Tydi-Chisel: Collaborative and Interface-Driven Data-Streaming Accelerators

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

DOI
10.1109/NorCAS58970.2023.10305451

Publication date
2023

Document Version
Final published version

Published in
Proceedings of the 2023 IEEE Nordic Circuits and Systems Conference (NorCAS)

Tydi-Chisel: Collaborative and Interface-Driven Data-Streaming Accelerators

Casper Cromjongh* Yongding Tian* Peter Hofstee*† Zaid Al-Ars*
*Delft University of Technology, Delft, The Netherlands
†IBM Infrastructure, Austin, TX, US
Email: C.Cromjongh@student.tudelft.nl

2023 IEEE Nordic Circuits and Systems Conference (NorCAS) | 979-8-3503-3757-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/NorCAS58970.2023.10305451

***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 is 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. In contrast, Chisel, with its high level of abstraction and customizability offers a suitable platform to implement Tydi-based components. In this paper, Tydi-Chisel is presented along with an A-to-Z design-process description. 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 and syntax sugar. In combination Chisel and Tydi help bridge the hardware-software divide, making solo-design and collaboration between designers easier.**

Project repository: https://github.com/ccromjongh/Tydi-Chisel

***Index Terms*—big data, streaming interfaces, HW design, testing**

I. Introduction

The deceleration in performance gain of CPUs signals the advent of the post-Moore's Law era [1, 2]. 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 [3], and HLS4ML [4], designed to streamline FPGA development, challenges remain. These challenges are amplified in the big data domain [5], 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 [6] accentuates this disparity, advocating for a transition from Hardware Description Language (HDL) to Accelerator Design Language (ADL). Chisel has emerged as a promising way to achieve this transition.

Chisel [7] 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.

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 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."* [8, pp. 123].

To address these issues, the idea for Tydi (Typed Dataflow Interface) was proposed. Figure 1 aims to illustrate the difference by showing a metaphorical representation of a raw data stream, handshaked stream, and Tydi stream.

Without a common interface standard like 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 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. 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 paper, 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 paper is organized as follows: in Section II, a background on Tydi and Chisel is given. Section III provides an overview of related works regarding streaming design and accelerator design. In Section IV, a Tydi-driven design workflow is explored with a simple example use-case. Additional

979-8-3503-3757-0/23/$31.00 ©2023 IEEE

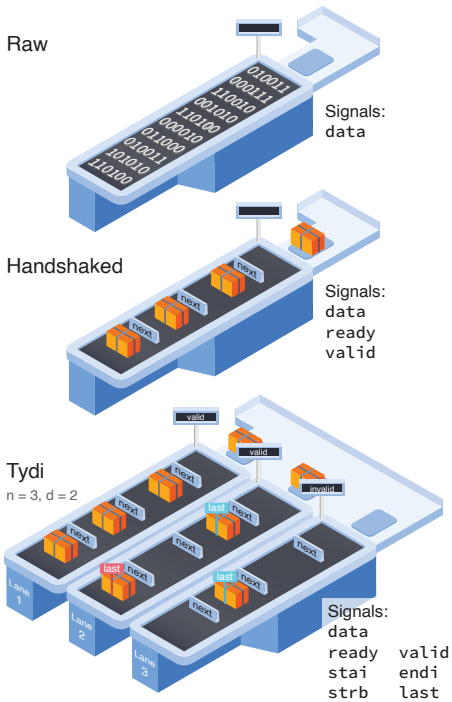

Fig. 1. A comparison of stream types represented as checkout conveyor belts. The Tydi stream has $n = 3$ lanes and a dimensionality of $d = 2$. `stai`, `endi`, and `strb` relate to data-lane validity. `last` transfers dimensionality information.

Tydi-Chisel features are covered in Section V. Section VI summarizes the whole paper.

II. Background

A. Tydi specification

The Tydi specification was first introduced in [8]. 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 [9]. Based on this refined version, Tian et al. proposed a high-level hardware description language to raise the abstraction level of typed streaming hardware and reduce the design effort for hardware designer [10]. In addition, Reukers et al. developed an intermediate representation tailored for hardware circuit design using the Tydi framework, accompanied by a compiler for VHDL translation [11]. 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 Table 4 of [8].

The basic data types used in Tydi intermediate representation are *Null*, *Bits*, *Group* and *Union*. The *Stream* is a wrapper of basic data types, adding extra streaming properties, such as *complexity*, *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 visually represents the protocol of complexities at levels 1 and 8. It is noteworthy that a source port with a lower complexity is able to connect with a sink port of a higher complexity.

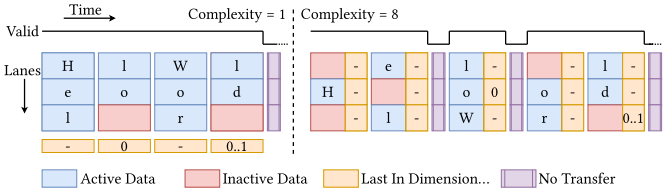

Fig. 2. The stream complexity property [11]

- **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-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.

Several projects have emerged that utilize Tydi-related methodologies. Among these, Tydi-JSON [12] 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 [12] and [11], JSON-TIL [13] 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 [14] incorporates Tydi interfaces. This initiative enables the generation of hardware blueprints for regular expression matchers that operate on UTF-8-encoded strings.

B. Chisel

Chisel [7] (Constructing Hardware In a Scala Embedded Language) is an open-source hardware construction language developed to facilitate the design of highly parameterized hardware components. Traditional HDLs primarily focus on

TABLE I
TYDI TERMS AND CORRESPONDING MEANING

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 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.

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.

C. Fletcher

The Fletcher project [15] 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 [16]. 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.

III. 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 [17–20] on multi-threaded CPUs and GPUs. Moreover, specific studies have focused on the intricacies of FPGA-oriented streaming [21, 22]. 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, [23] 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 [24, 25], 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 [26]. In response to the challenges posed by component interface compatibility, several industry standards have been established [27–29]. However, these existing works focus on the hardware signals rather than an effective representation of complex data, which is addressed by Tydi.

With `DecoupledIO`, Chisel has a built-in way to create handshaked connections, although support for dimensionality information and throughput scaling lacks. 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.

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.

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 this:

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

These steps and relevant toolchain projects are depicted in Figure 3.

B. Number pipeline example

To illustrate the aforementioned pipeline, we will work with an example. This example is purposefully kept simple and we want to remind the reader that the advantage of Tydi's ecosystem is greater with more complex projects. In this

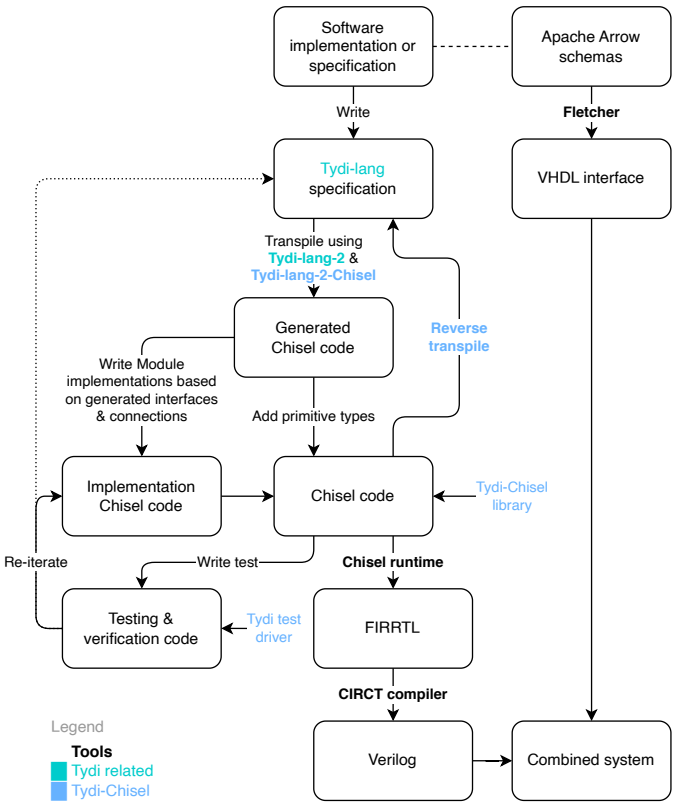

Fig. 3. Tydi toolchain components

example, we take in a stream of numbers with timestamps attached. This stream first gets filtered on $value \geq 0$ and then reduced to statistics: min value, max value, sum of values, and average. The block schedule for this system is given in Figure 4 In Apache Spark, one could execute this process as in Listing 1.

Listing 1. Example Spark code

```
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")
)
```

Fig. 4. Number pipeline structure

C. Tydi-lang specification & 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, components that require practical implementations are left empty. Tydi-Chisel code for this specification can be obtained by first running `tydi-lang-2`, obtaining a JSON description that is used to generate the Chisel code with `Tydi-lang-2-Chisel`.

Listing 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 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;
}
```

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 re-align the items when the sequence is required.

Listing 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 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 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 pack1. */
class NumsToStats_interface extends TydiModule {
    // ... code for NumberGroup in, Stats out
}
// ... other interface and implementation definitions

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

/** Implementation, defined in 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 4. Example implementation of single-lane filter

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

Tydi-Chisel's library was designed for ease of use both in new projects and in converting existing code. Next to syntax, care is given to the implementation of Tydi-Chisel's components. This implementation in Chisel consists of a few parts.

- Tydi *Element* types
  The *Element* types are implemented as superclasses of Chisel's `Bundle` class.
- 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.
  - Connecting streams is done using Chisel's directional `:=` notation.
- *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. Advanced number pipeline example

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 number of lanes is arbitrary, but $n = 4$ is chosen for this example.

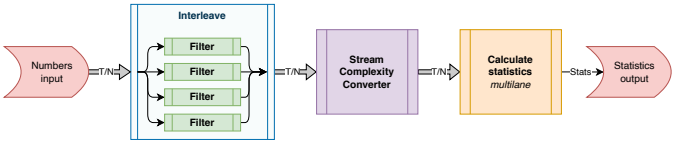

Fig. 5. Advanced number pipeline structure (T/N stands for timestamped numbers)

In the original number pipeline, filter and reducer components are available or created 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 (described in Section V). 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 stream 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 (Section V) 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 5 and code in Listing 5. This code showcases advanced use of Tydi-Chisel utilities, in contrast to the generated code before.

E. Stream-processing modules

The generated code expressed in Listing 3 is a functional representation of the hardware that is described for this pipeline and does not need to be altered if all connections are specified. When writing a component's implementation 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 5. 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.

Listing 5. Compositing of advanced pipeline in Chisel

```
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))
}
```

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.

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 6.

Listing 6. Testing a `TydiModule`

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

  // 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)
  })
```

All Tydi-Chisel utilities and Tydi compliance have been verified. See the project repository or thesis [30].

V. Additional utilities

A. Helper components

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

As explained in Subsection II-A, in Tydi, connections with complexities $C_{sink} \geq C_{source}$ are compatible. Connecting a high-complexity source to a low-complexity sink thus requires a conversion step. A *stream complexity converter* component is available that does this by taking in an incoming high-complexity stream and outputting a low-complexity outgoing stream. The *multi-processing* or *interleaving* component can be used to split a multi-lane stream to multiple components operating on a single 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 [30, Ch. 4].

B. Reverse-transpilation

The example in Section IV 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 is also available, as seen in Figure 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 generate a first draft spec when converting existing projects.

VI. Summary

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 trough Apache Arrow. In the future, Tydi-related tooling can be expanded to aid developers in various stages of accelerator development.

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

Acknowledgment

This research was performed with the support of the Eureka Xecs project TASTI (grant no. 2022005).

REFERENCES

[1] L. Truong and P. Hanrahan, "A golden age of hardware description languages: Applying programming language techniques to improve design productivity," p. 21 pages, 2019. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10550/

[2] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3282307

[3] M. U. Tariq and F. 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, pp. 4752–4761.

[4] E. Mageiropoulos, N. Chrysos, N. Dimou, and M. Katevenis, "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, pp. 277–277, ISSN: 2576-2621.

[5] 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 and Systems Magazine*, vol. 21, no. 2, pp. 30–47, 2021.

[6] A. Sampson. From hardware description languages to accelerator design languages. [Online]. Available: https://www.sigarch.org/hdl-to-adl/

[7] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, 2012, pp. 1212–1221, ISSN: 0738-100X.

[8] 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, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9098092/

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

[10] Y. Tian, M. A. Reukers, Z. Al-Ars, P. Hofstee, M. Brobbel, J. Peltenburg, and J. van Straten, "Tydi-lang: A language for typed streaming hardware." [Online]. Available: http://arxiv.org/abs/2212.06259

[11] M. A. Reukers, Y. Tian, Z. Al-Ars, P. Hofstee, M. Brobbel, J. Peltenburg, and J. van Straten, "An intermediate representation for composable typed streaming dataflow designs," *Joint Proceedings of Workshops at the 49th International Conference on Very Large Data Bases (VLDB 2023)*, vol. 3462, 2023.

[12] A. Hadnagy, M. Brobbel, and J. Haenen, "Tydi-json," Nov. 2022. [Online]. Available: https://github.com/jhaenen/tydi-json

[13] J. Haenen, "JSON-TIL: A tool for generating/reducing boilerplate when creating and composing streaming JSON dataflow accelerators using tydi interfaces," 2023. [Online]. Available: https://github.com/jhaenen/JSON_hierachy/blob/master/TIL_JSON.pdf

[14] J. van Straten and J. Haenen, "vhdre: a VHDL regex matcher generator," Dec. 2022. [Online]. Available: https://github.com/jhaenen/vhdre

[15] 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 *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 270–277, ISSN: 1946-1488.

[16] J. Peltenburg, J. van Straten, M. Brobbel, H. Hofstee, and Z. Al-Ars, "Supporting columnar in-memory formats on fpga: The hardware design of fletcher for apache arrow," in *Applied Reconfigurable Computing*, 2019, pp. 32–47.

[17] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "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*, ser. OOPSLA '10. New York, NY, USA: Association for Computing Machinery, Oct. 2010, pp. 89–108. [Online]. Available: http://doi.org/10.1145/1869459.1869469

[18] J. Thomas, P. Hanrahan, and M. 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*. New York, NY, USA: Association for Computing Machinery, Mar. 2020, pp. 639–651. [Online]. Available: http://doi.org/10.1145/3373376.3378495

[19] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A Survey on the Evolution of Stream Processing Systems," Aug. 2020. [Online]. Available: http://arxiv.org/abs/2008.00842

[20] 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. 154 300–154 316, 2019.

[21] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, "Optimus: Efficient realization of streaming applications on FPGAs," in *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '08. New York, NY, USA: Association for Computing Machinery, Oct. 2008, pp. 41–50. [Online]. Available: http://doi.org/10.1145/1450095.1450105

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

[23] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *International Conference on Compiler Construction*, Grenoble, France, Apr. 2002. [Online]. Available: http://groups.csail.mit.edu/commit/papers/02/streamit-cc.pdf

[24] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, "LLHD: A Multi-level Intermediate Representation for Hardware Description Languages," *arXiv:2004.03494 [cs]*, Apr. 2020. [Online]. Available: http://arxiv.org/abs/2004.03494

[25] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2017, pp. 209–216.

[26] LLVM Community, ""CIRCT" / Circuit IR Compilers and Tools," LLVM, May 2022. [Online]. Available: https://github.com/llvm/circt

[27] Arm Limited, "AMBA® AXI-Stream Protocol Specification," Apr. 2021. [Online]. Available: https://developer.arm.com/documentation/ihi0051/b

[28] Intel Corporation, "5. Avalon® Streaming Interfaces," Jan. 2022. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/streaming-interfaces.html

[29] J. Pontes, R. Soares, E. Carvalho, F. Moraes, and N. Calazans, "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. [Online]. Available: http://ieeexplore.ieee.org/document/4601950/

[30] C. Cromjongh, "Tydi-Chisel: Collaborative and Interface-Driven Data-Streaming Accelerator Design," Master's thesis, Delft University of Technology, Delft, Oct. 2023. [Online]. Available: http://resolver.tudelft.nl/uuid:06a0d9e6-6120-4ddc-8fd1-ea196bb85f91