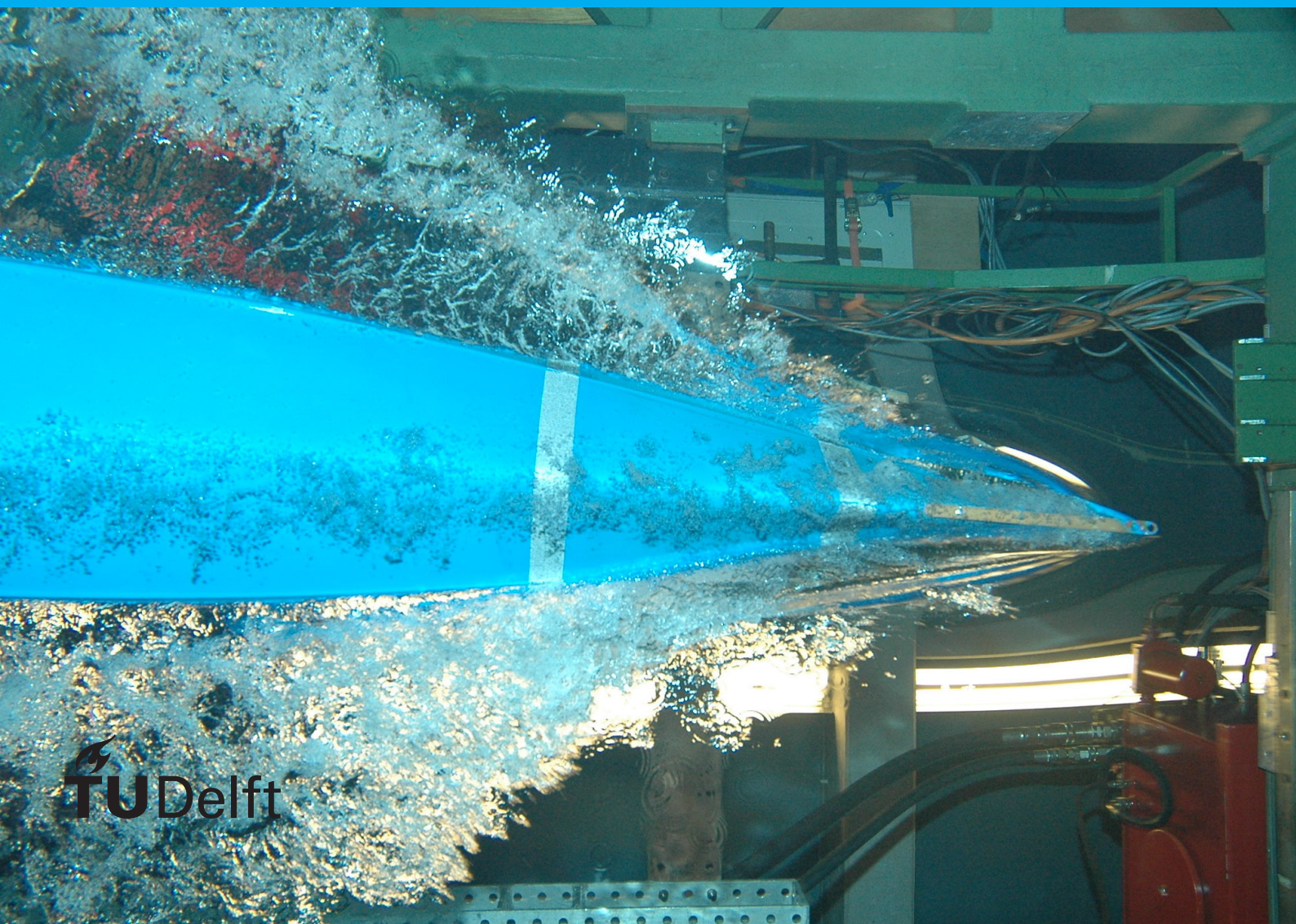
A Toolchain for Streaming Dataflow Accelerator Designs for Big Data Analytics

Defining an IR for Composable Typed Streaming Dataflow Designs

M. A. Reukers



A Toolchain for Streaming Dataflow Accelerator Designs for Big Data Analytics

Defining an IR for Composable Typed Streaming Dataflow Designs

by

M. A. Reukers

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday July 8, 2022 at 10:00 AM.

Student number:	4227220	
Project duration:	November 25, 2021 – July 8, 2022	
Thesis committee:	Prof. dr. H. P. Hofstee,	TU Delft, supervisor
	Dr. ir. T. G. R. M. van Leuken,	TU Delft
	Dr. ir. Z. Al-Ars,	TU Delft
	Dr. ir. J. Peltenburg,	Voltron Data

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

TU Delft

Abstract

Tydi is 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. This provides a higher-level method for defining interfaces between components as opposed to existing bit- and byte-based interface specifications.

In this thesis, an open-source intermediate representation (IR) is introduced which allows for the declaration of Tydi's types. The IR enables creating and connecting components with Tydi Streams as interfaces, called Streamlets. It also lets backends for synthesis and simulation retain high-level information, such as documentation. Types and Streamlets can be easily reused between multiple projects, and Tydi's streams and type hierarchy can be used to define interface contracts, which aid collaboration when designing a larger system.

The IR codifies the rules and properties established in the Tydi specification and serves to complement computation-oriented hardware design tools with a data-centric view on interfaces. To support different backends and targets, the IR is focused on expressing interfaces, and complements behavior described by hardware description languages and other IRs. Additionally, a testing syntax for the verification of inputs and outputs against abstract streams of data, and for substituting interdependent components, is presented which allows for the specification of behavior.

To demonstrate this IR, a grammar, parser, and query system have been created, and paired with a backend targeting VHDL.

Preface

This thesis proved to be considerably challenging, and an excellent learning experience. It was very interesting to explore various technologies around hardware accelerator design, compilation, and language parsing, and learn some of the specifics of VHDL syntax while implementing the backend.

I would like to thank Prof. dr. Peter Hofstee for his continued support during my work on the thesis, and for pushing me to constantly evaluate the novelty and direction of my work. Likewise, I would like to thank Dr. Zaid Al-Ars, Yongding Tian and the entire ABS Group for the interesting meetings and support over this time. Finally, I would like to thank Johan, Matthijs Brobbel and Joost, for their ideas and the extensive discussions, for answering my questions around the Tydi specification, and for so thoroughly reviewing my paper before I submitted it to ICCAD.

Some of the contents of this thesis also appear in “An Intermediate Representation for Composable Typed Streaming Dataflow Designs”, a paper which I submitted to ICCAD 2022. [35]

M. A. Reukers
Delft, July 2022

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 1
- 1.2 Methodology . . . 2
- 1.3 Contributions . . . 3

2 Background **5**
- 2.1 Stream Processing . . . 5
  - 2.1.1 Data Streams . . . 5
  - 2.1.2 Interface Specifications . . . 5
- 2.2 Tydi . . . 6
  - 2.2.1 Element-manipulating Types . . . 6
  - 2.2.2 Streams . . . 7
  - 2.2.3 Physical Streams . . . 8
- 2.3 Alternatives . . . 12
  - 2.3.1 Hardware Description Languages . . . 12
  - 2.3.2 Design Tools and High-Level Synthesis . . . 12
  - 2.3.3 Frameworks and Embedded Domain-Specific Languages . . . 13

3 Intermediate Representation: Composition **15**
- 3.1 Type Declarations and Interface Design . . . 15
  - 3.1.1 Type Declarations . . . 15
  - 3.1.2 Interfaces as Contracts . . . 17
  - 3.1.3 Compatibility . . . 18
- 3.2 Component Composition and Implementation . . . 19
  - 3.2.1 Structural Composition . . . 19
  - 3.2.2 Linked Implementations . . . 20
- 3.3 Recommendations for Language and Compiler Features . . . 21
  - 3.3.1 Type Parameters . . . 22
  - 3.3.2 Generation . . . 23
  - 3.3.3 Intrinsics . . . 24
  - 3.3.4 Annotations . . . 26
- 3.4 Project Structure and Reusability . . . 27
  - 3.4.1 Project Properties . . . 28
  - 3.4.2 Import Behavior . . . 28
  - 3.4.3 Notes on Reusability . . . 29

4 Intermediate Representation: Specification **31**
- 4.1 High-level Assertions . . . 31
  - 4.1.1 Parallel by Default . . . 31
  - 4.1.2 Sequences . . . 32
  - 4.1.3 Descriptive Errors . . . 33
  - 4.1.4 Asserting Equality . . . 33
  - 4.1.5 Issues with Explicit Assignment and Comparison . . . 35
- 4.2 Proof of Concept . . . 36
  - 4.2.1 Physical Transfers . . . 36
  - 4.2.2 Demonstration . . . 36
  - 4.2.3 Results and Future Work . . . 38

4.3 Complex Test Cases . . . 38
4.3.1 Limitations of High-Level Assertions . . . 38
4.3.2 Using Test Streamlets for Verification . . . 39
4.3.3 Substitution . . . 39
4.4 Setting up Subjects . . . 40

5 Implementation **43**
5.1 Query System . . . 43
5.2 Grammar and Parser . . . 44
5.2.1 Parsers . . . 44
5.2.2 Grammar . . . 46
5.2.3 Parser Implementation . . . 47
5.3 VHDL Backend . . . 48
5.3.1 Components and Organization . . . 50
5.3.2 Linked Implementations . . . 51
5.3.3 Structural Implementations . . . 51
5.3.4 Additional and Future Functionality . . . 52
5.4 Example . . . 53
5.5 Partial Implementations . . . 53

6 Evaluation **55**
6.1 Tydi Specification . . . 55
6.1.1 Directly nested Streams which must both be retained . . . 55
6.1.2 Significance of Strobe and Index Signals . . . 56
6.1.3 Transferring Empty Outer Sequences at Lower Complexities . . . 56
6.1.4 Indicating Inactive Lanes at Lower Complexities . . . 57
6.1.5 Minor Inconsistencies . . . 57
6.2 Readability . . . 58
6.2.1 Readable Output . . . 58
6.2.2 Type Identifiers . . . 58
6.3 Hardware Description Effort . . . 59
6.4 Parser . . . 61
6.4.1 Merits of Chumsky . . . 61
6.4.2 Issues Using Chumsky . . . 61
6.4.3 Recommendations for TIL Parser . . . 62

7 Conclusion **63**
7.1 Conclusions and Summary . . . 63
7.2 Recommendations for Future Work . . . 63

A Complete TIL Example **65**

B VHDL Backend Example **69**

C AXI4 Specification **71**

D AXI4 TIL Definition and VHDL Output **73**

1 Introduction

In order to transfer streaming data between components within digital circuits, designers have a choice to either design their own interfaces, or use general interface specifications such as Intel's Avalon-ST [21] or Arm's AXI4-Stream [7]. By using an interface specification, it is easier for other designers to connect components, as the signals and how they relate to data transfers are standardized. This can promote reuse, and is used by hardware design tools to provide IP (Intellectual Property) libraries and automate integration [9, 24].

The aforementioned specifications do not specify how data structures are represented, however, and as a result designers must still design, document and share these representations. Additionally, the IP integration tools are proprietary, reducing the simplicity of integrating such IPs outside of these specific tools. Addressing the first issue, Peltenburg et al. proposed Tydi (Typed dataflow interface) [32], an open specification which allows designers to explicitly define the data which is being transferred by providing a type system for composite and variable-length data structures, in addition to defining how data elements are organized in transfers and the requirements on transfers. This thesis aims to address the second issue, by utilizing the Tydi specification as part of an IR (intermediate representation) for defining interfaces and connecting components.

The goal of the IR is not to serve as a complete hardware description language, but to provide a simple and robust way to declare Tydi's types, define interfaces and connect components which adhere to the Tydi specification, serving as part of a toolchain in order to integrate and reuse components within and across projects. To this end, the IR is not capable of directly implementing behavior, but should instead be combined with transaction-level verification to *specify* intended behavior.

To demonstrate the potential of such an IR, and to explore potential approaches towards implementing a toolchain built around it, a prototype toolchain has been conceived and implemented. This consists of a query system, which tracks and computes information defined through the IR, a grammar called TIL (Tydi Intermediate Language) and parser, as a more portable, text-based way of representing designs in the IR, and finally a VHDL backend to emit designs defined in the IR.

1.1. Problem Statement

While much research is focused on developing and accelerating algorithms for streaming data in both hardware [31, 34] and software [22], many designs for low-level hardware still have to transfer streams over interfaces which are either custom or based on generic, bit- and/or byte-oriented specifications such as AXI4-Stream [7] and Avalon-ST [21]. As a result, higher-level information about data structures and how streams of data are organized over transfers must be devised and implemented by designers, and are not reflected by the declaration of the interface in a traditional HDL.

Some of this design effort can be alleviated through the use of high-level synthesis: tools such as Vivado HLS can be employed to leverage C, C++ or SystemC combined with IP-blocks using *ap_fifo* or AXI4-Stream to handle data streams [4], while synthesizing compilers such as Optimus [17] have been developed in the past to leverage StreamIt [42], a language specifically for streaming applications. At the same time, many researchers are working on improved hardware description languages and IRs, such as Chisel [11], FIRRTL [23] and LLHD [40].

These are not suitable replacements for a higher-level interface specification, however: HLS tools either obfuscate the interfaces between low-level hardware and/or use proprietary IRs and tools to connect components, making reuse more difficult. While the HDLs and IRs mentioned are aimed at more general hardware designs, so still require custom interfaces for streaming data transfers.

As such, the aim of this thesis is to develop a free, open-source IR for defining high-level streaming dataflow interfaces mapped onto hardware and for connecting these interfaces. This would complement existing HDLs and IRs which describe behavior, and enable components designed in higher-level front-end languages for HLS to propagate more type information to the resulting interfaces.

Long-term, the Accelerated Big Data Systems group aims to develop a toolchain for streaming dataflow accelerator designs for big data analytics. The work done for this thesis is part of such a toolchain, providing a grammar and parser, query system for the IR, and a compiler to VHDL. At the same time, Yongding Tian has been working on a front-end language for his thesis and as another component of this toolchain, enabling designers to express behavior as well.

1.2. Methodology

As the aim of this thesis is not only to define an intermediate representation, but contribute to a toolchain, there was an increased focus on *implementing* such tools. Essentially, by creating and iterating on a "vertical slice" of a partial toolchain, it is possible to evaluate the effectiveness of the IR and the feasibility of the proposed toolchain overall.

An ideal vertical slice would have the following properties and components:

1. A (partial) specification for the intermediate representation; i.e., what (additional) concepts should it be able to express.
2. A means to integrate a compiler, using one or both of:
   (a) A grammar and a parser, taking a text-based representation and allowing a subsequent backend to interpret the results.
   (b) A *query system* not unlike the one employed by the *Rust* compiler [37], which would allow a backend to perform queries to retrieve and/or compute information from a definition in the IR, reducing the need for separate optimizing passes. (As such, the information stored in the system does not necessarily need to be the result of a (single) parser, but can be input programmatically.)
3. A backend for emitting designs defined in the IR as a conventional hardware description language suitable for simulation and synthesis. Due to familiarity and broad support, this language will be VHDL(-93). The backend should be capable of as many of the following as possible:
   (a) Emit Streamlets with structural implementations; i.e., Streamlets which contain and connect other Streamlets.
   (b) Link behavioral implementations.
   (c) Emit a testbench based on high-level assertions defined in the IR.

Based on interim progress towards these features and results of finished (prototype) implementations, the next step would be to:

- Continue working on and/or expanding specific features. (The initial implementation is successful and/or promising, or the feature requires further evaluation.)
- Revise goals and/or the IR specification. (The feature is not feasible, or an alternative appears more effective.)
- Omit them and instead recommend their implementation as future work. (The feature is feasible and promising, but cannot be implemented satisfactorily within the time frame of this thesis.)

As an example, if a concept expressed in the IR is impossible or very difficult to express in VHDL (or any target HDL), the solution would be to either revise the IR to include more information (i.e., the concept is possible to express, but requires more/different input), or to summarize these findings in this thesis and remove it or recommend it as future work.

The results of this methodology are described in Chapter 6.

1.3. Contributions

The contributions of this thesis can be summarized as follows, with references to the relating chapters and sections:

- **An intermediate representation for composition and linking behavior** — This thesis proposes an intermediate representation for composing streaming dataflow designs, using the Tydi specification. It features means to define Streams and the data they carry, to create interfaces with clock and reset domains tied to specific ports, create components (Streamlets) and connect them, and link to implementations of behavior. Sections 3.1 and 3.2
- **Recommendations for further improvements to the intermediate representation** — After evaluation of the intermediate representation, it was determined certain language and compiler-oriented features would improve the intermediate representation's ability to describe designs and aid backends in emitting them to a target language. Specifically, the addition of type parameters (3.3.1), the inclusion of code generation constructs (3.3.2), the addition of limited, behavioral intrinsic functions (3.3.3), the inclusion of annotations for backends (3.3.4) and changes to the existing representation to improve the readability of the output (6.2). This thesis also discusses potential difficulties when implementing them, as features need to translate well to many potential target languages.
- **Proposals for (and partial, preliminary implementation of) a high-level testing framework** — As the intermediate representation primarily exposes typed interfaces, tests can be performed as high-level assertions against transfers of typed data (Sections 4.1 and 4.2), while the inclusion of substitutions helps when testing more complex or incomplete dependencies (4.3). This thesis also discusses the potential problems (and some solutions) when setting up (resetting) subject components in Section 4.4.
- **A complete toolchain as proof of concept, from intermediate representation to target language** — As part of the work on this thesis, a query system for the intermediate representation (5.1), a VHDL backend (5.3), and a text-based grammar (Tydi Intermediate Language, TIL) and parser (5.2) were implemented. These are provided in a free, open-source repository along with a simple example application which allows a user to compile a TIL file to VHDL, described in Section 5.4.
- **Validating the Tydi interface specification** — By implementing the Tydi interface specification programmatically, a number of unaddressed and contradictory situations were brought to light, for which the specification should be amended. Section 6.1
- **Evaluation of the intermediate representation's ability to describe interfaces and connections** — The intermediate representation was evaluated for its ability to produce human-readable and traceable output in Section 6.2. The evaluation of its effectiveness in representing streaming interfaces and connections between them is described in Section 6.3, using the existing AXI4 and AXI4-Stream standards as a reference point.

2 Background

2.1. Stream Processing

2.1.1. Data Streams

Stream processing refers to means of processing data which is produced or consumed incrementally, rather than a set of data which is known and stored ahead of time on the system. The order of the data and rate at which it arrives cannot necessarily be controlled, and the number of elements is potentially unbounded, requiring the system to process elements as they arrive and before the next element does.

Examples of practically unbounded data streams would be analyzing real-time weather events or human behavior, but even limited sets of data can be treated as streams when timing is critical to performance, such as when encrypting and decrypting data to and from a storage medium. As a result, stream processing has been actively researched for over 20 years, with software paradigms and hardware acceleration being worked on in parallel in attempts to improve performance and establish effective data and execution models [15, 22].

Software models In software, stream processing has been approached in many different ways to various ends. More recent examples of stream processing include Kafka Streams, which is a stream processing library of Apache Kafka [39], and Spark Streaming [49, 5] (now Structured Streaming [6]). Both aim to provide a useful subset of high-level functions for processing data streams, mapped onto their existing domains. There are also more wholesale approaches, such as StreamIt [42], which is a language specifically designed for streaming applications.

Hardware Acceleration In hardware acceleration, the constraints of stream processing are less uncommon; hardware designs are already heavily constrained by timing, and do not necessarily have a notion of state. More specifically addressing recent needs of stream processing, there are a number of frameworks such as Fleet [43] and S2FA (Spark-to-FPGA-Accelerator) [48] which use FPGAs to accelerate streaming operations which conventional processors may struggle with.

2.1.2. Interface Specifications

When designing digital circuits for stream processing hardware accelerators, internal communication will likewise take the form of unbounded streams of messages between sources and sinks. There exists a number of interface specifications to ensure these streams of data are correctly transferred and represented: For example, ARM devised the AXI4-Stream protocol [7], and Intel defines the Avalon-ST interface specification [21].

Both protocols are able to optionally organize sequences of data into packets over transfers; AXI4-Stream uses the *last* signal to indicate that a transfer is the last in a sequence making up a packet, while Avalon-ST uses the *startofpacket* and *endofpacket* signals to do the same. Likewise, both interface specifications incorporate means to indicate whether data is being transferred from a source (using a *valid* signal) and can be transferred to a sink (using a *ready* signal), and allows for valid transfers to be indicated as entirely or partially empty. These properties ensure that sequences of data can be transferred over time and without needing to account for the rate at which individual elements arrive, nor for the total size of a sequence.

Additionally, by using a standard and well-defined interface, designers can not only ensure that transfers are consistent within a project, but can share components between different projects and even across organizations. For example, this is employed by AMD/Xilinx [3] and Intel [20] to establish libraries of IP cores for designs implemented on their respective FPGAs. Finally, when representing streams of data in high-level synthesis, such standards ensure data can be consumed from or produced for such IP cores; e.g., Vivado HLS uses AXI4-Stream and *ap_fifo* for this purpose [4].

2.2. Tydi

The Tydi specification and type system was introduced by Peltenburg et al. [32] and defines an abstract way to describe data structures transferred over hardware streams. Tydi promises to reduce the design effort of creating hardware for streaming dataflow computing, by providing clear and intuitive ways to map composite, variable-length data structures onto a hardware streaming protocol. An open-source repository and documentation [44] expanding on the specification and providing example code for mapping Tydi's streams onto VHDL component ports is now available.

The specification defines five *logical types*: the stream-manipulating Stream type, and the element-manipulating Null, Bits, Group and Union types.

2.2.1. Element-manipulating Types

Element-manipulating types are how Tydi represents kinds of data; that is to say, these types represent arbitrary data as well as specific data structures.

- The *Null* type is for transfers of one-valued data, its only valid value is null. This can be used to indicate (part of) a transfer being valid and active, but no data being transferred.
- The *Bits(N)* type represents a data signal of N bits. It is used to transfer arbitrary data.
- The *Group* type contains named *fields*, which are themselves any logical type. Groups are compositions, and represent all fields being active simultaneously.
- The *Union* type is comparable to the Group type, in that it contains named *fields* of logical types. Unlike Groups, Unions are exclusive disjunctions; only one field may be active at a time. Figure 2.1 further illustrates the difference.

Field *Name*s of Groups and Unions consist of (ASCII) letters, digits and/or underscores [45], and may not contain two or more consecutive underscores. Names within a Group or Union must be unique, cannot start or end with an underscore, and cannot start with a digit. The latter constraints ensure broad compatiblity with potential target HDLs, while consecutive underscores are reserved for use in *Path Name*s, discussed in Section 2.2.3.

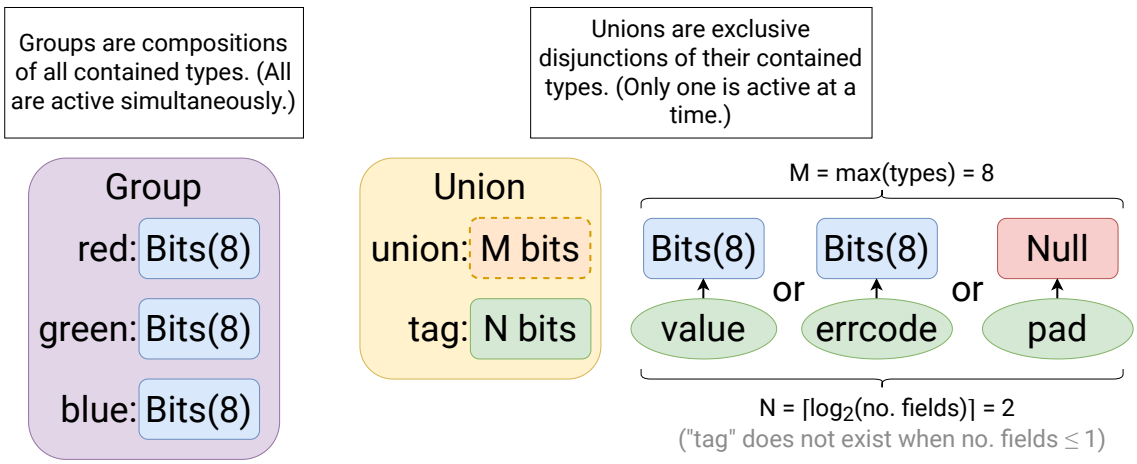


Figure 2.1: An illustration of the difference between Group and Union types.

The element-manipulating types alone can represent many data structures, for example:

- *Bits(N)* can be used to transfer primitive data types such as numbers, booleans and characters.
- A *Union* of *Null* and another type can indicate optional data.
- *Group*s can be used to directly represent records of data.

However, as Streams are also logical types, Groups, Unions and Streams themselves can carry further nested logical Streams, each with their own data and properties.

2.2.2. Streams

The Stream type adds a further layer of flexibility to these element-manipulating types. It does not only represent the physical stream and signals carrying the element-manipulating types, but also features properties for further describing data structures. Notably, Streams have a *dimensionality* property, which indicates whether the data being transferred is part of a sequence. In hardware, this is translated to a "last" signal; when this signal is driven high, it indicates that the data being transferred is the last element in a sequence, and a Stream with a higher dimensionality will have multiple last bits, to indicate nested sequences.

A *last* signal is typically used to reflect sequences or other kinds of variable-length data. Both AXI4-Stream and Avalon-ST lack Tydi's ability to assign multiple *last* bits to a transfer or element, however. This gives Tydi interfaces more flexibility in naturally reflecting different data types, or combining multiple variable-length data structures. Figure 2.2 illustrates how this can be used to better reflect a UTF-8 encoded string transferred as bytes; the inner dimension is used to represent UTF-8 characters, which can be between 1 and 4 bytes long, and the outer dimension is used to represent the string as a whole. Of course, UTF-8 itself already encodes whether a byte is part of a group making up a character, but this simplifies processing downstream, and alleviates the need for similar encoding on other data types. For instance, when representing a video with an arbitrary resolution, three dimensions can be used to indicate the end of a row, the end of a frame, and the end of the video overall respectively.

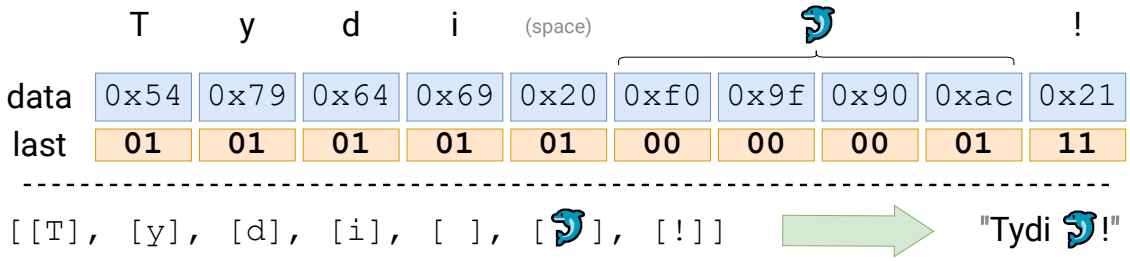


Figure 2.2: Using multiple last bits to transfer a UTF-8 encoded string as bytes, where the inner dimension is used to easily distinguish groups of bytes making up single characters.

Another noteworthy property is *direction*, which indicates whether a Stream flows in the same direction as its parent, or in reverse. This allows designers to express that certain Streams have a relation: As an example, a Group can have both a "Forward" and "Reverse" Stream to indicate that interdependent data is transferred between the sink and source, such as a memory address and the data retrieved from that address. Giving a parent and child Stream different directions can also be used to indicate that one Stream (direction) directly controls another; expanding on the previous example, consider a Stream which can be used to read and write to memory, but which prevents reading and writing simultaneously. One way to implement this is as follows: The parent Stream carries a Group with the fields "address" and "read_write", "address" is simply a Bits(N) type, but "read_write" is a Union carrying both a Forward "write" and Reverse "read" field. As illustrated by figure 2.3, this allows the parent Stream to control whether read or write is active by setting the Union's *tag*.

In addition to dimensionality and direction, Streams have properties for describing how transfers should be organized in space and time, the specifics of their implementation will be described in the next section:

- *Throughput* is a positive, rational number indicating how many elements are expected to be transferred per individual handshake, or relative to its parent Stream. The number of element lanes is a Stream's *throughput* multiplied by that of all parent Streams, rounded up to a natural number.

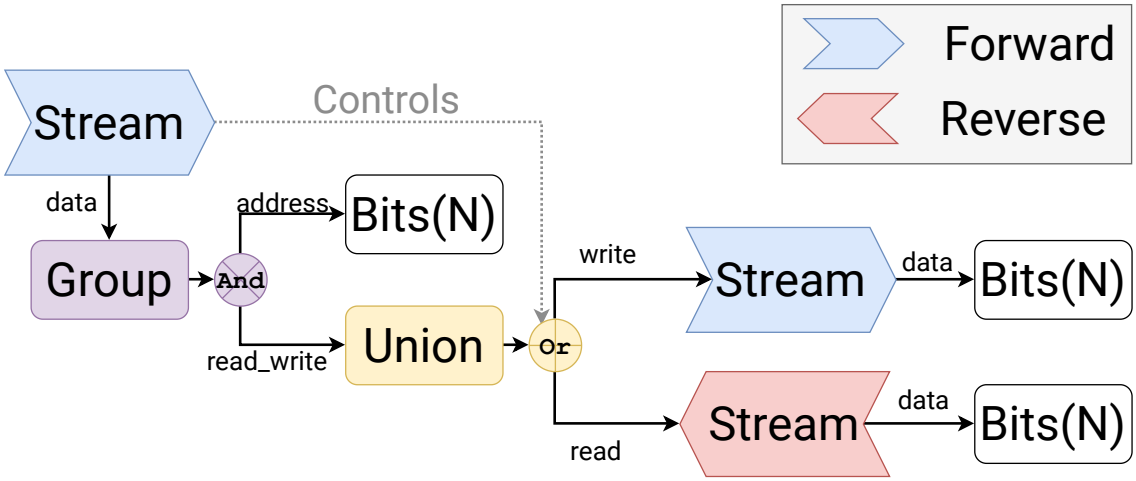


Figure 2.3: A Union of Streams with opposite directions allows a parent Stream to control whether data should be read or written.

- *Synchronicity* refers to how strong the relation between a child Stream and its parents are with regards to dimensional information. “Sync” indicates that for each element transferred on the parent, the child has a matching transfer, while “Desync” indicates that the child may have transfers of arbitrary size. Both options also have a “Flat” variant, which results in redundant last signals on the child being omitted.
- *Complexity* is a number which encodes guarantees on how elements of a sequence are transferred. In brief, lower complexities place more constraints on *source* streams, such as by requiring that transfers of sequences occur over consecutive cycles.
- A *keep* property can be used to ensure a *logical* Stream is synthesized into physical signals, as nested Streams may otherwise be combined into a single physical stream.

Finally, in the event these properties are insufficient for a use-case, Streams can also have a *user* signal carrying an element-manipulating type. This user signal can be used to provide additional information independent from transfers or clock cycles.

2.2.3. Physical Streams

Many of the properties described in the previous section have no impact on the kinds of data being transferred, but instead affect *how* it will be transferred. These changes are reflected in the *physical streams* [46] resulting from a logical Stream definition. A physical stream “canonically” consists of some variation on the following signals:

- *ready*: 1 bit, when driven high, this indicates that the sink device is prepared to accept transfers.
- *valid*: 1 bit, when driven high, this indicates that the source device is transferring valid data.
- *data*: $E \times N$ bits, carries an element-manipulating logical type (of size $0 \leq E < \infty$ bits), and may be composed of one or more data lanes ($1 \leq N < \infty$).
- *last*: $D \vee D \times N$ bits, indicates whether a particular transfer, or element, represents the end of one or more sequences. If complexity $C \geq 8$, there is a last signal/slice per element lane, otherwise, this signal refers to the entire transfer. Its size is equal to dimensionality D.
- *endi*: $0 \vee \lceil log_2(N) \rceil$ bits, the “end index”: Only exists when the number of element lanes $N > 1$, and complexity $C \geq 5$ or dimensionality $D \geq 1$. Indicates the end of all *active* element lanes in a valid transfer.
- *stai*: $0 \vee \lceil log_2(N) \rceil$ bits, the “start index”: Only exists when the number of element lanes $N > 1$, and complexity $C \geq 6$. Indicates the start of all *active* element lanes in a valid transfer. Can be combined with *endi* to form a range of active elements, but cannot be used to mark an entire transfer as inactive. (The value of *stai* must be smaller than or equal to that of *endi*.)

- *strb*: $0 \vee 1 \vee N$ bits, the "strobe" signal: When dimensionality $D \geq 1$, can be used to mark all of a transfer's elements inactive, as a single bit. When complexity $C \geq 7$, all element lanes N have an individual *strb* bit, allowing for individual element lanes to be marked inactive, rather than the ranges supported by the start- and end indices.
- *user*: U bits, this signal carries the element-manipulating type defined in the Stream's *user* property, its properties are entirely user-defined.

Any signals sized 0 are omitted entirely, and Tydi allows for the *ready* and *valid* signals to be omitted when the physical stream is always ready or always valid, respectively. Physical streams are not necessarily directly equivalent to logical Streams; this is a result of Tydi making Streams themselves logical types, allowing for nested Streams in a Stream's data property. As the *data* signal itself cannot represent a Stream, such logical Streams will be split into multiple physical streams.

The exact procedures for converting logical types into physical streams are defined in the Tydi specification as the *split*, *fields* and *synthesis* functions. In brief:

- Logical Streams are *split* into a list of named physical streams. Names of physical streams are based on potential field *Name*s of Groups and Unions, which are concatenated hierarchically as *Path Name*s; Path Names are emitted as Names joined by two underscores. As a root Stream or directly nested Stream is not part of a field, split Stream names may be empty. When two Streams are directly nested, they may be *flattened*, combining their properties into a single Stream (by for example multiplying their *throughput*s and determining an absolute *direction*).
- Non-Stream logical types are converted into *fields*, which are lists of named bitfields, based on their size. These fields eventually make up the *data* signal. As before, Names of Group and Union fields are used to determine these names, and are concatenated into Path Names. Likewise, a field name may be empty if it is directly part of a Stream's data or user property.
- The *synthesis* function converts all *split* Streams into a list of named $PhysicalStream(E, N, D, C, U)$ definitions:
  - E is the *element content*, derived from the *fields* function on a split Stream's *data* property.
  - N is the number of *element lanes*, which is equal to the split Stream's *throughput* $\lceil t \rceil$.
  - D is the physical stream's *dimensionality*, equal to that of the split Stream.
  - C is the physical stream's *complexity*, equal to that of the split Stream.
  - U is the *user content*, derived from the *fields* function on a split Stream's *user* property.

The *synthesis* function also accounts for defining separate, user-defined signals which flow in parallel to a physical stream (in addition to the *user* signal), but this is not currently relevant to the use-case of specifically generating Tydi-compliant interfaces. It is also worth noting that the output of the *split* function is not discarded after *synthesis*: The *direction* property is not part of physical streams, so must be retrieved from their respective Stream definition.

The Tydi specification also permits *alternative representations* of physical streams, bundling element types into aggregate/record types in the target language. For example, in VHDL, rather than simply using a bit vector to represent a Group's data over multiple lanes, it is possible to instead use a record type with field names corresponding to those of the Group's, and then using an array of this record type to represent the data signal overall. However, it recommends that any "outer" interfaces still use the "canonical" representation described at the start of this section, to ensure interoperability between potential IP blocks.

The *number of element lanes* N applies to the data signal, multiplying the total bit width defined by *element content* E. It does not apply to the *user* signal.

The *complexity* C affects how elements and sequences are organized over element lanes and consecutive lanes. Overall, a lower complexity imposes more restrictions on a source, in the inverse, this results in a higher complexity making it more difficult to implement a sink. As an example, a complexity of ≤ 2 requires that elements of an inner sequence are transferred over consecutive cycles by a source, while higher complexities allow it to stall independently from the sink. The specification currently defines 8 levels of complexity [46]. Table 2.1 illustrates the cumulative changes between complexity levels.

Complexity	Dimension Information	Sequence Transfer Requirements	Notes
1	Per transfer, on active data	*Entire sequence* must be transferred in consecutive lanes, with all possible lanes active, aligned to lane 0, over consecutive cycles.	Uses endi (end index) to pad the end of a sequence when using multiple element lanes. (E.g., when fitting a sequence of 2 elements in 3 element lanes.)
2	Per transfer, on active data	Innermost sequences (dimension 0) must be transferred in consecutive lanes, with all possible lanes active, aligned to lane 0, *over consecutive cycles*.	May set valid = '0' after an innermost sequence.
3	Per transfer, *on active data*	Innermost sequences (dimension 0) must be transferred in consecutive lanes, with all possible lanes active, aligned to lane 0.	May set valid = '0' after any transfer.
4	Per transfer	Innermost sequences (dimension 0) must be transferred in consecutive lanes, *with all possible lanes active*, aligned to lane 0.	May use strb = '0' with valid = '1' to transfer dimension information after a transfer with active data.
5	Per transfer	Innermost sequences (dimension 0) must be transferred in consecutive lanes, *aligned to lane 0*.	Does not need to fill all element lanes until done transferring an innermost sequence. (May split an innermost sequence of 2 elements over 2 transfers, even when 2+ lanes are available.)
6	Per transfer	Innermost sequences (dimension 0) must be transferred in consecutive lanes.	Can use stai (start index) to set the starting lane.
7	Per *transfer*	None	Can use strb (strobe) to set individual lanes inactive.
8	Per element	None	Has a last signal per element lane.

Table 2.1: The changes between Tydi's *complexity* levels. For clarity, constraints which will be omitted next complexity level are *italicized* and marked red, and any potential replacement constraint or property is underlined.

Figure 2.4 illustrates how a higher *complexity* allows for transfers to be organized differently. When transferring `[[H, e, l, l, o], [W, o, r, l, d]]`, at *complexity* $= 1$ all elements must be aligned to the first lane, *last* data is asserted per transfer, and all data must be transferred over consecutive cycles and lanes. At *complexity* $= 8$, there are no requirements for how elements are aligned, transfers may be postponed (asserting *valid* low), and *last* data is asserted per lane, and may be postponed (using an inactive lane to assert *last* for a previous lane or transfer).

The *synchronicity* and *keep* properties of the original logical Stream type are not directly reflected by the physical stream, but do affect how they are synthesized and how transfers are expected to behave. As mentioned in the previous section, *synchronicity* s indicates whether the transfers of a child Stream are constrained by its parent Stream, and only applies if the child Stream's *dimensionality* $d > 0$.

- If $s = Sync$, for each element in the parent Stream, one sequence must be transferred on the child Stream. E.g., both the parent and child Stream have *dimensionality* $d = 1$, and the parent Stream features a *Group(a: Bits(1), b: Stream(data: Bits(1), ...)*, where *b* is this child Stream. To transfer the sequence `[ (a: 1, b: [1, 0, 1] ), (a: 0, b: [0, 0]) ]`, the parent Stream will use its *data* signal to transfer the value of *a*, while the child Stream will transfer the sequence on *b* independently, but before the next transfer on the parent Stream. The

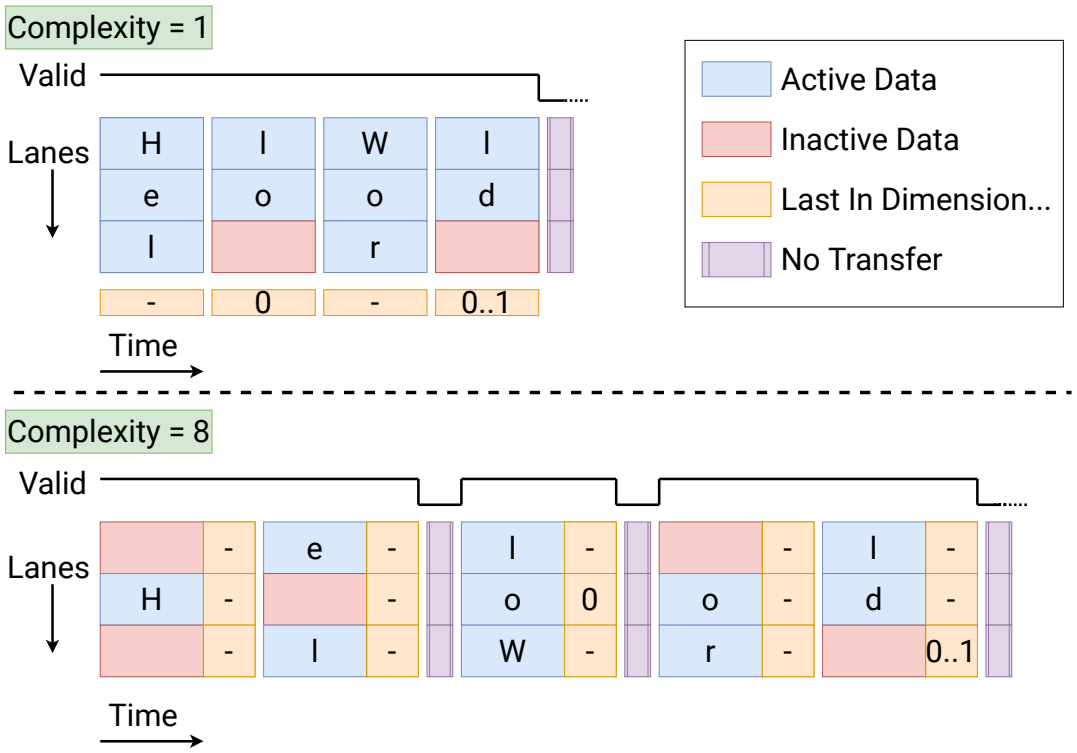


Figure 2.4: Streams determine which signals are used and valid to organize elements in transfers, and how transfers are organized over time.

child physical stream will have $d = 2$, and duplicate the *last* information of the parent transfer. (Resulting in a transfer `[ (a: 1), (a: 0) ]` on the parent, and `[ [1, 0, 1], [0, 0] ]` on the child.)

- If $s = Flatten$, the child Stream will behave as *Sync*, but the child physical stream will not duplicate the the (redundant) *last* information of the parent physical stream. (Hence, the previous examples transfer would become `[1, 0, 1], [0, 0]` for the child.)

- If $s = Desync$, the child Stream will still inherit the parent's dimensionality (i.e., if the parent and child have $d = 1$, the resulting child physical stream will still have $D = 2$), but its transfers will not be constrained by those on its parent Stream. I.e., its transfers will still be shaped as `[ [ ... ], [ ... ] ]`, but they will have no (apparent) relation to the parent Stream's transfers. This does not preclude designers from using the *user* signal to provide context, instead.

- $s = FlatDesync$ is the equivalent of *Flatten* for *Desync*, in that it does not duplicate the parent Stream's dimensionality for the child Stream. In effect, the child Stream now has no apparent relation to the parent Stream.

In the event two Streams are directly nested, they are *flattened* by default in the *split* function: Rather than a parent Stream with no *data* signal, whose only purpose is to transfer the outer *last* bits, the function will instead produce a single Stream combining the dimensionalities of parent and child. This process will only occur when both the *data* and *user* signals of the parent would otherwise be empty. The *keep* property mentions before prevents a Stream from being *flattened* regardless of its data and user signals being empty. This partly avoids issues with the parent and child Stream not having unique names (which would otherwise be derived from the Group or Union field name for the child Stream), requiring one replace the other, though the use of the *keep* and *user* properties can nonetheless introduce this issue, as discussed later in Section 6.1.1.

2.3. Alternatives

This thesis aims to simplify development of streaming dataflow accelerator designs for big data analytics by improving reusability and making it easier to connect different components through the use of the Tydi specification. Outside of the application of Tydi, however, these are existing problems for which solutions are already being developed, and which can ostensibly be further adapted to suit more specific needs. This section lists alternative solutions which have been considered, and how they align with the goals of the IR and toolchain.

2.3.1. Hardware Description Languages

Strictly speaking, it would be possible to implement hardware accelerators with Tydi interfaces in existing hardware description languages manually, rather than building a toolchain to emit them. It would also be possible to limit the scope of the toolchain, by directly compiling from a more abstract front-end language to an existing HDL, or by providing a standard library of components, or by generating templates in a target HDL. In particular, there are a number of languages which promote reuse and/or are suited to expressing streaming data processing efficiently.

Lime [10], StreamIt [42] and HPVM [25] are able to express streaming data processing, with StreamIt in particular being designed for this purpose alone. StreamIt and HPVM are not explicitly designed as hardware description languages, but nontheless map very well to hardware (using a suitable synthesizing compiler in StreamIt's case [17]).

Chisel [11], FIRRTL [23] and LLHD [40] are more general HDLs which aim to simplify expression of hardware and hardware interfaces, and promote reuse. All have since been incorporated into the CIRCT (Circuit IR Compilers and Tools) project [27] as different parts of an overall toolchain.

The IR is, first and foremost, an extension of the existing Tydi interface specification. In that it codifies the rules for designing and connecting interfaces, how to define data types, and how to transfer data. As such, the goal is not to outright replace any of the aforementioned languages, but serve a complementary role by expressing Tydi streams and Streamlets as efficiently as possible. It also aims to propagate high-level information down to the languages a backend might emit, including documentation. By intentionally limiting the IR's scope compared to conventional HDLs, it should also serve as an intermediary for very different kinds of front-ends. For instance, its focus on composable interfaces can also be applied to more visually-oriented design tools, such as Vivado's "block design" view discussed in the next section.

2.3.2. Design Tools and High-Level Synthesis

At the same time, there are multiple ongoing efforts to improve the tools used for designing such hardware accelerators, in the form of new hardware description languages [11, 23], intermediate representations [40] and compilers [27], high-level synthesis based on software programming languages [30], and more general program representations for heterogeneous systems [34, 25].

High-level synthesis can help programmers who are unfamiliar with HDLs and hardware design in general to more quickly implement their ideas. This is especially relevant when the goal is to use a hardware accelerator to speed up an algorithm which was previously implemented in software, as it allows for these ideas to be translated more easily. However, the ideas expressed in HLS rarely propagate very far to the resulting hardware descriptions and simulations, making it more difficult to perform verification and analyze issues and targets for optimization from the same perspective. Some of this can be addressed by also building simulators for the high-level language, and introducing additional directives and macros to better match hardware, as in SystemC [2].

Comprehensive **design tools** may also incorporate ways to encourage and improve avenues for reuse. For instance, Vivado includes a "block design" view which allows for individual components (IP blocks) to be connected using standard AXI4(-Lite/-Stream) or *ap_fifo* interfaces, as shown in Figure 2.5. This is combined with Xilinx's existing HLS tools and IP block library [3] to allow for integrating these components as well, such as through pragmas (directives) indicating particular parameters or variables in a high-level language should correspond to a given interface type [4].

Such tools are undoubtedly easy to use, especially when IP blocks surface configuration items to allow them to be modified from the same visual block design interface, as in Figure 2.6. The reusability enabled by such tools is less clear, however; they are proprietary, and any components designed for them must adhere to the constraints set by Vivado to enable the most desirable features, such as easily

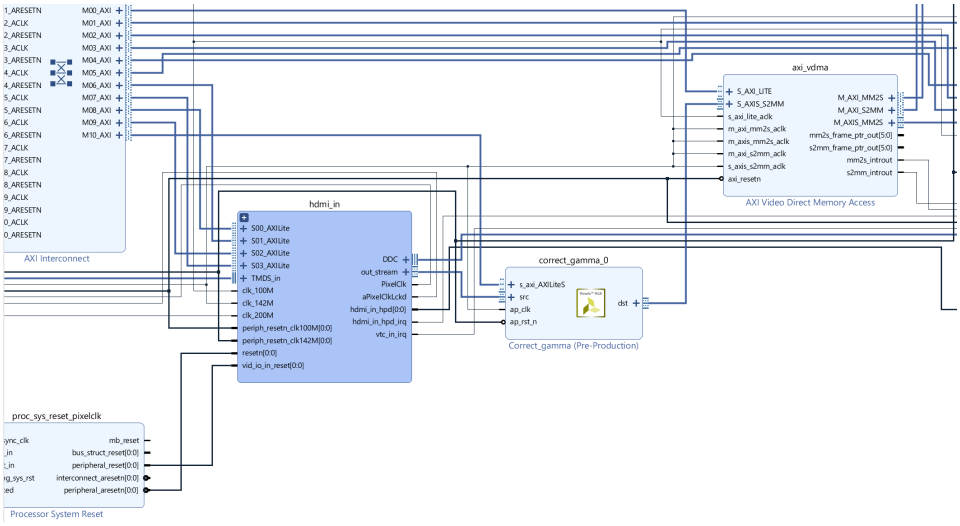


Figure 2.5: Using Vivado's "block design" interface to connect individual IP blocks using, among others, AXI4-Lite interfaces.

connected interfaces and exposed configuration properties. It can very much promote reuse, but only within a closed ecosystem.

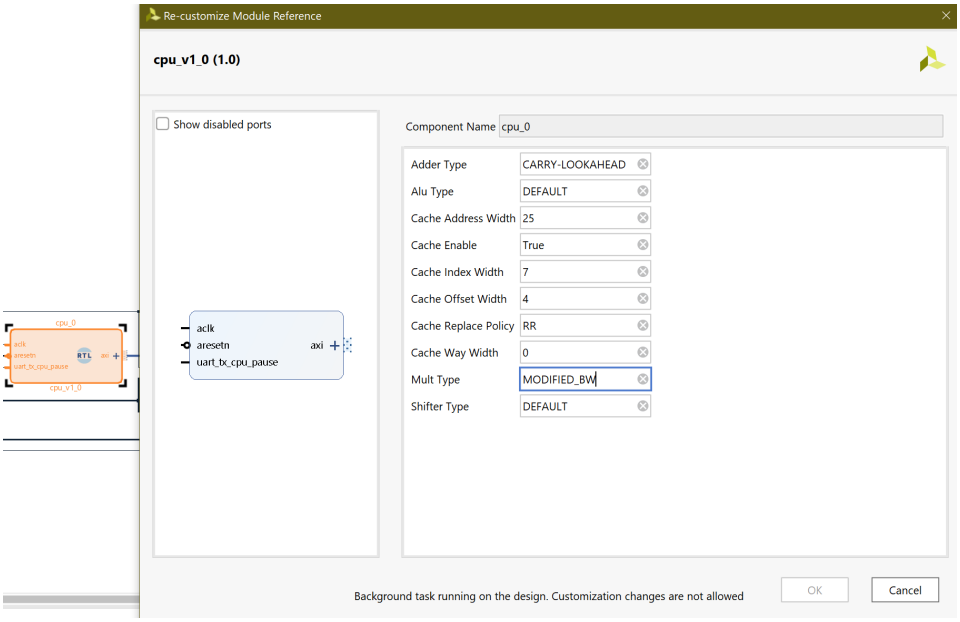


Figure 2.6: Using Vivado's "block design" interface to configure properties of certain IP blocks.

2.3.3. Frameworks and Embedded Domain-Specific Languages

One approach to building hardware accelerators for a specific domain is to instead surface hardware-oriented language and/or methods within an existing language. For instance, Fleet [43], S2FA [48] and Melia [47] all promise to easily integrate FPGA accelerators into existing software data processing environments. These improve ease-of-use by bringing the accelerator designs closer to their intended targets, and enable reuse of accelerator "kernels" throughout a project or over different projects. This approach can be quite effective, but is too specific to benefit hardware accelerator design reuse across different domains and potential processing frameworks.

3

Intermediate Representation: Composition

The primary purpose of the intermediate representation is to define Tydi Streams and Interfaces, and use these to compose Streamlets. This chapter describes the ways this functionality is implemented, its use-cases, and the considerations which have gone into the IR's design overall.

To illustrate the various IR concepts described in this chapter, there are listings in TIL (Tydi Intermediate Language) a grammar for the IR which was designed (and can be parsed) as part of the overall prototype toolchain. For more details on TIL and its implementation, see Section 5.2.

3.1. Type Declarations and Interface Design

3.1.1. Type Declarations

As described in Section 2.2, Tydi features 5 "logical types", with Groups, Unions and Streams themselves having fields or properties containing these logical types. The IR must be able to represent definitions of all types, account for being able to nest types, and enable comparison between types to ensure compatibility between interfaces. Listing 3.1 demonstrates expressions for the four "element-manipulating" types.

```
Null

Bits(7)

Group (
  field_name1: Bits(2),
  field_name2: Bits(7),
)

Union (
  field_name1: Bits(7),
  field_name2: Group (
    field_name1: Bits(2),
    field_name2: Bits(3),
  ),
  field_name3: Null,
)
```

Listing 3.1: Expressions for element-manipulating logical types in TIL

The "stream-manipulating" logical type, *Stream*, is defined in a similar way. As it is the only type with explicit properties, however, it also features a number of *default* values for some of these properties when they are omitted, as explained in the comments in Listing 3.2. Note that Streams can be used in the exact same way as any other logical type, in that it can be used as a Group or Union's field,

or another Stream's *data* property; the only exception is the Stream's *user* property, which may only contain element-manipulating types.

```
Stream (
  data: Bits(8),
  throughput: 2.0,     // 1.0 by default
  dimensionality: 0,
  synchronicity: Sync,
  complexity: 4,
  direction: Reverse, // Forward by default
  user: Bits(2),      // Null by default
  keep: true,         // false by default
)
```

Listing 3.2: Expression the stream-manipulating logical type in TIL

As tracking deeply nested types can become convoluted for a compiler emitting to the IR, and make the output hard to read, the IR also features the ability to declare types within a *namespace* and give them a unique identifier to track them by. Listing 3.3 showcases how these identifiers can be used; identifiers serve as an alternative to explicit type definition expressions.

```
namespace namespace_name {
  type bits_type_name = Bits(7);

  type group_type_name = Group (
    field_name1: Bits(2),
    field_name2: bits_type_name,
  );

  type union_type_name = Union (
    field_name1: bits_type_name,
    field_name2: group_type_name,
    field_name3: Null,
  );

  type stream_type_name = Stream (
    data: union_type_name,
    dimensionality: 0,
    synchronicity: Sync,
    complexity: 4,
  );

  type parent_stream_type_name = Stream (
    data: stream_type_name,
    dimensionality: 1,
    synchronicity: Sync,
    complexity: 4,
  );
}
```

Listing 3.3: Statements declaring logical types in TIL

So as not to diverge from the Tydi specification, which does not feature identifiers as a property of logical types, these identifiers only exist as a property of the namespace, and should not affect the output of a compiler for the IR. That is to say, a *Bits(8)* is always the same as any other *Bits(8)*, regardless of the identifier it was given, and whether it was given an identifier at all. The merits and demerits of this approach are elaborated on in Sections 3.1.3 and 6.2.

In order to represent each type definition in the query system, each type definition is *interned*: Each distinct type is stored as an immutable entry in memory and tracked using a unique identifier. This has a number of advantages:

- It reduces the amount of data stored by the query system. (No need for multiple entries in memory for identical type definitions.)

- Types which contain nested types as fields or properties only need to contain the identifier, instead of a copy of the definition or a direct reference to memory.
- Comparison between types is trivial, as it is only necessary to compare intern identifiers: If types are different, their identifiers will be different, as well.
- The query system will not have to track different kinds of type expressions: Whether they are namespace identifiers or direct definitions, each type ultimately becomes an intern identifier.

3.1.2. Interfaces as Contracts

As Section 2.2 would suggest, Tydi's types can convey a significant amount of information; not just what data is transferred, but also how it is transferred, and how sequences of elements relate to one another. In effect, a sufficiently detailed Stream definition can be treated as a *contract* between components (and in a sense, designers) on how a stream of data will be implemented.

The intermediate representation builds on this when declaring *Interfaces*. In its simplest form, an Interface represents a collection of ports on a component (Streamlet), each of which carries a logical Stream either into or out of the component. Any streamlet must have an interface; as a result, all streamlet definitions can be subsetted into interfaces, as shown in Listing 3.4. By default, backends are not expected to emit interface declarations which are not part of streamlet definitions, and the names of interface declarations should not have any effect on the resulting output. It is however allowed to define a streamlet without any implementation, consisting only of an interface - defining an interface without any ports is also allowed.

```
interface my_interface = (a: in stream, b: out stream);

streamlet my_streamlet = my_interface;

streamlet my_impl_streamlet = (
  a: in other_stream,
  b: out stream
) {
  impl: ...
};

streamlet subsetted_streamlet = my_streamlet {
  impl: ...
};
```

Listing 3.4: Statements declaring interfaces and streamlets in TIL

However, each Interface and its ports may also feature *documentation*. Distinct from comments on a grammar, documentation is an actual property of a port or interface, and is expected to be implemented by a backend, typically by generating matching comments on the related output. Documentation being propagated from higher-level descriptions to the actual computation-oriented design tools that the IR complements is primarily useful when either implementing a component based on an interface template, or when trying to identify how physical signals relate to their abstract definition.

While Tydi's Streams assume a single clock and reset signal, which together make up their clock and reset domain, regardless of how many physical streams they are composed of, the ports of an Interface do not need to rely on the same clock and reset signals. Instead, an Interface may have one or more uniquely named *domains* which represent a clock and reset signal, each of which is associated with one or more of the Interface's ports.

Subsequently, while the intermediate representation does not feature the ability to define a specific clock or how a reset signal should be handled, designers can use these domains to ensure multiple clock and reset signals are available on a component, and that ports which belong to different domains are not directly connected. In the event no domain is specified on the Interface, a default domain is instead created and assigned to all ports, as Tydi currently only defines Streams in the context of a clock.

It is worth noting, as a recommendation for future work, that the use of ready-valid signals *should* make it possible to represent fully asynchronous (clock-less) micropipelines [41] using Tydi. Even if the specification currently assumes the existence of a clock, many of the timing constraints enforced

through it can be replaced by the ready and valid signals serving as events for forward and reverse propagation.

3.1.3. Compatibility

The ports of Interfaces are compatible with one another when they have the same logical type, appropriate directions (for each physical stream, there is a source and matching sink), and the same (clock) domain.

A domain in Tydi and the IR consists of a clock and reset signal; while the reset signal does not have any specific constraints to the clock signal, the reset behavior, requiring the *valid* and *ready* signals to be driven low during a reset, is constrained by it. The Tydi specification generally assumes a single clock and reset signal, but this only applies in the context of a Stream and its compatibility with other Streams. Therefor, it is possible to surmise that a (clock) domain and a Stream are intrinsically linked; the compatibility of two interfaces using Tydi Streams is contingent on them being part of the same clock domain.

To reflect these properties, the IR assumes a single “default” domain, but allows for the definition of additional/alternative domains and for linking them to specific interface Streams; the actual clock speed is irrelevant to compatibility, only whether a designer indicates something is a different domain.

```
// As no domain has been defined, the ”default” is assigned
(a: in stream, b: out stream) {
  impl: {
    // Sharing one domain, these are compatible
    a -- b;
  }
}

// Declaring new domains removes the default domain, and
// requires that they are assigned to individual ports.
<'a_domain, 'b_domain>(
  a: in stream 'a_domain,
  b: out stream 'b_domain,
) {
  impl: {
    // As these now have different domains, these are incompatible
    a -- b;
  }
}
```

Note that while types in the IR may be defined with identifiers, these identifiers are not a property of the logical type in question, and only exist within the namespace. This choice was made to restrict the IR to properties defined in the Tydi specification.

As a result, types with different names but otherwise identical properties are fully compatible; on an abstract level, this can be interpreted as a kind of implicit casting between types. Although when evaluating this with respect to readability of backend output, discussed in Section 6.2, and in light of the potential added value of a stricter type system, this approach may need to be reconsidered in the future. An alternative approach might make identifiers an intrinsic property of types, and separately support type aliases for functionality similar to the current behavior - depending on the language being targeted, such aliases could even be propagated to the backend.

However, while type identifiers are not currently relevant to compatibility, *field* identifiers are an actual property of the Group and Union types. Hence, a `Group(a: Null)` is not compatible with a `Group(b: Null)`, regardless of whether they are physically identical.

```
type bits8 = Bits(8);
type byte = Bits(8);
// bits8 and byte are compatible

type a_group = Group(a: bits8);
type b_group = Group(b: bits8);
```

```
type group_a = Group(a: byte);
// a_group and group_a are compatible, but neither are compatible with b_group
```

Finally, while *complexity* is a property of the Stream type, the Tydi specification does conditionally allow Streams with different complexities but otherwise identical properties to be connected. Specifically, a physical *source* stream may be connected to a *sink* if its complexity is equal to or lower than that of the sink. Note however that this applies to physical streams: logical Streams do not have a notion of sinks and sources, and may contain child Streams which flow in reverse directions, resulting in them containing both sink and source physical streams.

As such, the IR considers the Streams of ports incompatible when their complexity is not identical. While the process of connecting compatible physical streams can be optimistically automated to improve reuse, as discussed later in Section 3.3.3, designers should generally strive for a shared, normalized complexity between Streams.

3.2. Component Composition and Implementation

In addition to Interfaces, the IR introduces the ability to declare components, referred to as *Streamlets*. These Streamlets consist of an Interface and optionally an Implementation. In effect, there are two different kinds of Implementation for a Streamlet: a *structural* implementation, which can be used to combine instances of streamlets into a larger design, and a *link* to an implementation of behavior in the target language or format.

Streamlets are the intended output of a project; Types, Interfaces and Implementations are not expected to be included in a backend's emissions unless they are part of a Streamlet, but can be shared between IR projects.

As Streamlets always have an Interface, they can be *subsetted* to Interfaces, which can be used to express alternate implementations of the same component, e.g. when versioning a component or when substituting one for the purposes of testing as described in Section 4.3.

3.2.1. Structural Composition

As the goal of both Tydi and the IR is to improve compatibility and reuse of primitive components, the IR features the ability to connect Streamlets to one another. The IR refers to this as a *Structural* implementation.

Structural implementations can contain *instances* of Streamlets and connections between ports of Streamlets. Instances consist of a local name and a reference to a Streamlet declaration, the ports of their interfaces are assigned separately through connections. If the parent interface has named domains, these must also be assigned to the Streamlet instance.

```
// Creating an instance with a default domain
instance_name = streamlet_name;

// Creating an instance and assigning domains
instance_name = streamlet_name<'parent_domain_name>;
// Or:
instance_name = streamlet_name<'streamlet_domain_name = 'parent_domain_name>;
```

Listing 3.5: Statements for instantiating instances of Streamlets in TIL.

Connections can be created between the ports of both Streamlet instances and the containing Streamlet which is being implemented, and require both ports to have identical types and clock domains (for the reasons described in Section 3.1.3). Connections are explicitly not "assignments", as the direction of a port is already known, and there is not necessarily one overall direction for a Stream type due to the possibility to define Streams which are *Reversed* (such as when representing request and response streams). Hence, the *source* and *sink* between two ports of a connection is determined during lowering for each resulting Physical Stream.

```
instance_name.instance_port1 -- instance_name.instance_port2;
instance_name.instance_port -- parent_port;
```

```
parent_port1 -- parent_port2;
```

Listing 3.6: Statements for connecting ports in TIL.

By default, the IR requires that each port of each Streamlet is connected to exactly one other port. Leaving ports unconnected is against the Tydi specification, which requires that a default signal is driven for omitted signals [46]. While HDLs such as VHDL and Verilog support one-to-many and many-to-one connections at a signal-level, these are not allowed by the IR due to the fact that ports represent Streams with handshake signals, which would need to be combined.

```
streamlet example_streamlet = <
  'parent_domain1,
  'parent_domain2,
> (
  parent_port1: in stream 'parent_domain1,
  parent_port2: out stream 'parent_domain1,
) {
    impl: {
        parent_port1 -- parent_port2;

        // dom_example has two domains, one for ports a and d, and one for port b and c
        different_domains = dom_example<'parent_domain1, 'parent_domain2>;
        different_domains.a -- different_domains.d;
        different_domains.b -- different_domains.c;

        // By assigning them the same domain, a and b and c and d can nonetheless be connected
        same_domains = dom_example<'parent_domain1, 'parent_domain1>;
        same_domains.a -- same_domains.b;
        same_domains.c -- same_domains.d;

        // For clarity, when assigning domains it's also possible to specify
        // which domain of the instance is being assigned to, rather than using their order.
        explicit_doms = blank_doms<'c = 'parent_domain1, 'a = 'parent_domain2, 'b ='parent_domain2>;

        // It's also possible to mix named assignments with ordered assignments,
        // provided the named assignments succeed all ordered assignments.
        mixed_assignments = blank_doms<'parent_domain2, 'c = 'parent_domain1, 'b ='parent_domain2>;
    }
};
```

Listing 3.7: A full Structural implementation in TIL.

While combining the *ready* signals of multiple sinks could be achieved with simple logical *and* expressions for a one-to-many connection, combining multiple transfers in a many-to-one connection has no clear, universally applicable, solution. Even the aforementioned one-to-many implementation is not universal, as some designs may call for only one of the many to alternately accept transfers. Finally, as a connection does not necessarily have a single direction, a one-to-many connection between ports may well contain physical many-to-one transfers.

Instead, the solution to unconnected and one-to-many ports would be to explicitly define their behavior. In the current implementation, that means designing specific Streamlets for this purpose; but as there is a common subset of expected behavior (drive default, sink and ignore input, transmit transfers to all in a one-to-many source-to-sink configuration, etc.), the ultimate goal would be to provide intrinsics which automatically implement this behavior, as described in Section 3.3.3.

3.2.2. Linked Implementations

The intermediate representation intentionally omits expressions for implementing or simulating arbitrary behavior of components. Designing a language or set of expressions for functional hardware design and simulation is a difficult problem which is already being addressed by many researchers and organizations, as explained in Sections 2.1 and 2.3.1. Instead, "behavioral implementations" in the IR exist only as *links* to directories, which contain the relevant code in languages more suited for expressing behavior.

How these links are used is left up to the backend, though a simple use-case would be to create or copy a file in the target output language based on the Streamlet's name. As these are directories, multiple such files can exist side-by-side for different targets, and implementations do not need to be constrained to a single file; a linked directory could even be used to refer to a project or library consisting of multiple files, provided this exposes the Interface of the Streamlet being implemented.

It is worth noting that linked implementations are not to be treated like *imports*: A linked implementation is still fundamentally part of the IR project, and should be included with its sources/output. A link

should not refer to a common library directory, and linked implementations should use relative paths (relative to the project root), rather than absolute paths. Provided front-end languages use these same constraints, this has the added benefit of ensuring projects can be easily shared between developers and tracked in version control systems.

In TIL's grammar, links are simply written as path strings enclosed in double quotes. Whether a path uses valid formatting and characters is determined by the query system, though the query system will not verify whether the directory actually exists; it is up to the connected backend (and its potential configuration) to decide whether to treat a non-existent directory as a failure condition, or to instead create the directory if it does not exist.

```
streamlet example_streamlet = <
  'parent_domain1,
  'parent_domain2,
> (
  parent_port1: in stream 'parent_domain1,
  parent_port2: out stream 'parent_domain1,
) {
    impl: "./path/to/a/directory"
};
```

Listing 3.8: A linked implementation in TIL.

Note that, as shown in Listing 3.8, linked implementations still require a complete Streamlet definition, consisting of a name and an interface in addition to its implementation. As mentioned before, the Streamlet name is used to let the backend determine which file or set of files to use from a directory. It also ensures that its interface definition can be included in the target language's project structure, and instances can be created inside structural implementations. E.g., when emitting to VHDL, the interface definition and name are used to create a component definition in the emitted project's *package* file(s), and these components are used inside the emitted architectures of structural implementations. The interface definition and name can also be used to automatically generate a correctly named and structured *template* in the linked directory, if the target does not already exist.

Figure 3.1 illustrates how linked implementations fit within a partial toolchain and workflow, consisting of Streamlets, structural implementations and tests defined in the IR, combined with behavior defined in a target language (VHDL, in this example) by a suitable backend. Not pictured are tools for simulating the testbenches produced by the backend, further passes on the output, nor any potential frontend language.

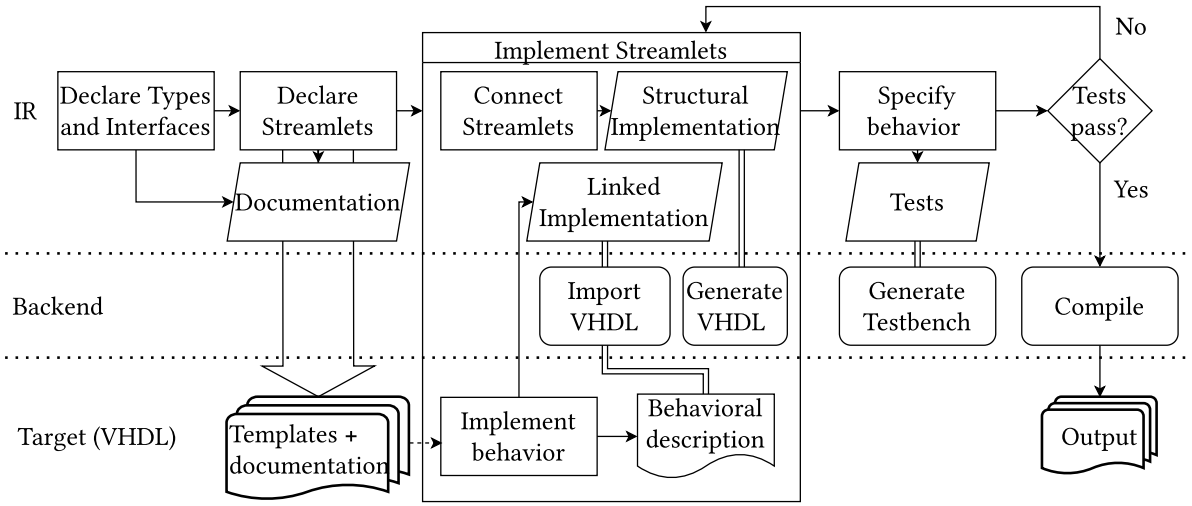


Figure 3.1: An example workflow, demonstrating how Streamlets are implemented using the IR, a suitable backend, and behavior defined in the target language.

3.3. Recommendations for Language and Compiler Features

While the previous sections cover the primary functionality of the IR, which is to describe types, interfaces, Streamlets, and implementations of Streamlets, there are still a number of language features

which can improve or simplify the IR's ability to describe designs overall. To preface these language features, however, it is important to note that these are not (solely) intended to make the IR easier to write or read by humans; as this is an IR, it will primarily be emitted as output of a more ergonomic, front-end language and read by a compiler via the query system. As such, language features in the IR should not be "syntactic sugar", which is ultimately different styles of expression for existing language constructs. New features should meaningfully translate to something a compiler can directly implement, and should not expand to existing IR constructs (except potentially as a fallback). By extension, these features should be possible to implement in common HDLs.

3.3.1. Type Parameters

Type parameters allow for the creation of variable types (also known as *generic types*), by parameterizing specific properties and allowing variations upon the type to be instantiated as needed. Listing 3.9 shows what such type parameters might look like in TIL. Such type parameters can be useful in simplifying and organizing the expression of related interfaces which share many properties, for example, if a certain (collection of) Streams represents a memory interface, its bus width can be parameterized.

```
type generic_stream<
  a: bitcount,
  b: type,
  c: complexity,
  d: throughput,
  e: keep,
  f: direction,
> = Stream(
  data: Group(a: Bits(a), a2: Bits(a), b: b),
  complexity: c,
  throughput: d,
  keep: e,
  synchronicity: Sync,
  dimensionality: 0,
  direction: f,
);

type concrete_stream = generic_stream<6, Bits(5), 4, 3.14, false, Forward>;
```

Listing 3.9: Theoretical grammar for generic types in TIL.

The omission of type parameters was not purely due to implementation time constraints, but due to the properties of such parameters being subject to debate, relating to the quality described in the preface: Language features should be possible to translate by a backend to their target language. There are two distinct ways to implement generic types in the IR:

1. Generic types are evaluated before compilation, either by the parser or by the query system before a compiler requests a definition. Only concrete types exist in the resulting output.

2. Type parameters are stored as properties on the IR, and can be used during compilation to translate to equivalent language features.

The first option is easiest to implement; as suggested, evaluating generic types to concrete types does not even need to be a feature of the IR itself, but can be handled by the parser for TIL. This makes them similar to identifiers on the namespace: A feature for tracking and reusing types, rather than something to be propagated to a compiler's output. For instance, one could save time on implementing multiple Streams with the same complexity, synchronicity and dimensionality by making the relevant properties type parameters.

However, like type identifiers, these generics are not *strictly* required to be part of the IR or even TIL: A front-end language and compiler can define generics themselves, and simply expand these to concrete types for the IR.

The second option provides significantly more added value, but would also be significantly more difficult to implement. After all, while many (hardware description) languages which could be targeted may feature similar type parameters, not all languages do, and not all type parameters are equally

flexible. For example, while VHDL(-93) has type parameters for determining the size of certain arrays, and newer versions (2019) even include support for types as parameters (like the “b” parameter in Listing 3.9), there are no parameters which omit ports altogether (when passing a Tydi Null, changing the *complexity*, or setting the *keep* property from true to false) or change the direction of ports (the “f” parameter in the example Listing).

There are a few potential ways around these limitations:

1. The IR can be limited in which properties can be parameterized (e.g., only *bitcount* and *throughput* can be parameterized).

2. A compiler using the query system can selectively request certain parameters to be evaluated ahead of time.

3. Require that compilers for the Tydi IR must support all generic parameters.

Each solution is ultimately flawed in some way, in that the first solution does not account for languages which lack any type parameters, and the third solution omits a large number of HDLs. The second solution’s flaws are less obvious, this solution means that functionality is at least equivalent to the alternative interpretation of generics (always evaluating to concrete types), but also greatly limits their potential; it requires that it *must* be possible to evaluate any generic type ahead of time.

This constraint means that designers cannot define generic Streamlets or Interfaces, or cannot use them effectively as instances. Additionally, to convert otherwise entirely generic types (and/or Streamlets) into concrete versions, the IR will have to generate unique names for each variant. E.g., an instance may simply be defined as `streamlet_name<a = 3, b = 4>`, where *b* is a bitcount and supported, but *a* is the complexity level and not supported: To resolve this, the query system returns a `streamlet_name__a_is_3` with a bitcount type parameter, and any number of other combinations.

Though with that said, it is still possible to make a recommendation for maximum functionality:

- Type parameters should be a part of the IR, and not evaluated beforehand. As the minimal possible outcome is one which matches functionality with pre-compilation evaluation.

- All properties, and types themselves, should be possible to use as parameters. As different target languages have different limitations.

- Compilers should be able to indicate which parameters they support, at which point the query system should do one of the following, potentially based on further configuration:

    1. Fail if the project contains type parameters which the compiler/target language does not support.

    2. Only return types and Streamlets which feature the supported type parameters. (If no type parameters are supported, only return concrete types and Streamlets.)

    3. Generate uniquely named variants of types and Streamlets which are concrete (have fully-defined parameters), but feature unsupported type parameters.

3.3.2. Generation

As a more general language feature for structural implementations, the IR could expose forms of *generation*. Specifically, generating multiple instances of the same Streamlet definition in *arrays*, and generating connections between such instances in *for loops*. For example, rather than expressing a number of instances with unique names, such generation would leverage the target language’s ability to generate similar arrays:

```
a_1 = streamlet_decl_a;
a_2 = streamlet_decl_a;
a_3 = streamlet_decl_a;
b_1 = streamlet_decl_b;
b_2 = streamlet_decl_b;
b_3 = streamlet_decl_b;
a_1.out_port -- b_1.in_port;
a_1.in_port -- b_1.out_port;
a_2.out_port -- b_2.in_port;
a_2.in_port -- b_2.out_port;
a_3.out_port -- b_3.in_port;
a_3.in_port -- b_3.out_port;
```

```
a[3] = streamlet_decl_a;

b[3] = streamlet_decl_b;

for i in 0..3 {
  a[i].out_port -- b[i].in_port;
  a[i].in_port -- b[i].out_port;
}
```

Listing 3.10: Arrays of instances and loops for connections (right) compared to equivalent, explicit instances and connections (left).

These forms of generation are relatively safe to include as part of the IR, because many common HDLs already support equivalent functionality:

- VHDL provides `for ... generate` to generate multiple *port map*s and connect arrays of signals based on an index. [19, Section 11.8]
- Verilog likewise allows for modules to be instantiated in `generate for ...` loops. [18, Section 12.4]

By adding support for such loops, it is possible to generate more readable output: If a front-end allows for the expression of (for) loops, propagating these loops to the IR and the eventual target language better reflects the designer's intentions.

In the event that a language does not feature compatible constructs, it is possible to safely expand these loops to explicit instantiations and connections in the IR. This can be performed as a fallback function of the query system, rather than requiring a compiler to implement it, and can simply generate instatiations with index numbers as part of a "Path Name" (which normally reflect namespaces or nested fields of logical types). To expand on why this is safe: The Tydi specification (and by extension the IR) does not allow for *Name*s to start with numbers, and normally requires *Path Name*s to consist of valid *Name*s; by using a number as part of a *Path Name* for this expansion, we are guaranteed unique output names. (Conversely, directly appending numbers to *Name*s could result in conflicts with otherwise valid *Name*s.)

Combining generation with type (and specifically Streamlet definition-bound) parameters can also be incorporated into the IR, but does not necessarily feature equivalent constructs in potential target languages, and may not be safely expanded for the same reasons as expressed in the previous section.

3.3.3. Intrinsics

While the intermediate representation does not support expressing *arbitrary* functionality, there is arguably a subset of functionality useful for implementing Tydi-based components and streaming dataflow designs in general. For general design purposes, small components which aid with building pipelines and ensuring consistent parallel operation, such as slices, buffers and synchronizers, are relatively simple in their implementation and commonly used. In the specific context of Tydi, there are are number of limitations enforced by the specification and IR which can be addressed somewhat easily; for instance, all ports of an interface must be connected - to address this, a designer needs to drive a default or constant value to this port, or simply indicate it is unused (by driving ready and/or valid low).

Hence, there is cause to establish a minimal, portable set of intrinsic functions, or *intrinsics*, to be implemented by any backend. Specifically, intrinsics should only cover commonly used, simple functionality which cannot be implemented by a library of fixed component designs; as an example, slices are commonly used and simple in both their functionality and implementation, but a fixed library cannot address each possible interface design. The same applies to driving default values to physical streams,

which are specified and relatively easy to apply, but too variable to incorporate into a library. Furthermore, neither functionality would commonly be implemented as a Streamlet, but instead incorporated into one (as a lower-level component, or directly).

Intrinsics will primarily be useful in the context of structural implementations, but may also be useful when defining types and interfaces. For example, in the absence of type parameters, or to provide a more explicit way of expressing it, an intrinsic which *reverses* a given Stream type could be implemented.

To include such intrinsics in TIL, accounting for the different scopes they may be applicable in, and avoiding an excess of keywords potentially conflicting with identifiers, it would make sense to preface intrinsics with a control character. For the purposes of the next examples, this control character will be `!`, as it is otherwise unused, and not valid as part of any identifier, as shown in Listing 3.11.

```
type my_rev_stream = !reverse(stream_identifier);
...
instance.port_a -- !default;
...
instance.port_a -- !buffer(3) -- parent_port;
```

Listing 3.11: Using `!` as a control character for intrinsic functions in TIL

Listing 3.11 also demonstrates that intrinsics benefit from being able to modify operators and produce different kinds of statements, by overriding what a port may be connected to (port connecting to `!default`), and producing different kinds of connections (a `!buffer` between a connection). As intrinsics will be part of the IR, and not user-definable, this will be possible to account for and implement.

As a recommendation for future work and summary, the following intrinsic functions are likely to be suitable for the IR:

1. *Slices* and (FIFO) *buffers*, to break up combinatorial paths and/or account for different operations taking a variable number of cycles.
2. A *synchronizer*, which attempts to combine the ready/valid control signals of multiple input and/or output Streams.
3. A *parallelizer*, which converts a single source Stream/port and attempts to split elements or outer sequences into separate Streams. (I.e., this only applies to Streams without dimensionality, or selects the cut-off point based on the outer dimension. For simplicity, this should only apply to Streams which are represented as single physical streams.)
4. A *serializer*, which converts multiple matching source Streams into a single Stream, based on their outer (or lack of) dimension. The inverse of the parallelizer above.
5. A throughput *reshaper*, which results in a Stream with a different number of element lanes, either combining multiple transfers into one, or splitting up transfers into multiple.
6. An intrinsic which automatically drives *default* values to the physical streams that make up a given Stream, as defined by the Tydi specification [46].
7. A way to explicitly mark a port as *unconnected*, circumventing the IR's validation against unconnected ports, and optionally driving ready/valid low.
8. A *constant value* source, based on the assertion system described in Section 4.1, allowing designers to quickly stub connections beyond driving their default values.
9. An optimistic source-to-sink *complexity bridging connection*: Provided a port's source Stream(s) have a lower complexity and otherwise perfectly match another port's sink Stream(s), these ports can be connected according to the Tydi specification. (Note that this will need to account for a port's parent (source) Stream potentially containing reversed child (sink) Streams, which cannot have lower complexities than their counterparts.)
10. A configurable *complexity downshifter*: Using sufficiently deep buffers, transfers from a higher-complexity source Stream may be converted to match the constraints of lower complexities. (A buffer is required because lower complexities can impose timing constraints on transfers, and may not allow transfers with inactive lanes.)

11. A *reverse* function for Stream expressions, which takes an existing Stream type and simply switches its direction property between Forward and Reverse. (This functionality can potentially be implemented on the query system or TIL parser, rather than by a compiler.)

12. Different, configurable intrinsics for *N-to-M connections* based on parallelizers and serializers, and potentially by simply sending transfers from a source Stream to multiple sinks by synchronizing their ready/valid signals.

This is not an exhaustive list, and more specific functions may arise based on actual use, but these should serve as relevant, minimal examples of which functions could be incorporated into the IR directly.

These proposed intrinsic functions also illustrate another property to account for: Not every intrinsic will necessarily succeed, and it may not be possible to determine their successfulness based on static evaluation. For example, the complexity bridging connection will simply fail on static evaluation if it is determined that the connection includes a source with a higher complexity than its sink counterpart. Conversely, whether the "complexity downshifter" will succeed depends entirely on whether the buffer is sufficiently large to concatenate all incoming transfers to match the target complexity's constraints; complexity 1 requires that *all* transfers occur over consecutive clock cycles, requiring either the buffer to be sufficiently deep for all expected sequence lengths, or for the source to not postpone transfers often enough to let a smaller buffer run out. The latter case can only be evaluated by the designer or potentially in simulation.

Finally, there will likely be a number of potential intrinsic functions which are broadly applicable, but can only be used in a subset of scenarios and/or target languages. For example, a simple clock generator for simulation purposes could be expressed as an intrinsic, but would not be possible or useful to synthesize. To this end, such subsets should have a hierarchical naming scheme to indicate they are not generally available and/or expected to be implemented by all backends. The clock generator example could be expressed as `!sim.clockgen(...)`, for example.

3.3.4. Annotations

One final suggested addition to the language as a whole is support for *annotations*, which are syntactic metadata to be interpreted by compilers. Up until now, the additional language features discussed were intended to be generally applicable between most compilers and potential target languages. By contrast, annotations are intended for including information which is specific to a target language, compiler, or hardware platform, or may be interpreted (very) differently between them. As an example of the latter scenario: A "deprecated" annotation could produce warnings, be ignored altogether, or outright prevent compilation depending on the target, compiler, and configuration.

The IR should not predicate the properties or implementation of annotations, and it should be possible to add annotations to any language construct: That is to say, annotations can be used to provide metadata for any expression, statement, or intrinsic, and may be combined with other annotations. To avoid unintentional conflicts of annotations, to ensure they can be parsed as text in TIL and can be stored in the query system, and to avoid incompatibility between IR projects, the following constraints and properties are recommended:

1. An IR project being valid should *never* be contingent on the existence of annotations. I.e., a compiler must be able to ignore all annotations and produce valid (albeit not necessarily correct or desirable) output. This can be enforced by the TIL parser and IR query system not making use of the metadata defined in annotations. Note that compilers may still fail on *incorrect* annotations, based on their interpretation.

2. Use a unique control character as delimiters for an annotation in TIL. For example, the `@` symbol, which is otherwise unused and unsupported by the IR.

3. All annotations must have a "namespace": Namespaces can be used to indicate certain annotations belong to specific compilers, languages, or operations. This ensures a compiler can easily query only relevant annotations within a project. To prevent further conflicts within a specific namespace (such as ones for a target language which is served by different compilers), a common forum to establish their meaning will be necessary.

    - E.g.: `@namespace.sub_namespace.property@`

- This also allows for the organization of multiple properties based on namespaces, as shown in Listing 3.12.

4. An annotation can be one of the following kinds of properties:
   (a) A *flag*; their existence implies a boolean *true*. E.g.: `.property`
   (b) An *assignment*; a name, followed by a single value being assigned. E.g.: `.property = value`
   (c) A *constructor*; a name, followed by multiple properties of that name being assigned a value. E.g.: `.property(a = value, b = value)`
5. Annotations can be assigned one of the following kinds of values:
   - A *number*: These can be positive, negative or floating point, but will not be evaluated by the parser and simply stored as strings. E.g.: `1, -1, 1.0, -1.0`
   - A *unit value*: An arbitrary sequence of (non-control or otherwise conflicting) characters preceding and/or succeeding a number. Stored as an optional string, a number (also a string) and another optional string. E.g.:
     – `$ 100.00`
     – `40 GHz`
     – `after 10 ns`
   - A *string*: Arbitrary sequences of characters, enclosed by double-quotes. Control characters should be escaped, but it may be possible to provide a “raw string” syntax in TIL, as well.
   - A *name*: Sequences of (non-control or otherwise conflicting) characters, can be used to imply constants or values of enumerations, are stored as strings. E.g.: `.target_hw = HAL9000`
   - An *object*: Arbitrary collections of (unique) names and nested values, stored as maps using a string (name) as key. E.g.: `{a: value, b: value}`
   - A *list*: A sequence of values, the parser and query system will not enforce they are the same kind of value. E.g.: `[”string”, 1.0, name]`

```
@namespace.namespace2 {
  namespace3.namespace4.flagvalue,
  namespace5 {
    constructorvalue(a = 2, b = 3),
    flagvalue2,
    namespace6 {
       flagvalue3,
       assignvalue = ”a string”,
    }
  }
}@
```

Listing 3.12: A collection of annotated properties

Combined, these properties should allow for the clear, readable expression of virtually any kind of metadata. Note that annotations overall simply amount to collections of (tagged) strings, ensuring they do not burden the query system or TIL parser with evaluating specific constraints. By extension, any kinds of values not addressed (e.g., various non-decimal representations of numbers) can simply be passed directly as strings.

3.4. Project Structure and Reusability

In order to support organizing information over different files, and to provide further options for configuration, the IR should also feature some form of “projects”; definitions of which files in a given directory or set of directories belong to one another. This also opens the way for *imports*, not just of namespaces within a file or project, but between different projects altogether.

At this time, the TIL parser does not support using multiple files, does not allow for imports between namespaces, and the only configuration is the input file and output directory through the example application described Section 5.4. Additionally, the starting position of relative paths is defined relative to the directory from which the example application is run. Many of these issues can be addressed through a project definition; the project file would define which TIL files are part of the project, what the desired output directory is, and its location could serve as the root location for relative paths (alternatively, the file could make this configurable).

The notion of projects, and that of imports between both projects and namespaces already exists to an extent on the query system; all internable structures even implement a `MoveDb` trait, which as the name implies allows for them to be moved (or copied) between query system databases, generating new identifiers as needed. This is necessary as some structures will themselves contain identifiers which would otherwise refer to their old database. Imports generally will not need to be reflected in a target language, meaning the query system does not need to reflect any new structures. As such, there are two aspects to define: What information should a "project" describe, and how should importing declarations between namespaces and projects behave?

3.4.1. Project Properties

It is important to distinguish which properties are relevant to the TIL *parser*, and which properties are relevant to the query system's notion of a project. As such, the query system's project should contain the following information:

- A name for the project.
- The location to use as root for any relative paths.
- Any configuration relevant to the backend, such as the desired output directory, and potentially backend-specific configuration items.
- Potential configuration relevant to the query system, such as how to handle specific type parameters, as discussed in Section 3.3.1.

In addition to these properties, the *parser*'s project (file) should perform the following functions:

- Serve as the root for relative paths, or configure it.
- Contain the configurations described above, or point towards configuration files for these properties.
- Configure parser-specific properties; e.g., should there be a way to optimize the parse speed by not creating and emitting an error report, this should be configurable.

3.4.2. Import Behavior

There are two main questions to answer about how imports should be have:

1. How should imported declarations be identified?
2. Which declarations can be imported?

Identifiers The first question is relevant to situations where multiple identifiers overlap, either between imports, or between an import and the namespace being imported to. The query system simply provides an `import_as` function, which allows the name of the import to be set afterwards. This does not specify any constraints as to how they should be named, however. One of, or more likely some combination of, the following methods can be employed:

1. Duplicate identifiers should result in an error.
2. Imports should always be prefixed by their namespace, and further prefixed by their project name if imported from another project. (Note that multiple projects may have different names, however.)
3. Imports are arbitrarily aliased, determined by whichever means emits TIL.

4. Imports are arbitrarily prefixed, determined by whichever means emits TIL.

Declarations to import As the use-cases for imports have not been fully realized at this point, future work on the IR or similar projects should consider the following questions:

- Should it be possible to selectively import declarations from a namespace?
- Would declarations in a namespace benefit from a public/private distinction, or similar?
- May certain namespaces be excluded from being imported altogether?

3.4.3. Notes on Reusability

While being able to import declarations between projects can aid reusability, this ultimately depends on whether these imports are actually possible from the front-end emitting the IR. A front-end language can also establish its own project structure and import behavior, and simply emit “flat” TIL. However, namespaces are designed to be able to reflect any project structure a front-end may utilize, and imports should be designed to do the same. The ideal outcome would be for importing IR projects generated by different front-ends to be feasible, by making it as easy as possible to map between the IR and any structures the front-end may use.

4

Intermediate Representation: Specification

While the intermediate representation lacks the ability to completely implement behavior, it can nonetheless allow for the specification of behavior through tests.

Unlike the previous chapter, the listings shown in this chapter feature a theoretical, suggested grammar, as work on tests at the query system- and VHDL compiler-level did not advance far enough to warrant implementation of a parser.

4.1. High-level Assertions

As the IR is used to represent ports consisting of Streams carrying logical types, it is best suited for transaction-level verification. Inputs and outputs should be verified against abstract streams of data, upon which the IR combined with a backend will generate the necessary signalling behaviour and assertions. This enables designers to verify the behaviour of components and correctness of their interfaces without needing to concern themselves with the target language.

There are two key properties to consider when designing and generating tests for Interfaces based on transactions:

1. Ports of an Interface are not required to be interdependent or synchronized with one another.
2. A port's Stream does not necessarily have a single direction, as child Streams can be *Reversed*.

To address these, the recommended testing grammar has the following properties:

1. Transaction verification on ports should be assumed to happen in parallel by default, rather than in the sequence assertions are declared. Discussed in Section 4.1.1.
2. Rather than explicit *assign* and *compare* methods, the IR should automatically determine whether physical streams are sinks or sources. Discussed in Sections 4.1.4 and 4.1.5.

These properties by themselves will still not allow high-level assertions to cover every possible use of Tydi interfaces, however. Section 4.3 describes how to mitigate these limitations.

4.1.1. Parallel by Default

Within a test scope, all statements should be assumed to occur in parallel, rather than being executed sequentially. If the input of one or more streams are required before an output can be produced, this should be implemented through the *ready* and *valid* signal(s) of the Streamlet. For example, implementing a Streamlet which adds two inputs could be represented as follows, assuming the output "result" does not assert *valid* until it has received and added two inputs:

```
test test_name {
  adder = adder_def;
```

```
  adder.result = "010";
  adder.in1 = "01";
  adder.in2 = "01";
}
```

As assertions occur in parallel, it is not possible to perform multiple separate assertions on a port in the same scope. However, if the Streamlet being tested also implements proper *ready* signalling and/or buffers to ensure one set of inputs ("in1" and "in2") corresponds to one output, the following can be asserted:

```
adder.result = ("010", "001", "011");
adder.in1 = ("01", "01", "10");
adder.in2 = ("01", "00", "01");
```

Where `("01", "00", "01")` represents a series of `Bits(2)` to be transferred over a Stream without dimensionality. This is to be transferred depending on throughput; e.g., one port could support two elements per transfer and require only two transfers, while another might only support one element per transfer and require three. In this proposed syntax, square brackets would be used to indicate dimensionality: `[["1", "0"], ["0"]]` represents a Stream with data `Bits(1)` and dimensionality 2.

4.1.2. Sequences

While transactions on ports are not *necessarily* interdependent, it is reasonable to expect that they will be in many cases regardless. While stateless behavior can be tested in parallel, as each transfer still requires a valid handshake, components which do observe state require that transactions on ports can be asserted in a specific sequence. For example, a counter which accumulates based on input transfers and always drives its output with its current value, or an instruction for a state machine, require that the transfer on the input succeeds before the value on the output is tested.

To this end, the proposed testing grammar also includes *sequences* of explicit stages, each with their own scope; the assertions within each stage still happen in parallel, but each stage must successfully pass before the assertions in the next stage are performed:

```
sequence sequence_name {
  initial_state {
    counter.count = "0000";
  }, increment {
    counter.increment = "1";
  }, result_state {
    counter.count = "0001";
  },
};
```

In simulation, such stages could be implemented by creating specific flags for each assertion in that scope, then requiring that the normally parallel processes wait until each flag in that stage is set.

For the purposes of tracking their progress and for giving flags descriptive names, sequences *must* have a unique *Name* (in their scope), while individual stages *may* have unique names. Stages are propagated as *Path Names*, using the enclosing scope (the sequence) as their root; if a stage does not have a name, *stage#* is used instead, with # being the number of that stage (starting from 1, and counting stages which do have names). The flag names of individual assertions within a scope are to be determined by the backend; a descriptive example would be to simply use the name of the instance and port being asserted on, as these are guaranteed to be unique within that scope.

It is possible to specify multiple sequences in a single test, which will occur in parallel. Likewise, nesting sequences is allowed, in which case the nested sequence will occur in parallel with any other

assertions in that scope (including other sequences). Note that *any* port asserted on in a sequence is thereby excluded from being asserted on in the sequence's parent scope.

4.1.3. Descriptive Errors

For organization purposes and to emit descriptive error messages when tests fail, (optional) labels on assertions, sequences and sequence stages can be added to tests. By specifying intended error messages for the backend, it is possible to better reflect the high-level assertions described in the IR. The proposed syntax for labels/messages on assertions is as follows:

```
"overall test label": test test_name {
  ...

  "this is an assertion label": component.port = "1010";

  "sequence label": sequence sequence_name {
    "stage label": stage_with_name {
      ...
    },
    "stage label": { // No stage name
      ...
    }
  }
}
```

That is to say, labels will consistently use the <string><colon> syntax, behaving similarly to documentation. Test labels may be combined with their parent labels for further clarity; ergo, the query system should provide an ordered list of all parent labels.

In the event no label was supplied for an assertion, it will be up to the backend to generate messages/names if necessary, based on other properties of that assertion. Regardless of the existence of labels, it will be prudent for the query system to also store ancillary information about all assertions, for the backend to optionally use. For instance, if the tests were parsed from a TIL file, line numbers and character spans of assertions and stages can be propagated to the backend to provide more descriptive errors, even when labels are provided. By extension, it is also possible for a frontend emitting the IR to use labels to propagate such (line number and character span) information from its own test definitions.

4.1.4. Asserting Equality

The IR should automatically determine whether physical streams are sinks or sources, rather than requiring explicit language to drive or compare a signal. The latter property means that something closer to mathematical equality is implemented; "the transaction on port *a* is equal to data *x*", whereupon it is automatically determined whether *x* should be driven, or observed and compared.

While this has very little effect on the examples in the previous section, other than potentially removing unnecessary keywords or operators:

```
adder.result == ("010", "001", "011");
adder.in1 = ("01", "01", "10");
adder.in2 = ("01", "00", "01");
// OR:
assert adder.result = ("010", "001", "011");
act adder.in1 = ("01", "01", "10");
act adder.in2 = ("01", "00", "01");
```

It greatly simplifies assertions on nested and reversed child streams. For example, we can use the same adder concept described before, but combine its ports into a single Stream and port with a Reversed child Stream to indicate a response:

```
add: in Stream(
  data: Group(
    in1: Stream(
      data: Bits(2),
      direction: Forward,
      ...
    ),
    in2: Stream(
      data: Bits(2),
      direction: Forward,
      ...
    ),
    result: Stream(
      data: Bits(3),
      direction: Reverse,
      ...
    ),
  ),
  direction: Forward,
  ...,
  keep: false,
)
```

Subsequently, the assertion can be represented as follows:

```
adder.add = {
  in1: ("01", "01", "10"),
  in2: ("01", "00", "01"),
  result: ("010", "001", "011"),
};
```

Listing 4.1: Representing the *in* and *result* Streams as independent.

Or as follows, to emphasize the relation between transfers on the Streams:

```
adder.add = ({
  in1: "01",
  in2: "01",
  result: "010",
}, {
  in1: "01",
  in2: "00",
  result: "001",
}, {
  in1: "10",
  in2: "01",
  result: "011",
});
```

Listing 4.2: Representing the *in* and *result* Streams as interdependent.

In the above examples, the parent "add" stream will have been flattened into "in1" and "in2", but this physical implementation detail has little bearing on the assertion itself. Furthermore, if we decide to turn "in1" and "in2" into simple *Bits(2)* fields rather than *Stream*s, the assertion in Listing 4.2 will remain unchanged.

It is worth noting that Listing 4.1 and Listing 4.2 should produce identical results, with independent, parallel transfers on the *in* and *result* Streams attempting to match the specified throughput (and physical element lanes). Synchronicity across multiple child Streams relative to a parent Stream at

an element level can only be respected if the parent Stream has dimensionality or transfers elements itself. This is why the parent “add” Stream can be flattened into its child “in” Streams.

It is of course still possible for a Streamlet to be implemented such that it enforces synchronicity through ready-valid signals on the child Streams, or by setting *keep* = *true* on the parent Stream. Conversely, it is allowed for the *adder* example to have a higher throughput on the “result” Stream, and buffer results (or inputs) to provide them all as multiple elements in a single transfer.

An example of a naturally synchronous set of Streams without the parent having dimensionality is a *Union* of Streams, in which the parent Stream transfers the *tag* (indicating the active field) over its data signal. This can be asserted as follows, and should result in sequential assertions on the child Streams, synchronized to transfers on the parent Stream:

```
storage.query = ({
  request: [”0001”, ”0100”], // derive tag value based on the field name
}, {
  response: [”11001010”, ”10010000”, ”00110001”, ...],
}, {
  request: ...
}, ...)
```

Listing 4.3: Asserting on a Stream carrying a Union with an alternating input (request) and output (response) Stream. The parent (query) Stream controls the *tag* determining which Stream should be active.

4.1.5. Issues with Explicit Assignment and Comparison

Conversely, consider a way to represent the initial scenario from Listing 4.1 using = and == operators:

```
adder.add = {
  in1: (”01”, ”01”, ”10”),
  in2: (”01”, ”00”, ”01”),
  result: (”010”, ”001”, ”011”),
};
```

```
adder.add = {
  in1 = (”01”, ”01”, ”10”),
  in2 = (”01”, ”00”, ”01”),
  result == (”010”, ”001”, ”011”),
};
```

Listing 4.4: Two approaches to using explicit assign and compare operators.

The left example of Listing 4.4 is clearly incorrect, as it is using a “assign” operator, but actually “comparing” *result*. The right example is more subtly wrong, however: The parent “add” Stream does not actually exist, making the initial `add =` statement irrelevant. If we were to make “add” an *out* port, and reverse *in1* and *in2* instead of *result*, we may instead assert it as follows:

```
adder.add == {
  in1 = (”01”, ”01”, ”10”),
  in2 = (”01”, ”00”, ”01”),
  result == (”010”, ”001”, ”011”),
};
```

However, the physical implementation of either design is identical, as is the actual assertion. The outer `add <operator>` has no bearing on the inner scope; it requires work to track the direction of the parent Stream to no practical benefit. So alternatively, we may split up all statements as follows:

```
adder.add.in1 = (”01”, ”01”, ”10”);
adder.add.in2 = (”01”, ”00”, ”01”);
adder.add.result == (”010”, ”001”, ”011”);
```

This requires knowledge of which Streams are converted to physical streams, however. Moreover this would not work when the parent stream does exist (as the Group may contain element types in addition to Streams), and could not be used to perform the Union assertion in Listing 4.3.

Finally, in order for compilers to emit (or designers to write) these statements in the IR, they will need to track the specific direction of each physical stream, based on the logical types and interfaces.

As this process is necessary regardless, it is more effective to offload it to the query system (or compiler) consuming the IR, which already concerns itself with converting Interfaces and logical Streams to physical streams. This also ensures the front-end compilers (or designers) will only need to concern themselves with the abstract, logical definitions of transactions, rather than their exact physical implementation.

4.2. Proof of Concept

4.2.1. Physical Transfers

Work on high-level assertions had begun as part of this thesis, though they could not be implemented in full. Assertions were developed bottom-up, in that the focus was on the results to be emitted by the VHDL backend, with the intent of raising the level abstractions to match the proposed syntax from there.

Specifically, there is support within the query system and the VHDL backend for assertions on *physical streams*, with a `PhysicalTransfers` trait being able to arbitrarily switch between driving signals and asserting them against a given transfer. The `PhysicalTransfers` trait is implemented automatically by any object implementing the `PhysicalSignals` trait, which features methods for returning the direction of a physical stream, and for automatically driving or comparing the signals (*data*, *endi*, etc.) based on this direction.

Combined with a set of `handshake` (driving ready or valid, and waiting for and/or asserting the inverse) methods, a *sequence* of elements can be easily transferred over multiple cycles. To further allow for the verification of various timing constraints, the handshake signals can be driven to either be held high (resulting in a transfer over consecutive cycles), or to be driven low after a cycle. The `PhysicalTransfers` trait reflects this by having separate `open_transfer` and `close_transfer` methods, along with a “test_staggered” parameter on `transfer`.

The transfers themselves are called `PhysicalTransfer`, featuring properties for how the transfer should behave derived from the physical stream being driven, such as which lanes may be inactive, and whether the transfer needs to occur over consecutive cycles. The contents of the transfer are taken from a more free-form `LogicalTransfer`, which is either explicitly an “empty sequence”, or an iterator of simple *elements*.

Each logical element contains an optional *data* field which is either Null, Bits, or a Group or Union of further *element data*, it also contains an optional *last* property to indicate that the element represents the end of one or more dimensions in a sequence. If *last* is not set, it simply means the *last* signal should not be driven. However, if *data* is not set, that indicates that the data lane itself is inactive.

When a `LogicalTransfer` is set on a `PhysicalTransfer`, it is verified whether this transfer is possible given the physical stream’s constraints. For instance, it will fail if the elements contain data types which do not match the physical stream, or if it attempts to drive *last* for multiple elements despite the physical stream’s complexity being $C < 8$.

4.2.2. Demonstration

The results of this work are demonstrated by `process_tests`[1], as such:

Taking the following “physical transfers”, representing the sequence: `[ [ [ 11, -, 11, 10 ], [ 01, 00, 10 ] ], - ], -` (With - representing inactive lanes.)

```
let transfer_1 =
    PhysicalTransfer::new(Complexity::new_major(8), Positive::new(3).unwrap(), 2, 3, 3)
        .with_logical_transfer(([Some("11"), None, Some("11")], "101"))?; // [[[11, -, 11
let transfer_2 =
    PhysicalTransfer::new(Complexity::new_major(8), Positive::new(3).unwrap(), 2, 3, 3)
        .with_logical_transfer([("01", Some(0..0)), ("10", None), ("00", None)])?; // 10], [01, 00
let transfer_3 =
    PhysicalTransfer::new(Complexity::new_major(8), Positive::new(3).unwrap(), 2, 3, 3)
        .with_logical_transfer([("01", Some(0..1)), ("-", Some(2..2)), ("-", None)])?; // 10]], -], -
```

An input physical stream can be addressed as follows:

```
drive_stream.open_transfer()?;
drive_stream.transfer(transfer_1.clone(), false, "test message drive 1")?;
```

[1] https://github.com/matthijsr/til-vhdl/blob/main/crates/til_vhdl/tests/process_tests.rs

```
drive_stream.transfer(transfer_2.clone(), false, "test message drive 2")?;
drive_stream.transfer(transfer_3.clone(), false, "test message drive 3")?;
drive_stream.close_transfer()?;
```

Producing the following VHDL, driving its ports with the correct values and timings:

```
process is
begin
  a__x_valid <= '1';
  a__x_data(1 downto 0) <= "11";
  a__x_data(5 downto 4) <= "11";
  a__x_last(2 downto 0) <= (others => '0');
  a__x_last(5 downto 3) <= (others => '0');
  a__x_last(8 downto 6) <= (others => '0');
  a__x_strb <= "101";
  a__x_user(2 downto 0) <= "101";
  wait until rising_edge(clk) and a__x_ready = '1';
  a__x_data(1 downto 0) <= "10";
  a__x_data(3 downto 2) <= "01";
  a__x_data(5 downto 4) <= "00";
  a__x_last(2 downto 0) <= "001";
  a__x_last(5 downto 3) <= (others => '0');
  a__x_last(8 downto 6) <= (others => '0');
  a__x_strb <= "111";
  a__x_stai <= std_logic_vector(to_unsigned(0, 2));
  a__x_endi <= std_logic_vector(to_unsigned(2, 2));
  wait until rising_edge(clk) and a__x_ready = '1';
  a__x_data(1 downto 0) <= "10";
  a__x_last(2 downto 0) <= "011";
  a__x_last(5 downto 3) <= "100";
  a__x_last(8 downto 6) <= (others => '0');
  a__x_strb <= "100";
  wait until rising_edge(clk) and a__x_ready = '1';
  a__x_valid <= '0';
  wait until rising_edge(clk);
end process a__x;
```

While an output physical stream is addressed in the same way:

```
compare_stream.open_transfer()?;
compare_stream.transfer(transfer_1.clone(), false, "test message compare 1")?;
compare_stream.transfer(transfer_2.clone(), false, "test message compare 2")?;
compare_stream.transfer(transfer_3.clone(), false, "test message compare 3")?;
compare_stream.close_transfer()?;
```

But is automatically converted to comparisons in VHDL:

```
process is
begin
  wait until rising_edge(clk) and a__y_valid = '1';
  assert a__y_data(1 downto 0) = "11" report "test message compare 1";
  assert a__y_data(5 downto 4) = "11" report "test message compare 1";
  assert a__y_last(2 downto 0) = (others => '0') report "test message compare 1";
  assert a__y_last(5 downto 3) = (others => '0') report "test message compare 1";
  assert a__y_last(8 downto 6) = (others => '0') report "test message compare 1";
  assert a__y_strb = "101" report "test message compare 1";
  assert a__y_user(2 downto 0) = "101" report "test message compare 1";
  a__y_ready <= '1';
  wait until rising_edge(clk) and a__y_valid = '1';
  assert a__y_data(1 downto 0) = "10" report "test message compare 2";
  assert a__y_data(3 downto 2) = "01" report "test message compare 2";
  assert a__y_data(5 downto 4) = "00" report "test message compare 2";
  assert a__y_last(2 downto 0) = "001" report "test message compare 2";
  assert a__y_last(5 downto 3) = (others => '0') report "test message compare 2";
  assert a__y_last(8 downto 6) = (others => '0') report "test message compare 2";
  assert a__y_strb = "111" report "test message compare 2";
  assert a__y_stai = std_logic_vector(to_unsigned(0, 2)) report "test message compare 2";
  assert a__y_endi = std_logic_vector(to_unsigned(2, 2)) report "test message compare 2";
  a__y_ready <= '1';
  wait until rising_edge(clk) and a__y_valid = '1';
  assert a__y_data(1 downto 0) = "10" report "test message compare 3";
  assert a__y_last(2 downto 0) = "011" report "test message compare 3";
  assert a__y_last(5 downto 3) = "100" report "test message compare 3";
  assert a__y_last(8 downto 6) = (others => '0') report "test message compare 3";
  assert a__y_strb = "100" report "test message compare 3";
  a__y_ready <= '1';
  wait until rising_edge(clk) and a__y_valid = '1';
  a__y_ready <= '0';
```

```
  wait until rising_edge(clk);
end process a__y;
```

4.2.3. Results and Future Work

While work did not complete within the span of the thesis, this proof of concept does demonstrate that high-level assertions such as those described in Section 4.1 are possible. Notably, even this reduced version greatly improves the ergonomics of performing transfer-level assertions, as they require fewer lines of code (scaffolding around the query system required for the test notwithstanding) and better represent the high-level intentions.

To expand on this, the next steps would be to:

1. Use the query system to convert an arbitrary (but type-appropriate) sequence into multiple transfers automatically.

2. Use the query system to convert assertions on logical Streams such as those in 4.1 into multiple assertions on the corresponding physical streams.

Note that these steps do not require further input on the backend; provided the backend implements the requisite `PhysicalTransfers` trait, all further logic can be implemented for *all* possible backends on the query system itself. And when this is successful, one may create a minimal grammar and parser, to reduce the amount of scaffolding when testing these functions.

4.3. Complex Test Cases

4.3.1. Limitations of High-Level Assertions

Of course, not all behavior can be tested through transfer-based, high-level assertions. There are a few specific properties which make a Streamlet difficult to test, or make a test scenario difficult to implement:

- **A *user* signal** — The Tydi specification allows for Streams to have a *user* signal, which exists specifically to address use-cases not covered by Tydi's transfer specification. There are very few constraints on the user signal, other than it being a *signal*, and not its own (physical) stream. The user signal can be driven independently from transfers and clock cycles, what constraints there are to its behavior are entirely determined by the designer, and so cannot be translated to the assertion system described in the previous sections. Hence, the user signal will be omitted from the assertion system, and designers of highly custom interfaces should implement tests manually.

- **Testing large ranges of inputs** — The assertions described before use *constants*, making more exhaustive testing difficult to implement. For example, when testing a 8-bit adder, one would expect it to work for any combination of inputs in that 8-bit range. This *can* be implemented through an exhaustive series of assertions, but is better served by a (random) number generator, or another external source of inputs and expected outputs.

- **Testing against randomness** — In the same vein as the previous limitation, the use of constants in assertions makes accounting for Streamlets which itself produces random outputs more difficult. The most simple example of this would be the Streamlet itself being a random number generator, with the test attempting to assert that its output is sufficiently statistically random. (A more complex scenario could involve a Streamlet using randomness for the purposes of encryption.)

- **Unimplemented dependencies** — If a design is made up of multiple Streamlets, assertions can only be performed against the completed product and against its individual components. If one or more pieces are not yet implemented, the tests cannot succeed.

- **Verifying the use of dependencies** — On a more abstract level, it is not possible to verify that a composite design actually employs its intended dependencies. E.g., does the "encryption" Streamlet actually use the verified "random number generator" Streamlet, or does it implement its own (potentially incorrect) random number generation?

- **Creating predictable dependencies** — Conversely, a dependency may have been implemented, but not be conducive to testing. In the earlier example of an "encryption" Streamlet depending on a random number generator, it may be desirable to test to the Streamlet with non-random numbers for more predictable assertions. In effect, the goal is to isolate only one Streamlet's functionality.

4.3.2. Using Test Streamlets for Verification

In scenarios where high-level assertions cannot serve as a useful source of inputs, and/or cannot properly verify outputs, it is still possible to use the IR's ability to link and compose Streamlets to instead create "test Streamlets", and connect these to the subject's input and/or output.

For instance, in the previously described scenario of wanting to test a range of inputs against an adder, it is possible to create a Streamlet which generates numbers and verifies outputs (either against a known-good adder, or by drawing from an external source), and connect it to the adder. The same principle can be used to create a Streamlet which tests a random number generator's randomness, or one which drives the user signal.

These applications are obviously not very different from manually creating a testbench in the target language, but they ensure that the tests remain organized through the IR, and make it easier to reuse certain solutions. Additionally, these "test Streamlets" can be combined with high-level assertions for mixed test scenarios, or to use the assertions as configuration or verification on the test Streamlet.

Such "test Streamlets" can already be implemented in the IR through simple namespaces, but should be declared in tests or test files instead for better organization of both the IR and the back-end's output, e.g:

```
streamlet general_test_streamlet = <definition>;

test test_name {
  streamlet very_specific_test_streamlet = <definition>;
  streamlet test_tld = (correct: out test_result) {
    impl: {
      tester = very_specific_test_streamlet;
      subject = actual_streamlet;

      subject.input -- tester.output;
      subject.output -- tester.input;
      tester.correct -- correct;
    }
  };

  test_subject = test_tld;

  test_subject.correct = "1";
}
```

4.3.3. Substitution

In order to isolate a composite Streamlet's functionality from its dependencies, it will be helpful for the IR to provide some way of redefining or *substituting* Streamlet definitions. The test Streamlets in the previous sections are helpful when the dependencies are internal, or part of a structural implementation; when Streamlets are embedded in a behavioral implementation, there are no such options.

Provided the behavioral implementation depends on Streamlets tracked by and generated from the IR, it should be possible to redefine their implementation when creating the test, or test project. E.g., in VHDL, it would include different architecture definitions in the workspace of the testbench.

This can be used for the following purposes:

- Performing proper "unit tests", by removing all other dependencies from a test of a composite Streamlet and replacing them with more predictable, simpler Streamlets.
- "Stubbing" an unimplemented dependency; even if the purpose is not to perform a unit test, it may help to temporarily substitute a dependency, to verify a larger design.
- Simulating a dependency which cannot otherwise (efficiently) run in software; e.g., if a dependency would normally draw data from a hardware component (such as memory, device storage,

or a sensor), replacing it with something which reads or produces data in software enables the overall design to be tested.

- Verifying that a dependency is being used; e.g., by substituting the random number generator Streamlet, it is possible to assert that it transferring a specific number is actually propagated to the output of the dependent Streamlet.

The syntax for this functionality can be fairly simple, e.g.:

```
// substitute(<streamlet_declaration>, <replacement implementation>)
substitute(streamlet_decl, "/test/path/");
substitute(streamlet_decl, !an_intrinsic);
substitute(streamlet_decl, { input--output; });
```

Where such substitution statements are allowed is to be determined based on actual implementation. As it will depend on the complexity of performing a substitution across different target languages. Should it prove very simple in most relevant languages, it can be performed per test - otherwise, it may make more sense per test file, or at the level of the entire (test) project.

4.4. Setting up Subjects

The last property to address when creating tests is appropriate setup steps for a subject Streamlet. Even when considering tests are simulated, it is still likely that a test may involve a reset procedure before or during assertions; a designer may also want to verify reset behavior itself.

Additionally, Streamlets will have one or more domains consisting of a clock and reset signal each, which may be different. The Tydi specification does not place constraints on the reset signal, other than requiring that the *ready* and *valid* signals are released during a reset. As such, the reset signal can have any sensitivity and synchronicity, and a reset may take any number of cycles.

As such, reset/setup syntax needs to minimally account for the following properties:

- Whether the reset signal is sensitive on a positive or negative edge.
- Whether the reset signal needs to be held for a certain number of cycles.
- When accounting for multiple domains, whether each domain's reset signal behaves differently, and whether they can be reset simultaneously.

Specific configuration of clock signals and frequency is less important, but may prove helpful when testing components which explicitly require different clock speeds to operate correctly. Initially, clock behavior is best left to annotations (described in Section 3.3.4), as the actual implementation of a clock may differ between target languages.

How to specifically represent these properties has not been fully evaluated, and more considerations may arise from actual implementation. However, in the absence of such an implementation and based on the previously established properties, the following syntax can be proposed:

```
domain 'a { reset: low };
domain 'b { reset: low };
domain 'c { reset: high };
domain 'd { reset: high };

subject1 = streamlet_def_name1<'a, 'b>;
subject2 = streamlet_def_name2<'c>;
subject3 = streamlet_def_name2<'d>;

process arrange1 = sequence arrange1_sequence {
  {
    reset('a, 3);
  }, {
    wait('a, 1);
  }, {
    reset('b, 1);
    reset('c, 1);
  }
};

process arrange2 = {
  reset('d, 1);
};
```

```
"sequence label": sequence sequence_name {
  "initial setup": {
    arrange1; // Note that nested sequences are allowed.
    arrange2; // arrange2 will occur in parallel to all of arrange1
  }, "test stage": {
    // Perform some assertions
  }, "partial reset": {
    arrange_name2; // The contents of arrange_name2 are inserted into this scope.
    reset('a, 3);
  }, ...
}
```

Listing 4.5: "Arranging" subject streamlets, by driving their reset signals.

To elaborate on this syntax, and address some properties not (clearly) included in this example:

- `domain`s are defined with `reset: low` or `reset: high`, indicating whether the reset signal needs to be held low or high. Other properties may be added if they are useful and generally applicable in defining domains (e.g., (relative) clock speed).
- `process`es are stored sequences or operations, to be easily reused. These are useful for arranging subjects, but may also be used for other kinds of test organization.
- `reset(<1>, <2>)` is a pre-defined process with two parameters:
  1. Which *reset signal (domain)* to drive. The reset signal will be inverted from its default state.
  2. How many cycles to *hold* the reset signal. (Currently assumed to be an integer ≥ 1.)
- `wait(<1>, <2>)` is a pre-defined process, with two parameters:
  1. Which *clock signal (domain)* to wait relative to.
  2. How many cycles to *wait* for. (Currently assumed to be an integer ≥ 1.)
- `reset(...)` and `wait(...)` may only be used in the context of a sequence. If a domain is being reset or waited on, any subject streamlets which depend on that domain cannot be asserted on. Likewise, resets and waits can be applied in parallel with other resets and waits (e.g., waiting on two domains at the same time, moving to the next stage if both have passed), but cannot be applied to the same domain in the same scope.
- As in structural implementations, Streamlet definitions with only the *default* domain can still be assigned a domain on instantiation. Tests require that domains are explicitly declared.

5 Implementation

As discussed in the Methodology Section (1.2), in order to demonstrate the intermediate representation's capabilities and evaluate various approaches, a prototype toolchain was implemented[1] over the course of the thesis. This toolchain consists of a query system for storing and retrieving the IR's declarations and expressions on-demand, a preliminary grammar and parser which stores its results in the query system, and a backend which uses the query system and emits VHDL.

5.1. Query System

The first component of the prototype toolchain is the query system for storing and computing information of the IR. The decision to use a query system rather than more traditional passes of compilation was inspired by ongoing work on the Rust compiler [37] and implemented using the Salsa framework [38]. The advantage of such a system is that information can be retrieved or computed on-demand, and the results of previously executed queries are automatically stored, and only re-computed when their dependencies change.

The query system currently performs the following tasks:

- **Storing information** — The query system stores types, Interfaces, Streamlets and Implementations. The query system also tracks Namespaces, Projects, and the declarations therein, but those declarations are ultimately stored as identifiers of the query system's database.
- **Validation** — The query system is responsible for validating definitions against the Tydi specification, well before a backend is able to extract any information. For example:
  - *Name*s used as identifiers for ports, Streamlets and declarations must be formatted correctly, according to the Tydi specification (as mentioned in Section 2.2). Likewise, it ensures that identifiers are unique where relevant (such as in ports of Interfaces).
  - All ports of an Interface with explicitly named *domain*s must be assigned a domain.
  - *Link*s to behavior must be correctly formatted paths.
  - Streamlet *instances* in a structural implementation are correctly assigned their domains, and assigned default domains if the parent Streamlet has a default domain.
  - *Connection*s between ports of interfaces must have compatible directions, types, and domains, as explained in Section 3.1.3.
  - Once a structural implementation is being stored, all ports of all instances and the parent Streamlet must have been connected.
- **All Streamlets** — Regardless of how they are organized in namespaces, the query system is able to retrieve all declared Streamlets and automatically set appropriate *Path Name*s based on the namespace they were a part of.

[1] `https://github.com/matthijsr/til-vhdl`

- **Physical streams** — In order to represent the logical Stream definitions used for ports in hardware, they must be converted to physical streams, as described in Section 2.2.3. The query system performs this conversion, and also tracks which logical types the physical streams themselves originally related to.

Another use-case for the query system is the high-level assertions described in Section 4.1; converting abstract streams of data on a logical Stream into appropriate, generic calls to the signals that make up its physical streams. Through these functions, a backend would only need to implement the methods for addressing physical streams in order to support these complex, abstract assertions.

While these are still a work in progress, Section 4.2 showcases how the query system is already capable of taking abstract transfers of data structures and converting them into appropriate addressing of a physical stream. As explained in Section 4.2.3, the query system can take on the bulk of the work implementing high-level assertions once a backend implements the specifics for addressing individual physical streams.

5.2. Grammar and Parser

While the query system is effectively an implementation of the IR in its own right, text-based representations are more portable and can allow for more flexible expressions. Furthermore, a purpose-built language reduces the amount of scaffolding required when testing complete projects in the IR, as compared to setting up the query system manually.

To this end, the prototype toolchain also features a simple grammar (referred to as Tydi Intermediate Language, or *TIL*) and parser, implemented using Chumsky [13]. Using the parser, a project expressed in TIL can be stored in the query system. TIL also served as a more stable target for a front-end, computation-oriented language (called Tydi-lang) which was being developed in parallel with the IR by Yongding Tian, as mentioned in Section 1.1.

5.2.1. Parsers

Before designing a grammar, it was necessary to determine which libraries were available to parse the intermediate language. The initial requirements for such libraries were relatively simple:

1. The library needs to target Rust, as this is what the query system was written in. Adding an interfacing pass between another language and Rust would not be a productive use of time.

2. The parsing method needs to support lexing (tokenization) and (integrate with) some form of evaluation, in addition to conventional syntax parsing (producing an abstract syntax tree). The goal is to avoid needing to rely on multiple different parser libraries.

3. Defining a grammar in the parser needs to be well-documented, ideally with examples provided as part of the documentation, or through other users' projects. As building a parser and defining a grammar is not the primary goal of this thesis, it should not require too much time.

Based on these requirements, *crates.io*'s list of most downloaded "grammar" crates[2] and a cursory search through *Y Combinator* and the Rust subreddit (e.g., [1, 36, 50]), it was possible to produce a shortlist of potential candidates based on other users' experiences. This shortlist is included here as Table 5.1.

Name	Kind
lalrpop [26]	LR(1) (**L**eft-to-right, **R**ightmost derivation in reverse, **1** lookahead symbol)
lrpar [16]	LR(1)
nom [14]	combinator
chumsky [13]	combinator
rust-peg [29]	PEG (Parsing Expression Grammar)
pest [33]	PEG

Table 5.1: A list of parser libraries evaluated for this project

[2] `https://crates.io/keywords/grammar?sort=downloads`

Figure 5.1: Errors from the evaluation pass, rendered in the terminal by *Ariadne*.

Of these, *nom* and *Chumsky* directly provide (macro) functions to build and combine parsers in Rust. *LALRPOP*, *lrpar* and *rust-peg* all allow users to define grammars in a separate syntax, which is subsequently converted into Rust code to be referenced; *lrpar* is somewhat notable for using the existing *Yacc* syntax, rather than a custom one. *pest* also relies on an external grammar definition, but only exposes pre-defined functions, rather than generated ones to be imported.

In order to quickly determine which parser library was best suited for quickly defining a grammar and parser, each library's documentation (and possible examples) were followed to the point it was possible to define and evaluate a simple programming language. Of these, *lalrpop*, *Chumsky* and *nom*'s documentation was easiest to follow, featuring clear tutorials and ample examples, though *nom*'s largely stopped short of parsing programming languages. *Chumsky*'s separate example parsers were not immediately functional, due to the methods they relied on receiving breaking changes in the interim, but were relatively easy to repair after finishing the tutorial and using the IDE to automatically suggest changes. The fact that *Chumsky* directly employs Rust functions was ultimately what resulted in a decision in its favor, as this is what allowed it to integrate with the IDE's (Visual Studio Code with *rust-analyzer*) existing analysis and suggestion capabilities.

This direct integration with Rust also meant that a parser built in Chumsky could directly use existing types and functionality built for the query system, such as using the `Name` constructor to determine and store valid identifiers, or simple use-cases such as re-using the existing *Direction* enumeration for "Forward" and "Reverse". This benefit extends to the evaluation pass, which amounts to more Rust functions to interpret the abstract syntax tree; this meant that rather than building custom functionality for tracking identifiers and validating statements, it was possible to directly store declarations in the query system, and rely on its errors and validation.

A final, interesting but non-essential quality of Chumsky is that its errors (using *spans* of character positions) are easily rendered by its sister project, *Ariadne* [12]. *Ariadne* allows a compiler/parser to emit labelled and color-coded errors to the terminal, not unlike Rust's own *rustc* compiler. The quality of these errors will ultimately depend on the implementation, but chumksy's error recovery strategies and reporting is quite flexible. Figure 5.1 illustrates how errors emitted by the parser are human-readable, how Chumsky's error recovery allows for multiple errors to be detected and reported within the same file, and how spans can be propagated even to the evaluation pass.

Theoretically, it would be possible to use the spans and abstract syntax tree emitted by the parsers built in Chumsky to build a *language server* for an IDE (specifically Visual Studio Code's Language Server Protocol), to provide support for in-line color-coding of syntax, linting of errors, and other analysis. There is no straightforward path to building such a service, however, and so no effort was made towards it beyond an initial cursory exploration of the possibilities.

5.2.2. Grammar

TIL features expressions for declaring namespaces, types, Interfaces, Streamlets and Streamlet implementations, as well as some syntax sugar for subsetting Streamlets into interfaces. This grammar has been fully implemented in the prototype toolchain, in that it can also be emitted to VHDL using the backend described in the next subsection.

Namespaces are simple containers for other declarations, their only innate property is their name, which can be expressed as a *path*. Note that paths in this context are purely abstract, and do not reflect any hierarchy in the grammar or IR itself, they can simply be used to *communicate* hierarchy to a backend, and/or propagate it from a front-end.

```
namespace example::name::space {
    ...
}
```

Path separator

The **types** described in Section 2.2 can be declared using the *type* keyword, an identifier, and an expression. Type expressions either reference these identifiers, or directly describe the type's properties.

```
type identifier = Type Expression ;

identifier

Null

Bits(8)

Group(field_name: Type Expr., field2: ...)

Union(field_name: Type Expr., ...)

Stream(data: Type Expr., throughput: ...)
```

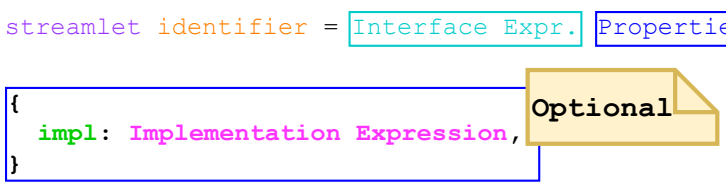
Interfaces, as described in Section 3.1.2 are collections of ports and (clock and reset) domains. They can be separately declared with an identifier, to enable reuse.

```
interface identifier = Interface Expr. ;

identifier

(port_name: in Stream Type Expr., port2: out ...)

<'domain, ...>(port_name: in Stream Type Expr. 'domain, ...)
```

There are two kinds of **implementations**, *links* to behavior, and *structural* implementations which connect Streamlets declared in the IR. This is elaborated on in Section 3.2. Links simply use double-quotes to enclose a path to a directory, while structural implementations are scopes with two kinds of statements: One to create a Streamlet *instance* and connect the Interface's domains, and another to connect ports between instances and/or the enclosing Streamlet.

```
impl identifier = Implementation Expr.;

identifier

"./path/to/directory"

{
  instance_name = Streamlet Identifier;
  parent_port -- instance_name.instance_port;
}

instance = id<'parent_domain, 'instance_dom2 = 'parent_dom2>;
```

Streamlets are a combination of the expressions above, and consist of an Interface and optionally an implementation. These are intended to be the output of a backend.



Finally, **Documentation** is expressed by enclosing text with # signs, and must precede their subject, as shown in Listing 5.1. As explained in Section 3.1.2, documentation is distinct from comments in that it is an actual property of Streamlets, ports, and implementations.

```
#documentation (optional)#
streamlet comp1 = (
    // This is a comment
    a: in stream,
    b: out stream,
    #this is port
documentation#
    c: in stream2,
    d: out stream2,
);
```

Listing 5.1: Documentation Example

For a complete example of TIL, see Listing A.1.

5.2.3. Parser Implementation

The overall TIL parser performs the following passes:

1. Lexing; converting the initial text file(s) into tokens.
    - The primary token categories (in order of parsing priority) are: Documentation, versions (e.g., *1.0.1*), numbers, path strings, operators, control characters, keywords, and identifiers.
    - Anything not interpreted as a token (mainly whitespace) is treated as padding, and ignored. This includes comments. (Comments start with `//`, or are enclosed by `///` when multi-line.)
    - Identifiers and keywords are continuous strings of characters, not separated by whitespace, operators or control characters.
    - Documentation is any character (except `#`), enclosed by `#`s.
    - The lexer pass recovers by simply skipping to the next input and retrying once all subsequent inputs have been parsed. This is not especially robust (i.e., it may not detect multiple errors), but simple to implement.
2. If the lexer pass succeeds, its resulting tokens are parsed to an abstract syntax tree (AST).
    - The root node of any syntax tree is the *namespace*; a namespace contains any number of statements.
    - Statements in the current version of TIL are only declarations, but may be expanded to include imports and potentially certain intrinsic functions.
    - There are four kinds of definitions to be declared (in order of parsing priority): Types, implementations, interfaces, and streamlets.
    - Definitions themselves are expressions, and may use an identifier to reference a prior declaration.
    - Identifier tokens are further combined into Path Names (Names separated by `::`), (port/field) labels (Names followed by a `:`) and domain names (Names preceded by a `'`). Though whether each parser is applied depends on context; e.g., labels only exist in ports lists of interfaces, while domain names only exist in domain lists of interfaces and domain assignments.

- Structural implementation definition bodies, despite being part of an expression, parse a list of statements. Structural body statements are Streamlet instantiations or port connections.
- This expression parsing can recover from some errors by looking for the next delimiter (e.g., if an error occurs parsing an interface `( ... )`, it will simply look for the closing `)`) and storing this as an `Error` node.

3. If the AST parsing pass succeeds, all declarations are *evaluated*.

   - Identifiers are converted to Names and Path Names, using the existing `try_` constructors built for the query system, which consume an arbitrary string and then validate whether they match the Tydi requirements for identifiers.
   - Each declaration is immediately stored into the query system's database during evaluation.
   - The *interned* identifiers of each declaration are stored in a HashMap for that type of declaration, using the identifer as a key. While the query system already ensures that identifiers are unique and exist, this method allows for better error recovery and error messages. (And while this was not implemented, pairing the original declaration with a span could allow for error messages which directly reference the previous declaration.)
   - When evaluation of a declaration fails, this pass recovers by storing an `EvalError` node. Subsequent declarations are still evaluated, and all EvalErrors are reported (and rendered) once all declarations have been evaluated.
   - Structural definition body statements are also evaluated during this pass (as part of the implementation expression/declaration evaluation), errors are still recovered as part of the complete declaration, however. (I.e., only the first error in a structural body will be reported, and all other statements are ignored.)

Note that while in this implementation, each pass is only executed if the previous succeeded, this is not actually a requirement of Chumsky or the parsers implemented in it. Provided a pass has sufficiently robust error recovery, performing the next pass is a viable way of reporting more comprehensive errors. Figure 5.2 illustrates what happens when evaluation is allowed to occur despite AST parsing errors (this can be achieved by removing the initial `return Err` from the parser library's `into_query_storage` function). This is an optimistic scenario, however, as some AST errors (especially those involving delimiters) will still prevent all subsequent parsing.

5.3. VHDL Backend

In order to verify that the IR could actually be compiled to a hardware description, a VHDL backend was incorporated into the prototype toolchain. As all concepts expressed in the IR would need to be emitted to VHDL, this helps explore which properties are necessary or helpful for targeting hardware.

VHDL was chosen as the target because it is well-supported by multiple toolchains for both synthesis and simulation, and simply because its syntax was personally most familiar. Similar methods as those for emitting VHDL can be employed when emitting other hardware description languages, such as Verilog, FIRRTL and LLHD.

The "passes" used when emitting to VHDL in this example backend are intentionally very simple (for instance, while namespaces could correspond to their own VHDL packages, all namespaces are instead combined into a single package), though they do leverage the the query system's ability to incrementally compute and retrieve information:

1. The "all streamlets" query described in Section 5.1 is used to retrieve all the Streamlet declarations in the project.

2. For each Streamlet, the Streams that make up its Interface are split into physical streams, of which the signals are converted into ports. These ports make up a component with a unique name based on the Streamlet declaration and the namespace in which it was declared. These components are added to a single VHDL package.

3. For each Streamlet, an architecture declaration is either imported or generated, as discussed in the next subsections.

Figure 5.2: Errors simultaneously reported from both the abstract syntax tree parsing pass and evaluation pass.

5.3.1. Components and Organization

As mentioned in the preface, each Streamlet is converted to a unique *component* definition, regardless of how it is implemented. Taking the following namespace and Streamlet declaration, as an example:

```
namespace my::example::space {
    type stream = Stream (
        data: Bits(8),
        dimensionality: 0,
        synchronicity: Sync,
        complexity: 4,
    );

    #Streamlet documentation#
    streamlet comp1 = (
        a: in stream,
        b: out stream,
        #Port
documentation#
        c: in stream,
        d: out stream,
    );
}
```

Listing 5.2: A simple namespace containing a Streamlet declaration

We see that there is a namespace called *my::example::space* and a single Streamlet declaration called *comp1*. When the VHDL backend imports all Streamlets, the namespace is combined with the declaration identifier to form a unique identifier for the whole project, *my::example::space::comp1*. As multiple underscores are not valid in Tydi's *Name*s, we can safely use two underscores to represent the path separators, making for `my__example__space__comp1_com`. The `_com` suffix being a holdover from the original Tydi VHDL interface generator, which used it to distinguish *canonical* representations of interfaces compared to the *"fancy"* equivalents which used record types for better readability, as implementing similar functionality in the new backend is recommended as future work (as described in Section 6.2).

Likewise, the signals that make up the physical stream(s) split from each port receives a prefix based on that physical stream's name, for clarity. E.g., the *data* signal of port `a` becomes `a_data`. This results in the following package and component definition:

```
package proj is

  -- Streamlet documentation
  component my__example__space__comp1_com
    port (
      clk : in std_logic;
      rst : in std_logic;
      a_valid : in std_logic;
      a_ready : out std_logic;
      a_data : in std_logic_vector(7 downto 0);
      b_valid : out std_logic;
      b_ready : in std_logic;
      b_data : out std_logic_vector(7 downto 0);
      -- Port
      -- documentation
      c_valid : in std_logic;
      c_ready : out std_logic;
      c_data : in std_logic_vector(7 downto 0);
      d_valid : out std_logic;
      d_ready : in std_logic;
      d_data : out std_logic_vector(7 downto 0)
    );
  end component;

end proj;
```

Listing 5.3: The TIL from Listing 5.2 converted to a VHDL package.

Note that the documentation from Listing 5.2 is converted into comments in Listing 5.3. Documentation on Streamlet declarations is added above the respective VHDL component and entity declarations, documentation on a structural implementation is added above the architecture declaration, and documentation on the (IR) ports of Interfaces is added above the sets of (VHDL) ports that make up the logical Stream.

The *clock* and *reset* signals of the component are simply called *clk* and *rst*, as the Streamlet's interface only features the unnamed *default* domain. Should it feature explicitly named domains, however,

such as <'domain_name>, this will take the form domain_name__clk and domain_name__rst instead. The relation between ports and specific domains is only tracked within the IR, as there is no way to reflect this property in VHDL directly.

The architecture definitions are stored in separate .vhd files named after the produced components, e.g., my__example__space__comp1.vhd. This output is emitted to the same directory as the package definition. Each architecture also imports the package it was declared in by default, which simplifies the use of other components in structural implementations:

```
library work;
use work.proj.all;
```

5.3.2. Linked Implementations

Linked implementations are expressed as paths to a directory, as explained in section 3.2.2. For example:

```
streamlet comp2 = comp1 {
    impl: "./vhdl_dir"
};
```

How to use these paths is to be decided by each backend: The current VHDL backend simply checks whether a file matching the naming scheme specified before exists in the directory (my__example__space__comp2.vhd, in this case), and generates an empty architecture at that location if one does not exist. Then, the file is directly copied to the output. Note that as Streamlets are independently converted to component definitions for the package, any linked implementation must match the generated definition exactly for the project (and other uses of the Streamlet) to work correctly.

5.3.3. Structural Implementations

Structural implementations represent the bulk of the VHDL backend's computation, as these involve defining a (non-empty) architecture. The statements and properties the backend must implement are as follows:

1. Instantiating Streamlets (a = comp1); the backend must implement creating a named instance of a Streamlet, to be used throughout the rest of the implementation.
2. Connecting ports (a.a -- a.b); the backend must allow for ports of both instances and the parent Streamlet (made up of one or more physical streams and therefore multiple signals) to be connected.
3. Assigning domains to instances (a = comp<'a, 'b>); the backend must assign the appropriate clock and reset signals from the parent streamlet to an instantiated child Streamlet.

To demonstrate how the backend implements this functionality, we will define a simple example, extending the previous listings:

```
streamlet domains_only = <'a, 'b, 'c>();

streamlet comp3 = <'x, 'y>(
    q: in stream 'x,
    r: out stream 'x,
) {
    impl: {
        dom_ex = domains_only<'x, 'y, 'y>;
        inst = comp2<'x>;
        q -- inst.a;
        r -- inst.b;
        inst.c -- inst.d;
    }
};
```

Listing 5.4: Declaring a Streamlet with multiple domains, and a structural implementation.

To start, instances of Streamlets can be represented as simple *port mapping*s of their respective component in the architecture. The name of the instance can be reflected as a label, e.g.:

```
dom_ex: my__example__space__domains_only_com port map( ...
```

As ports of instances need to be connected to both the parent Streamlet and other instances, the backend first defines a set of signals matching each port, allowing them to be connected in different parts of the architecture, rather than during port mapping. These signals are given unique names by suffixing their name with the instance's name, e.g.: `inst__a_valid`

Domain assignments are defined directly on instantiation, however, and always draw from the parent Streamlet's domains. This means the clock and reset signals can be assigned directly on port mapping:

```
inst: my__example__space__comp2_com port map(
  clk => x__clk,
  rst => x__rst,
  ...
```

Finally, one last property to account for is that while physical streams have a single overall "direction" in Tydi itself, they are still made up of signals with different directions. In particular, the *ready* signal will always be reversed relative to the other signals; as such, its assignment must be reversed in the resulting VHDL:

```
inst__a_valid <= q_valid;
q_ready <= inst__a_ready;
inst__a_data <= q_data;
r_valid <= inst__b_valid;
inst__b_ready <= r_ready;
r_data <= inst__b_data;
```

The full architecture (and TIL namespace) can be found in Appendix B.

5.3.4. Additional and Future Functionality

While structural implementations are the most complex fully-implemented feature of the backend, they do not reflect the full functionality implemented over the course of the thesis. For reference, the VHDL backend actually consists of two *crate*s (Rust libraries): *til-vhdl* and *vhdl*. The latter is a library which is solely focused on programmatically defining and *validating* VHDL, independent from Tydi or the IR, while the former simply uses it to convert the IR into VHDL.

These libraries were split to ensure that the concerns of *generating correct VHDL syntax* and of *converting the IR to VHDL* could be separated. As a result, the *vhdl* library is capable of generating various VHDL statements, expressions and properties which were not used by *til-vhdl*, such as:

- *Processes*, which were employed by the proof-of-concept for high-level assertions described in Section 4.2, along with...
- *Assertions*, which can compare values/signals and report an error message if they do not match.
- *Types* other than *std_logic* and *std_logic_vector*; *vhdl* actually supports booleans, *time*s, and arbitrary *array* and *record* types, along with constant expressions of *severity*.
- *Constant expressions* and *relations*, as shown in Section 4.2, the *vhdl* library is also able to emit constant expressions of bits and bit vectors, though it can do the same for arrays, records, booleans and times (in various units). It can combine these as relations using using various operators (equality, greater/less than, logical operators, etc.), which themselves form (associative, boolean) relations.
- *Imports*; while the current implementation only requires the *ieee* and project package imports, the vhdl library can easily track multiple imports and prevent duplicate imports.
- Support for *variables* and *constants* in addition to *signals*.

The reason the *vhdl* library has support for these is because, while *til-vhdl* often only required a small subset of VHDL, it was not significantly more difficult to implement a more complete and correct set of the relevant VHDL syntax. As a result, future functionality of *til-vhdl* can be implemented more easily.

5.4. Example

As an end-to-end example of the toolchain, the repository [28] features a `demo-cmd` command-line application project which incorporates the parser, query system and VHDL backend. This project has been verified to compile and run using the Rust compiler `rustc`[3] version 1.61.0 on both Windows 10 and Ubuntu 20.04 (through Windows Subsystem for Linux 2). To try the demonstration yourself, follow the following steps:

1. Clone the repository (e.g., `git clone https://github.com/matthijsr/til-vhdl.git`)
2. Switch to the `demo-cmd` directory (`cd demo-cmd` from the repository's root, or `cd ./til-vhdl/demo-cmd/` if you have just cloned the repository)
3. Build the application with `cargo build`, this will also install any dependencies.
4. Run the application with `cargo run ./til_samples/paper_example.til ./output`, where `./til_samples/paper_example.til` is the input TIL file, and `./output` is the output directory.
5. Once compilation has succeeded, there should be a `proj` directory in the chosen output directory, inside of this directory are the resulting `.vhd` VHDL architecture definitions and package. (As projects were not implemented in TIL, "proj" is used as the default project name.)

```
PS C:\Users\matth\Projects> cd ./til-vhdl/demo-cmd/
PS C:\Users\matth\Projects\til-vhdl\demo-cmd> cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.09s
PS C:\Users\matth\Projects\til-vhdl\demo-cmd> cargo run ./til_samples/paper_example.til ./output
    Finished dev [unoptimized + debuginfo] target(s) in 0.09s
     Running `C:\Users\matth\Projects\til-vhdl\target\debug\til-demo.exe ./til_samples/paper_example.til ./output`
PS C:\Users\matth\Projects\til-vhdl\demo-cmd>
```

Figure 5.3: Successfully building and running the `demo-cmd` example application.

Once the application has been set up, users are free to try other TIL files and output directories. For example, the `./til_samples/evaluation_axi.til` file was used for the evaluation in Section 6.3. Users may write their own TIL files, or modify the existing ones. (For instance, to try out error reporting by changing `same_domains = dom_example<'parent_domain1, 'parent_domain1>;` to `same_domains = dom_example<'parent_domain1, 'parent_domain2>;`.)

5.5. Partial Implementations

As noted in their respective sections, work on some features had begun, but did not reach a complete or satisfactory state within the timeframe of the thesis. To summarize:

- (Section 4.2) Transaction-level assertions were partially implemented at the physical stream level, but lack support for logical streams or other features discussed in the chapter overall.
- (Section 3.4) Imports of declarations from other sources, and minimal support for projects have been implemented on the query system, but lack support from the parser.

[3] `https://doc.rust-lang.org/cargo/getting-started/installation.html`

6

Evaluation

The prototype toolchain was developed not just as a demonstration, but also to test different approaches and verify their effectiveness. This section contains the evaluation of the prototype toolchain and its features.

6.1. Tydi Specification

As a result of explicitly translating the Tydi specification to code, a few oversights and contradictions in the specification came to light. Fortunately, it was possible to discuss the intent with the designers of the specification (Johan Peltenburg, Jeroen van Straten, and Matthijs Brobbel), who were originally part of the ABS group. As such, in addition to submitting these as issues on the specification's GitHub, I was able to directly propose and utilize (interim) solutions for the purposes of my toolchain, or determine the original intent.[1] The following subsections describe the issues I have found and their proposed and/or interim solutions, their contents are adapted from the submitted reports.

6.1.1. Directly nested Streams which must both be retained

Report `https://github.com/abs-tudelft/tydi/issues/221`

Background When a Stream contains another Stream as its *data*, the *Split* function[2] assigns both the parent and child streams "∅" (empty name), and employs "flattening" to combine their *throughput*, *synchronicity*, *dimensionality* and *direction*.

When a Stream has no element-manipulating data (*data* is either Null or a Stream) and no *user* property, it is discarded from the result. In effect, this creates a new physical stream with the original child Stream's *data*, combined with the parent Stream's properties.

Issue When *keep* (x) is true and/or *user* (T_u) is non-Null, the parent Stream must be retained.

If a parent Stream has $x = true$ and/or a non-Null T_u property, both Streams are still assigned "∅", but the parent Stream will conflict with the child Stream. Implementing the result of the Split function as a map in code, this means that either the child Stream simply replaces the parent Stream altogether (thereby losing the parent Stream's user property), or the Split function fails. Additionally, this does not account for child Streams having a *synchronicity* which flattens its *last* signal on the assumption that the (now non-existent) parent stream will drive the *last* bits for its enclosing dimensions.

However, this behavior is not described in the specification, it only specifies that the names resulting from the Split function "are case-insensitively **unique**, emptyable strings consisting of letters, numbers, and/or underscores, not starting or ending in an underscore, and not starting with a digit" (emphasis mine).

Interim solution In the toolchain's implementation, the Split function will fail when it encounters a situation where two physical Streams have identical names.

[1] Also tracked on this project's repository, here: `https://github.com/matthijsr/til-vhdl/issues/81`
[2] `https://abs-tudelft.github.io/tydi/specification/logical.html#split-function`

Proposed solutions Explicitly specify that it is illegal for nested Streams (Streams which only have another Stream type as their *data*) to have a *keep* and/or *user* property on more than one of these Streams, and "flattening" should incorporate the singular user property into the resulting physical stream.

Alternatively, Streams should have a non-empty *name* property, or directly-nested children with retained parents should automatically receive one (e.g., `data` or `child`). This will avoid conflicts in the result of the Split function, regardless of whether these names are unique. (As nested Streams will simply have the parent and child Stream names joined in a "Path Name", as is currently the case for field names in Groups and Unions.)

6.1.2. Significance of Strobe and Index Signals

Report `https://github.com/abs-tudelft/tydi/issues/223`

Background Tydi provides three different signals to indicate whether the lanes of a *data* signal are active during a transfer. At *complexity* $C \geq 7$, the strobe (*strb*) signal can encode the validity of every lane independently. At *complexity* $C < 7$, the start index (*stai*) and end index (*endi*) encode which *range* of lanes is active instead; as both are 0-indexed, however, such Streams also have a single *strb* bit to indicate whether *all* of the transfer's data lanes are inactive.

When a Stream with $C < 7$ is a source to a Stream with $C \geq 7$, it will drive all *strb* bits simultaneously from its single *strb* bit, and the higher-complexity sink must instead interpret the *stai* and *endi* signals.

Issue While Tydi's *data* signal specification[3] indicates that the *stai* and *endi* signals are redundant when $C \geq 7$, it is not entirely clear whether they are still *significant*.

Since the *strb* signal still exists at $C < 7$ as a single bit to indicate that all of transfer's data lanes are inactive, considering the inverse, *stai* and *endi* must be significant despite *strb* indicating that "all lanes" are active. Otherwise, these signals would be unable to encode lane activity. At the same time, while *stai* and *endi* are insignificant when all *strb* bits are driven low, it is not clear whether this applies when some *strb* bits are high and some are low.

As an example of these conflicts, consider the following examples:

1. $stai = 0$, $endi = 0$ and $strb = "010"$
2. $stai = 0$, $endi = 0$ and $strb = "101"$

In both instances, the start- and end-indices indicate that only the first lane is active. However, in the first instance the *strb* indicates that the first lane is inactive, but the second lane is active. In the second instance, the *strb* does indicate that he first lane is active, but also indicates that the third lane is active.

Proposed solution The start- and end index signals should only be significant when *all* strobe signal bits are driven high. This ensures lower-complexity sources can still connect to higher-complexity sinks, but does not allow for sources with $C \geq 7$ to create confusing transfers.

6.1.3. Transferring Empty Outer Sequences at Lower Complexities

Report `https://github.com/abs-tudelft/tydi/issues/224`

Background The specification for the *last* signal[4] notes different constraints for complexities $C < 4$ and at $C < 8$, with the intent of placing certain requirements on sources transferring sequences. The constraints at $C < 4$ are as follows:

1. "It is illegal to assert a *last* bit for dimension j without also asserting the last bits for dimensions $j' < j$ in the same lane."
2. "It is illegal to assert the *last* bit for dimension 0 when the respective data lane is inactive, except for empty sequences."

The intention of these rules is to prevent source Streams with $C < 4$ from postponing *last* flags of outer dimensions to subsequent transfers. For instance, when transferring *[[data, data]]*, the source

[3] `https://abs-tudelft.github.io/tydi/specification/physical.html#data-signal-description`
[4] `https://abs-tudelft.github.io/tydi/specification/physical.html#last-signal-description`

cannot first perform a transfer with all data and $last = "01"$ (last in dimension 0), followed by an empty transfer with $last = "10"$ (last in dimension 1).

Issue The first constraint prevents Streams with $C < 4$ from transferring empty outer sequences. That is to say, they cannot perform a transfer *[]* ($last = "10"$), and may only transfer *[[]]* ($last = "11"$) instead. As a result, the example sequence from the *last* signal specification cannot actually be transferred at lower complexities:

```
["Hello", "World"], ["Tydi", "is", "nice"], [""], []
```

Complexity is meant to be a property affecting *how* data can be transferred and how Streams are physically implemented: These constraints mean that the complexity property also affects what kind of data can be transferred.

Proposed solution The first rule for $C < 4$ (requiring *last* in dimensions $j' < j$) should be amended with an exception for empty sequences, just like the second rule. This ensures *complexity* does not affect what kinds of data can be transferred, while still ensuring *last* flags cannot be postponed at lower complexities.

6.1.4. Indicating Inactive Lanes at Lower Complexities

Report `https://github.com/abs-tudelft/tydi/issues/226`

Background The specification for signal omission[5] places the following constraints on the start index (*stai*), end index (*endi*) and strobe (*strb*) signals which govern whether element (*data*) lanes in a transfer are active, based on complexity C, number of element lanes N, and dimensionality D:

1. *endi* is contingent on $(C \geq 5 \vee D \geq 1) \wedge N > 1$
2. *stai* is contingent on $C \geq 6 \wedge N > 1$
3. *strb* is contingent on $C \geq 7 \vee D \geq 1$

If these constraints are not met, a physical stream is unable to indicate whether individual element lanes are inactive. As these constraints are part of the signal omission specification, it is implied that such streams have no need to do so.

Issue When a Stream has properties $C < 5 \wedge D = 0 \wedge throughput > 1$, its physical implementation will have multiple element lanes, but lack the ability to indicate whether they are inactive. This means each transfer *must* consist of exactly N elements

Proposed solutions This constraint is either an oversight, or should be clarified to be an actual requirement for lower complexity Streams, rather than a physical implementation detail:

1. If it is an oversight, the requirement for *endi* being contingent on $(C \geq 5 \vee D \geq 1) \wedge N > 1$ should be changed to being solely contingent on $N > 1$.
2. If it is intentional, and physical streams should be able to transfer arbitrary sets of elements, the number of element lanes N being greater than 1 should be contingent on $D > 0 \vee C \geq 5$.
3. If it is intentional overall, the requirement for Streams with complexity $C < 5$, $D = 0$ and *throughput* $t > 1$ to transfer only sets of elements equal to or divisible by $\lceil t \rceil$ should be specified as part of the logical Stream specification, as well.

6.1.5. Minor Inconsistencies

Reports

1. `https://github.com/abs-tudelft/tydi/issues/222`
2. `https://github.com/abs-tudelft/tydi/issues/225`

[5] `https://abs-tudelft.github.io/tydi/specification/physical.html#signal-omission`

Background, Issue and Solution These contradictions are relatively minor inconsistencies related to phrasing and constraints conflicting over multiple separate requirements:

1. The signaling specification states that when $C < 8$, the *last* bits for lanes 0 through $N-2$ must be driven low, suggesting that the singular *last* value applies to lane $N-1$. One of the constraints for $C < 4$ mentions that "It is illegal to assert the *last* bit for dimension 0 when **the respective data lane** is inactive, except for empty sequences." (emphasis mine). As a result, sequences which do not align with the number of element lanes (and do not have a start index signal, so must align to the first lane) would not be able to assert *last*.

    - The solution is to change the phrasing for the $C < 4$ rule to refer to *transfer* data, rather than a data lane.

2. The specification inconsistently refers to the strobe (*strb*) signal encoding whether individual data lanes are active, with some constraints suggesting this is only the case at $C \geq 8$, while others suggest this applies to $C \geq 7$.

    - This is simply an oversight: $C \geq 7$ must allow *strb* to encode individual lane activity, as otherwise $C = 7$ is identical in functionality to $C = 6$. ($C = 8$ adds the ability to encode a *last* value per lane over $C = 7$.)

6.2. Readability

6.2.1. Readable Output

As the IR relies on other languages to express functionality, it will generally be necessary for the descriptions a backend *does* generate to be readable by designers, barring a frontend emitting both the IR and the behavioral descriptions. To this end, the IR exposes "documentation" to backends, enabling designers to propagate some intent to component templates and interfaces. The prototype VHDL backend propagates this documentation as comments, and generates indented VHDL with port and signal names derived from the TIL port and field names.

There is one area in which much information and readability is lost, however: The physical streams emitted by the VHDL backend feature standard *data* and *user* signals as bit vectors, meaning that the names of element fields of Groups and Unions are lost. As described in Section 3.3.3, the Tydi documentation describes alternative ways to represent physical streams to retain this information. For instance, Groups and Unions could be expressed as record types in VHDL, multiple element lanes as arrays of the base type, and even physical streams themselves could be collected into records (split into separate records for up and downstream signals). These are not only useful for implementation, but can also provide more information when simulating a design.

In fact, the *Implementations* section of the original Tydi paper [32] assumes that designers would prefer such a solution, and illustrates that automatically generating such records from Tydi logical types would greatly reduce the number of lines of code designers would need to write. To better enable such alternative representations, making changes to the IR to require type identifiers, rather than storing only the official properties of logical types may prove beneficial, as described in Section 3.1.3. Doing so would allow a backend to generate alternative representations with meaningful type names, which could then be directly reused by multiple interfaces, albeit at the expense of the ability to directly connect physically compatible types.

6.2.2. Type Identifiers

The initial approach towards the IR was to stay very close to the Tydi specification itself, and avoid any added or divergent functionality. As the Tydi specification did not feature identifiers for types, this would be diverging from the specification. Even if these identifiers were only tracked by the IR itself and not propagated to the backend, using them to determine "compatibility" appeared to be too opinionated.

However, as the previous section and Section 3.1.3 reflect, identifiers being a property of types can yield significant benefits. Moreover, insisting that types must be compatible based on their definition, and not their identifier, is opinionated in its own way. As such, there are a number of different approaches to implementing type identifiers:

1. Whether identifiers affect compatibility is *configurable*; the existing TIL syntax remains unchanged, but the query system's functions for determining compatibility are configurable, and may either behave as it currently does, or require identical identifiers as well. This allows either the backend or frontend to specify which behavior is preferred.

2. Create explicit distinctions between *anonymous* types, *named* types, and *aliases* of types. This enables the frontend (or designer) to choose which behavior is preferred on a per-project or per-type level, and would be implemented as follows:

   - *Anonymous* types are the current type definitions, they are not declared, but used in declarations or directly on Interface ports: `Bits(8)`
   - *Named* types modify the properties of a type, setting the name; when checking whether types are compatible, their names must also match:
     - `type byte = Bits(8);`
     - `type char = byte;`
     - `char != byte`
   - *Aliases* are how current type declarations work, they are identifiers for types, but do not modify the type's properties, and are only tracked as part of namespaces:
     - `alias char = byte;`
     - `char == byte`
     - `alias reg = Bits(8);`
     - `reg == Bits(8)` and `reg != char`

3. Types *must* have identifiers. This is the most divergent approach from both the current IR and of the Tydi specification, but does have a number of merits:

   - It guarantees that backends have unique identifiers for "fancy" types, ensuring they are also compatible in the target language (e.g., record types in VHDL).
   - By giving Streams explicit names, there can be no conflicts between physical stream names, such as those described in Section 6.1.1.

As an aside, it is worth noting that technically, frontends can already implement a kind of identifier-based compatibility between types, by declaring every type as a `Group` with one field (e.g., `Group(byte: Bits(8))`). Though this puts the onus of tracking the uniqueness of identifiers on said frontends. More importantly, the fact that Tydi already supports compatibility restrictions based solely on identifiers calls the accuracy of anonymous types enforcing "physical compatibility" into question.

6.3. Hardware Description Effort

The goal of the IR is to describe streams carrying complex data structures more effectively than conventional HDLs. As such, while "lines of code" is not an especially relevant metric for an IR overall, it can be applied to the amount of effort required to express interfaces and connections. To evaluate the IR's effectiveness in this regard, Tydi equivalents of the AXI4-Stream [7] and AXI4 [8] interface standards were declared in TIL.

Table 6.1 shows the signal specification of AXI4-Stream, while Listing 6.1 shows how it was implemented as a Tydi Stream for the purposes of this evaluation, along with the resulting (VHDL) signals in Listing 6.2. AXI4 was spread over 5 Streams for Address Write, Write Data, Write Response, Address Read, and Read Data. Appendix C shows the full signal specification of AXI4, while Appendix D shows the full TIL definitions of both AXI4-Stream- and AXI4-equivalent Streams, and the resulting VHDL component.

Signal	Source	Description
ACLK	Clock source	The global clock signal. All signals are sampled on the rising edge of ACLK.
ARESETn	Reset source	The global reset signal. ARESETn is active-LOW.
TVALID	Master	TVALID indicates that the master is driving a valid transfer. A transfer takes place when both TVALID and TREADY are asserted.
TREADY	Slave	TREADY indicates that the slave can accept a transfer in the current cycle.
TDATA[(8n-1):0]	Master	TDATA is the primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes.
TSTRB[(n-1):0]	Master	TSTRB is the byte qualifier that indicates whether the content of the associated byte of TDATA is processed as a data byte or a position byte.
TKEEP[(n-1):0]	Master	TKEEP is the byte qualifier that indicates whether the content of the associated byte of TDATA is processed as part of the data stream. Associated bytes that have the TKEEP byte qualifier deasserted are null bytes and can be removed from the data stream.
TLAST	Master	TLAST indicates the boundary of a packet.
TID[(i-1):0]	Master	TID is the data stream identifier that indicates different streams of data.
TDEST[(d-1):0]	Master	**TDEST provides routing information for the data stream.**
TUSER[(u-1):0]	Master	**TUSER is user defined sideband information that can be transmitted alongside the data stream.**

Table 6.1: The AXI4-Stream signal specification, source: [7]

```
type axi4stream = Stream (
    data: Union (
        data: Bits(8),
        null: Null, // Equivalent to TSTRB
    ),
    throughput: 128.0, // Data bus width
    dimensionality: 1, // Equivalent to TLAST
    synchronicity: Sync,
    complexity: 7, // Tydi's strobe is equivalent to TKEEP
    user: Group (
        TID: Bits(8),
        TDEST: Bits(4),
        TUSER: Bits(1),
    ),
);

streamlet example = (
    axi4stream: in axi4stream,
```

Listing 6.1: An AXI4-Stream-equivalent interface in TIL.

```
axi4stream_valid : in std_logic;
axi4stream_ready : out std_logic;
axi4stream_data : in std_logic_vector(1151 downto 0);
axi4stream_last : in std_logic;
axi4stream_stai : in std_logic_vector(6 downto 0);
axi4stream_endi : in std_logic_vector(6 downto 0);
axi4stream_strb : in std_logic_vector(127 downto 0);
axi4stream_user : in std_logic_vector(12 downto 0);
```

Listing 6.2: Result of Listing 6.1 in VHDL.

Once a Stream type has been declared, it can be easily reused for any number of ports, and ports only require one statement (`port_a -- port_b;`) to connect, which is far fewer than the signals which make up a stream (or AXI4 channel). Table 6.2 illustrates this difference: The AXI4-Stream equivalent requires a single Stream overall, while AXI4 requires a Stream per channel, and can be either split across multiple ports, or combined into a Group with Reverse Streams for the Read Data and Response channels, depending on the use case. Both result in identical physical streams, but using multiple ports allows for them to be connected to different Streamlets if necessary.

As an aside, these AXI4 and AXI4-Stream definitions are areas where the type parameters discussed in Section 3.3.1 could be applied very effectively. A type could define the basic requirements for an AXI4(-Stream)-equivalent interface, while using type parameters to set the variable properties of AXI4(-Stream), such as data bus width.

	Type Declaration	Interface
AXI4 equiv. (TIL)	48*	5
AXI4 equiv. (TIL, Group)	59*	1
AXI4 equiv. (VHDL)	-	28
AXI4	-	44
AXI4-Stream equiv. (TIL)	15*	1
AXI4-Stream equiv. (VHDL)	-	8
AXI4-Stream	-	9

Table 6.2: Lines of code to represent an interface in TIL, compared to the resulting number of signals in VHDL or for an equivalent interface standard. *Only required once.

6.4. Parser

As development of a text-based grammar and parser was secondary to development of the query system and VHDL backend, it did not receive as much attention, and work and research started later in the course of the thesis overall. Despite these limitations, the parser developed using *Chumsky* was satisfactory overall; it was possible to very quickly and relatively easily define a grammar and build a parser which translated to the majority of concepts expressed in the query system. Hence, this section will summarize the specific merits and demerits of Chumsky, based on experience using it to build the TIL parser, as well as recommend potential improvements to the TIL parser.

6.4.1. Merits of Chumsky

Overall, Chumsky was easy to work with, and can certainly be recommended for continued work on a TIL parser, or any other projects which might require a domain-specific language parser built in Rust. To summarize the specific advantages:

1. It is (comparatively) easy to work with, featuring a short but descriptive tutorial[6] and many built-in parser functions, such as `delimited_by` to indicate grammar is enclosed by specific symbols, and `foldr` and `foldl` for *folding* right- and left recursive grammar into nested expressions.

2. As parser definitions are fully native to Rust, they integrate well with IDEs (detecting errors, suggesting functions, providing documentation hints) and with other functionality written in Rust (such as functions and types originally created for the query system).

3. Parsers can be split over multiple functions or variables, making them reusable and easy to organize over files and directories.

4. It features built-in error-recovery strategies to use with parsers, enabling it to generate partial ASTs and/or evaluate more of a file, even when errors are encountered. This is not necessarily unique, but is made very accessible.

5. While not directly part of Chumsky, its sister project *Ariadne* [12] can be used to render error reports in the terminal in a clear, color-coded way.

6.4.2. Issues Using Chumsky

While Chumsky proved to be a good fit for the project, and provides many useful features, it does have a number of issues to account for:

1. Chumsky is explicitly not designed to be a high-performance parser, as its repository description notes, “Chumsky focuses on high-quality errors and ergonomics over performance.”[7] While parsing speed was not an issue for the TIL parser’s implementation, it could scale poorly to larger, multi-file projects, and may be ill-suited to other kinds of parsers.

2. While being able to include *spans* of character positions in parsed lexical tokens and nodes of the abstract syntax tree is very useful for error reporting, there does not appear to be a way to quickly

[6] `https://github.com/zesterer/chumsky/blob/master/tutorial.md`
[7] `https://github.com/zesterer/chumsky#performance`

remove them from all nodes/tokens. This is inconvenient when attempting to write unit tests against the parser to assert its output is correct - needing to either manually include the expected spans in the comparison, or write a custom method to remove them. As different parser stages also expect to receive tokens with spans, this also convolutes their input when testing.

3. Chumsky is not portable between different languages: While its parser definitions being native to Rust was marked as an advantage, this also means that the TIL parser cannot be easily translated to a different language altogether. Using independent grammar definitions can reduce the effort of implementing parsers in different languages, although of the parser libraries evaluated, only *lrpar* [16] employed a syntax (*Yacc*) for which libraries exist in other languages.

4. In the event a parser composed of multiple different parsers fails (notably due to stack overflows), it is extremely difficult to find out through debugging or error messages what part of the code caused the issue. This is due to the expanded code resulting from Chumsky's inline and macro functions being difficult to trace back to the source. This is not unique to Chumsky, however, as tracing the source of a stack overflow is rather difficult regardless.

As a note on the last issue, stack overflows were consistently resolved by splitting up larger parsers into multiple functions (to better reason about their operation), and by removing potential sources of ambiguity. One such source of ambiguity causing stack overflows in the TIL parser was that originally, the AST parser would attempt to parse any kind of definition expression (i.e., whether it was an identifier, type, interface, implementation or Streamlet) after a declaration, and select the appropriate ones afterward. Splitting up the parsers for the definition expressions into separate functions, and only combining them with the declaration parser where they were appropriate resolved the stack overflows.

6.4.3. Recommendations for TIL Parser

The following aspects of the TIL parser can be improved, or may be useful additions (note, these are not additions to TIL itself):

1. More robust error recovery strategies applied to all passes, e.g.:
    - The lexer pass currently uses `skip_then_retry`, it may be possible to use `skip_until` for certain constructs, or create a custom strategy.
    - The AST pass has difficulty with unclosed delimiters, it should be possible to create a recovery strategy which skips until the next declaration keyword.
    - The evaluation pass does not recover inside structural implementation definitions, and instead recovers the entire declaration. Instead, an error recovery strategy on every structural implementation statement should be implemented.
2. Attempt to produce more helpful errors; e.g, when a duplicate identifier error occurs, also point towards the previous declaration.
3. As a larger change/addition, attempt to emit the parser's results to a language server (protocol), so an IDE can provide syntax highlighting and error linting.

7 Conclusion

7.1. Conclusions and Summary

This thesis presents an IR for defining interfaces and integrating components using the Tydi specification. The prototype toolchain used to evaluate and demonstrate the ideas in this thesis features the ability to efficiently express Tydi interfaces and connect components using a simple grammar, and emit these as VHDL components and architectures.

Of note is the ability to propagate high-level, abstract properties such as documentation down from the IR (and any potential front-end) to the target language, to improve readability and more easily verify its outputs. As an extension of this, emitting alternative representations for Tydi's interfaces to retain type information could improve readability further. The thesis outlines potential changes to the IR to better enable this, in particular the addition of identifiers as properties of types.

This thesis also proposes the use of and a potential syntax for high-level assertions against interfaces defined in the IR, and a partial proof-of-concept for such tests was implemented in conjunction with the prototype toolchain. Alongside the high-level assertions, the limitations of such tests were discussed, and a potential solution was proposed in the form of "test Streamlets" and substitutions of Streamlet implementations. Then, the requirements for setting up individual Streamlets for testing were described, in order to aid future work.

As part of the overall evaluation, a number of inconsistencies in the Tydi specification were also identified and reported. The identification of these issues was a direct result of implementing the Tydi specification programmatically.

Overall, the work done as part of this thesis has been effective in demonstrating and testing the limits of an intermediate representation and toolchain specifically for composing components using the Tydi specification. However, there are many avenues to improve both the IR and toolchain, as summarized in the next section.

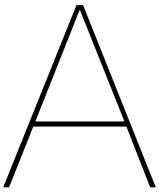
7.2. Recommendations for Future Work

Based on the findings of the thesis, we can make the following recommendations for future work:

- Implement a framework for typed, transaction-level assertions as described in Section 4.1.
- Document the workings of the IR and any ancillary components outside of this thesis and the published paper as part of the open-source repository.
- Define a number of intrinsics, and attempt to emit them through a backend, as described in Section 3.3.3. Having the ability to verify their functionality through the transaction-level assertions would be ideal, as this would also allow them to be verified over different backends.
- Add support for other language features, such as type parameters (3.3.1), generation (3.3.2) and annotations (3.3.4).

- Consider adding support for Streams without a clock domain, enabling asynchronous transfers as discussed in Section 3.1.2. Do note that this will also require modifications to the Tydi specification.
- Implement and use “projects” in both the query system and TIL, as described in Section 3.4.
- Add support for “substituting” implementations of Streamlets for the purposes of testing, as described in Section 4.3.
- Consider means for setting up “subjects” of tests correctly, given potentially unusual reset behavior and other requirements, as described in Section 4.4.
- Make identifiers a (potentially optional) component of type compatibility, as described in Section 6.2.
- As also described in Section 6.2, emit interfaces which better reflect their original logical type definitions.
- Make improvements to the TIL parser, as described in Section 6.4.3.
- Should performance become a concern, establish a “benchmark” TIL project and attempt to measure the speed with which the TIL parser and query system + backend perform their tasks. Of particular interest is whether the query system is effectively storing and reusing previous queries.
- Once the query system, parser, and other components have been appropriately iterated on and fully documented, publish them as crates on *crates.io*, so that others may use them more easily. Also ensure that their respective dependencies are subsequently converted to dependencies on the published crates, rather than their relation in the repository.

A number of these recommendations are also being tracked as issues on the `til-vhdl` repository: `https://github.com/matthijsr/til-vhdl/issues`



Complete TIL Example

```
namespace my::example::space {
    // Type declarations
    type byte = Bits(8);
    // Type expressions can be identifers or in-line declarations
    type select = Union(val: byte, empty: Null);
    type rgb = Group(r: select, g: select, b: select);
    // Streams have many properties, but some are optional
    type stream = Stream (
        data: rgb,
        throughput: 2.0, // 1.0 by default
        dimensionality: 0,
        synchronicity: Sync,
        complexity: 4,
        direction: Forward, // Forward by default
        user: Null, // Null by default
        keep: false, // false by default
    );
    type stream2 = stream;

    // A streamlet declaration
    #documentation (optional)#
    streamlet comp1 = (
        // Ports are *name* : *direction* *stream expression*
        a: in stream,
        b: out stream,
        # port documentation #
        c: in stream2,
        d: out stream2,
    );

    // An independent interface declaration
    interface iface1 = (a: in stream, b: out stream);

    #streamlet documentation
newline documentation#
    streamlet comp2 = iface1;

    // Implementation declarations
    #This is implementation documentation.#
    impl struct = (
        a: in stream,
        b: out stream,
        c: in stream2,
        d: out stream2,
    ){
        // Ports can be connected with --
        a -- b;

        // Streamlet instances are declared with
        // *instance name* = *streamlet name*
        a = comp1;
        b = comp1;

        // Ports on streamlet instances can be addressed with .
        a.a -- b.b;
        a.b -- b.a;

        // Ports on instances can also be connected to local ports
        c -- a.c;
```

```
    d -- b.d;
    a.d -- b.c;
};

// Linked implementations are paths enclosed by double quotes.
impl link = comp1 "./vhdl_dir";

streamlet comp3 = comp1 {
    impl:
    #This is implementation documentation, too.#
    {
        p1 = comp2;
        p2 = comp2;
        a -- p1.a;
        b -- p1.b;
        c -- p2.a;
        d -- p2.b;
    }
};

streamlet comp4 = comp1 { impl: struct };

streamlet comp5 = comp1 { impl: "./vhdl_dir" };

// 'domains represent combined clock and reset domains, and how they relate
// to a port's stream.
streamlet dom_example = <
 'domain1,
 'domain2,
>(
    a: in stream 'domain1,
    b: out stream 'domain2,
    c: in stream 'domain2,
    d: out stream 'domain1,
);

streamlet blank_doms = <'a, 'b, 'c>();

// In the above example, the domains of ports a and b are different, making them incompatible
// despite having the same type. a and d, and b and c can be connected, however.
//
// However, a structural implementation can assign the same domain twice,
// making a and b, and c and d compatible again.
streamlet struct_dom_example = <
  'parent_domain1,
  'parent_domain2,
> () {
    impl: {
        different_domains = dom_example<'parent_domain1, 'parent_domain2>;

        // Try changing these to <'parent_domain1, 'parent_domain2>
        // to see what happens when domains don't match.
        same_domains = dom_example<'parent_domain1, 'parent_domain1>;

        different_domains.a -- different_domains.d;
        different_domains.b -- different_domains.c;

        same_domains.a -- same_domains.b;
        same_domains.c -- same_domains.d;

        // For clarity, when assigning domains it's also possible to specify
        // which domain of the instance is being assigned to, rather than using their order.
        explicit_doms = blank_doms<'c = 'parent_domain1, 'a = 'parent_domain2, 'b = 'parent_domain2>;

        // It's also possible to mix named assignments with ordered assignments,
        // provided the named assignments succeed all ordered assignments.
        mixed_assignments = blank_doms<'parent_domain2, 'c = 'parent_domain1, 'b = 'parent_domain2>;
    }
};

// When a parent interface has no explicit domains, it is instead given a "default" domain.
// This default domain is also automatically assigned to any instances which do have explicit domains.
streamlet default_domains = (
    a: in stream,
    b: out stream,
    c: in stream,
    d: out stream
) {
    impl: {
        explicit_domains_instance = dom_example;

        explicit_domains_instance.a -- a;
        explicit_domains_instance.b -- b;
        explicit_domains_instance.c -- c;
```

```
            explicit_domains_instance.d -- d;
        }
    };
}
```

Listing A.1: A full TIL namespace with explanations as comments.

B

VHDL Backend Example

```
namespace my::example::space {
    type stream = Stream (
        data: Bits(8),
        dimensionality: 0,
        synchronicity: Sync,
        complexity: 4,
    );

    #Streamlet documentation#
    streamlet comp1 = (
        a: in stream,
        b: out stream,
        #Port
documentation#
        c: in stream,
        d: out stream,
    );

    streamlet comp2 = comp1 {
        impl: "./vhdl_dir"
    };

    streamlet domains_only = <'a, 'b, 'c>();

    streamlet comp3 = <'x, 'y>(
        q: in stream 'x,
        r: out stream 'x,
    ) {
        impl: {
            dom_ex = domains_only<'x, 'y, 'y>;
            inst = comp2<'x>;
            q -- inst.a;
            r -- inst.b;
            inst.c -- inst.d;
        }
    };
}
```

Listing B.1: The full TIL namespace defined over the course of Section 5.3

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.proj.all;

entity my__example__space__comp3_com is
  port (
    x__clk : in std_logic;
    x__rst : in std_logic;
    y__clk : in std_logic;
    y__rst : in std_logic;
    q_valid : in std_logic;
    q_ready : out std_logic;
    q_data : in std_logic_vector(7 downto 0);
    r_valid : out std_logic;
    r_ready : in std_logic;
    r_data : out std_logic_vector(7 downto 0)
```

```
  );
end my__example__space__comp3_com;

architecture my__example__space__comp3 of my__example__space__comp3_com is
  signal inst__a_valid : std_logic;
  signal inst__a_ready : std_logic;
  signal inst__a_data : std_logic_vector(7 downto 0);
  signal inst__b_valid : std_logic;
  signal inst__b_ready : std_logic;
  signal inst__b_data : std_logic_vector(7 downto 0);
  signal inst__c_valid : std_logic;
  signal inst__c_ready : std_logic;
  signal inst__c_data : std_logic_vector(7 downto 0);
  signal inst__d_valid : std_logic;
  signal inst__d_ready : std_logic;
  signal inst__d_data : std_logic_vector(7 downto 0);
begin
  dom_ex: my__example__space__domains_only_com port map(
    a__clk => x__clk,
    a__rst => x__rst,
    b__clk => y__clk,
    b__rst => y__rst,
    c__clk => y__clk,
    c__rst => y__rst
  );
  inst: my__example__space__comp2_com port map(
    clk => x__clk,
    rst => x__rst,
    a_valid => inst__a_valid,
    a_ready => inst__a_ready,
    a_data => inst__a_data,
    b_valid => inst__b_valid,
    b_ready => inst__b_ready,
    b_data => inst__b_data,
    c_valid => inst__c_valid,
    c_ready => inst__c_ready,
    c_data => inst__c_data,
    d_valid => inst__d_valid,
    d_ready => inst__d_ready,
    d_data => inst__d_data
  );
  inst__a_valid <= q_valid;
  q_ready <= inst__a_ready;
  inst__a_data <= q_data;
  r_valid <= inst__b_valid;
  inst__b_ready <= r_ready;
  r_data <= inst__b_data;
  inst__c_valid <= inst__d_valid;
  inst__d_ready <= inst__c_ready;
  inst__c_data <= inst__d_data;
end my__example__space__comp3;
```

Listing B.2: The VHDL architecture output by the VHDL backend for *comp3* of Listing B.1

C

AXI4 Specification

The following tables were taken from [8].

Write Address (AW) channel signals	AXI version
AWVALID	AXI3 and AXI4
AWREADY	AXI3 and AXI4
AWADDR[31:0]	AXI3 and AXI4
AWSIZE[2:0]	AXI3 and AXI4
AWBURST[1:0]	AXI3 and AXI4
AWCACHE[3:0]	AXI3 and AXI4
AWPROT[2:0]	AXI3 and AXI4
AWID[x:0]	AXI3 and AXI4
AWLEN[3:0]	AXI3 only
AWLEN[7:0]	AXI4 only
AWLOCK[1:0]	AXI3 only
AWLOCK	AXI4 only
AWQOS[3:0]	AXI4 only
AWREGION[3:0]	AXI4 only
AWUSER[x:0]	AXI4 only

Table C.1: Write Address

Write Data (W) channel signals	AXI version
WVALID	AXI3 and AXI4
WREADY	AXI3 and AXI4
WLAST	AXI3 and AXI4
WDATA[x:0]	AXI3 and AXI4
WSTRB[x:0]	AXI3 and AXI4
WID[x:0]]	AXI3 only
WUSER[x:0]	AXI4 only

Table C.2: Write Data

Write response (B) channel signals	AXI version
BWVALID	AXI3 and AXI4
BWREADY	AXI3 and AXI4
BRESP[1:0]	AXI3 and AXI4
BID[x:0]	AXI3 and AXI4
BUSER[x:0]	AXI4 only

Table C.3: Write response

Read Address (AR) channel signals	AXI version
ARVALID	AXI3 and AXI4
AREADY	AXI3 and AXI4
ARADDR[31:0]	AXI3 and AXI4
ARSIZE[2:0]	AXI3 and AXI4
ARBURST[1:0]	AXI3 and AXI4
ARCACHE[3:0]	AXI3 and AXI4
ARPROT[2:0]	AXI3 and AXI4
ARID[x:0]	AXI3 and AXI4
ARLEN[3:0]	AXI3 only
ARLEN[7:0]	AXI4 only
ARLOCK[1:0]	AXI3 only
ARLOCK	AXI4 only
ARQOS[3:0]	AXI4 only
ARREGION[3:0]	AXI4 only
ARUSER[x:0]	AXI4 only

Table C.4: Read Address

Read Data (R) channel signals	AXI version
RVALID	AXI3 and AXI4
RREADY	AXI3 and AXI4
RLAST	AXI3 and AXI4
RDATA[x:0]	AXI3 and AXI4
RRESP[1:0]	AXI3 and AXI4
RID[x:0]	AXI3 and AXI4
RUSER[x:0]	AXI4 only

Table C.5: Read Data

D

AXI4 TIL Definition and VHDL Output

```
namespace evaluation {
    type axi4stream = Stream (
        data: Union (
            data: Bits(8),
            null: Null, // Equivalent to TSTRB
        ),
        throughput: 128.0, // Data bus width
        dimensionality: 1, // Equivalent to TLAST
        synchronicity: Sync,
        complexity: 7, // Tydi's strobe is equivalent to TKEEP
        user: Group (
            TID: Bits(8),
            TDEST: Bits(4),
            TUSER: Bits(1),
        ),
    );

    type axi4_address = Stream (
        data: Group (
            ADDR: Bits(32),
            SIZE: Bits(3),
            BURST: Bits(2),
            CACHE: Bits(4),
            PROT: Bits(3),
            ID: Bits(4),
            LEN: Bits(8),
            LOCK: Bits(1),
            QOS: Bits(4),
            REGION: Bits(4),
        ),
        dimensionality: 0,
        synchronicity: Sync,
        complexity: 1,
        user: Bits(4),
    );
    type axi4_write_data = Stream (
        data: Bits(8),
        throughput: 256.0, // Max transfers
        dimensionality: 1, // Equivalent to LAST
        synchronicity: Sync,
        complexity: 7, // Adds a strobe
        user: Bits(4),
    );
    type axi4_read_data = Stream (
        data: Bits(8),
        throughput: 256.0, // Max transfers
        dimensionality: 1, // Equivalent to LAST
        synchronicity: Sync,
        complexity: 7, // Adds a strobe
        user: Group (
            RESP: Bits(2),
            ID: Bits(4),
            USER: Bits(4),
        ),
    );
    type axi4_response = Stream (
        data: Group (
            RESP: Bits(2),
            ID: Bits(4),
```

```
        ),
        dimensionality: 0,
        synchronicity: Sync,
        complexity: 1,
        user: Bits(4),
    );

    type axi4 = Stream (
        data: Group (
            AW: axi4_address,
            W: Stream (
                data: Bits(8),
                throughput: 256.0, // Max transfers
                dimensionality: 1, // Equivalent to LAST
                synchronicity: Sync,
                complexity: 7, // Adds a strobe
                user: Bits(4),
            ),
            B: Stream (
                direction: Reverse,
                data: Group (
                    RESP: Bits(2),
                    ID: Bits(4),
                ),
                dimensionality: 0,
                synchronicity: Sync,
                complexity: 1,
                user: Bits(4),
            ),
            AR: axi4_address,
            R: Stream (
                direction: Reverse,
                data: Bits(8),
                throughput: 256.0, // Max transfers
                dimensionality: 1, // Equivalent to LAST
                synchronicity: Sync,
                complexity: 7, // Adds a strobe
                user: Group (
                    RESP: Bits(2),
                    ID: Bits(4),
                    USER: Bits(4),
                ),
            ),
        ),
        dimensionality: 0,
        synchronicity: Sync,
        complexity: 1,
    );

    streamlet example = (
        axi4stream: in axi4stream,
        axi4_aw: out axi4_address,
        axi4_w: out axi4_write_data,
        axi4_b: in axi4_response,
        axi4_ar: out axi4_address,
        axi4_r: in axi4_read_data,
        axi4: out axi4,
    );
}
```

Listing D.1: The full TIL definition of AXI4-Stream and AXI4

```
  component evaluation__example_com
    port (
      clk : in std_logic;
      rst : in std_logic;
      axi4stream_valid : in std_logic;
      axi4stream_ready : out std_logic;
      axi4stream_data : in std_logic_vector(1151 downto 0);
      axi4stream_last : in std_logic;
      axi4stream_stai : in std_logic_vector(6 downto 0);
      axi4stream_endi : in std_logic_vector(6 downto 0);
      axi4stream_strb : in std_logic_vector(127 downto 0);
      axi4stream_user : in std_logic_vector(12 downto 0);
      axi4_aw_valid : out std_logic;
      axi4_aw_ready : in std_logic;
      axi4_aw_data : out std_logic_vector(64 downto 0);
      axi4_aw_user : out std_logic_vector(3 downto 0);
      axi4_w_valid : out std_logic;
      axi4_w_ready : in std_logic;
      axi4_w_data : out std_logic_vector(2047 downto 0);
      axi4_w_last : out std_logic;
      axi4_w_stai : out std_logic_vector(7 downto 0);
```

```
      axi4_w_endi : out std_logic_vector(7 downto 0);
      axi4_w_strb : out std_logic_vector(255 downto 0);
      axi4_w_user : out std_logic_vector(3 downto 0);
      axi4_b_valid : in std_logic;
      axi4_b_ready : out std_logic;
      axi4_b_data : in std_logic_vector(5 downto 0);
      axi4_b_user : in std_logic_vector(3 downto 0);
      axi4_ar_valid : out std_logic;
      axi4_ar_ready : in std_logic;
      axi4_ar_data : out std_logic_vector(64 downto 0);
      axi4_ar_user : out std_logic_vector(3 downto 0);
      axi4_r_valid : in std_logic;
      axi4_r_ready : out std_logic;
      axi4_r_data : in std_logic_vector(2047 downto 0);
      axi4_r_last : in std_logic;
      axi4_r_stai : in std_logic_vector(7 downto 0);
      axi4_r_endi : in std_logic_vector(7 downto 0);
      axi4_r_strb : in std_logic_vector(255 downto 0);
      axi4_r_user : in std_logic_vector(9 downto 0);
      axi4__AW_valid : out std_logic;
      axi4__AW_ready : in std_logic;
      axi4__AW_data : out std_logic_vector(64 downto 0);
      axi4__AW_user : out std_logic_vector(3 downto 0);
      axi4__W_valid : out std_logic;
      axi4__W_ready : in std_logic;
      axi4__W_data : out std_logic_vector(2047 downto 0);
      axi4__W_last : out std_logic;
      axi4__W_stai : out std_logic_vector(7 downto 0);
      axi4__W_endi : out std_logic_vector(7 downto 0);
      axi4__W_strb : out std_logic_vector(255 downto 0);
      axi4__W_user : out std_logic_vector(3 downto 0);
      axi4__B_valid : in std_logic;
      axi4__B_ready : out std_logic;
      axi4__B_data : in std_logic_vector(5 downto 0);
      axi4__B_user : in std_logic_vector(3 downto 0);
      axi4__AR_valid : out std_logic;
      axi4__AR_ready : in std_logic;
      axi4__AR_data : out std_logic_vector(64 downto 0);
      axi4__AR_user : out std_logic_vector(3 downto 0);
      axi4__R_valid : in std_logic;
      axi4__R_ready : out std_logic;
      axi4__R_data : in std_logic_vector(2047 downto 0);
      axi4__R_last : in std_logic;
      axi4__R_stai : in std_logic_vector(7 downto 0);
      axi4__R_endi : in std_logic_vector(7 downto 0);
      axi4__R_strb : in std_logic_vector(255 downto 0);
      axi4__R_user : in std_logic_vector(9 downto 0)
    );
  end component;
```

Listing D.2: The full VHDL component using the AXI4-Stream and AXI4 equivalent Tydi Streams

Bibliography

[1] *A Python Interpreter Written in Rust | Hacker News*. 2019. URL: https://news.ycombinator.com/item?id=19064069 (visited on 06/22/2022).

[2] Accellera Systems Initiative. *Systemc.Org*. 2022. URL: https://systemc.org/about/systemc/overview/ (visited on 06/26/2022).

[3] Advanced Micro Devices, Inc. *Intellectual Property*. Xilinx. 2022. URL: https://www.xilinx.com/products/intellectual-property.html (visited on 05/18/2022).

[4] Advanced Micro Devices, Inc. *Interfaces for Vivado IP Flow • Vitis High-Level Synthesis User Guide (UG1399) • Reader • Documentation Portal*. Apr. 20, 2022. URL: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Interfaces-for-Vivado-IP-Flow (visited on 05/21/2022).

[5] Apache Software Foundation. *Spark Streaming - Spark 3.3.0 Documentation*. 2022. URL: https://spark.apache.org/docs/latest/streaming-programming-guide.html (visited on 06/25/2022).

[6] Apache Software Foundation. *Structured Streaming Programming Guide - Spark 3.3.0 Documentation*. 2022. URL: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html (visited on 06/25/2022).

[7] Arm Limited. *AMBA 4 AXI4-Stream Protocol Specification*. Mar. 3, 2010. URL: https://developer.arm.com/documentation/ihi0051/a/Introduction/About-the-AXI4-Stream-protocol (visited on 04/29/2022).

[8] Arm Limited. *An Introduction to AMBA AXI*. Mar. 12, 2021. URL: https://developer.arm.com/documentation/102202/0200/ (visited on 05/17/2022).

[9] Adam Arnesen et al. “Increasing Design Productivity through Core Reuse, Meta-data Encapsulation, and Synthesis”. In: *2010 International Conference on Field Programmable Logic and Applications*. 2010 International Conference on Field Programmable Logic and Applications. Aug. 2010, pp. 538–543. DOI: 10.1109/FPL.2010.106.

[10] Joshua Auerbach et al. “Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures”. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’10. New York, NY, USA: Association for Computing Machinery, Oct. 17, 2010, pp. 89–108. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869469. URL: http://doi.org/10.1145/1869459.1869469 (visited on 04/29/2022).

[11] J. Bachrach et al. “Chisel: Constructing Hardware in a Scala Embedded Language”. In: *DAC Design Automation Conference 2012*. June 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[12] Joshua Barretto. *Ariadne*. June 21, 2022. URL: https://github.com/zesterer/ariadne (visited on 06/22/2022).

[13] Joshua Barretto. *Chumsky*. May 12, 2022. URL: https://github.com/zesterer/chumsky (visited on 05/12/2022).

[14] Geoffroy Couprie. *Nom, Eating Data Byte by Byte*. June 21, 2022. URL: https://github.com/Geal/nom (visited on 06/21/2022).

[15] Marios Fragkoulis et al. “A Survey on the Evolution of Stream Processing Systems”. Aug. 3, 2020. DOI: 10.48550/arXiv.2008.00842. arXiv: 2008.00842 [cs]. URL: http://arxiv.org/abs/2008.00842 (visited on 05/18/2022).

[16] *Grammar and Parsing Libraries for Rust*. Software Development Team, June 21, 2022. URL: https://github.com/softdevteam/grmtools (visited on 06/21/2022).

[17] Amir Hormati et al. “Optimus: Efficient Realization of Streaming Applications on FPGAs”. In: *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES ’08. New York, NY, USA: Association for Computing Machinery, Oct. 19, 2008, pp. 41–50. ISBN: 978-1-60558-469-0. DOI: 10.1145/1450095.1450105. URL: http://doi.org/10.1145/1450095.1450105 (visited on 04/29/2022).

[18] “IEEE Standard for Verilog Hardware Description Language”. In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (Apr. 2006), pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.

[19] “IEEE Standard VHDL Language Reference Manual”. In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (Jan. 2009), pp. 1–640. DOI: 10.1109/IEEESTD.2009.4772740.

[20] Intel Corporation. *1. Introduction to Intel® FPGA IP Cores*. Intel. Apr. 10, 2021. URL: https://www.intel.com/content/www/us/en/docs/programmable/683102/21-3/introduction-to-cores.html (visited on 05/18/2022).

[21] Intel Corporation. *5. Avalon® Streaming Interfaces*. Intel. Jan. 24, 2022. URL: https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/streaming-interfaces.html (visited on 05/18/2022).

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

[23] A. Izraelevitz et al. “Reusability Is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2017, pp. 209–216. DOI: 10.1109/ICCAD.2017.8203780.

[24] M.F. Jacome and H.P. Peixoto. “A Survey of Digital Design Reuse”. In: *IEEE Design Test of Computers* 18.3 (May 2001), pp. 98–107. ISSN: 1558-1918. DOI: 10.1109/54.922806.

[25] Maria Kotsifakou et al. “HPVM: Heterogeneous Parallel Virtual Machine”. In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’18. New York, NY, USA: Association for Computing Machinery, Feb. 10, 2018, pp. 68–80. ISBN: 978-1-4503-4982-6. DOI: 10.1145/3178487.3178493. URL: http://doi.org/10.1145/3178487.3178493 (visited on 04/29/2022).

[26] *LALRPOP*. lalrpop, June 20, 2022. URL: https://github.com/lalrpop/lalrpop (visited on 06/21/2022).

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

[28] matthijsr. *TIL -> VHDL*. June 23, 2022. URL: https://github.com/matthijsr/til-vhdl (visited on 06/23/2022).

[29] Kevin Mehall. *Parsing Expression Grammars in Rust*. June 19, 2022. URL: https://github.com/kevinmehall/rust-peg (visited on 06/21/2022).

[30] Razvan Nane et al. “A Survey and Evaluation of FPGA High-Level Synthesis Tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (Oct. 2016), pp. 1591–1604. ISSN: 1937-4151. DOI: 10.1109/TCAD.2015.2513673.

[31] Tony Nowatzki et al. “Stream-Dataflow Acceleration”. In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). June 2017, pp. 416–429. DOI: 10.1145/3079856.3080255.

[32] Johan Peltenburg et al. “Tydi: An Open Specification for Complex Data Structures Over Hardware Streams”. In: *IEEE Micro* 40.4 (July 2020), pp. 120–130. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2996373.

[33] *Pest. The Elegant Parser*. pest, June 21, 2022. URL: https://github.com/pest-parser/pest (visited on 06/21/2022).

[34] Franjo Plavec. “Stream Computing on Fpgas”. CAN: University of Toronto, 2010. ISBN: 9780494731772.

[35] Matthijs A. Reukers et al. “An Intermediate Representation for Composable Typed Streaming Dataflow Designs”. Unpublished. 2022.

[36] rodyamirov. *What Should I Use to Build a Parser in Rust Today?* r/rust. Apr. 6, 2020. URL: www.reddit.com/r/rust/comments/fvwe9t/what_should_i_use_to_build_a_parser_in_rust_today/ (visited on 06/22/2022).

[37] Rust Compiler Team. *Queries: Demand-Driven Compilation - Guide to Rustc Development*. 2021. URL: https://rustc-dev-guide.rust-lang.org/query.html (visited on 05/11/2022).

[38] salsa-rs. *Salsa*. salsa, May 9, 2022. URL: https://github.com/salsa-rs/salsa (visited on 05/11/2022).

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

[40] Fabian Schuiki et al. "LLHD: A Multi-level Intermediate Representation for Hardware Description Languages". Apr. 7, 2020. arXiv: 2004.03494 [cs]. URL: http://arxiv.org/abs/2004.03494 (visited on 04/29/2022).

[41] Ivan E. Sutherland. "Micropipelines". In: *ACM Turing Award Lectures*. New York, NY, USA: Association for Computing Machinery, Jan. 1, 2007, p. 1988. ISBN: 978-1-4503-1049-9. URL: http://doi.org/10.1145/1283920.1283946 (visited on 12/15/2021).

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

[43] James Thomas, Pat Hanrahan, and Matei Zaharia. "Fleet: A Framework for Massively Parallel Streaming on FPGAs". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, Mar. 9, 2020, pp. 639–651. ISBN: 978-1-4503-7102-5. URL: http://doi.org/10.1145/3373376.3378495 (visited on 04/29/2022).

[44] Jeroen Van Straten, Johan Peltenburg, and Matthijs Brobbel. *Introduction - Tydi*. 2021. URL: https://abs-tudelft.github.io/tydi/index.html (visited on 05/16/2022).

[45] Jeroen Van Straten, Johan Peltenburg, and Matthijs Brobbel. *Logical Streams - Tydi*. 2021. URL: https://abs-tudelft.github.io/tydi/specification/logical.html (visited on 05/04/2022).

[46] Jeroen Van Straten, Johan Peltenburg, and Matthijs Brobbel. *Physical Streams - Tydi*. 2021. URL: https://abs-tudelft.github.io/tydi/specification/physical.html (visited on 05/04/2022).

[47] Ze-ke Wang et al. "Melia: A MapReduce Framework on OpenCL-based FPGAs". In: *IEEE Transactions on Parallel and Distributed Systems* 27 (Dec. 1, 2016), pp. 1–1. DOI: 10.1109/TPDS.2016.2537805.

[48] Cody Hao Yu et al. "S2FA: An Accelerator Automation Framework for Heterogeneous Computing in Datacenters". In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC). June 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465827.

[49] Matei Zaharia et al. *Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing*. UCB/EECS-2012-259. EECS Department, University of California, Berkeley, Dec. 2012. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-259.html.

[50] zesterer. *Chumsky, a Parser Combinator Crate That Makes Writing Error-Tolerant Parsers with Recovery Easy and Fun!* r/rust. Oct. 28, 2021. URL: www.reddit.com/r/rust/comments/qhyzwd/chumsky_a_parser_combinator_crate_that_makes/ (visited on 06/22/2022).