Hardware-Accelerator Design by Composition
Dataflow Component Interfaces with Tydi-Chisel

Cromjongh, Casper; Tian, Yongding; Hofstee, H. Peter; Al-Ars, Zaid

DOI
10.1109/TVLSI.2024.3461330

Publication date
2024

Document Version
Final published version

Published in
IEEE Transactions on Very Large Scale Integration (VLSI) Systems

Hardware-Accelerator Design by Composition: Dataflow Component Interfaces With Tydi–Chisel

Casper Cromjongh, Yongding Tian, H. Peter Hofstee, *Member, IEEE*, and Zaid Al-Ars, *Member, IEEE*

***Abstract*— As dedicated hardware is becoming more prevalent in accelerating complex applications, methods are needed to enable easy integration of multiple hardware components into a single accelerator system. However, this vision of composable hardware is hindered by the lack of standards for interfaces that allow such components to communicate. To address this challenge, the Tydi standard was proposed to facilitate the representation of streaming data in digital circuits, notably providing interface specifications of composite and variable-length data structures. At the same time, constructing hardware in a Scala embedded language (Chisel) provides a suitable environment for deploying Tydi-centric components due to its abstraction level and customizability. This article introduces Tydi–Chisel, a library that integrates the Tydi standard within Chisel, along with a toolchain and methodology for designing data-streaming accelerators. This toolchain reduces the effort needed to design streaming hardware accelerators by raising the abstraction level for streams and module interfaces, hereby avoiding writing boilerplate code, and allows for easy integration of accelerator components from different designers. This is demonstrated through an example project incorporating various scenarios where the interface-related declaration is reduced by 6–14 times. Tydi–Chisel project repository is available at https://github.com/abs-tudelft/Tydi-Chisel.**

***Index Terms*— Big data acceleration, data types, HW design, streaming interfaces, testing.**

I. Introduction

In THE past decade, hardware accelerators [such as GPUs and field-programmable gate arrays (FPGAs)] have emerged as next-generation alternative computing platforms to meet the ever-increasing computing demands in various compute-intensive application domains, especially in fields, such as machine learning and big data analytics [2], [3]. This resulted in the rise of heterogeneous computing platforms, where multiple accelerators can cooperate to address the computationally intensive parts of an application. However, there is a big difference in the development effort required for these accelerators, where hardware development for FPGAs stands out for being exceptionally difficult to design, optimize, and debug.

Tool frameworks do exist that attempt to reduce this challenge of high field-programmable gate array (FPGA) design effort, such as high-level synthesis (HLS), OpenCL [4], and HLS4ML [5]. However, the continued increase in the abstraction level that software developers use to program their applications limits the effectiveness of such tool frameworks, especially in application domains, such as big data analytics [6], where developers can typically write a few lines of SQL to execute a query, whereas translating the same query to FPGAs requires thousands of lines of hardware description code. Sampson [7] accentuates this disparity, advocating for a transition from hardware description language (HDL) to accelerator design language (ADL). Constructing hardware in a Scala embedded language (Chisel) has emerged as a promising way to achieve this transition.

Chisel [8] aims at lowering design complexity by providing designers with more powerful design tools. These tools empower designers to craft highly parameterized generator components, seamlessly manipulate complex aggregate signals, and utilize high-level programming paradigms. Since Chisel is intended for general hardware design, however, it is possible to further refine the design process through a domain-specific strategy, particularly for data-streaming accelerators.

A central challenge in designing data-streaming accelerators is related to the transfer of structured and dynamically sized data between components in a flexible manner. However, there is no open-source hardware development framework that is able to address this challenge, including classical HDLs and contemporary ones (e.g., Cλash, Chisel, and Spatial). These HDLs do support compound types, but lack support for dynamically sized aggregate types for data streaming [9]. The languages allow expressing composite signals in space, but a strategy for transferring variable-length data, where the time domain is incorporated, is left to the designer.

To address these issues, the idea for typed dataflow interface (Tydi) was proposed [9]. Fig. 1 aims to illustrate the advantages Tydi offers by showing a metaphorical representation of a raw data stream, handshaked stream, and Tydi stream.

Without a common interface standard, such as Tydi, designers are often left designing their own communication protocol. While, in simple cases, this is often trivial, design complexity frequently increases during development and optimization. With increasing complexity, communication solutions will

Received 8 April 2024; revised 31 July 2024; accepted 22 August 2024. Date of publication 4 October 2024; date of current version 6 December 2024. This work was supported by the Eureka Xecs Project TASTI under Grant 2022005. An earlier version of this paper was presented at the NorCAS 2023 [DOI: 10.1109/NorCAS58970.2023.10305451]. *(Corresponding author: Casper Cromjongh.)*

Casper Cromjongh, Yongding Tian, and Zaid Al-Ars are with the Department of Electrical Engineering, Mathematics and Computer Science (EEMCS), Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: C.Cromjongh@tudelft.nl).

H. Peter Hofstee is with the Department of Electrical Engineering, Mathematics and Computer Science (EEMCS), Delft University of Technology, 2628 CD Delft, The Netherlands, and also with IBM Infrastructure, Austin, TX 78758 USA.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2024.3461330.

Digital Object Identifier 10.1109/TVLSI.2024.3461330

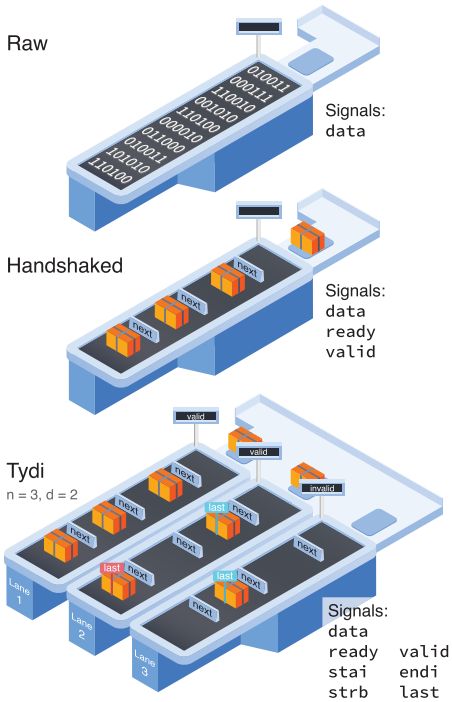

Fig. 1. Comparison of stream types represented as checkout conveyor belts. The Tydi stream has $n = 3$ lanes and a dimensionality of $d = 2$. `stai`, `endi`, and `strb` relate to data-lane validity. `last` transfers dimensionality information.

become more specific and divergent. When adopting IP or working between projects, this creates a lot of unnecessary overhead in specification and conversion. Debugging and interpreting the communication flow easily become very difficult. Standards and tooling can help alleviate hardship in design choices, implementation effort, and debugging and interpretation. Tydi aims to be a standard that can offer this. In this article, 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.

This article is organized as follows. Section II provides an overview of related work on streaming design and accelerator design. In Section III, a background on Tydi and Chisel is given. In Section IV, a Tydi-driven design workflow is explained with a minimal example use case. In Section V, an example is examined resembling a real-world data-processing system. With this example, various integration methods are highlighted. Additional Tydi–Chisel features are covered in Section VI, notably the stream-complexity converter component that allows connecting components with differing stream protocol complexity levels. A discussion about the role of Tydi-lang and Tydi–Chisel and the utility of language features is presented in Section VII. Section VIII ends with the conclusion.

II. Related Work

A stream-based processing system generally consists of interconnected computation kernels/modules that communicate through streaming channels. Such a system, therefore, has three components. There is the compute implementation, the communication between modules, and the composition/hierarchy of the modules. The communication model can be subdivided into the protocol and the payload data structure.

From the software side, 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 [10], [11], [12], [13] on multithreaded CPUs and GPUs. Thies et al. [14] proposed the holistic language StreamIt, crafted for universal streaming logic. With software's flexibility, specifying implementation, hierarchy, and communication with complex, variable-length data types is straightforward, even in lower level languages, such as C.

For hardware design, the landscape is more fractured, and tools generally focus on one or two of the mentioned aspects. Neuendorffer and Vissers [15] provide an overview of differences between streaming in hardware and software. In hardware, various studies have demonstrated the performance advantage of streaming dataflow systems, especially in signal-processing tasks. Some of these [16], [17], [18], [19] have focused on tools for development of such FPGA-oriented dataflow systems. Thomas et al. [19] also use Chisel as the language for module implementation. These dataflow systems focus on developing implementations of stream-processing modules, rather than streams from an interface perspective.

Hormati et al. [20] created a framework that does both with Optimus [14], utilizing the aforementioned StreamIt language. They define macro- and micro-functionality: "macro-functional concerns address how components (modules) are assembled to implement larger more complex applications. Micro-functional issues deal with synthesis issues of the module internals." Thereby, Optimus combines module implementation with the module composition that Tydi promotes.

The interfaces that are used in hardware generally use some variation of a ready-valid handshaked connection protocol, with bit stream data. Such a handshaked connection mechanism is built into Chisel in the form of `DecoupledIO`. Several industry standards for handshaked streaming connections have been established [21], [22], [23]. These standards help address the challenges posed by component interface compatibility, but are rather simplistic, often requiring a custom adaptation that negates the adaptation effort advantage. Frameworks, such as the dsptools [24] and Clash-Protocols [25] libraries, can help by allowing one to abstract the underlying signaling interface away in Chisel and Clash, respectively.

For interaction with host software, other tools have been designed that help with communicating with the host in a streaming manner [26], [27], [28]. Among these is Fletcher, which is mentioned in Section III-A.

Yet, these standards, frameworks, and studies primarily address simple data transfer at the bit stream level, commonly not supporting complex data structures with inner lists, which is required by the complexity and dynamic nature of the data from the software side, where it comes naturally.

In conclusion, though efforts have been made to improve the ease of implementation and usage of simple stream communication as well as the composition of modules, no general

purpose standard or tool exists that focuses on the transfer of complex data structures (with variable-length sequences and nesting). Though Tydi and Tydi–Chisel also do not cover all aspects of a streaming system, many tools exist that allow a designer to create an implementation, catering to various contexts and needs. Tydi was, therefore, conceived as a language-agnostic standard, allowing designers to choose the implementation tool themselves or work with the existing IP. Tydi leads to interface-driven design, a successful concept in software development.

III. Background

A. Fletcher and `vhlib`

The Fletcher project [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 [29]. In the hardware design, Fletcher uses `vhlib` [30] streams, which can be seen as a predecessor to Tydi streams, approximately equal to Tydi streams with protocol complexity $C \leq 5$. Despite Fletcher's capabilities in generating components for memory data access, the challenge of designing a system to transfer more complexly structured data remained, akin to the in-memory data structures that tend to be both complex and dynamic. This gave rise to the development of Tydi.

B. Tydi Specification

The Tydi specification was first introduced in [9]. 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 [31]. Based on this refined version, Tian et al. [32] proposed a high-level HDL to raise the abstraction level of typed streaming hardware and reduce the design effort for hardware designer. In addition, Reukers et al. [33] developed an intermediate representation tailored for hardware circuit design using the Tydi framework, accompanied by a compiler for VHDL translation. The terms utilized within the Tydi intermediate representation and their meanings are summarized in Table I. For a broader perspective, a comparative study between Tydi and prevalent protocols, such as AXI and Avalon, can be consulted in [9, Table IV].

The basic data types used in Tydi intermediate representation are *Null*, *Bits*, *Group*, and *Union*. The *Stream* type is a wrapper of basic data types, adding the streaming properties, such as *complexity*, *dimension*, and *throughput*. These are explained next. *Group*, *Union*, and *Stream* support nesting of other types to create arbitrary data structures.

1) *Complexity:* Denotes the intricacy of the physical protocol. In other words, how orderly or flexibly the data elements can be sent over the stream bus. The present Tydi specification delineates eight distinct complexity levels, ranging from 1 to 8. A lower complexity value imposes more rules for stricter transfer continuity. This implies more straightforward data reception, yet, correspondingly, the source component may necessitate increased complexity to guarantee the availability of the data in the required form. Fig. 2 visually represents the protocol of complexities at levels 1 and 8. Because lower complexities are more restrictive and higher complexities more flexible, a source port with a lower complexity is able to connect to a sink port of the same or higher complexity.

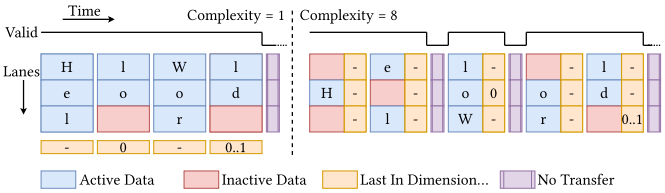

Fig. 2. Stream-complexity property [33].

2) *Dimension:* Indicates the number of dimensions 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-D 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)`.
3) *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, such as manual protocol conversion.

C. Tydi Ecosystem

Several projects have emerged that utilize Tydi-related methodologies. Among these, Tydi-JSON [34] 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 [33] and [34], JSON-TIL [35] 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. In addition, 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.

D. Chisel

Chisel [8] is an open-source hardware construction language developed to facilitate the design of highly parameterized

TABLE I
TYDI TERMS AND CORRESPONDING DEFINITIONS

Term	Type	Software equivalent	Chisel equivalent	Definition
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 width would be the sum of child elements.
Union	Tydi logical type	Union	Bundle & tag	A union of several other logical types. The active field can be selected with the tag. This field can also be a stream.
Stream	Tydi logical type	Bus	–	Represents a stream of a Tydi logical type. The stream can also specify the data dimension, protocol complexity, hardware synchronicity, and throughput.
Streamlet	Tydi hardware element	Interface	Trait with IO definitions	Represents the port map of a component. This term is almost the same as the "entity" term in VHDL.
Impl	Tydi hardware element	Class with functionality	Module	"impl" is the abbreviation of "implementation", representing the inner structure of a component.

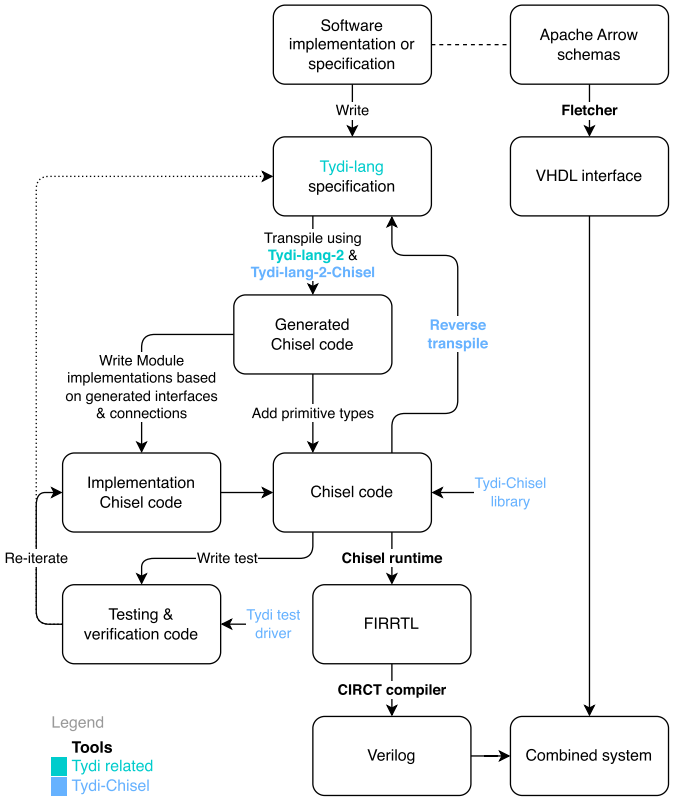

Fig. 3. Tydi toolchain components.

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.

IV. CHISEL EXTENSION WITH TYDI

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 an example.

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

Listing 1. Example spark code.

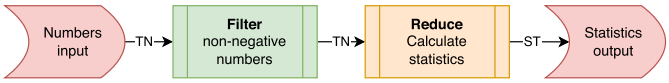

Fig. 4. Number pipeline structure. TN: timestamped number. ST: statistics.

A. Conceived Design Pipeline

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 the following.

1) Idea/software definition.
2) Write interface specifications in Tydi-lang code.
3) Describe additional communication specification.
4) Transpile with Tydi-lang-2 and 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 Fig. 3.

B. Number Pipeline Example

To illustrate the aforementioned pipeline, we work with a minimal example. This example is purposefully kept simple. Section V investigates a more advanced example that better shows the advantage of Tydi's ecosystem. In this section's example, we take in a stream of numbers with timestamps attached. This stream first gets filtered on value ≥ 0 and then reduced to statistics: min value, max value, sum of values, and average. The block schedule for this system is given in Fig. 4. In Apache Spark, one could execute this process as in Listing 1.

```
#### 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 NumsFilter_interface {
    std_out : pack0.NumberGroup_stream out;
    std_in : pack0.NumberGroup_stream in;
}

impl NonNegativeFilter of NumsFilter_interface {}

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

impl Reducer of NumsToStats_interface {}

impl PipelineExample of NumsToStats_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;
}
```

Listing 2. Example Tydi-lang source code.

C. Tydi-Lang Specification and Transpiling

As explained before, Tydi-lang was designed to close the gap between software and hardware design. Listing 2 shows Tydi-lang code for our example. Connections between components can be specified within Tydi-lang, and components that require practical implementations are left empty. Tydi–Chisel code for this specification can be obtained by first running *Tydi-lang*, obtaining a JSON description that is used to generate the Chisel code with *Tydi-lang-2–Chisel*.

A snippet of the generated Chisel code is given in Listing 3. The code shows the transformed *Element* datatypes, interface specifications (from streamlets), and implementation skeletons. Since Tydi focuses on the structure of the data, not 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. A simple implementation of the filter function is given in Listing 4, where the data lane is turned off for filtered items. The cost of this simple implementation is that the output stream complexity is raised to $C \geq 7$. The next component must do work to realign the items when the sequence is required.

Tydi–Chisel's library was designed for ease of use both in new projects and in converting existing code; see Section VII for a discussion about the library's role. Next to syntax, care is given to the implementation of Tydi–Chisel's components. This implementation in Chisel consists of a few parts.

```
object MyTypes {
    /** Bit(64) type, defined in package [[Pack0]] */
    def UInt_64_t: UInt = UInt(64.W)
    assert(this.UInt_64_t.getWidth == 64)

    /** Bit(64) type, defined in package [[Pack0]] */
    def SInt_64_t: UInt = UInt(64.W)
    assert(this.SInt_64_t.getWidth == 64)
}

/** Group element, defined in package [[Pack0]]. */
class NumberGroup extends Group {
    val time = MyTypes.UInt_64_t
    val value = MyTypes.SInt_64_t
}

/** Group element, defined in package [[Pack0]]. */
class Stats extends Group {
    val average = MyTypes.UInt_64_t
    val max = MyTypes.UInt_64_t
    val min = MyTypes.UInt_64_t
    val sum = MyTypes.UInt_64_t
}

/** Stream, defined in package [[Pack0]]. */
class NumberGroupStream extends PhysicalStreamDetailed(e=
     new NumberGroup, n=1, d=1, c=1, r=false, u=Null())

object NumberGroupStream {
    def apply(): NumberGroupStream = Wire(new
         NumberGroupStream())
}
// ... other stream definitions

/** Streamlet, defined in package [[Pack1]]. */
class NumsFilter_interface extends TydiModule {
    /** Stream of [[in]] with input direction. */
    val inStream = StatsStream().flip
    /** IO of [[inStream]] with input direction. */
    val in = inStream.toPhysical
    /** Stream of [[out]] with output direction. */
    val outStream = NumberGroupStream()
    /** IO of [[outStream]] with output direction. */
    val out = outStream.toPhysical
}

/** Streamlet, defined in package [[Pack1]]. */
class NumsToStats_interface extends TydiModule {
    // ... code for NumberGroup in, Stats out
}
// ... other interface and implementation definitions

/** Implementation, defined in package [[Pack1]]. */
class NonNegativeFilter extends NumsFilter_interface {}

/** Implementation, defined in package [[Pack1]]. */
class PipelineExample extends NumsToStats_interface {
    // Modules
    val filter = Module(new NonNegativeFilter)
    val reducer = Module(new Reducer)

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

Listing 3. Chisel output code from Tydi-lang transpilation.

```
class NonNegativeFilter extends NonNegativeFilter_interface
      {
  outStream := inStream
  outStream.strb := inStream.strb(0) && inStream.el.value
       >= 0.S
}
```

Listing 4. Example implementation of single-lane filter.

1) *Tydi Element Types:*
   The *Element* types are implemented as superclasses of Chisel's `Bundle` class.
2) *Tydi Stream Implementation:*
   A *Stream* is from Tydi's perspective also an *Element* and is, therefore, also implemented as a `Bundle`. This allows nesting streams and using the stream directly for IO.
   a) Connecting streams is done using Chisel's directional `:=` notation.
3) *TydiModule Base Module:*
   This module has methods and overrides to allow Chisel → Tydi-lang transpilation.

By staying close to Chisel's normal components and paradigms, it is expected that working with Tydi–Chisel will feel familiar to Chisel programmers, and adapting it should be easy and intuitive for new and existing projects.

D. Communication Specification

The Tydi specification establishes a standard for communicating data over streams. It also specifies how data structures 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 Verilog compiled representation, however, these streams end up as parallel streams. Inherently, nested streams 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. Since, for this example, all components have only one input stream and one output stream without nested streams, no further communication specification need be made.

E. Component Implementation

The fifth step in the given design pipeline is to acquire or build the component implementations. There are currently three types of implementations in two categories. An implementation is usually a wrapper component, routing signals between its IO and inner instances, or an implementation component containing logic.

Wrapper or interconnect components can be fully defined by their Tydi-lang definitions and are included in the generated code. Implementation components can be expressed in two ways: internal and external. An internal component means that the logic will be defined in Chisel and included in the Chisel → Verilog compilation. An external component will be emitted as a (subclass of) `ExtModule`. Chisel will then consider the component a black box, and the emitted Verilog code will just use the component without including an implementation, assuming it will be available. An `ExtModule` can only have IO defined and no wires or logic.

For the lines of code analysis, each component is treated as external, such that the overhead of signal definitions can be clearly established.

```
test(new PipelineWrap) { c =>
  // Initialize signals
  c.in.initSource()
  c.out.initSink()

  // Generate list of random numbers
  val nums = randomSeq(n = 50)
  val stats = processSeq(nums)  // Software impl.

  // Test component
  parallel({
    for ((elem, i) <- nums.zipWithIndex) {
      c.in.enqueueElNow(_.time -> i.U, _.value -> elem.S)
    }
    c.in.enqueueEmptyNow(last = Some(c.in.lastLit(0->1.U)))
  },{
    c.out.waitForValid()
    // Utility for comprehensively printing stream state
    println(c.out.printState(statsRenderer))
    c.out.expectDequeue(_.min -> stats.min.U, _.max ->
        stats.max.U, _.sum -> stats.sum.U, _.average ->
        stats.average.U)
  })
```

Listing 5. Testing a `TydiModule`.

F. Testing Utilities

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. To aid designers with writing these functional tests for Tydi-interface using components, a test driver was developed for Tydi stream signals. This driver is based on the `DecoupledIO` driver from the `chisel-test` package. An example of a test for our Tydi-based module can be seen in Listing 5.

All Tydi–Chisel utilities and Tydi compliance have been verified. Details can be found in the project repository or thesis [37].

V. Illustrative Example

After establishing the general concept of Tydi, Tydi-lang, and Tydi–Chisel, a more challenging system can be examined that more closely resembles a real-world scenario. The system presented in this section contains several sections that each execute a common function in data processing. Fig. 5 depicts the system. From a macroscopic viewpoint, this example system processes a compressed set of records about students and their grades for different courses and computes their similarity vectors. Specifically, the data flow and manipulation in the system are as follows.

1) Take in snappy compressed data.
2) Decompress these data.
3) Parse the JSON records containing the info about the students.
4) Filter the parsed records based on some criteria.
5) Encode the student record in a similarity encoder.

6) Output the similarity vector.

Each part of the system is successively described in this section. This description contains details about intended functionality, stream data types and IO, and integration in the system. Together, the parts used in the example serve to illustrate the following.

1) Usage of Tydi-lang to create a high-level description of the dataflow in the system.
2) How an existing component without Tydi interface can be easily wrapped to allow usage from within Tydi-lang.
3) How a custom JSON parser can automatically be generated by using Tydi-interfaced components and included in the design.
4) How Tydi–Chisel can be used to more easily write a component's functionality.

The schema for the data used in this example is illustrated by the following sample JSON; see Listing 6.

The sources for this example are available online at https://github.com/ccromjongh/Tydi-TVLSI-example together with instructions.

A. Decompression

The first block of the example system is the data-decompression component. This component has a single input stream with snappy compressed bytes and a single output stream with decoded bytes. Listing 7 shows the Tydi-lang definition of these streams and the component.

One of the goals of this example is to show how existing IP can be integrated into a system with Tydi dataflow. The Snappy decompressor IP in question is `vhsnunzip` [38], by Van Straten. The strategy to include such an IP is to simply create a wrapper for it that has Tydi interfaces. To this end, an internal implementation of the desired streamlet definition is created, `VhSnUnzipUnbufferedWrap`. The Chisel code that is generated can then be filled in with the wrapping functionality. The IP itself, `vhsnunzip` in this case, can be specified as an `ExtModule` with IO matching the component. When this wrapper is created, instances of the implementation can be used anywhere in the design. Since the `vhsnunzip` component uses `vhlib` [30] streams, as mentioned in Section III-A, a predecessor to Tydi, creating the wrapper is in this case trivial.

B. JSON Parser

The second block of the example system is the JSON parser component, responsible for converting the JSON character stream into streams of values of the field types. This block consists of a top-level component with various subcomponents each operating parsing a specific part of the data. The top-level component has a single input stream consisting of the decoded bytes and one output stream per field. The generation of the Tydi-lang code for this component and its dependencies is done automatically by the JSON-Analyzer tool, further described in Sections III-C and VI-F. Listing 8 shows snippets of the generated Tydi-lang definition of these streams, the components, and usage.

C. Filter

The filter block's function is twofold. First, it consolidates the separated output streams from the JSON parser into a single stream with a hierarchical data structure. Second, it filters the data entries based on some criteria. The exact filter criterion and implementation are arbitrary and outside the scope of this example. This block aims to showcase different data representations. Listing 9 shows the Tydi-lang definition of the data structure, mirroring the JSON sample of Listing 6.

D. Similarity Encoder

The final block of the system is a placeholder for a component that encodes the data entry for a student in a similarity vector. Such a similarity vector could be used to gain insights into patterns in the data. In this example case, the performance of various types of students for specific courses could be analyzed.

This component takes in the structured student data stream from the filter component and has an output stream that consists of 32-bit floating points. We do not focus on the exact representation and translation of the data in vector space, but rather on how such a component can be integrated into a system. For example, the throughput parameter can be chosen, such that it matches the desired system throughput of records. Listing 10 shows the Tydi-lang definition of the stream, streamlet, and externally defined implementation.

E. IO

The diagram shown in Fig. 5 describes the construction of a data-processing system but has unspecified data input and output blocks. The authors wish to emphasize details of how Tydi and the related tools can be used to realize a system with different kinds of components. Of course, in the end, the input and output streams cannot be left unconnected. Implementation specifics will be platform-dependent, likely revolving around some form of direct memory access.

F. Effort Analysis

Providing a comprehensive estimate of the reduction of development effort that can be achieved by the introduction of new tools can be challenging. A general statistic that is often included in code transformation and generation research is lines of code of input and output. The number of lines of code for various representations of the systems discussed in this section is analyzed. Table II shows the results.

The source material consists of a total of 99 project-specific lines of Tydi-lang (*snappy.td* and *example.td*), the JSON schema for the parser generation (*student_schema.json*), and 48 lines to wrap the existing decompressor component (*DecompressorWrap.scala*). After going through transpilation to Scala (*ExampleMain.scala* and *GenerateExampleVerilog.scala*) and compilation by Chisel, a Verilog output file is produced (*example.v*). For this analysis, Tydi-lang-2–Chisel was run with the option to only emit "external" implementations, as defined in Section IV-E. This method produces an output consisting only of module and interconnect

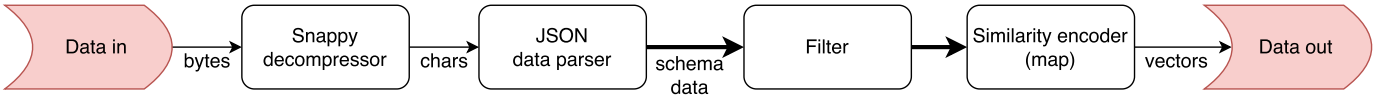

Fig. 5. Pipeline structure for example.

```
{
  "student_number": "S123456789",
  "name": "John Doe",
  "birthdate": "2000-05-15",
  "study_start": "2021-05-15",
  "study_end": null,
  "study": "Computer Science",
  "email": "john.doe@example.com",
  "exams": [
    {
      "course_code": "CS101",
      "course_name": "Introduction to Computer Science",
      "exam_date": "2023-12-10",
      "grade": 80
    },
    {
      "course_code": "MATH201",
      "course_name": "Calculus",
      "exam_date": "2023-12-15",
      "grade": 60
    }
  ]
}
```

Listing 6. JSON sample of student entry.

```
byte = Bit(8);
byte_stream = Stream(byte, t=8.0, d=1, c=1);

streamlet decompressor {
    # Compressed data #
    co: byte_stream in;
    # Decompressed data #
    de: byte_stream out;
}

impl VhSnUnzipUnbufferedWrap of decompressor {}
```

Listing 7. Data, streams, streamlet, and implementation definition for the snappy decompressor.

routing code, without implementation logic. This gives the best indication of a complete interconnect file in Verilog, as no signals are left out through optimization by Chisel's compilation framework. Using this methodology, the resulting Verilog lands at over 2300 lines of code. The majority of this consists of routing of the stream signals for the JSON parser, as it has a lot of components. Compiling the Verilog for the parser as the top-level component results in 1772 lines (74%), giving us an "overhead" of 611 lines (26%) for the rest of our system. The ratio between the source code size and the output code size for the JSON parser is such that one would not write this by hand, the chance of error being very high. However, even assuming the parser component is available, the effort saved in interconnect routing is considerable. This only increases as systems become more complex, as a nested stream (commonly used for streams) adds up to seven extra IO signals to route.

In conclusion, the added value of using Tydi and the tools and utilities of this document are encountered at various stages of development and not solely quantifiable in lines of code of generated boilerplate. Indeed, alongside facilitating easier implementation, the analysis shows a considerable reduction in lines to write, up to several times the code's size. Eventually,

```
streamlet t <d: int> {
JSONStream = Stream(
    Bit(8),
    throughput = 4.0,
    dimension = d,
    synchronicity = "Sync",
    complexity = 8
);

IntParserStream = Stream(
    Bit(64),
    throughput = 1.0,
    dimension = d,
    synchronicity = "Sync",
    complexity = 2
);
...
}

streamlet int_parser_L4_00 {
    NESTING_LEVEL = 3;

    input: t<NESTING_LEVEL+1>.JSONStream in;
    output: t<NESTING_LEVEL>.IntParserStream out;
}

impl int_parser_L4_00_impl of int_parser_L4_00 @External {
    }

streamlet grade_matcher_L3_00 {
    input: t<0>.MatcherStrStream in;
    output: t<0>.MatcherMatchStream out;
}

impl grade_matcher_L3_00_impl of grade_matcher_L3_00
    @External { }

...

streamlet top {
    input: t<1>.JSONStream in;
    output_string_parser_L1_00_inst: t<2>.JSONStream out;
    ...
    output_int_parser_L4_00_inst: t<3>.IntParserStream out;
}

impl top_impl of top {
    instance string_parser_L1_00_inst(
        string_parser_L1_00_impl);
    ...

    self.input => record_parser_L1_00_inst.input;
    ...
}
```

Listing 8. Data, streams, streamlet, and implementation snippets for the JSON parser.

the value provided by this methodology will manifest itself through the lack of research required in custom communication solutions.

Naturally, designers will need to familiarize themselves with the Tydi protocol and Tydi-related tooling. By staying close to familiar syntax and concepts of Chisel and C, it is expected that learning tool usage is easy. Protocol implementation may be harder, but will not exceed a custom solution.

VI. Additional Utilities and Tools

To prevent unnecessary verbosity in common use cases, several helper components and utilities were developed.

```
Char = Bit(8);
n = 4.0;
String = Stream(Char, t=n, d=1, c=8);

Group Exam {
    course_code: String;
    course_name: String;
    exam_date: String;
    grade: Bit(7);
}

Group Student {
    student_number: String;
    name: String;
    birthdate: String;
    study_start: String;
    study_end: String;
    study: String;
    email: String;
    exams: Stream(Exam, t=1.0, d=1);
}

StudentStream = Stream(Student, t=1.0, d=1);

streamlet StudentFilterInterface {
    input_string_parser_L1_00_inst: student_schema_parser.t
        <2>.JSONStream in;
    ...
    input_int_parser_L4_00_inst: student_schema_parser.t
        <3>.IntParserStream in;

    output: StudentStream out;
}

impl StudentFilterImpl of StudentFilterInterface {}
```

Listing 9. Data, streams, streamlet, and implementation snippets for the filter block.

```
Float32 = Bit(32);
SimilarityVectorStream = Stream(Float32, t=64.0, d=1, c=1);

streamlet SimilarityEncoderInterface {
    # Student data structure input. #
    student_input: StudentStream in;
    # Student similarity vector output. #
    similarity_output: SimilarityVectorStream out;
}

impl SimilarityEncoderImpl of SimilarityEncoderInterface
    @External {}
```

Listing 10. Data, streams, streamlet, and implementation for the similarity encoder.

TABLE II

LINES OF CODE COMPARISON; VERILOG IS GENERATED USING ONLY "EXTERNAL" IMPLEMENTATIONS TO SEE INTERCONNECT OVERHEAD. ITALICS INDICATE GENERATED CODE.*AUTOGENERATED SCALA CODE IS NOT OPTIMAL. HAND-WRITTEN CODE WOULD BE MUCH SHORTER

File	Lines of code
student_schema.json	18
snappy.td	13
example.td	86
student_schema_parser.td	546
ExampleMain.scala	3455*
DecompressorWrap.scala	48
GenerateExampleVerilog.scala	25
example.v	2381

A. Stream-Processing Modules

The generated code expressed in Listing 3 of Section IV-C is a functional representation of the hardware that is described for the pipeline and does not need to be altered if an interconnecting implementation is desired. When writing a component's implementation in Chisel, it is rather verbose, however, and seems far off from the original software example (Listing 1) that uses method chaining, a paradigm used by many big-data frameworks because of its convenience and conciseness. Therefore, a 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 11. 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. The `convert` method does this with a stream-complexity converter with specified buffer size.

```
class PipelineExampleModule(bufferSize: Int) extends
    SimpleProcessorBase(new NumberGroup, new Stats) {
  out := in.processWith(new NonNegativeFilter)
           .convert(bufferSize)
           .processWith(new Reducer())
}
```

Listing 11. Compositing of advanced pipeline in Chisel.

At the time of writing, this notation is not used in automatic generation. Since this notation is not as of yet used in Tydi-lang, additional analysis would be required to implement this.

B. Stream-Complexity Converter

Section III-B briefly explains the stream-complexity system for physical streams in Tydi. Connections with complexities $C_{\text{sink}} \geq C_{\text{source}}$ are compatible for a lower C, which is a more bounded version of a higher C. Connecting a high-complexity source to a low-complexity sink, $C_{\text{sink}} < C_{\text{source}}$, does not satisfy this requirement and, thus, requires changing the components or doing a conversion. A *stream-complexity converter* component is developed that can perform this stream-complexity conversion for an arbitrary physical stream (i.e., it takes in an incoming high-complexity stream and outputs a low-complexity outgoing stream). To offer one solution that fits all situations, the component has a $C_{\text{source}} = 8$ and $C_{\text{sink}} = 1$ and is parameterizable for element width (e), dimensionality (d), and number of lanes (n).

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 stream-complexity rules. 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 addition, 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 *realign* the dimensionality information with the element data.

Based on these requirements, the component takes in the data and dimensionality information at C_{source}, buffers and compresses it, and outputs the information again at the required C_{sink}. As shown in Fig. 6, the execution of these tasks is split up into three stages: input processing, buffer management, and output generation.

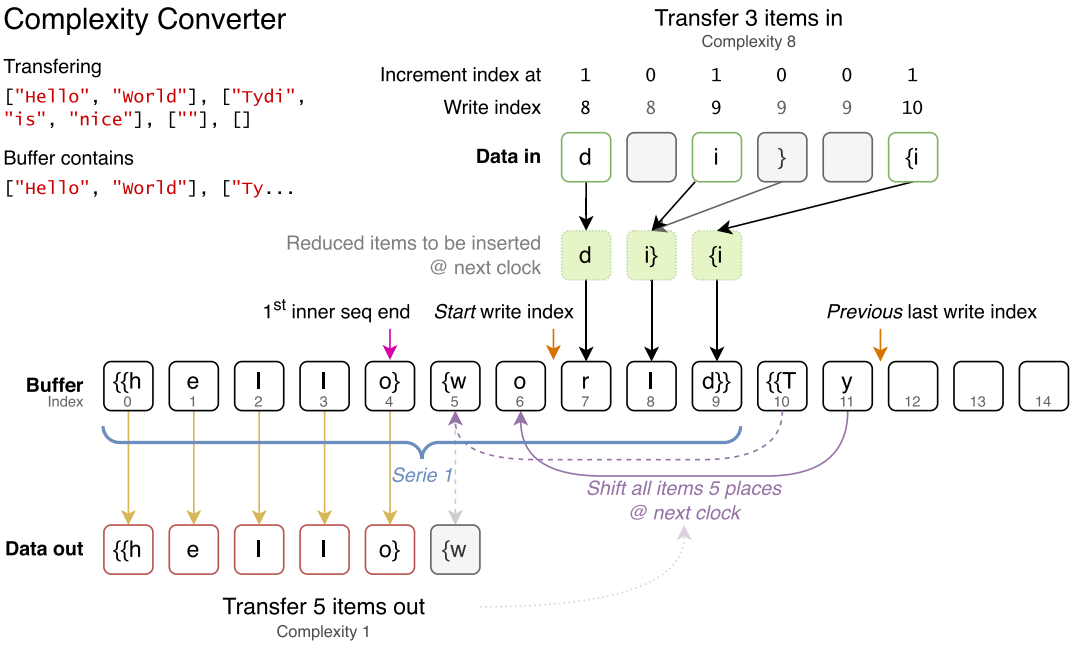

Fig. 6. Visual explanation of complexity converter operation.

1) *Input processing* consists of realigning and reducing optional delayed dimensional data, concluding when an empty element/sequence occurs, and computing where all element data should go. The order of the elements is also important here.
2) For the *buffer*, it is important that the data are coherently stored, in the right order. New data are coming in, and stored data are going out when at least one finished sequence is stored. Since the input and output are uncoupled, this can happen in the same clock cycle.
3) *Output generation* consists of the logic to form a valid $C = 1$ stream.

The most challenging part of the stream-complexity converter design is the input processing. At $C = 8$, dimensionality information can be delayed with respect to the element data. In addition, an empty sequence of $1 \leq d_{\text{seq}} \leq d$ dimensions can be transmitted by closing asserting `last` bits on inactive element data lanes. Simply executing a reduction of the dimensionality information from one valid data lane to the next is, therefore, not possible. While this problem is inherently stateful, it can, nevertheless, be solved combinationally by utilizing a structure similar to a ripple-carry adder. Based on the (reduced) `last` information of the previous lane and the data-lane validity and `last` information of the respective input lane, the element computes the reduced `last` information and whether a new sequence starts based on dimensionality information alone. This process is displayed in Fig. 7.

C. Multilane Slicing

Another utility component that can help insert a component into a design with specific requirements is the *multiprocessing* or *interleaving* component. This component can be used to split a multilane stream into multiple components operating on a single stream. In effect, this creates a component that operates on a multilane stream. This can be used to easily scale up throughput if the data elements can be processed on an element-by-element basis. For an overview, see [37, Ch. 4].

D. Stream Duplicator and Voider

The stream duplicator and stream voider are standard components that were already introduced in [32]. The stream duplicator copies one output stream onto multiple output streams. 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. The stream voider simply consumes all data that are sent to it by always presenting a high `ready` signal. This component is mainly used to connect unused (sub-)streams, avoiding unconnected signal errors.

E. Reverse Transpilation

The design pipeline in Section IV-A 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, such as workflow, a "reverse" transpiler is also available, as seen in Fig. 3. 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 generating a first draft spec when converting the existing projects. See Section VII for a more intricate analysis between Tydi–Chisel and Tydi-lang.

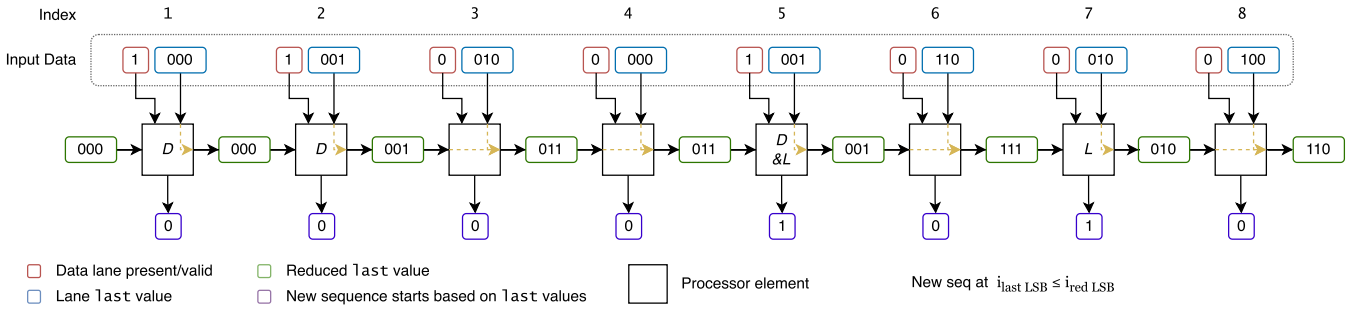

Fig. 7. Last sequence processor component schematic.

F. Updates to JSON-TIL

Section V-B describes the JSON parser block of the illustrative example system. To make this work, some updates to the JSON-TIL project [35] were required. JSON-TIL, as the name implies, was created with Tydi-IR as output target, to be further compiled by the `til` tool to VHDL. While this could have been used in the example by including the top-level component in Tydi-lang as external, Tydi-IR is currently not under active development and misses the functionality to easily duplicate streams. Tydi-lang has a sugaring feature that automatically inserts stream duplicator components with the required number of output ports between the output and input ports when the output stream is used multiple times. For these two reasons, the JSON-Analyzer code of the JSON-TIL project was updated to output Tydi-lang code.

VII. Discussion

Most snippets given in Section V are in Tydi-lang-2 dialect, not in Chisel/Scala. This representation was chosen for a few reasons. Tydi-lang was historically meant as the front-end description for Tydi using dataflow systems. Its syntax was chosen to clearly describe Tydi concepts. In addition, Tydi-lang features sugaring features, such as automatic insertion of stream *duplicator* and *voider* components, where they are necessary, which is essential to the JSON parser generation.

The Tydi–Chisel library was developed with two goals in mind: support Tydi logic types (these include the streams) in Chisel and easily compose systems. In that sense, it was built to either work with generated code or standalone, without requiring any separate tooling than Chisel's. In fact, systems can sometimes be described more efficiently in Chisel when making use of in-place definitions of data types and/or Tydi–Chisel's module shorthands and utilities. Both these things will not happen when generating code, simply because the exact purpose of definitions is not known in Tydi-lang and because Tydi-lang is target-language agnostic.

Regarding this reasoning, two main differences stand out between Tydi-lang and Tydi–Chisel. Tydi-lang's generality prevents the systems it describes from being as concise and specific as with Tydi–Chisel. On the other hand, the generality allows it to evaluate to other HDLs as output.[1] There is, thus, a tradeoff in specificity. A concept that cannot easily be implemented in Chisel is the automatic insertion of stream duplicators. This would require advanced *runtime* introspection and administration, where the power of Scala lies in its strong *compile-time* type checking. Tydi-lang can, thus, be said to be more suitable for implementing advanced design checks and modifications. An interesting question is what the ideal language, or languages and toolchain would look like for Tydi-based hardware development and what standard utilities it should contain.

[1]Since, at the time of writing, til [33] is not maintained anymore, Chisel is the only target language. However, others can easily be supported based on previous work, either by introducing a Tydi library or by more verbose code generation.

VIII. Conclusion

This article introduced Tydi–Chisel, a library that integrates the Tydi standard within Chisel, along with a toolchain and methodology for designing data-streaming accelerators. This toolchain reduces the effort needed to design streaming hardware accelerators and allows for easy integration of accelerator components from different designers.

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.

Through this and previous projects, the tools developed encompass specification of dataflows in the design, creation of hardware design boilerplate code from this specification, utilities for writing the implementations, testing, and generating software–hardware interfaces for communication through Apache Arrow. Tydi–Chisel plays an essential role in the development process for Tydi interface-based systems presented in this work. In the future, Tydi-related tooling can be expanded to aid developers in various stages of accelerator development. Tydi–Chisel could gain more advanced testing tools for data en-/de-queueing, data transfer visualization, and stream protocol compliance verification. Interoperability with the existing streaming protocols could be developed, alongside more real-world examples.

Eventually, the authors envision 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 with the data communication specification, which is easy to implement, and not the component's implementation, avoiding implementation-dependent design.

REFERENCES

[1] C. Cromjongh, Y. Tian, P. Hofstee, and Z. Al-Ars, "Enabling collaborative and interface-driven data-streaming accelerator design with tydi-chisel," in *Proc. IEEE Nordic Circuits Syst. Conf. (NorCAS)*, Nov. 2023, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10305451

[2] L. Truong and P. Hanrahan, "A golden age of hardware description languages: Applying programming language techniques to improve design productivity," in *Proc. 3rd Summit Adv. Program. Lang. (SNAPL)*, 2019, p. 21. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10550/

[3] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, 2019, doi: 10.1145/3282307.

[4] M. U. Tariq and F. Saeed, "Parallel sampling-pipeline for indefinite stream of heterogeneous graphs using OpenCL for FPGAs," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2018, pp. 4752–4761.

[5] E. Mageiropoulos, N. Chrysos, N. Dimou, and M. Katevenis, "Using hls4ml to map convolutional neural networks on interconnected FPGA devices," in *Proc. IEEE 29th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2021, p. 277.

[6] J. Hoozemans, J. Peltenburg, F. Nonnemacher, A. Hadnagy, Z. Al-Ars, and H. P. Hofstee, "FPGA acceleration for big data analytics: Challenges and opportunities," *IEEE Circuits Syst. Mag.*, vol. 21, no. 2, pp. 30–47, 2nd Quart., 2021.

[7] A. Sampson. *From Hardware Description Languages to Accelerator Design Languages*. Accessed: Aug. 7, 2023. [Online]. Available: https://www.sigarch.org/hdl-to-adl/

[8] J. Bachrach et al., "Chisel: Constructing hardware in a scala embedded language," in *Proc. DAC Design Autom. Conf.*, 2012, pp. 1212–1221.

[9] J. Peltenburg, J. Van Straten, M. Brobbel, Z. Al-Ars, and H. P. Hofstee, "Tydi: An open specification for complex data structures over hardware streams," *IEEE Micro*, vol. 40, no. 4, pp. 120–130, Jul. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9098092/

[10] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: A java-compatible and synthesizable language for heterogeneous architectures," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, New York, NY, USA, Oct. 2010, pp. 89–108, doi: 10.1145/1869459.1869469.

[11] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A survey on the evolution of stream processing systems," *VLDB J.*, vol. 33, no. 2, pp. 507–541, Mar. 2024, doi: 10.1007/s00778-023-00819-8.

[12] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, "A survey of distributed data stream processing frameworks," *IEEE Access*, vol. 7, pp. 154300–154316, 2019.

[13] M. J. Sax, "Apache Kafka," in *Encyclopedia of Big Data Technologies*, S. Sakr and A. Zomaya, Eds., Cham, Switzerland: Springer, 2018, pp. 1–8, doi: 10.1007/978-3-319-63962-8_196-1.

[14] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proc. Int. Conf. Compiler Construct.*, France, Apr. 2002, pp. 179–196. [Online]. Available: http://groups.csail.mit.edu/commit/papers/02/streamit-cc.pdf

[15] S. Neuendorffer and K. Vissers, "Streaming systems in FPGAs," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, M. Ber N. Dimopoulos, and S. Wong, Eds., Berlin, Germany: Springer, 2008, pp. 147–156.

[16] J. C. Penha et al., "ADD: Accelerator design and deploy—A tool for FPGA high-performance dataflow computing," *Concurrency Comput., Pract. Exper.*, vol. 31, no. 18, Sep. 2019, Art. no. e5096, doi: 10.1002/cpe.5096.

[17] L. Guo et al., "TAPA: A scalable task-parallel dataflow programming framework for modern FPGAs with co-optimization of HLS and physical design," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, no. 4, pp. 1–31, Dec. 2023, doi: 10.1145/3609335.

[18] L. Guo et al., "RapidStream: Parallel physical implementation of FPGA HLS designs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2022, pp. 1–12, doi: 10.1145/3490422.3502361.

[19] J. Thomas, P. Hanrahan, and M. Zaharia, "Fleet: A framework for massively parallel streaming on FPGAs," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, New York, NY, USA, Mar. 2020, pp. 639–651, doi: 10.1145/3373376.3378495.

[20] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, "Optimus: Efficient realization of streaming applications on FPGAs," in *Proc. Int. Conf. Compil., Archit. Synth. Embedded Syst.*, New York, NY, USA, Oct. 2008, pp. 41–50, doi: 10.1145/1450095.1450105.

[21] Arm Limited. (Apr. 2021). *AMBA AXI-Stream Protocol Specification*. [Online]. Available: https://developer.arm.com/documentation/ihi0051/b

[22] Intel Corporation. (Jan. 2022). *5. Avalon Streaming Interfaces*. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/streaming-interfaces.html

[23] J. Pontes, R. Soares, E. Carvalho, F. Moraes, and N. Calazans, "SCAFFI: An intrachip FPGA asynchronous interface based on hard macros," in *Proc. 25th Int. Conf. Comput. Design*, Lake Tahoe, CA, USA, Oct. 2007, pp. 541–546. [Online]. Available: http://ieeexplore.ieee.org/document/4601950/

[24] C. Markley, A. Wang, and P. Rigge. (Jul. 2024). *Dsptools*. Accessed: Aug. 10, 2016. [Online]. Available: https://github.com/ucb-bar/dsptools

[25] (Jul. 2024). *Clash Protocols* Accessed: Dec. 2, 2020. [Online]. Available: https://github.com/clash-lang/clash-protocols

[26] J. Peltenburg, J. van Straten, L. Wijtemans, L. van Leeuwen, Z. Al-Ars, and P. Hofstee, "Fletcher: A framework to efficiently integrate FPGA accelerators with apache arrow," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 270–277.

[27] Z. Ruan, T. He, B. Li, P. Zhou, and J. Cong, "ST-accel: A high-level programming platform for streaming applications on FPGA," in *Proc. IEEE 26th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2018, pp. 9–16. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8457626

[28] M. Jacobsen and R. Kastner, "RIFFA 2.0: A reusable integration framework for FPGA accelerators," in *Proc. 23rd Int. Conf. Field Program. Log. Appl.*, Sep. 2013, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/6645504

[29] J. Peltenburg, J. van Straten, M. Brobbel, H. P. Hofstee, and Z. Al-Ars, "Supporting columnar in-memory formats on FPGA: The hardware design of fletcher for apache arrow," in *Proc. Int. Symp. Appl. Reconfigurable Comput.*, 2019, pp. 32–47.

[30] J. Van Straten, M. Brobbel, and J. Peltenburg. (May 2021). *Abs-tudelft/vhlib*. Accessed: May 27, 2019-05-27. [Online]. Available: https://github.com/abs-tudelft/vhlib

[31] M. Brobbel, J. Peltenburg, and J. van Straten. (Apr. 2023). *Tydi*. [Online]. Available: https://github.com/abs-tudelft/tydi

[32] Y. Tian et al., "Tydi-lang: A language for typed streaming hardware," 2022, *arXiv:2212.06259*.

[33] M. A. Reukers et al., "An intermediate representation for composable typed streaming dataflow designs," in *Proc. Joint Workshops 49th Int. Conf. Very Large Data Bases (VLDB)*, vol. 3462, 2023, pp. 1–11.

[34] A. Hadnagy, M. Brobbel, and J. Haenen. (Nov. 2022). *Tydi-Json*. [Online]. Available: https://github.com/jhaenen/tydi-json

[35] J. Haenen. (2023). *JSON-TIL: A Tool for Generating/Reducing Boilerplate When Creating and Composing Streaming JSON Dataflow Accelerators Using Tydi Interfaces*. [Online]. Available: https://github.com/jhaenen/JSON_hierachy/blob/master/TIL_JSON.pdf

[36] J. van Straten and J. Haenen. (Dec. 2022). *Vhdre: A VHDL Regex Matcher Generator*. [Online]. Available: https://github.com/jhaenen/vhdre

[37] C. Cromjongh, "Tydi-Chisel—Collaborative and interface-driven data-streaming accelerator design," M.S. thesis, Dept. Elect. Eng., Math. Comput. Sci. (EEMCS), Delft Univ. Technol., Delft, The Netherlands, Oct. 2023. [Online]. Available: http://resolver.tudelft.nl/uuid:06a0d9e6-6120-4ddc-8fd1-ea196bb85f91

[38] J. Van Straten. (Dec. 2022). *Abs-Tudelft/Vhsnunzip*. Accessed: Sep. 6, 2019. [Online]. Available: https://github.com/abs-tudelft/vhsnunzip