference 2012*. DAC Design Automation Conference 2012. ISSN: 0738-100X. June 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[6] Matthijs Brobbel, Johan Peltenburg, and Jeroen van Straten. *Tydi*. original-date: 2020-01-03T17:52:17Z. Apr. 17, 2023. URL: https://github.com/abs-tudelft/tydi (visited on 08/24/2023).

[7] Casper Cromjongh et al. "Enabling Collaborative and Interface-Driven Data-Streaming Accelerator Design with Tydi-Chisel". In: NorCAS 2023. Aalborg, Nov. 1, 2023, p. 7.

[8] Marios Fragkoulis et al. *A Survey on the Evolution of Stream Processing Systems*. Issue: arXiv:2008.00842. Aug. 2020. DOI: 10.48550/arXiv.2008.00842. arXiv: 2008.00842. URL: http://arxiv.org/abs/2008.00842 (visited on 05/18/2022).

[9] Google. *XLS: Accelerated HW Synthesis*. URL: https://google.github.io/xls/ (visited on 09/28/2023).

[10] Akos Hadnagy, Matthijs Brobbel, and Jasper Haenen. *Tydi-json*. original-date: 2022-11-21T12:59:39Z. Nov. 21, 2022. URL: https://github.com/jhaenen/tydi-json (visited on 08/24/2023).

[11] Jasper Haenen. "JSON-TIL: A tool for generating/reducing boilerplate when creating and composing streaming JSON dataflow accelerators using Tydi interfaces". In: (Jan. 2023). URL: https://github.com/jhaenen/JSON_hierachy/blob/master/TIL_JSON.pdf.

[12] John L. Hennessy and David A. Patterson. "A new golden age for computer architecture". In: *Communications of the ACM* 62.2 (Jan. 28, 2019), pp. 48–60. ISSN: 0001-0782. DOI: 10.1145/3282307. URL: https://dl.acm.org/doi/10.1145/3282307 (visited on 08/25/2023).

[13] Amir Hormati et al. "Optimus: Efficient Realization of Streaming Applications on FPGAs". In: *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '08. New York, NY, USA: Association for Computing Machinery, Oct. 2008, pp. 41–50. ISBN: 978-1-60558-469-0. DOI: 10.1145/1450095.1450105. URL: http://doi.org/10.1145/1450095.1450105 (visited on 04/29/2022).

[14] Intel Corporation. *5. Avalon® Streaming Interfaces*. Publication Title: Intel. Jan. 2022. URL: https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/streaming-interfaces.html (visited on 05/18/2022).

[15] Haruna Isah et al. “A Survey of Distributed Data Stream Processing Frameworks”. In: *IEEE Access* 7 (2019), pp. 154300–154316. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2946884.

[16] Adam Izraelevitz et al. “Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). Irvine, CA: IEEE, Nov. 2017, pp. 209–216. ISBN: 978-1-5386-3093-8. DOI: 10.1109/ICCAD.2017.8203780. URL: http://ieeexplore.ieee.org/document/8203780/ (visited on 08/07/2023).

[17] Florent Kermarrec et al. “LiteX: an open-source SoC builder and library based on Migen Python DSL”. In: Open Source Design Automation (OSDA). Florence, Mar. 29, 2019.

[18] David Koeplinger et al. “Spatial: a language and compiler for application accelerators”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’18: ACM SIGPLAN Conference on Programming Language Design and Implementation. Philadelphia PA USA: ACM, June 11, 2018, pp. 296–311. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192379. URL: https://dl.acm.org/doi/10.1145/3192366.3192379 (visited on 09/28/2023).

[19] Max Korbel. “Rapid Open Hardware Development Framework”. In: *Workshop on Open-Source EDA Technology (WOSET)*. Workshop on Open-Source EDA Technology (WOSET). Nov. 2022.

[20] Yi-Hsiang Lai et al. “HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’19: The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. Seaside CA USA: ACM, Feb. 20, 2019, pp. 242–251. ISBN: 978-1-4503-6137-8. DOI: 10.1145/3289602.3293910. URL: https://dl.acm.org/doi/10.1145/3289602.3293910 (visited on 09/28/2023).

[21] Jean-Christophe Le Lann, Hannah Badier, and Florent Kermarrec. “Towards a Hardware DSL Ecosystem : RubyRTL and Friends”. In: *Proc.\ Design, Automation and Test in Europe Conference (DATE ’20). Open Source Design Automation Workshop.* Grenoble, France: IEEE, Mar. 2020.

[22] LLVM Community. *”CIRCT” / Circuit IR Compilers and Tools*. May 2022. URL: https://github.com/llvm/circt (visited on 05/18/2022).

[23] Derek Lockhart, Gary Zibrat, and Christopher Batten. “PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research”. In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. ISSN: 2379-3155. Dec. 2014, pp. 280–292. DOI: 10.1109/MICRO.2014.50. URL: https://ieeexplore.ieee.org/abstract/document/7011395 (visited on 09/28/2023).

[24] Evangelos Mageiropoulos et al. “Using hls4ml to Map Convolutional Neural Networks on Interconnected FPGA Devices”. In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). ISSN: 2576-2621. May 2021, pp. 277–277. DOI: 10.1109/FCCM51124.2021.00062.

[25] Rachit Nigam et al. “Predictable accelerator design with time-sensitive affine types”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’20: 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation. London UK: ACM, June 11, 2020, pp. 393–407. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385974. URL: https://dl.acm.org/doi/10.1145/3385412.3385974 (visited on 09/28/2023).

[26] Johan Peltenburg et al. “Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow”. In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019 29th International Conference on Field Programmable Logic and Applications (FPL). ISSN: 1946-1488. Sept. 2019, pp. 270–277. DOI: 10.1109/FPL.2019.00051.

[27] Johan Peltenburg et al. “Supporting Columnar In-memory Formats on FPGA: The Hardware Design of Fletcher for Apache Arrow”. In: *Applied Reconfigurable Computing*. Ed. by Christian Hochberger et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 32–47. ISBN: 978-3-030-17227-5. DOI: 10.1007/978-3-030-17227-5_3.

[28] Johan Peltenburg et al. “Tydi: An Open Specification for Complex Data Structures Over Hardware Streams”. In: *IEEE Micro* 40.4 (July 1, 2020), pp. 120–130. ISSN: 0272-1732, 1937-4143. DOI: 10.1109/MM.2020.2996373. URL: https://ieeexplore.ieee.org/document/9098092/ (visited on 08/07/2023).

[29] Julian Pontes et al. “SCAFFI: An Intrachip FPGA Asynchronous Interface Based on Hard Macros”. In: *2007 25th International Conference on Computer Design*. Lake Tahoe, CA, USA: IEEE, Oct. 2007, pp. 541–546. ISBN: 978-1-4244-1257-0. DOI: 10.1109/ICCD.2007.4601950. URL: http://ieeexplore.ieee.org/document/4601950/ (visited on 12/17/2021).

[30] M A Reukers. “A Toolchain for Streaming Dataflow Accelerator Designs for Big Data Analytics”. In: ().

[31] Matthijs A Reukers et al. “An Intermediate Representation for Composable Typed Streaming Dataflow Designs”. In: *Joint Proceedings of Workshops at the 49th International Conference on Very Large Data Bases (VLDB 2023)* 3462 (2023).

[32] Matthijs A. Reukers. *TIL -> VHDL*. original-date: 2021-12-08T08:57:23Z. Apr. 24, 2023. URL: https://github.com/matthijsr/til-vhdl (visited on 08/24/2023).

[33] Adrian Sampson. *From Hardware Description Languages to Accelerator Design Languages*. SIGARCH. June 29, 2021. URL: https://www.sigarch.org/hdl-to-adl/ (visited on 08/07/2023).

[34] Matthias J. Sax. “Apache Kafka”. In: *Encyclopedia of Big Data Technologies*. Ed. by Sherif Sakr and Albert Zomaya. Cham: Springer International Publishing, 2018, pp. 1–8. ISBN: 978-3-319-63962-8. DOI: 10.1007/978-3-319-63962-8_196-1. URL: https://doi.org/10.1007/978-3-319-63962-8_196-1 (visited on 05/18/2022).

[35] Fabian Schuiki et al. “LLHD: A Multi-level Intermediate Representation for Hardware Description Languages”. In: *arXiv:2004.03494 [cs]* (Apr. 2020). arXiv: 2004.03494. URL: http://arxiv.org/abs/2004.03494 (visited on 04/29/2022).

[36] Jeroen van Straten and Jasper Haenen. *vhdre: a VHDL regex matcher generator*. original-date: 2022-12-05T12:35:47Z. Dec. 20, 2022. URL: https://github.com/jhaenen/vhdre (visited on 08/24/2023).

[37] Muhammad Usman Tariq and Fahad Saeed. “Parallel Sampling-Pipeline for Indefinite Stream of Heterogeneous Graphs using OpenCL for FPGAs”. In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018 IEEE International Conference on Big Data (Big Data). Dec. 2018, pp. 4752–4761. DOI: 10.1109/BigData.2018.8621979.

[38] William Thies, Michal Karczmarek, and Saman Amarasinghe. “StreamIt: A Language for Streaming Applications”. In: *International Conference on Compiler Construction*. Grenoble, France, Apr. 2002. URL: http://groups.csail.mit.edu/commit/papers/02/streamit-cc.pdf.

[39] James Thomas, Pat Hanrahan, and Matei Zaharia. “Fleet: A Framework for Massively Parallel Streaming on FPGAs”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20: Architectural Support for Programming Languages and Operating Systems. Lausanne Switzerland: ACM, Mar. 9, 2020, pp. 639–651. ISBN: 978-1-4503-7102-5. DOI: 10.1145/3373376.3378495. URL: https://dl.acm.org/doi/10.1145/3373376.3378495 (visited on 08/09/2023).

[40] Yongding Tian. *Tydi-lang-2*. original-date: 2023-04-05T14:38:14Z. May 8, 2023. URL: https://github.com/twoentartian/tydi-lang-2 (visited on 08/24/2023).

[41] Yongding Tian. “Tydi-lang: a language for typed streaming hardware”. master thesis. Delft: Delf University of Technology, July 5, 2022. URL: http://resolver.tudelft.nl/uuid:7ac8d6cd-13eb-4d81-972e-cc23d79b2f22.

[42] Yongding Tian et al. “Tydi-lang: A Language for Typed Streaming Hardware”. In: *2023 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 2023 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC). San Francisco, 2023, p. 8. arXiv: 2212.06259[cs]. URL: http://arxiv.org/abs/2212.06259 (visited on 08/07/2023).

[43] Transaction Processing Performance Council (TPC). *TPC Benchmark™ H*. Apr. 28, 2022. URL: https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp (visited on 09/28/2023).

[44] Lenny Truong and Pat Hanrahan. “A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity”. In: (2019). In collab. with Michael Wagner, 21 pages. DOI: 10.4230/LIPICS.SNAPL.2019.7. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10550/ (visited on 08/25/2023).