Shockwaves & Tydi-Clash

Raising the abstraction level of the Haskell HDL Clash through typed waveforms and complex streaming interfaces

Marijn Adriaanse

TU Delft

Shockwaves & Tydi-Clash

Raising the abstraction level of the Haskell HDL Clash through typed waveforms and complex streaming interfaces

Thesis report

by

Marijn Adriaanse

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on June 26, 2025 at 14:00

Thesis committee:

Chair:	Prof. Dr. H. Peter Hofstee
Core member 2:	Dr. Ir. Chris Verhoeven
Core member 3:	Dr. Ir. Zaid Al-Ars
External advisor:	Dr. Ir. Christiaan Baaij
Place:	Hall H, Faculty of Electrical Engineering, Mathematics & Computer Science, Delft
Project Duration:	November, 2024 - June, 2025
Student number:	5346878

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

Faculty of Electrical Engineering, Mathematics & Computer Science · Delft University of Technology

Abstract

This work contains two systems created to raise abstraction for the Haskell-based HDL Clash.

A common tool in hardware design is the waveform viewer. Although Clash could already generate waveform files, these only contained binary representations of the values. Without translating these to Haskell values, they are difficult to interpret. Shockwaves was created to perform this translation. Unlike other typed waveform solutions, Shockwaves performs the translation fully in the Haskell runtime, and stores the results in lookup tables. This gives the programmer full control over the waveform representation of data. There are two methods of generating VCD files from Clash, and Shockwaves was designed to work with both. The system is fully functional for signals traced during direct simulation. The alternative approach of simulating a design after compiling it to a different HDL depends on the Clash compiler adding type annotations. This requires an overhaul of the Clash compiler beyond the scope of the project.

The second system, Tydi-Clash, is a library for the Tydi streaming specification in Clash. Tydi was designed around transferring complex data structures, and allows for multiple related streams carrying typed, multi-dimensional data. The Tydi-Clash library supports Tydi data types, physical streams, and logical stream constructs. To encourage correct usage of the streams, the internal signals are encapsulated in algebraic and abstract data types that prevent defining or accessing undefined values. Additionally, tests are supplied for behavioral restrictions. An example implementation revealed implementations using Tydi-Clash are unfortunately still a bit cumbersome, but this is believed to be solvable by adding a library of utility modules for common situations.

Preface

When I was just a little kid, I played a game. And in that game, there were logic gates. And so I started making logic circuits to make systems in the game years before I even started programming. Now, as I am about to finish my master's, I can see that the love for hardware design that started back then never left me, and hopefully never will.

In my bachelor's degree Electrical Engineering I quickly gravitated towards digital logic once again, and it led me to my master's in Computer and Embedded Systems Engineering. There, I picked a single course that taught me about Haskell and Clash, which ultimately brought me where I am today.

Now, I would like to thank my supervisors, Peter Hofstee, Zaid Al-Ars, and Christaan Baaij for their help and support throughout the project. I would also like to thank my fellow students working on Tydi, and colleagues at QBayLogic for their help and input. I would like to thank Frans Skarman for his support in modifying Surfer. And I would like to thank everyone not mentioned here that came before and created the systems upon which my work was built.

And finally, I would like to thank my family and friends - in my hometown, in Delft, and in the CodeBugs community - that kept me going these past months.

Marijn Adriaanse
Enschede, June 2024

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

Contents

List of Figures

List of Tables

1 Introduction

In the past decade, advancements in hardware development have not been able to keep up with Moore's law. Yet, the amount of data we process is ever increasing. As such, the industry has shifted its focus from trying to improve the performance of regular processors to using accelerators and domain-specific hardware [1].

Modern hardware technologies allow for custom hardware implementations that are more complex than ever. Furthermore, the art of system design is an ever developing field. With this growth of knowledge and resources, it becomes increasingly difficult for developers to be able to wield the vast complexity at their disposal. As such, there is a need to create languages and tools that allow advanced systems to be created with as little effort as possible.

Thus, this need for abstractions and less cumbersome implementations has given rise to numerous modern HDLs. While languages such as VHDL and Verilog are still used for hardware design, these modern HDLs allow for a higher level of abstraction, such as advanced data types and programming constructs. However, development of new languages also requires tools and libraries to be developed to fully leverage their advantages over classical HDLs.

To this end, two systems were developed for Clash, a modern, Haskell-based HDL. The first is *Shockwaves*, a typed waveform system for Clash. This raises the abstraction level of waveform-based debugging to the same level as the rest of Clash. The second is *Tydi-Clash*, which deals with the implementation of Tydi streams. These streams can be used to create standardized, reusable, complex hardware interfaces.

Consequently, after a common introduction to Haskell and Clash in Chapter 2, the contents of this work are divided into two parts. Part I covers Shockwaves. Shockwaves is later used in Part II, which covers Tydi-Clash. Both parts have their own introduction, background information, discussion and summary. Finally, the work is concluded in Chapter 22.

2

Common background

2.1. Haskell

This work deals with the Haskell [2] programming language, and thus a proper understanding of some of its mechanics are useful to be aware of. Haskell is a general purpose functional programming language, and uses immutable data. It is statically typed, but has a high degree of type inference. The following sections provide an introductions to some core concepts of Haskell used throughout this thesis.

2.1.1. Basic syntax

Haskell does not use parentheses for function calls like most languages. The operator with the highest precedence is actually the whitespace between identifiers/values, and denotes application. This means that `a b c` may be read as `(a(b))(c)` or even `a(b,c)`.

Operators works as binary operators. They combine the arguments to the left and right, though they can be grouped in parentheses to make them act like normal functions. Thus, `a + b` reads as `(+) a b`. It is possible to define custom operators, and set their associativity and precedence.

Data types can be defined after `::`: `x :: Int`. Function types are denoted by `input -> result`. Multi-parameter functions do not have multiple inputs; instead, the result after taking one argument is a new function that takes the rest of the arguments. Hence, in `f :: a -> b -> c`, `f` is a function that takes inputs of types `a` and `b` and produces a result of type `c`.

2.1.2. Lazy evaluation

As a language that works by definitions, rather than being imperative, Haskell relies heavily on lazy evaluation: expressions are only evaluated when they are needed, and only to the extent that is absolutely necessary. Since this behavior is not always desired, Haskell has several tools for forcing evaluation of a value, even if it is not directly used.

Expressions may be set to so called *bottom values* such as `undefined` or `error`. These do not represent actual values of the associated type, but rather the notion that there is no way to determine the value. As such, attempting to evaluate such a bottom value results in an exception that, unless caught, crashes program execution.

2.1.3. Data types

Haskell has primitive data types, such as numbers and characters, but otherwise runs on algebraic data types. Such a type may have one or more constructors, which may in turn each have one or more data fields of varying types. The fields can be either be nameless, and provided like function arguments, or use record style.

```
1 data A = P | Q | R          -- multiple constructors
2 data B = V Int Int          -- regular style fields
3 data C = W{x::Int, y::Int} -- record syntax
```

Data types are may be recursive, and polymorphic, i.e. a type may take several types as parameters. A type that takes parameters is essentially a function at the type level: it takes one or more types as

arguments to produce a type. The "type of a type" is called it's *kind*. Besides data types and polymorphic types, there are also kinds such as `Nat` for natural numbers, or `Symbol` for strings.

```
data X = C              -- X has kind Type
data T a b = A a | B b -- T has kind Type -> Type -> Type
```

Data types don't have to always be explicitly defined, and they can often be kept polymorphic. For example, a function may have the type `a -> a`, indicating it takes one value of *any* type `a` and returns a value of that same type. When a type parameter cannot be inferred, it may be specified with the `@` operator.

```
show (def @Int) -- show the default value of Int
```

2.1.4. Common data types

There are a few common Haskell types that come up repeatedly in this work. `()` is the *unit type*, and only has the value `()`. It is a type that holds no actual data. `Bool` is a boolean, and has constructors `True` and `False`. There are multiple number types, but in this work, `Int` is used for arbitrary numerical values. It does not have a strictly defined number of bits.

Values may be grouped into *tuples*. Tuple types can hold heterogenous data, and there is a separate tuple data type for each length. For example, the type of a 3-long tuple would be `(a,b,c)`.

The final important type is the polymorphic type `Maybe`. The type `Maybe a` has constructors `Just a` and `Nothing`, and denotes optional data.

2.1.5. Classes

A class in Haskell is essentially a set of functions, types and values that may be defined for a some data types by implementing the class. Some common examples are `Show`, which allows a value to be represented as text, and `Generic`, which decomposes the type and values into their different parts (constructors, fields, etc.).

A class can be used as a *constraint* for a class or function. For example, the type `C a => a -> a` denotes a function that takes and produces a value of any type `a` *for which the implementation* `C a` *exists*.

Values may have default implementations. If all values have suitable defaults, the class can be *derived*: it is implemented using these default implementations without requiring the programmer to define everything manually. There are also different "deriving strategies" such as `derive via`, which allow the implementation to be copied from a wrapper type instead. This can be used to provide a secondary default implementation.

2.1.6. Type families

Type families are essentially functions over types. They may be supplied at the toplevel form, in which case they can look much like ordinary functions, or they may be "associated" with classes (for example, the class `C a` has a type `X a`, and the implementation of `C Bool` defines that `X Bool = Int`). Type families are a very powerful tool for performing operations on types, or defining more complex class instances.

2.1.7. Template Haskell

Template Haskell is an extension that adds meta-programming in compile time. It can be used to generate code in a way that would otherwise be impossible. As such, it is extremely powerful, but it does come at the cost of having much more complicated code. Therefore, it is generally only used if absolutely necessary.

2.2. Clash

Clash [3], [4] (stylized as CλasH) is an HDL based on Haskell. Haskell's purity and data immutability make it particularly suitable to be used as an HDL. It was first created in 2009, and has since been further developed and successfully used in many projects.

Clash consists of several Haskell libraries, containing things such as sequential logic constructs and hardware-friendly types, and the Clash compiler, which can translate a Clash design into other HDLs. The Clash compiler uses the Haskell compiler frontend, and replaces the backend. In turn, this backend has its own target language-specific backends.

2.2.1. HDL concepts in Clash

Hardware designs are generally a combination of combinatorial and sequential logic.

Creating combinatorial logic in Clash is trivial: due to Haskell's data immutability, any standard expression defining values will result in combinatorial logic. Any value is simply a signal in the HDL, while a function turns into a piece of combinatorial logic with inputs and outputs. Since Haskell is pure, the complete behavior of the function is captured by the logic's inputs and outputs. Control statements, such as `if` or `case`, are turned into multiplexers.

```
combinatorial_double :: Int -> Int
combinatorial_double x = 2 * x
```

For sequential logic, Clash uses the `Signal` type. This type represents a time domain signal in a certain clock domain in the design, and is essentially an array-like structure containing values for each clock cycle.

Combinatorial logic can be applied by mapping ordinary functions to these time domain signals. Special Clash functions, such as `register` and `mealy`, can be used to gain access to values from previous clock cycles, allowing the creation of sequential logic.

```
signal_double :: Signal dom Int -> Signal dom Int
signal_double x = fmap combinatorial_double x

accumulator :: Signal dom Int -> Signal dom Int
accumulator = mealy go 0   -- start at 0
  where go (a,x) = (a+x,a) -- (state,in) -> (state',out)
```

One cannot directly map combinatorial logic to multiple `Signal`s. To convert between structures of signals (in the same domain) and signals of structures, the `bundle` and `unbundle` functions are provided. Cross-domain connections are outside of the scope of this work.

```
-- bundle for tuples of 2 values
bundle :: (Signal dom a, Signal dom b) -> Signal dom (a,b)
```

2.2.2. Important Clash types

For the Clash compiler to be able to synthesize code, the data types used must be representable as a fixed number of bits. For example, Haskell's `Bool` type is fine, while lists are not due to their unknown length.

Some examples of commonly used Haskell types in Clash designs are `Bool` and `()`. Polymorphic types such as `Maybe` and the various tuples can be used as well, but only if their contained types are themselves synthesizable. In general, any algebraic data types can be used, as long as all of their subvalues are of representable types.

The Clash library comes equipped with some fixed-size versions of common types. `Signed` and `Unsigned` represent numerical values with a fixed number of bits, and `Index` represents numbers up to a certain value. `Vec` provides fixed-length arrays of values.

```
signed :: Signed 5 -- 5 bits
index  :: Index 10 -- an integer in the range [0,10)

vec :: Vec 3 Int  -- 3 Ints
vec = 0 :> 1 :> 2 :> Nil
```

Part I

Shockwaves

3

Introduction

Debugging is an inevitable part of development, and having effective tools for debugging is essential to the development process. While software developers have access to a multitude of testing facilities, hardware developers have fewer tools at their disposal.

In essencence, debugging tools allow the programmer to see the inner workings of their creations. This might be by showing execution in a stepwise manner, logging values throughout the design, or simply representing the system in a different format.

In hardware design, one of the most used tools is a waveform viewer: signals are logged throughout design simulation (typically in a VCD file), and their values are visualized in the time domain. This allows the developer to see the values of every signal during each clock cycle of the simulation.

Although Clash has support for generating VCD files, the most used logging format for hardware designs, one indispensible part is missing: since these VCD files only contain the binary representations of Haskell values, waveform viewers can only display these values using standard formats, such as binary, hexadecimal or signed integers. This makes the waveforms incredibly difficult to interpret, and greatly diminishes the value of waveforms.

3.1. Objectives

The Shockwaves project aims to create a system that shows logged values in their Haskell representation form, allowing Clash developers to properly interpret waveforms.

In creating the system, the following objectives were adhered to:

- Haskell values are displayed in a format that is close to their representation in Haskell.
- Complex nested data structures are decomposed into subsignals.
- It is possible to, with little effort, add display formats for new data types.
- The system is easy to apply to an existing design.
- The system can be used with, or is a close substitute to, existing waveform viewer pipelines.

Together, these objectives lead to a system that requires minimal effort to use, while being flexible and broadly applicable. An illustratory example of what the output might look like with and without Shockwaves can be seen in Fig. 3.1.

3.2. Outline

Background information is provided in Chapter 4. The main system design is discussed in Chapter 5, and further elabortated in Chapters 6 to 8. These chapters cover the waveform viewer integration, data representation, and translation mechanisms respectively. Chapters 9 and 10 detail and discuss the results, and finally the part is summarized in Chapter 11.

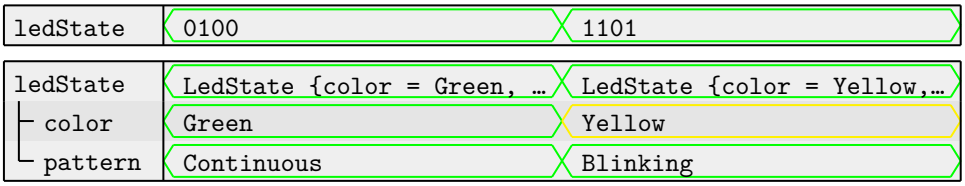

Figure 3.1: Illustratory example of a signal defining the state of an RGB LED before and after translation by Shockwaves.

4

Background & Related Work

This chapter provides some background information on waveform viewers, other HDLs, and Clash's methods of creating VCD files. Although there are many modern HDLs, this chapter is limited to languages that have custom waveform viewer support, and served as an inspiration for Shockwaves.

4.1. Surfer

Surfer [5] is a modern waveform viewer written in Rust. Surfer was designed to be easily extensible with translator modules, that may be used to represent the data in VCD files in different ways. These translators are responsible for both the direct display and division into subsignals of VCD signals. Additionally, translators have control over which (sub)signals are defined when, as well as their display style.

4.2. Spade

Spade [6] is a modern HDL inspired by Rust. As it was only created in 2022, the language is still quite heavily under development.

The Spade compiler was written in Rust, and is developed by, among others, the creator of Surfer. As such, it should come as no surprise that Surfer has typed waveform support for Spade. The Spade compiler stores it's 'state', including the signal type information, in a separate file. Surfer is given the location of this file in a configuration option, and restores the Rust data internally. The binary data in the VCD files is then translated using this type information.

4.3. Chisel and Tywaves

Tywaves [7] is a typed waveform viewing solution for Chisel [8], [9], a modern HDL based on Scala. Before Tywaves, Chisel lacked waveform support alltogether. Designs could be compiled by Chisel to FIRRTL [10], and then further to Verilog [11] using CIRCT [12]. Tywaves updates these tools by propagating type information to the FIRRTL code, and extending the debug output of CIRCT to include this extra information. This debug data is used by a Surfer translator to translate the simulation data. The pipeline is handled by the Tywaves-Chisel API.

4.4. MyHDL and GTKWave

MyHDL [13] is a Python-based HDL. It supports typed waveforms by directly putting string representations of values into the VCD file, circumventing binary values alltogether.

Strings in VCD files are an extension to the VCD format that is supported by the GTKWave [14] waveform viewer. Although GTKWave was not designed to be extended with arbitrary translators, the string support means that it can be used to display typed waveforms by including all translations in the VCD file directly. This does require including any subsignals in the VCD as well, and provides no control over details like the display color of a value.

4.5. Clash VCD Generation

In Clash, VCD files can be generated using the `Signal.Trace` library [15]. This library provides several functions for logging Clash signals. The Clash compiler takes no part in this form of simulation: the project is simply an ordinary Haskell program making use of the Clash Haskell libraries. The main function is responsible for setting up and running the simulation, as well as storing the results in appropriate files.

Tracing has several drawbacks. First of all, the `Signal.Trace` library requires Clash's `Signal` objects to function, while most computation in Clash are performed as concurrent logic on the data inside these signals. This means that tracing intermediate values often requires extra work. Second, tracing uses impure behavior to store the values of signals without propagating these to the toplevel entity. Unfortunately, this does mean that if by lazy evaluation the trace statement is never evaluated, the trace does not end up in the output. Although this of itself may be valuable information, it generally just makes it more difficult to debug a design. As long as one value in the signal is evaluated, the signal can be collected by the system and is fully evaluated.

Alternatively, the Clash compiler may be used to compile the Clash design to any supported HDL, before simulating the generated code using any appropriate third party tool. While this has the ability to capture all values in the design, the generated code may differ from the original design in several ways, include signal names, which make it harder to understand the output. Furthermore, testbenches can no longer be easily defined in Haskell since they must be compileable, and Haskell data types cannot be used when writing testbenches in different languages.

The advantages and disadvantages of both methods are summarized in Table 4.1. Because of these properties, both methods are used in practice.

Table 4.1: Advantages and disadvantages of both VCD generation methods

Using `Signal.Trace`	**Post-compilation code simulation**
✚ Haskell available for writing testbenches ✚ No extra tools required	✚ All signals are captured
✖ Only signals can be traced ✖ Traces may be lost due to lazy evaluation	✖ Testbench creation is difficult ✖ The compiled design might differ substantially from the Clash design ✖ Requires external tools

5 System Design

This chapters deals with the major design decisions made in the project. Section 5.1 explains the decisions regarding tracing and compiled design simulation, and Section 5.2 discusses different ways of translating values. Section 5.3 summarizes the design decisions, and lists the components required for the selected design.

Because of its design focus on extensibility, the amount of control over the output, and the fact it is already in use by Clash developers, the design will focus on using Surfer as the waveform viewer.

5.1. Tracing and Compiling

Because of the respective advantages and disadvantages of creating waveform output through tracing and simulation of the compiled design, the choice was made to attempt to make Shockwaves compatible with both simulation types.

Clash can compile to different HDLs, but due to the simplicity of the generated Verilog designs, this language was chosen.

5.2. Value Decoding and Representation

Unlike Spade and Chisel, Clash supports custom bit representations. This makes decoding values much more complex. Clash itself is equipped with an `unpack` function, which is automatically generated for types with custom bit representations. It would be possible to store this custom type representation, and recreate the functionality of Clash's `unpack` function in Rust for both ordinary types and types with custom representations, but that would lead to more code redundancy.

Alternatively, it is possible to use the Haskell runtime to translate the values, as this provides access to either the original Haskell value, or the `unpack` function. These are useful for both tracing and decoding Verilog values respectively. Additionally, the Haskell runtime has access to `show`, which can be used to display values as text, and is the standard display method for most types. In general, using the Haskell runtime would allow for both standard representation of data types, and custom display rules set from within the Clash design.

However, interfacing with the Haskell runtime introduces additional complexity. First of all, the type information needs to be converted back to runtime type selection. Secondly, there needs to be a way to pass information between Haskell and Surfer. Either this is done live, which would require an interface between Rust and Haskell, or in advance, by pre-translating the values.

Live translation requires a direction communication link between Rust and Haskell, as well as access to the project while running the waveviewer. This greatly increases the complexity of the setup, and would be even more difficult when using the browser version of Surfer.

Pre-translation is much easier - the values can be translated in advance and passed to Surfer by simply writing them to a file. This does come at the cost of having to translate all values (or all values for some subset of signals) in advance, as well as having to store all translations, which as a general rule are much larger than the bit representations.

Table 5.1: Advantages and disadvantages of different translation methods

Translation in Rust	Live translation in Haskell	Pre-translate in Haskell
✚ Translation on demand	✚ Translation on demand ✚ Values displayed as done in Haskell ✚ Fully customizable	✚ Values displayed as done in Haskell ✚ Fully customizable
✖ Repeated implementations of `unpack` and `show` ✖ Limited formats	✖ High complexity of joining Haskell and Rust in runtime ✖ Requires access to the Clash design while using the waveform viewer	✖ All values are always translated ✖ Potentially large file sizes

A summary of the advantages and disadvantages can be seen in Table 5.1.

Because of the desire to avoid re-implementing `unpack` and general control and flexibility for displaying values, Haskell-based translation was chosen. Due to the time constraints of the project, the implementation was restricted to pre-translation, but this may be extended later to also support live translation.

5.3. Selected Design

The selected system design uses Haskell to translate all values in the VCD file and store these translations in separate files. This determines the shared and per-approach required components when tracing and simulating compiled code.

When tracing signals, the translations are created and added to a translation table while tracing, which is stored alongside the VCD output. This requires modification of the current tracing library.

For compiled designs, the current pipeline consists of compiling the Clash design to another HDL such as Verilog, and simulating this compiled design to obtain the VCD output, which can be opened in a waveform viewer. Shockwaves will add the following steps to this pipeline:

- Modifying the Clash compiler to propagate type information to the generated HDL code
- Linking the signals in the VCD file to type information in the compiled design
- Translating the values for all typed signals and storing the result in a translation table

Both tracing and compiled design simulation require a translator module to be added to Surfer (Chapter 6), and a Haskell library for representing values in a Surfer-compatible data format (Chapter 7). The different translation procedures are covered in Chapter 8.

6

Surfer Integration

This chapter describes the creation of a new translator for Surfer. Section 6.1 covers the main translator implementation, while Sections 6.2 to 6.4 detail the way translation data for the translator is generator from Haskell and passed to the translator module.

6.1. Translator Implementation

Surfer works with *translators*, which determine how a signal is shown. For example, one translator might format data as a signed integer, while another formats the same data as a hexadecimal value. These translators can also generate subsignals, and change a signal's appearance.

Since Surfer was designed around the option of adding new translators, adding one is rather simple. The added translator simply reads the lookup tables generated, and makes the translations available to the rest of Surfer.

More precisely, the translator keeps two tables: one that lists the Haskell types of all signals, and one that stores, for each Haskell type, the structure and translation table. The structures and translations are stored directly as Surfer's internal types, but subsignals missing in the translation are filled with `NotPresent` values recursively upon lookup (see Section 6.2). This prevents the system from having to store a lot of data just to mark a signal as not present.

The translator looks for the lookup table files in the same directory as the VCD file when the VCD file is opened. The files must have the same base filename, but a different extension indicating their function.

6.2. Translation Data Format

Surfer requires two things to be able to translate a signal: structural information, and a way to translate values from the VCD file into values for all (sub)signals in the structure.

The structure is represented by a `VariableInfo` object. This can take the value `Compound` for signals with subsignals, or `Bits`, `Bool`, `Clock`, `String` or `Real`. These other variants have some effect on the way the signal's waveform is displayed. The `Compound` variant holds a list of string-`VariableInfo` tuples denoting the names and structures of subsignals.

The value translations take the form of a `TranslationResult` object, which contains a `ValueRepr` value, a `ValueKind` value, and a list of name-`SubFieldTranslationResult` pairs. `ValueRepr` contains the actual value displayed, while `ValueKind` determines what the wave itself looks like - this mostly boils down to the color, and whether it is displayed at all. Specifically, if a signal does not exist at a given time, its kind is `NotPresent`.

6.3. Lookup Tables in JSON

The lookup tables are created using the JSON format. While this format is not that efficient, it is human readable, easy to generate, and above all, can be directly deserialized by Rust into tables of Surfer's structure and translation data types.

There are two lookup table files, corresponding to the tables stored by the translator module:

- `<waveform>.types.json` is a simple dictionary linking signal names to type names, and has the format: `{signal:type,....}`.
- `<waveform>.trans.json` contains the structures and translations as a dictionary of all Haskell types in the format: `{type:[structure,{value:translation,...}],...}` where `structure` and `translation` are simply the serialized JSON of Surfer's `VariableInfo` and `TranslationResult` types.

Examples of both formats can be found in Appendix B.

For optimization, all field and variant names have been reduced to a single character. Furthermore, `ValueKind::Normal` is set as the default value during deserialization and omitted from the JSON.

6.4. Equivalent Haskell Types

The Haskell part of Shockwaves uses direct equivalents of the Rust data types used in Surfer, and includes a module for serializing these object to JSON format.

7

Data Representation

This chapter describes how Haskell data types can be displayed in the waveform viewer by generating data in the format specified in Chapter 6. Section 7.1 describes the Haskell classes created to generate this data. Sections 7.2 and 7.3 show how different data types can be displayed, and Section 7.4 lists the implementations used for different data types used in Clash.

7.1. The Haskell Classes Display and Split

The translation process does two essentially separate things: showing the value (creating the `ValueRepr` and `ValueKind` values) and defining and generating the structure of subsignals (the `VariableInfo` and `SubFieldTranslationResult` values). In Shockwaves, these actions are performed through the two new classes `Display` and `Split` respectively.

`Display` defines the `display` method, which generates a `(ValueRepr,ValueKind)` pair. By default, this uses Haskell's `show` method to turn a value into a string, and keeps the kind to `VKNormal`. This means the type only needs to derive `Show` for `Display` to be deriveable. `display` merely returns the result of the `repr` and `kind` methods, which can be overwritten individually. The library also has a flag to use `showX` instead of `show`, which is less generic but can handle undefined subvalues better. Alternatively, this behaviour can be derived for a single type by deriving via `DisplayX`.

`Split` defines `structure`, which returns a `VariableInfo` value, and `split`, which generates a list of `SubFieldTranslationResult`s. It has a default implementation for algebraic data types as described in Section 7.2.

For primitive types, such as numbers, the value is not split into subsignals at all. To facilitate deriving this behavior, a wrapper class `NoSplit` is provided; `Split` can be derived via `NoSplit` using the `derive via` statement to obtain this behavior.

The `display` and `split` functions may be called on (partially) undefined values, which would ordinarily crash the simulation. To prevent this, the functions `safeDisplay` and `safeSplit` exist, which catch these exceptions. These are used by the function `translate`, which combines the results from the two functions, allowing the value to be split even if it is not fully defined. The `translate` function is used for generating the translation tables.

7.2. Algebraic Data Types

Most Haskell data types are fundamentally algebraic: data types have a number of constructors, which correspond to a sum type, and each constructor may have a number of fields, which correspond to a product type.

In a waveform viewer, we want to be able to split signals up into subsignals for each of their constructors and fields. More accurately, every data type has subsignals for all its constructors, of which exactly one is defined at any time. Each constructor then has subsignals for all of its fields. If the data type only has one constructor, we may omit this subsignal for legibility, and directly add the subsignals for the constructor's fields to the signal of the data type. Some examples are shown in Fig. 7.1.

```haskell
data Month = Jan | Feb | ... deriving (...)
data Date = Date {month :: Month, day :: Int} deriving (...)
date :: Date
data Instr = And Bool Bool | Invert Bool deriving (...)
instr :: Instr
```

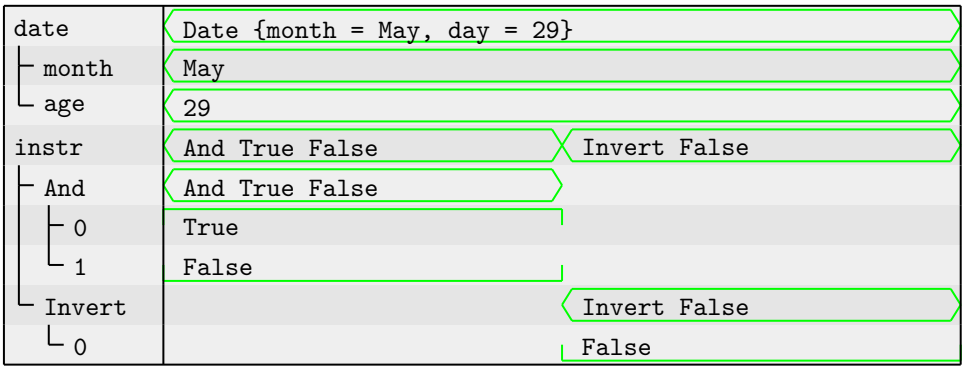

Figure 7.1: Default waveform viewer representation of algebraic data types.

It would be a tedious process to manually define these subsignals for every data type. Luckily, Haskell has a class `Generic` which can be derived for standard algebraic data types, and provides an interface to a representation of the type's structure. Using this structural representation, it is possible to create the subsignals as described.

7.3. Customized Representations and Abstract Data Types

Sometimes the default implementations of the `Display` and `Split` are insufficient to best display a data type, and custom implementations of the classes may be preferred.

A simple example would be assigning colors to the different constructors of a data value (see Fig. 7.2), to more easily discern what is happening in a design. The Shockwaves library includes a `Color` module that allows for arbitrary RGB values to be used, in addition to some default colors.

A custom implementation of `Split` is much more powerful. This is particularly useful for abstract datatypes, whose internal structure is less important (unless it is actively being debugged) than its interface. For example, array type structures such as Clash's `Vec` are defined recursively in Haskell, but creating a recursive structure of subsignals would be less readable than simply having a list (see Fig. 7.2). A custom implementation of the class makes this possible.

7.4. Builtin Types

Shockwaves comes with default implementations for all builtin Haskell and Clash types. Most of these types are either primitives, or function properly using the default implementation of `Split`. However, as mentioned in Section 7.3, there are a few types for which this is suboptimal:

- `Vec` has a custom implementation that creates a list of subsignals rather than a nested structure. The same is done for `BitVector`.
- `Either` is often used as a result value, where the `Left` constructor indicates an error. By setting a flag, the `Left` constructor is displayed using `VKWarn`.
- `Maybe` does not really benefit from having subsignals for it's constructors. Instead, it only has a subsignal for the contained value, which only exists if the `Maybe` value is of the `Just` variant. Furthermore, there is a flag to make `Nothing` display in gray.
- `Bit` and `Bool` are both single bit values, and thus use `VIBool` instead of `VIString` (the default for unsplit types).
- `RTree` has two subsignals for each of its children if it is a branch, and a single subsignal for the contained value if it is a leaf. While these are simply the two constructors's subfields, the type itself

already restricts the values to either one of these constructors. Hence, only the subsignals of the one realizable constructor are displayed.

A list of all supported types can be found in Appendix C.

```
data Color = Red | Green | Blue deriving (Show,Generic,Split,...)
instance Display Color where
  kind x = VKCustom $ case x of
    Red   -> C.red
    Green -> C.green
    Blue  -> C.blue
color :: Color
coords :: Vec 3 Int
```

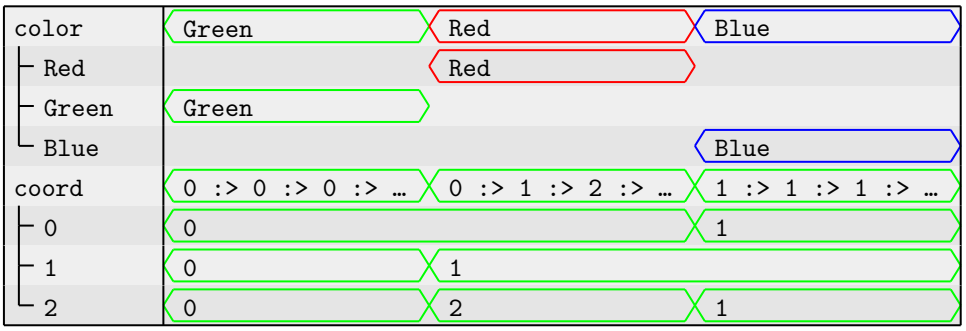

Figure 7.2: Examples of signals with customized implementations for `Display` and `Split`.

8

Translation

With the data requirements properly defined, all that is left is to actually generate the lookup tables. This step differs for the different methods of generating VCD files from Clash designs. Section 8.1 covers the solution when tracing using Clash's `Signal.Trace` module, and Section 8.2 covers the pipeline for translating VCD files generated by simulating the Verilog produced by the Clash compiler.

8.1. Tracing

For tracing, an adapted version of the `Signal.Trace` module is used. This module works by passing signals through a trace function, which uses impure behavior to store the signal in a global dictionary.

The changes to the module are relatively simple. In addition to the table of signal values, two other tables are stored globally: the signal-to-type map, and the table of structure and translation data per type. Since the simulation happens in the Haskell runtime, and the signal's type is directly available, these values can be trivially generated using the `translate` and `structure` functions. One detail of note is that the string representing the type must use the full type name, including its module of origin, to avoid namespace collisions.

`Shockwaves.Trace` acts as a drop-in replacement to `Signal.Trace`, and requires only minimal changes to store the results in 3 files, rather than just one. However, this also means the module has the same drawbacks of lazy evaluation and only being able to trace time-domain signals.

8.2. Post-simulation Translation of Verilog Simulations

The second method of obtaining a VCD file is simulating a different HDL generated by the Clash compiler. Several languages are available, but Verilog was deemed the most suitable since the generated code is relatively simple. For example, it does not turn a record-type Haskell data type into a record in the HDL, which does happen in VHDL generation.

To create VCD translations this way, several steps have to be performed:

- Linking the VCD signals to their Haskell types
  - Annotating the Haskell types in the generated Verilog code
  - Linking the signals in the VCD file to the type annotations of Verilog variables
- Translating the values of the VCD signals in Haskell
  - Gathering the values per type
  - Adding a Haskell script for performing the translation to the Clash project
  - Adding imports for all relevant types to the Haskell script
  - Generating a function linking the type signatures to translation functions for said type and adding it to the Haskell script
  - Running the script to generate the translation tables

An overview of the complete pipeline can be seen in Fig. 8.1.

8.2.1. Propagating Haskell types to the Verilog code

This is one of the most difficult steps, as it requires modifications to the Clash compiler.

In the Clash compiler, there are software types (the Haskell types) which later are transformed into hardware types (which end up in the generated HDL). Although at a type level, the compiler seems to support adding annotations to these hardware types, in practice it is not that simple. The backend of the Clash compiler uses extensive pattern matching on hardware types, which was not designed to support such annotations. As a result, adding software type annotations to hardware types breaks compilation completely.

An attempt was made to fix these issues. Unfortunately, the compiler was too complex to remove all bugs. The modified compiler was able to compile all but two of the sample programs included with Clash, and managed to add type annotations for most, but not all signals. The repository for the modified compiler fork can be found in Appendix A.

To fully make this step operational, a rather extensive overhaul of the Clash compiler and all its backends is required, which is not within the scope of the Shockwaves project, but certainly not impossible.

8.2.2. Linking VCD signals to type annotations

The next step is to link the signals in the VCD file to the type annotations in the generated Verilog.

Verilator [16] is used to generate an XML description of the Verilog design. This file is read by a Python script to extract the hierarchy of signals. The signals are then extracted from the VCD file and matched against this hierachy to find their source variables.

Verilator can accept special comments as annotations on variables, which get included in the XML file. Unfortunately, the placement of these comments is after the definition of the signal, making code generation difficult. Instead, the type information is added as a Verilog attribute annotation. When the variable is found in the hierarchy, its source location is used to look for this annotation in the Verilog code.

The mapping from signal to Haskell type is directly stored in a JSON file for Surfer to read.

8.2.3. Gathering type values

From the VCD file and signal-to-type mapping, all values of translatable signals are collected per type. This information is stored in a separate file, since it will need to be read by Haskell later. The file format is rather simple: every first line contains a type name, and every second line contains a space-separated list of values of that type.

8.2.4. Importing all relevant types

To be able to translate the bitrepresentations of the Haskell types, these types need to be in scope. This means they have to be imported.

From the full type names, all source modules are extracted and added to the import list of the translation script. This has the drawback that some types may be defined in hidden modules, and are made available through different modules. This complicates the imports greatly.

There are workarounds [17], but these are questionable practice and have not (yet) been incorporated into Shockwaves. A more proper way would require actual language support for importing non-exported data types.

8.2.5. Linking the type signatures to translation functions

Since the types of the translation functions needs to be present at compile time, while the data to be translated is only present at runtime, there needs to be a conversion table to transform string representations of a type into their type function.

The Python code generates such a function, which maches the string to the `structure` function of that type, as well as a translation function that turns string representations of the bitvalues back into values using `unpack`, and then uses `translate` to translate them to Surfer data.

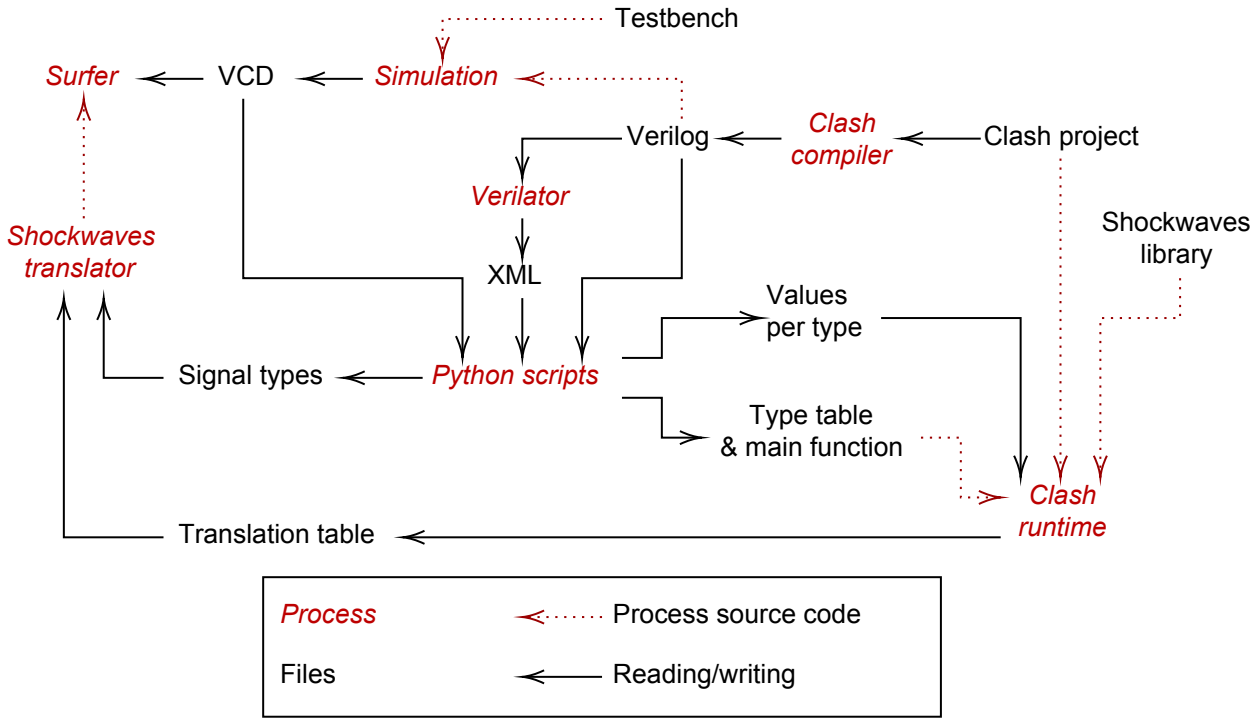

Figure 8.1: Pipeline of the Shockwaves system when using Verilog as an intermediary language.

8.2.6. Generating the translation tables

Finally, a simple main function reads the file containing the values of all types, uses this type to obtain translation functions, and uses those to generate structural information and translations for all values according to the lookup table format described in Section 6.3. These are then stored in a second JSON file for Surfer to read.

9 Results

Shockwaves was tested on both VCD file generation methods, with varying levels of success. Section 9.1 covers tracing, while Section 9.2 discusses the results of translating Verilog simulation output.

9.1. Tracing

The tracing method was tested on a real design for a USB controller, that was already using `Signal.Trace` for debugging. It was found that adapting the code required minimal effort, which was largely spent tracking down all data types used and adding the class derives to them.

Shockwaves was able to show Haskell representations for all types, an example of which can be seen in Fig. 9.1. The developer of the USB controller stated that this would have been very helpful during the debugging process.

In total, 35 signals were traced over 2118 clock cycles. The VCD file took up 119KB. The signal-to-type table used 42KB, and the translation table 514KB. Analysis of these files resulted in two observations:

- The representation of the Haskell types in string form contains largely of bytes unnecessary to uniquely identify the types. Manually removing most of these reduced the file size of the signal type table to only 8KB.
- The translation table contains a very large amount of double quotes, which are required by the JSON format. Removing these reduced the file size to 369KB, showing a different file format can greatly reduce the file size.

Although this does highlight multiple points of improvement, no performance issues were observed while testing.

9.2. Post-simulation Translation

The adapted Clash compiler was tested on the sample projects included with Shockwaves. Out of the 16 examples, compilation failed for two. For the other examples, the compiler was able to annotate some of the signals. Most of the internal signals and outputs were annotated, but most of the inputs were not. It is currently unknown what caused the compilation failures and annotation irregularities.

The full post-simulation translation pipeline was tested on a test project that included a simple accumulator, as well as a color value that changed every time the accumulator changed values. The compiler was able to compile the design, but the inputs and output were left unannotated. The remaining pipeline worked without issue, and successfully translated the signals.

Because of the compiler problems, the pipeline was not tested on a larger design.

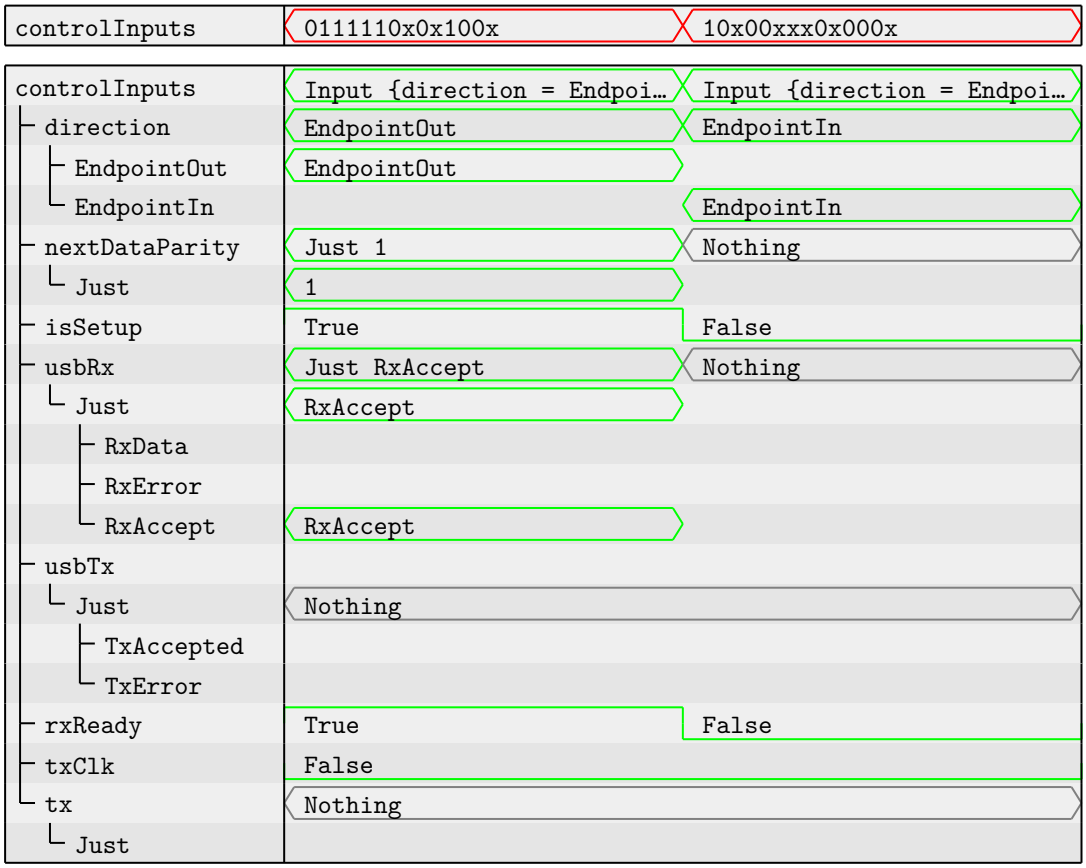

Figure 9.1: Example signal `controlInputs` from the USB interface as bitvector and after Shockwaves translation.

10 Discussion

In this chapter, the effectiveness of various parts of the system is discussed, and potential improvements and future work are suggested. Section 10.1 reflects on the choice to perform translation in advance outside of Surfer. Section 10.2 discusses the shared Haskell library code. The different methods for generating lookup tables are covered in Section 10.3 and Section 10.4.

10.1. Pre-translation

The use of lookup tables proved to work without issue for the project the system was tested on, and the file size of the lookup table remained easily manageable. However, for some projects this might not be the case. The method is particularly vulnerable to large data structures with many non-repeating values. For example, a data structure that contains a counter that is rarely reset causes each value of the total data structure to be different, resulting in a large amount of translation data. Moreover, the fact that these large data structures are now interpretable will lead to larger data structures being traced.

Instead of doing everything through lookup tables, some such data structures might be common enough to use Rust runtime translation instead. For example, instead of creating a lookup table for a vector of data types, the system could specify how this vector should be split, and use a lookup table for the child elements instead. Through the same mechanics that allow `BitPack` to be derived, rules for deserializing data could be derived automatically.

Having these methods available while keeping the option of using a lookup table would allow the user to choose more performant waveform representation options where possible or needed, while keeping the option to customize waveform viewer behavior from within the Clash design. It would, however, also require a major overhaul of the entire system, since this deserialization overrides both the `Display` and `Split` implementations, file formats, as well as the Surfer translator.

These changes would improve the performance of the system in some scenarios. Unfortunately, this comes at the cost of higher complexity, more maintenance, and general scope creep to recreate Haskell's display methods as closely as possible. Considering these drawbacks, we see no reason to change the system this way unless a project is found for which Shockwaves is rendered unusable by performance problems. Without majorly changing the system, some performance increases could be achieved through code optimization and better file formats.

10.2. Haskell Library

The Haskell library includes the translation code, default implementations for all major Haskell types, a modified tracing library and, in an 'experimental' module, code for post-simulation translation. The non-experimental modules are all directly usable.

The `Display` and `Split` classes successfully allowed for customized implementations. No use case came up where separate derivation of the classes was actually helpful, and since Haskell allows for implementations to partially rely on defaults, they might be merged into a single class to keep implementations cleaner.

Some other default implementations, such automatically color-coded constructors could be added to aid users. Another utility solution might be adding data wrappers with custom implementations, such as

a wrapper for vectors of `Maybe` values that directly displays the contained values instead of having each made available in a doubly nested structure. Such additions would be easy to add, but are very much optional nice-to-haves.

During the project, there was no opportunity to properly analyze potential runtime improvements in the system. Haskell has various laziness mechanics and alternative string representations that may be leveraged to obtain higher performance, and these could most likely be applied in various places of the library.

10.3. Tracing

The tracing module included in Shockwaves proved to be very effective. When testing the module on a real Clash design using `Signal.Trace`, adapting the code required very little effort, and had immediately clear results. The developer commented that it would have been very valuable to have Shockwaves available while debugging the design.

The most important drawback is that Shockwaves's tracing module will need to be kept up to date with Clash's `Signal.Trace` module, but this is unavoidable.

10.4. Post-compilation simulation

As mentioned before, the Shockwaves pipeline for translating values generated by simulating Verilog code has several problems. Most importantly, adding type annotations requires a major overhaul of the Clash compiler. This was too large of a task for this project, but may still be performed later.

Other parts of the pipeline are convoluted, and have several issues such as generating the import list for the Haskell translator. Some of these might be alleviated by extra compiler support, which would certainly be possible to add while upgrading the compiler. However, due to the complexity of the pipeline, it is expected most of this convolution would remain.

Despite the fact that the pipeline is most certainly not usuable in its current state, that does not mean the efforts that went into it were in vain. Though it might only function as a proof of concept for now, it shows that such a system is very much feasible, and adds value by making simulation results intelligible in the same way the tracing approach does.

Furthermore, aside from the benefits from using a compiled design rather than tracing that were previously listed in Section 4.5, a different use case has presented itself during the project: translating values from logic analysers. A currently ongoing project is focused on recording real-world signals on FPGAs running Clash designs, and a very similar pipeline could be used to translate the recorded values.

11

Summary

Waveform viewers are an essential tool in HDL development. To actually get value out of the waveforms, they must display the data in a format representatative of the original data types. Shockwaves aims to bring typed waveforms for Clash to the Surfer waveform viewer.

There are two methods to generate VCD files for Clash designs. The first is by adding traces in the design (roughly equivalent to logging statements in conventional coding) and simulating the design by executing it as a Haskell program. The second method is to compile the Clash design to a different HDL such as Verilog, and use existing simulation tools to simulate that code. Since both options have their own advantages and disadvantages, shockwaves was designed to support both these methods.

Unlike typed waveform solutions for higher level HDLs, Shockwaves uses Haskell to produce the waveform data. This data is then passed to Surfer using lookup tables. Keeping the translation in Haskell prevents reimplementation of existing Clash and Haskell functions in Surfer, and gives the programmer complete control over the way data is represented.

Shockwaves includes a tracing module that acts as a drop-in replacement to Clash's tracing module. It has been tested on a real project using tracing, and worked without issue. The size of the extra files produced combined was approximately 5 times the size of the VCD file. When used this way, Shockwaves meets all the objectives set in the introduction.

To make Shockwaves work with a compiled language, the type data needs to be propagated from the Haskell code to the VCD file. This includes having the compiler annotate the generated HDL. A proof-of-concept pipeline was developed, but the compiler modifications were too substantial to complete within the project. However, it did show the potential of the system.

It is possible to rewrite the Clash compiler to support type annotations, but this would require changes to every compiler backend for Clash. A different future goal lies in optimization, since in larger projects, the lookup tables might incur performance issues. This could be resolved by direct optimizations, as well as potentially moving translation of simple and common data types to Surfer.

Part II

Tydi

12 Introduction

When a hardware design uses data, this data often needs to be moved - into the system, out of the system, or between different submodules inside the system. Sometimes, this data is too big, possibly even unbounded, to be transferred in parallel, and is streamed instead. Streaming data between different modules generally requires some form of communication protocol.

Transferring complex data structures with variable-length fields in hardware requires *advanced* communication protocols. These may be designed on a case-by-case basis, but this leads to high implementation effort, more documentation, harder to understand designs, and as a result, more errors.

To provided a standard for many streaming applications, the Tydi specification was created. The specification was designed around transferring complex data structures, and allows for multiple related streams carrying typed, multi-dimensional data.

Of course, to be useful to developers, Tydi must be made available first. Tydi-Clash is an implementation of Tydi for Clash, meant to provide developers with the tools to use all parts of the specification.

12.1. Objectives

In the implementation of Tydi-Clash, the following objectives were pursued:

- Tydi streams are made available in Clash.
- The hardware representation of the streams is kept as close as possible to the Tydi specification.
- The design follows “the most important design guideline” [18]: make interfaces easy to use correctly, and hard to use incorrectly.
- Interfaces to the streams are provided using Haskell-friendly data types.
- Tydi-Clash is integrated with Shockwaves to improve debuggability.

Additionally, the implementation serves as a practical test of Tydi as an interface standard. Observations may be used to further improve future version of the specificaiton.

12.2. Related Work

There have been many previous advancements in the field of stream processing, resulting in a plethora of languages and frameworks both for software [19]–[22] and hardware design [23], [24]. This includes several industry standards [25]–[27], languages [10], [28] and projects attempting to integrate these with existing methods [12]. However, these address the problem at a bit stream level, rather than the much higher level of complex data types.

One interesting language is Delta [29], which allows for Haskell-like programming over time-domain streams. Nevertheless, this is still in a prototype stage, and holds no position to replace established languages such as Clash. Thus, a common standard such as Tydi is still valuable for these languages.

12.3. Outline

Chapter 13 provides background information about Tydi and several related Tydi projects, as well as some relevant Haskell and Clash libraries. Chapters 14 to 16 cover the implementation of different facets of the Tydi specification, and Chapter 17 extends this with additional verification modules. Chapter 18 provides information about the integration of Shockwaves.

To test Tydi-Clash, an example implementation was created, which can be found in Chapter 19. The observations made are discussed in Chapter 20, after which the part is summarized in Chapter 21.

13

Background

This chapter provides a background on Tydi and several Tydi projects, as well as some relevant Haskell libraries.

13.1. Tydi

Tydi [30] is a specification for complex streams. These streams support arbitrarily sized nested sequences of typed data, complex data types that require multiple separate streams, as well as reverse streams. Throughput can be managed by changing the number of parallel data lanes, and streams can be configured to several complexity levels. This section introduces most of the important aspects of Tydi, but we refer the reader to the official documentation [31] for a more complete overview.

13.1.1. Data types

Tydi defines four data types: $Null$, $Bits$, $Group$ and $Union$.

$Bits(n)$ denotes data consisting of n bits. $Null$ represents the lack of data, and is equivalent to $Bits(0)$.

$Group$ combines multiple values that exist in parallel. A $Group$ contains labeled, ordered fields that can be of any type: $Group(N_1 : T_1, N_2 : T_2, \dots)$ (labels N_i and their respective types T_i). The binary representation of a group is simply the concatenation of those of the different fields.

$Union$ combines multiple values of which only one exists at the same time. Like $Group$, the variants are labeled and ordered. In hardware, it has two fields tag and $union$ which represent which variant is contained, and the data of that variant, respectively. For example, for $Union(a : Bits(3), b : Bits(5))$, $tag = 0$ indicates the data is of variant a, and 3 out of the 5 bits of the $union$ field are used to contain the data of $Bits(3)$.

13.1.2. Physical streams

Physical streams are at the core of Tydi. They are the hardware level representation of streams, and they carry most of the complexity of the Tydi specification.

Physical streams have a number of parameters:

- c: the *complexity level* of a stream. This level determines which signals are used, and may set some behavioral restrictions. In low complexity streams, the source makes several guarantees, while in high complexity streams, the sink must support more usage options. A source may only be connected to a sink of equal of higher complexity.
- n: the number of data lanes present. Multiple lanes may be used in parallel to increase the throughput.
- d: the dimensionality of the stream. For example, $d = 2$ indicates data of the form `[[x]]`.
- T_e: the data type of data elements.
- T_u: the user data data type. Tydi streams may carry additional user data, of which the meaning is not defined by the Tydi specification.

Physical streams contain the following signals:

- $valid$ denotes whether any data is being sent.
- $data$ contains the data elements for all data lanes.
- $user$ contains the user data.
- $last$ contains bits indicating the end of a sequence.
- $stai$ indicates the first active data lane.
- $endi$ indicates the last active data lane.
- $strb$ indicates per lane whether this lane is active.
- $ready$ run from the sink to the source, and can be used to provide backpressure. Data is only transferred when $valid$ and $ready$ are both high at the same time.

Depending on the complexity level, some of these signals may be (partially) unused, or otherwise restricted. They are still defined, to allow them to be connected to higher complexity sinks that make use of these signals, but may be omitted in hardware.

13.1.3. Logical streams

Logical streams are abstract representations of collections of related physical streams. They exist as a mixture of nested streams and data, and form a bridge between the abstract, nested, complex data being transmitted, and the phsysical streams required to do so. Logical streams use slightly different parameters from physical streams:

- c: the complexity of the associated phsyical stream. The parameter may be omitted to use the complexity of the parent stream.
- t: the throughput relative to the parent stream. The number of lanes in the physical stream equals the product of the throughput values of that logical stream and all its parent streams, rounded up.
- d: the dimensionality. Depending on the synchronization mode (s), this represents either the total dimensionality of the stream, or the extra number of dimensions with respect to the parent stream.
- s: the synchronization mode. This determines whether the stream is flattened (i.e. does not include the dimensionality of the parent stream) and synchronized (one element in the parent stream corresponds to one data structure at the same dimensionality level as the parent).
- r: the direction of the stream. If the stream is reversed, in runs in the opposite direction of the parent stream.
- T_e and T_u still indicate the data type and user data data type. The stream data may include substreams.
- x: a boolean that may be used to indicate a stream should not be optimized away, even if T_u and T_e both carry no data.

Logical streams do not exist in hardware directly, but can be turned into a bundle of physical streams in a process called *synthesis*. As logical streams are transformed into physical streams, substreams are removed from the data type and further synthesized into separate physical streams.

The modules connected by Tydi streams are called *streamlets*.

13.2. Tydi-lang

Tydi-lang [32] is a programming languages developed for creating streaming architectures using Tydi.

Rather than aspiring to be a full HDL, the goal of Tydi-lang is only to describe the different streams and streamlets of a system. Streamlets may be connected together or decomposed into other streamlets, but otherwise have their implementation defined in other HDLs. Tydi-lang can generate templates for these external implementations.

13.3. Tydi-Chisel

Tydi-Chisel [33] is an implementation of Tydi for the Scala-based modern HDL Chisel [8], [9]. It was designed to be used with Tydi-lang to describe the different streams and streamlets. It also includes some utility modules. Typed waveforms for Tydi-Chisel is provided in Tywaves (see Section 4.3).

Although Tydi-Clash is in some ways the Clash equivalent of Tydi-Chisel, Tydi-Chisel puts more focus on the design pipeline and tooling outside Chisel itself.

13.4. Haskell Optics

Accessing data deeply hidden in nested data types can be rather inconvenient. In Haskell specifically, the immutability of data can make changing values rather painful. For this purpose, Tydi-Clash makes use of the Optics library [34], which provides access to (among other things) lenses and prisms.

A lens is essentially a reference to a point in a data structure. It may be used to obtain the data stored, or generate a new object with the indicated data replaced with a different value.

Optics defines *prisms* for data that may or may not be defined; in this case, retrieving the value results in a `Maybe` value, and changing the value only makes a difference when the value was previously already present.

Lenses and prisms may be combined to produce new lenses and prisms. Thus, a path to a deeply nested value may be composed of multiple optics, one per data structure.

13.5. Clash Protocols

Clash Protocols [35] is a library that contains machinery for working with interfaces. It comes equipped with implementations for several well-known interfaces such as AXI [25] and Avalon [26]. It also includes a “dataflow” protocol inspired by AXI, which consist of a simple forwards stream of one data type, and an acknowledgement signal for backpressure.

The library works with so-called *circuits* - modules with input and output interfaces - which can be connected together. Several testing tools are made available to verify the behavior of these circuits.

14

Tydi Data Types

Tydi has four main data types. The $Null$ type corresponds directly to Haskell's unit type `()`. $Bits$ denotes any serialized data; although this is most similar to Clash's `BitVector`, it is essentially isomorphic to any synthesizable Haskell type. $Group$ and $Union$ denote product and sum types respectively. Sections 14.1 and 14.2 cover the implementation of these types, and their relation to standard Haskell types is discussed in Section 14.3. Section 14.4 further elaborates on the conversion between Haskell and Tydi types, as well as between Tydi types.

14.1. Group

Tydi's $Group$ is a product type and contains a number of labeled fields. This is implemented through three Haskell types:

- A label type, wrapping the value of a field in a label existing purely at the type level
- A binary operator to join fields together
- A toplevel container type to denote the bounds of the type

Labels are constructed using the operator `>::`, which is designed to look like the standard type notation syntax `::`. It has the constructor `L`, wrapping the contained value.

```
label :: "myBool" >:: Bool
label = L True
```

To join the fields, the operator `:&:` was created. Its constructor mimics its type, and simply joins its two subtypes. The operator is right-associative and has a lower precedence than the label operator, allowing head-tail structures to be written without brackets.

```
joined :: "a" >:: Bool :&: "b" >:: Bool :&: "c" >:: Bool
joined = L True :&: L False :&: L False
-- equivalent to L True :&: (L False :&: L False)
```

Finally, the `Group` type wraps the joined fields to clearly denote the boundaries of the type. It is largely useful for readability.

```
type myGroup = Group ("a" >:: Bool :&: "b" >:: Bool)
```

It is not necessary to specify the whole structure to read or write a field, as this can be done though the `getField` and `setField` functions or the `_field` lens. These can be used to target a specific field by supplying the type level field label.

```
g :: Group ("a" >:: Bool :&: "b" >:: Maybe Bool)
g = L True :&: L Nothing
a :: Bool
a = view (_field @"a") g -- evaluates to True
```

14.2. Union

The implementation of `Union` is rather different from that of `Group`. While it would certainly be possible to represent the internal data, this has several issues:

- The operators would need to have a binary tree structure to not have a linear amount of bits specifying the variant. This structure would be more difficult to generate and navigate.
- Even in a tree structure, the location of the variant selection bits would be difficult to align.
- The data would deviate from the format specified in the Tydi specification, which has distinctly separate tag and $union$ fields.

Instead, the `Union` type only uses a binary operator (`:|:`) at the type level to describe the variants, and simply has a single record constructor with a `tag` and `union` field, the latter taking the form of a `BitVector`. This constructor is hidden, and data can only be accessed through the `getVariant` and `mkVariant` functions, or the `_variant` lens. Similarly to the access methods for `Group`, the variants are specified at the type level.

```
type U = Union ("bool" >:: Bool :|: "int" >:: Int)
u :: U
u = mkVariant @"bool" True
i :: Maybe Int
i = getVariant @"int" b -- evaluates to Nothing
```

14.3. Translating Haskell Types

Aside from primitives, Haskell data types are algebraic, and can thus be composed of `Group` and `Union` types. `Union` corresponds to a type having multiple constructors, while `Group` represents the data fields of a single constructor.

Tydi-Clash comes equipped with the `TydiConvert` class, which can be used to specify the Tydi representation of a Haskell type. The default implementation uses the `Generic` class to automatically derive the Tydi representation according to these rules:

- If a type has multiple constructors, its Tydi representation is a `Union` of these constructors, each variant having for its type the Tydi representation of the constructor.
- Otherwise, if there is a single constructor, the representation of the type is simply that of the constructor.
- Void types (types without constructors) cannot be represented.
- If a constructor has any fields, its representation is a `Group` of these fields. For record style constructors, the field names correspond to the record's field names; otherwise the fields are simply numbered.
- If the constructor has no fields, its type is `()`.

For example:

```
data T = A | B (Unsigned 3) | C {a :: Bool, b :: Bool}
-- Tydi representation:
type T' = Union (
    "A" >:: ()
:|: "B" >:: Group ("0" >:: Unsigned 3)
:|: "C" >:: Group (
            "a" >:: Bool
        :&: "b" >:: Bool
        )
)
```

During the project, a point of debate was whether to keep `Group`s containing only one field. The choice was made to keep this structure, to keep the Tydi representation closer to the original Haskell data type.

While it is fairly easy to generate Tydi types based on Haskell types, the same cannot be said about the reverse. Creating Haskell types dynamically would require using Template Haskell, and would

probably not result in the cleanest types since the programmer has little control over the result. In all likelihood, the required level of control for a practical result would be on par with simply writing the types by hand. Alternatively, one could write a script to externally generate template code instead, to expedite the process.

14.4. Type Conversion

Expanding from the type conversion between Haskell and Tydi types, a general type conversion was added, with the purpose of connecting any two compatible streams (see Section 15.4). The `convert` function can automatically convert between compatible data types according to the following four cases:

1. Conversion from a type to itself does nothing, and bypasses the entire conversion.
2. Conversion between standard Haskell types is outsourced to the `DataConvertible` class. This class has no default implementations, but may be connected to any existing conversion class though a single polymorphic instance.
3. Conversion between standard Haskell types and Tydi types works by converting the Haskell type to or from its Tydi representation, and then converting between the Tydi type and the Tydi type representation of the Haskell type.
4. Conversion between Tydi types happens based on structure: both types must have the same number of fields or variants, and these must be pairwise compatible. The field names are ignored.

Since the Tydi type is generally not simplified, the structures might not match exactly. Furthermore, the conversion ignores field names, which might result in unexpected behaviour if two fields have compatible types and appear in a different order. However, arbitrary reordering is in the general case not allowed, because it may change the behavior of the interface. Improvements of this conversion are left for future development.

15

Physical Streams

This chapter discusses how Tydi's phsyical streams can be represented in Clash. Section 15.1 describes how the streams are internally represented in Clash. Section 15.2 and Section 15.3 detail how Tydi streams can be interpreted as certain Haskell and Clash types, and how this determines the way these streams are used in a Clash design. Finally, Section 15.4 handles connecting streams and streamlet ports.

15.1. Streams and Ports in Clash

Unlike Chisel, Clash does not have a notion of "port objects". Instead, ports are simply the inputs and outputs of functions. This also means that the $ready$ signal present in a Tydi interface cannot be bundled together with the forwards signals, since a function's inputs and outputs are separated.

In the Clash design, the stream exists in the form of the values taken on by the different signals described in the Tydi specification. The forwards signals are bundled into a `PStream` object, while the $ready$ signal is represented by `PStreamReady`. A source is any function taking the `PStreamReady` signal as an input, and producing the corresponding `PStream` value as an output. Conversely, a function taking the `PStream` value and producing the `PStreamReady` acts as a sink.

Both `PStream` and `PStreamReady` are parameterized with the types `c`, `n`, `d`, `u`, and `e`. These denote the complexity level, number of lanes, dimensionality, user data type and data type respectively. Note that these correspond to the parameters of Tydi's physical streams directly, though they appear in a different order - this is done to increase readability when written in Haskell.

The types of the values inside the `PStream` data type depend on all of these parameters. `PStreamReady` objects do not use the types directly, but these types are still included to match the ready signal to its `PStream` counterpart. A simplified version of the definition can be seen below:

```
data PStream c n d u e = PStream
  { valid :: Bool
  , dat   :: Vec n e
  , user  :: u
  , last  :: LastType' c n d
  , stai  :: StaiType' c n
  , endi  :: Index n
  , strb  :: StrbType' c n
  }

data PStreamReady c n d u e = NotReady | Ready
```

The `PStream` data type was designed to only contain signals that are actually defined for the complexity level. This means that depending on the complexity level, some signals have different types. And overview of the types can be seen in Table 15.1.

- If the complexity level has last bits per lane, `LastType' c n d` evaluates to `Vec n (Vec d Bool)`. Otherwise, the type is `Vec d Bool`.

Table 15.1: Internal physical stream data types

c	valid	data	user	last	stai	endi	strb
1	Bool	Vec n e†	u	Vec d Bool	()	Index n	Bool
2							
3							
4							
5							
6					Index n		
7							Vec n Bool†
8				Vec n (Vec d Bool)			

Data types for a physical stream with `n` data lanes, `d` dimensions, data data type `e`, user data data type `u`, and complexity `c`.
† May contain undefined values even when `valid` is high.

- If the complexity level has a start index, `StaiType' c n` evaluates to `Index n`. Otherwise, the type is `()`.
- If the complexity level has independent strobe bits for the data lanes, `StrbType' c n` evaluates to `Vec n Bool`. Otherwise, the type is `Bool`.

15.2. Representing Control Signals

In Tydi streams, the $data$, $last$ and $user$ signals carry the actual data. The other signals, $valid$, $stai$, $endi$ and $strb$, are control signals that specify which (parts) of these data signals are actually defined.

In Clash, having separate control signals is generally considered bad practice. Instead, the data is encapsulated in types that represent the control. For example, a memory block that has a `address`, `value` and `write_en` input for writing, might in Clash have an input of the type `Maybe (Index memsize, ValueType)`: although the `address` and `value` signals always exist in hardware, their values are only defined when write enable is high.

This practice is in line with the most important design principle: it is made difficult to use the interface incorrectly, because a value must be defined when write enable is high, and cannot be supplied when it is low. At the same time, the functions available for the `Maybe` type make it easier to work with the combination of the enable signal and its related value signals. Therefore, we investigate and implement such representations for Tydi streams.

15.2.1. Analysis of Tydi control signals

At the top level, `valid` controls the validity of all other signals. `stai` and `endi` together form a mask leaving a slice of the data lanes, but also of the individual strobe signals of those data lanes. Finally, the individual `strb` bits act as per-lane masks. It should be noted here that at lower complexity levels, where the strobe consists only of a single bit, it might seem that this strobe signal should mask not only the data lanes, but `stai` and `endi` as well. After all, if the strobe bit is low, no data is transferred, and thus the values of `stai` and `endi` are irrelevant. However, the Tydi specification still mandates that these signals are properly defined in order to make it possible to connect the stream to a higher complexity sink.

15.2.2. Representing valid

Now all that remains is to select appropriate types to represent the different data signals combined with their control signals. Ideally, the set of values representable by the types chosen is exactly equal to the set of valid values of the Tydi stream. At the highest level, there is `valid`, acting as an enable of all other data. This can me modeled through Haskell's `Maybe` type, as it implements the optionality of the other signals. Modeling enable signals using `Maybe` is standard practice in Clash. Since it will be useful in the rest of the design, if `valid` is high, all signals apart from `valid` grouped together are from here on referred to as a `Transfer` object (Eqs. (15.1) and (15.2)). Since the `user`, `last`, `stai` and `endi` signals

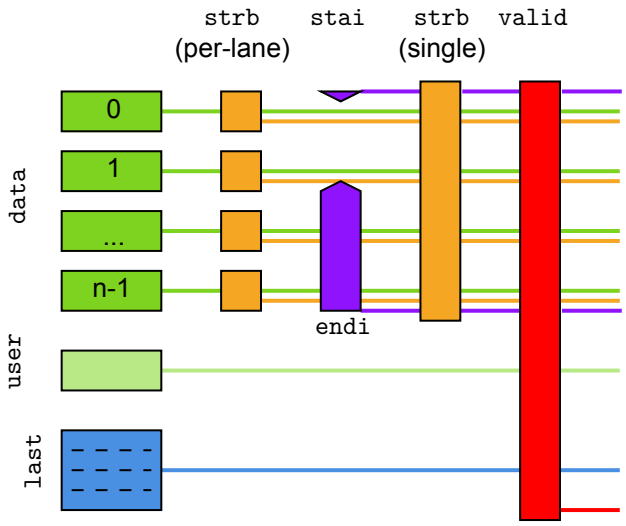

Figure 15.1: Signal validity masking in a physical stream. `data` is masked by individual strobes, if they exist, which are then masked by `stai` and `endi`, which are in turn masked by the single strobe bit if present. All signals are masked by `valid`.

of a `Transfer` never contain undefined values, they may be accessed directly.

$$\texttt{PStream} \equiv \texttt{Maybe Transfer} \tag{15.1}$$

$$\texttt{Transfer} \equiv \texttt{(data,user,last,stai,endi,strb)} \tag{15.2}$$

15.2.3. Representing strb

The strobe signal can be modeled in a similar fashion to `valid`. At low complexity levels, where there is only a single strobe bit, the data lanes can, as a whole, be wrapped inside of a `Maybe` type (Eq. (15.3)). At high complexity levels, when strobe provides per-lane enable signals, the data of the individual lanes is wrapped in `Maybe` instead (Eq. (15.4)).

$$\texttt{(data::Vec n e, strb::Bool)} \equiv \texttt{Maybe (Vec n e)} \tag{15.3}$$

$$\texttt{(data::Vec n e, strb::Vec n Bool)} \equiv \texttt{Vec n (Maybe e)} \tag{15.4}$$

15.2.4. Representing stai and endi

This leaves us with the `stai` and `endi` signals. If only `endi` is present, the valid lanes are a non-empty prefix of the data lanes (Eq. (15.5)). If both are present, they form a slice instead (Eq. (15.6)). These can then be combined with the strobe in the same way as before.

Clash does not have a type for vector prefixes or slices, so a new type is required. Although it is certainly possible to represent this slice as an algebraic data structure, the type would be rather complicated, and most likely hard to use in practice. Furthermore, the implementation would need to be rather complex to avoid unnecessary data shifting in hardware due to to the binary representation of the slice. Therefore, we instead choose to represent the data as an abstract data type wrapping a normal vector.

$$\texttt{(data::Vec n e, endi::Index n)} \equiv \texttt{Prefix n e} \tag{15.5}$$

$$\texttt{(data::Vec n e, stai::Index n,endi::Index n)} \equiv \texttt{Slice n e} \tag{15.6}$$

Table 15.2: External physical stream data types. Types that share the internal representation are omitted.

c	data	strb
1	`Maybe (Prefix n e)`	`Bool`
2		
3		
4		
5		
6	`Maybe (Slice n e)`	
7	`Slice n (Maybe e)`	`Slice n Bool`
8	`Vec n (Maybe e)`	`Vec n Bool`

The new types, `Slice` and `Prefix`, are implented as a `Vec` with an index range and upper bound respectively. They are protected against reading the values outside the range. All applicable standard vector functions, as well as some additional utility functions, are provided.

At high complexity levels, where a per-lane strobe is present, the fact that `stai` and `endi` create a consecutive set of data lanes becomes irrelevant. It is possible to transform `stai` and `endi` into two additional per-lane strobes, and combine these with `strb` to obtain a single strobe (Eq. (15.7)), which can then be applied to the data lanes (Eq. (15.8)). This greatly reduces implementation complexity of the sink. Similarly, when creating a transfer object in the source, just the strobe may be used to specify the validity of data lanes. In this case, `stai` and `endi` are simply set to enable the full range of data lanes. Generally, this means `stai` and `endi` signals can be removed in optimization. Because of its general usefulness, any prefix or slice may also be transformed to a vector of `Maybe` values. However, since this type does not guarantee the active data lanes are consecutive, it cannot be used to write data for complexity levels without a per-lane strobe.

$$\texttt{(stai::Index n, endi::Index n, strb::Vec n Bool)} \approx \texttt{Vec n Bool} \tag{15.7}$$

$$\texttt{(data::Vec n e, stai::Index n, endi::Index n, strb::Vec n Bool)} \approx \texttt{Vec n (Maybe e)} \tag{15.8}$$

The representations of `data` and `strb` in the presence of `stai`, `endi` and `strb` can be seen in Table 15.2.

15.3. Interacting with Physical Streams

To keep the synthesized code close to the Tydi representation, the `PStream` object is kept close to this representation in terms of structure. However, as discussed, the data can be presented differently to make it safer and easier to use. For this reason, the `PStream` constructor is kept hidden, and can only be interfaced with through functions and patterns, making it an abstract data type.

First of all, $valid$ can be represented as a `Maybe` type. We introduce the `PStreamTransfer` type, with the same parameters and structure as `PStream` except for the `valid` field. The `PStream` can then be turned into `Maybe PStreamTransfer` though the `getTransfer` function, and turned back using `transfer`. Alternatively, the `Transfer PStreamTransfer` and `NoTransfer` patterns are present to match the `PStream` directly, as if it had two constructors like `Maybe`:

```
case stream of        -- stream :: PStream c n d u e
  Transfer tf -> ... -- tf :: PStreamTransfer c n d u e
  NoTransfer  -> ...
```

All other signals are accessed through methods of this `PStreamTransfer` object. These functions require various class implementations dependent on the complexity level. To avoid having to add constrains for all of these, the classes are bundled under a single class `CompleteComplexity`. This class essentially indicates that the stream is properly defined.

15.3.1. Access functions

Most signals ($user$, $last$, $stai$, $endi$) are directly accessible using getter functions: `getUser`, `getLast`, `getStai`, `getEndi`. Since $stai$ and $last$ have types dependent on the complexity level, there are extended functions that add back unused signals, essentially lifting them to the maximum complexity representation: `getLastExt` and `getStaiExt`. These extended functions help with creating complexity-independent functions.

The two remaining signals may have undefined values in them, and thus have transformative access functions. For data, these are `getDataSliced` and `getDataStrobed`. The former returns the data in the complexity-dependent format constructed throughout Section 15.2, while the lattern returns individually strobed data regardless of the complexity level as per Eq. (15.8). Since some implementations might benefit from direct but unsafe access, `getDataRaw` return the data lanes directly.

Finally, $strb$ has both a data type dependent on the complexity level, thus prompting an extended getter variant, and may be partially obscured by $stai$ and $endi$, prompting raw getter functions. Thus, there are four getter functions: `getStrb`, `getStrbExt`, `getStrbRaw` and `getStrbExtRaw`.

15.3.2. Creator functions

The data types that have extended getter functions also have functions for creating values. `mkLast`, `mkStai` and `mkStrb` take both a simple and extended value, and return either based on the complexity level.

Finally, the different functions for obtaining the data all have respective functions for creating a `PStreamTransfer`. `fromSliced` creates a transfer from the sliced data, the last values, and user data. `fromStrobed` similarly uses the strobed data, but can only be used if the complexity level defines a per-lane strobe. `fromSignals` creates the transfer from the raw data signals. When creating the transfer this way, all signals undefined by the Tydi standard are replaced with `undefined`s.

15.4. Connecting Streams

Tydi allows sources of a certain complexity level to be connected to sinks of a higher complexity level. For this to work in Haskell, the data type needs to be converted. This is done through the `connect` function.

A `PStream c n d u e` object may be converted to a `PStream c' n' d' u' e'` if the following conditions are met:

- `c` $\leq$ `c'`: The complexity level cannot decrease.
- `n` $=$ `n'`,`d` $=$ `d'`: the number of lanes and dimensionality are both equal.
- `u` and `u'`, as well as `e` and `e'`, share an implementation of `TydiConvert`, i.e. they may be converted between as described in Section 14.4.

For `PStreamReady`, the same conditions are set, except for the complexity level: this much be equal or *lower*, since the $ready$ signal runs from the sink to the source.

The `connect` function does not allow connecting the ready signals of two incompatibly typed streams, since these streams cannot be connected to each other. However, it may be useful to propagate the ready signal of one stream to another. For this purpose, `connectReady` exists, which converts between any two subtypes of `PStreamReady`.

16

Logical Streams

This chapter details aspects of Tydi-Clash related to logical streams and stream bundles. Section 16.1 explains the Tydi-Clash equivalent of Tydi's logical streams. Section 16.2 describes how these are synthesized into structures of physical streams, which are further elaborated in Section 16.3.

16.1. Representation of Logical Streams

The logical streams are implemented through types, and converted to bundles of physical streams using type-domain computations. This allows the programmer to compose Tydi stream descriptions using a combination of Clash data types, Tydi data types, and logical streams. These logical streams have the same arguments as Tydi's logical streams, and the same shorthands are defined (such as `Rev` and `Dim`). The logical stream type does not have any constructors, and so no value can be instantiated, nor does it have a hardware equivalent. It is purely meant for synthesis to physical streams.

16.2. Synthesis into Phsyical Stream Bundles

Tydi's synthesis process produces a bundle of physical streams. As discussed in Section 15.1, the nature of Haskell makes it impossible to create a single bidirectional port. Instead, the synthesis produces two bundles of signals: one of all forwards signals, and one of all reverse signals. The synthesis process directly creates the forwards bundle, which can then be reversed to obtain the reverse bundle. Reversed streams are synthesized by reversing the synthesized forwards stream.

Tydi-Clash stream bundles differ from the Tydi specification in another major way: instead of flattening the logical stream, and renaming the physical streams according to their position in the hierarchy, the structure of the logical stream type is kept to represent the bundles. This allows substreams to be accessed including their own substreams.

In this hierarchy, all logical streams are turned into stream nodes that hold both the subhierarchy of the bundle and the physical stream associated with the logical stream. All data in this hierarchy is turned into the unit type `()`, since the data does not exist outside the streams. Similarly, for the physical streams, all nested streams are replaced with the unit type in the data data type. Because a `Union` of logical streams results in both being instantiated in parallel, all occurrences of `Union` in the data are transformed to `Group` in the bundle hierarchy.

Suppose we have a memory element that can receive addresses and stream back the values at those addresses. That can be represented using a reversed substream, as in Fig. 16.1. The value is a substream, since it is the direct response to an address value. Synthesis turns the logical stream description with a nested stream into a hierarchical structure. In the forwards physical stream bundle, the reverse stream shows up as the `ready` signal. The corresponding Haskell code then looks like:

```
type MemStream = New (C 5) 1 () (
    Group (
        "address" >:: Index MEMSIZE
    :&: "value"   >:: Rev CInherit 1 () (Unsigned 32)
    )
)
```

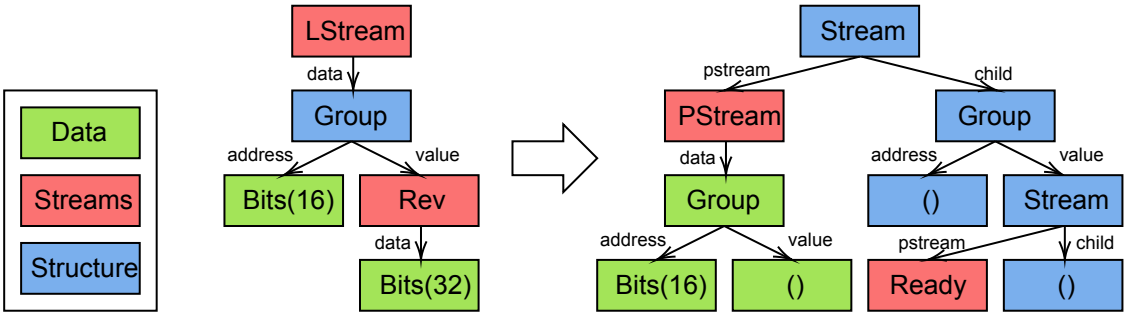

Figure 16.1: Synthesis of a logical stream to a forwards physical stream bundle.

```
type MemStreamBundle = TydiSynth MemStream
memory :: Signal dom MemStreamBundle -> Signal dom (Reverse WordStreamBundle)
```

The synthesis process currently does not optimize away streams that do not carry any data. Consequently, the boolean in the logical stream type denoting a stream may not be optimized away is left unused.

16.3. Interacting with Physical Stream Bundles

Physical stream bundles consist of only three types: `Group`, `()`, and `StreamNode`. Tydi-Clash has the lenses `_child` and `_stream` to access the subhierarchy and physical stream fields of these `StreamNode` values respectively. Combined with the `_field` lensen for `Group`, these can be used to access any value inside a stream bundle.

It is often useful to switch between signals of data structures and data structures of signals, and Clash defines the `bundle` and `unbundle` functions for this purpose. These are implemented for `Group` and `StreamNode`, as well as `PStream` and `PStreamReady`, to also allow this conversion for bundles of physical streams.

17

Behavioral Verification

Although typing may be able to prevent some incorrect usage, the Tydi standard also specifies some behavioral rules that extend beyond the domain of single cycle values. To facilitate testing whether these rules are adhered to, Tydi-Clash has several checks that can be included in a design for verification.

Sections 17.1 and 17.2 detail the checks performed on all physical streams, while Section 17.3 lists the additional checks for constraints imposed on low complexity streams. These checks are all performed by passing signals of both the `PStream` and its respective `PStreamReady` through a sequential logic funcion keeping track of the stream's state.

Section 17.4 sheds light on the behavioral requirements of nested streams, and why these cannot easily be checked.

17.1. Stable Data Transmission

Tydi specifies that transmission data must be stable: as soon as $valid$ is set to high, all other (defined) signals may no longer change until the transfer is completed by a high value bit in $ready$.

This property is easy to check: one simply needs to compare the current value of the stream to the previous stream and ready signals. If the previous clock cycle contained a transfer not accepted by $ready$, the current stream value must be that same transfer. For this, the values of `getDataSliced`, `getLast`, `getUser`, `getStai`, `getEndi` and `getStrb` are compared (as well as whether there is a transfer at all). If these values do not match, the stream is invalidated by replacing it with an `errorX`.

The check can only be performed if the stream's data types (i.e. the data and user data) are members of the `Eq` class.

17.2. Correct Sequence Termination

The Tydi specification states that sequences at one dimension cannot be terminated without, first or at the same time, terminating inner sequences. Tydi-Clash contains a checker for this restriction.

The presence of empty sequences make the check slightly more complicated: if no data is sent, but a last bit is sent for a certain dimension, that denotes an empty sequence, which from the perspective of outer dimensions means data has been transmitted. Although empty sequences are largely left untouched by the specification, Tydi-Clash was built to support these data structures.

The checker works according to the following rules:

- Data sent opens all dimensions.
- A last bit on one dimension opens all lower dimensions.
- A last bit terminates the sequence of the corresponding dimension. This happens after any dimensions have been opened, i.e. a dimension may be opened and closed within the same cycle.

The checker keeps track of which dimensions have been opened, and invalidates the stream if a dimension is closed while a higher level dimension is still opened. The checking logic is implemented as a systolic array, which can be seen in Fig. 17.1.

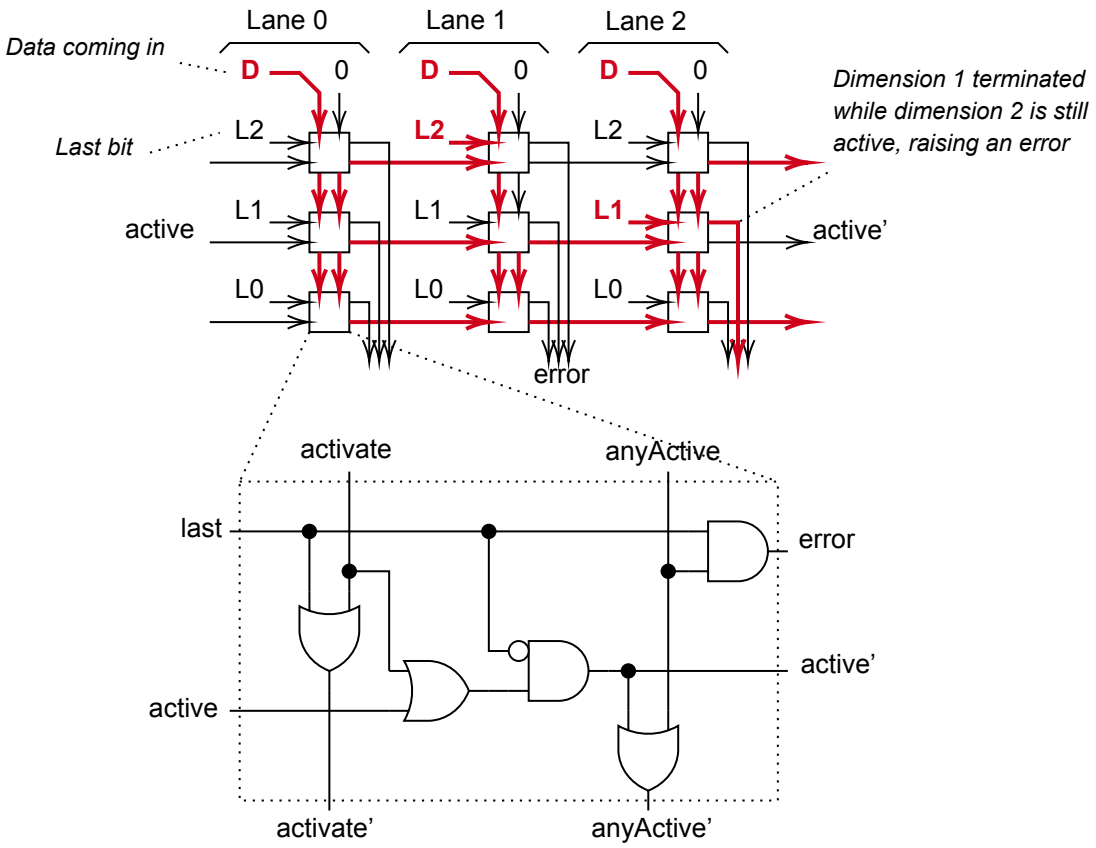

Figure 17.1: Sequence termination error detection logic of a stream with complexity 8, 3 data lanes and 3 dimensions. The highlighted signals show an example or erroneous data being detected.

17.3. Complexity Level Restrictions

Below complexity level 5, the complexity levels do not determine the presence of signals, but rather behavioral restrictions. For all four restrictions, a check is implemented to verify this behavior. Since all restrictions relate to sequences, the checks are only performed on streams with a dimensionality of at least 1.

The current Tydi documentation of the lower complexity levels is rather brief, and does not take into account the possibility of empty sequences. Hence, the checks will be extended to support empty sequences, and are based on the quoted parts of the Tydi specification [31].

> "C<5: All lanes must be active for all but the last transfer of the innermost sequence."

This is might seem easily checkable: "a transfer must have all lanes active, unless the last bit for the innermost sequence is set". However, the last bit might occur only in a later cycle. Furthermore, this check fails for the transfer of empty sequences in higher dimensions.

Instead, the check is implemented as follows: if a previous transfer contained data in a strict subset of the data lanes, without the innermost sequence being terminated by a `last` bit, any new data being sent will raise an exception.

> "C<4: The last flag cannot be postponed until after the transfer of the last element."

Again, this check is complicated if empty sequences are allowed. The rule is implemented as "If the last flag of a certain dimension is set, while that dimension previously had data, and that dimension currently has no data, an error is raised".

> "C<3: Innermost sequences must be transferred in consecutive cycles."

This check is much simpler: if data has been sent without terminating the innermost sequence, any `NoTransfer` invalidates the stream.

> "C<2: Whole outermost instances must be transferred in consecutive cycles."

The final check is similar, but needs an addendum for empty sequences. If data has been sent, *or any last flag has been set at a higher dimension level*, without terminating the *outermost* sequence, a `NoTransfer` invalidates the stream.

17.4. Inter-stream Dependencies

To avoid deadlocks, Tydi specifies an ordering for nested streams: streams are ordered depth-first, left-to-right, in a preordering. Although data does not have to be transferred in this order, neither the source nor the sink may rely on the other port supporting out of order data transfer.

The fact that ports may communicate out of order, yet are not allowed to depend on this behaviour makes it difficult to check. This is further worsened by the fact these structures are very application-dependent.

Rigorous analysis of Tydi's inter-stream behaviors is still being worked on. Tydi-Clash currently does not include any tools for asserting correct behavior of streamlets. It might be possible to add a layer between ports that blocks physical streams from sending data out of order, but this would probably often be less error-prone than out-of-order stream handling, and thus miss those bugs.

18

Shockwaves Integration

Although its use is far broader, the primary reason for the creation of Shockwaves was to be able to visualize the Tydi types in Tydi-Clash. Due to their high complexity, Tydi streams would be nigh impossible to debug using a waveform viewer that cannot interpret Haskell types. This chapter covers the integration of Shockwaves for Tydi-Clash data types.

18.1. Tydi Data Types

The Shockwaves implementations for the Tydi data types, `Group` and `Union`, are straightforward, since their equivalence to normal algebraic Haskell data types has already been established in Section 14.3, and these types already have Shockwaves implementations (see Section 7.2). `Group`s are split into their separate fields, while `Union`s are matched to their current variant, and only this variant is displayed. Examples can be seen in Fig. 18.1.

```
coord :: Group ("x" >:: Unsigned 6 :&: "y" >:: Unsigned 6)
filter :: Union ("none" >:: () :|: "lessThan" >:: Int)
```

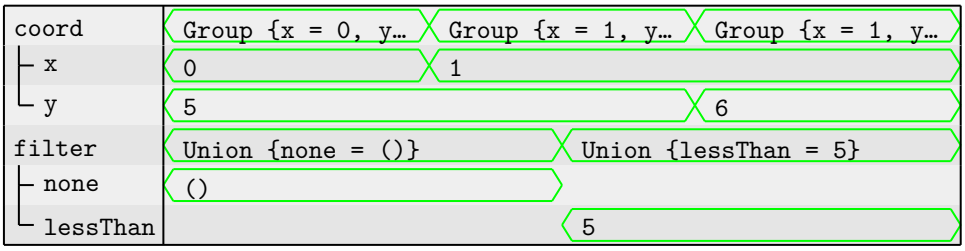

Figure 18.1: Shockwaves signals for `Group` and `Union`.

18.2. Slices and Prefixes

`Slice` and `Prefix` are similar to Clash's `Vec`, and are thus displayed in a similar fashion. Two changes are made: values outside of the sliced range are not displayed, and an extra signal is added to display the end (for a prefix) or range (for a slice) values. Examples can be seen in Figs. 18.2 and 18.3.

```
dat :: Prefix 3 Int
```

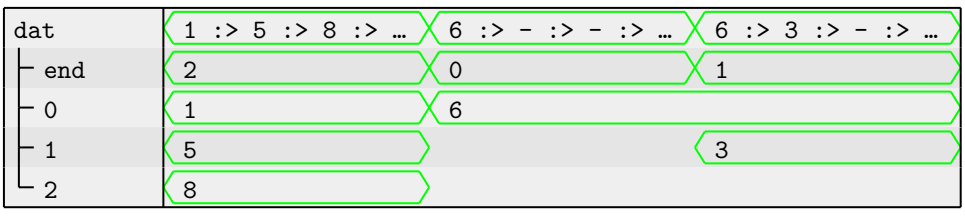

Figure 18.2: Shockwaves signals for `Prefix`.

```
dat :: Slice 3 Int
```

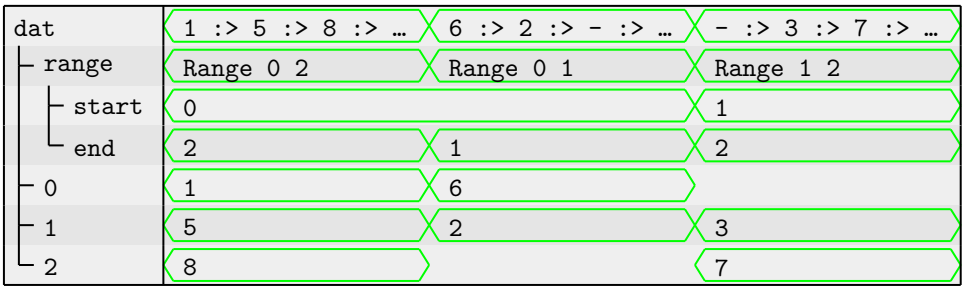

Figure 18.3: Shockwaves signals for `Slice`.

18.3. Physical Streams

The final data types to integrate are those for physical streams. An example can be seen in Fig. 18.4.

`PStream` are rendered like `Maybe` values: there is a single subsignal for the transfer, if present. This `PStreamTransfer` is displayed using the signals specified in Tydi. Most of these signals (`last`, `user`, `stai`, `endi`) are displayed directly as stored internally. The strobe data is displayed as returned by `getStrb`. The data signal is shown as obtained a vector, where, like in the implementations for `Slice` and `Prefix`, the individual data values are only present if defined.

`PStreamReady` has a custom implementation of `Display`, coloring the `NotReady` constructor red. This makes it easier to see when a stream is being blocked by the sink.

```
stream :: PStream (C 1) 2 0 () Int
ready :: PStreamReady (C 1) 2 0 () Int
```

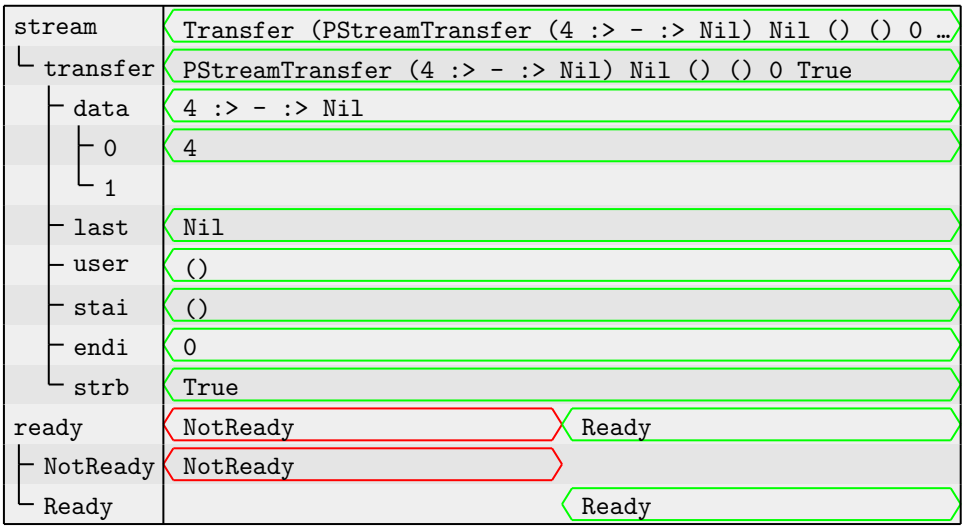

Figure 18.4: Shockwaves signals for `PStream` and `PStreamReady`.

19 Example Implementation

To test Tydi-Clash, an example implementation of a streamed system was made. The goal of the system in introduced in Section 19.1. The system and its implementation are described in Section 19.2 and Section 19.3, and findings are covered in Section 19.4.

19.1. Problem Statement

The problem is about parsing JSON files containing Scrabble data. Each file contains a list of words, and its number of occurances. Bonuses on the Scrabble board are ignored. The end goal is to, for each file, sum up the total scores achieved.

The input format is a list of JSON objects, which each include a word. Optionally, if the word occured more than once, a count field may be present. There may be other fields, which are only allowed to have string and integer values, but these are to be ignored. It may be assumed that the count field always precedes the word field. For simplicity, any strings, including field names, must be alphanumeric.

Some examples:

```
[
  {
    "count": 3,
    "word": "ABRACADABRA"
  },
  {
    "ignored": 5,
    "word": "ONCE",
    "ignored2": "X"
  }
]
```

```
[{"word":"A"}]
```

```
[]
```

19.2. Tydi Streamlet Pipeline

The process of parsing this data is split into a number of stages, aimed at testing various Tydi features. First, the files are streamed as loose characters. A state machine parses these into a higher dimension sequence of field-value pairs. These are then grouped together to words with their counts. The next stage computes the total score of each word, and the final state sums up all the scores per file.

The different modules and their outputs can be seen in Fig. 19.1 and Table 19.1.

19.3. Implementation

This section describes the pipeline modules in more detail.

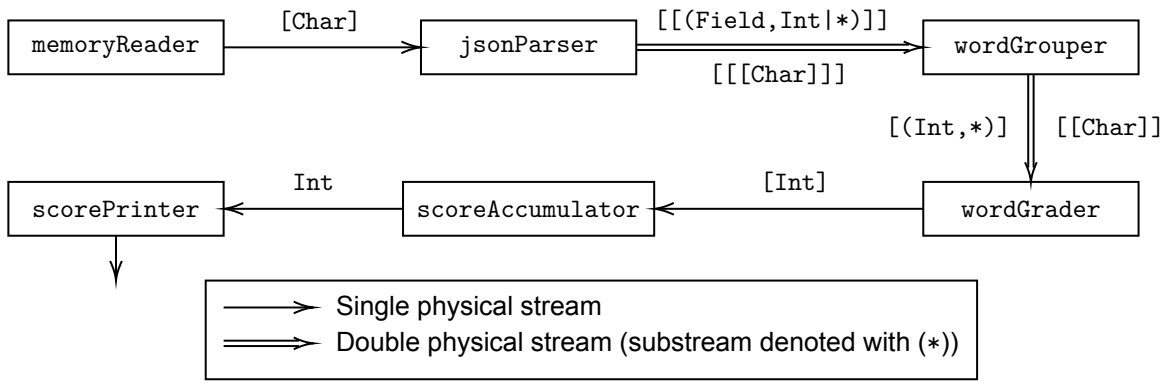

Figure 19.1: JSON parsing pipeline. Stream types are given in simplified notation.

Table 19.1: Modules of the JSON parsing pipeline.

Module	Output	Description
`memoryReader`	$Stream(c = 1, t = 16, e =$ `Char`$)$	Read the input files.
`jsonParser`	$Stream(c = 5, d = 2, e =$ $Group(field:$ `JsonField`$, value:$ $Union(int:$`Int`$, string:$ $Stream(t = 4, d = 1, e =$ `Char`$))))$	Parse the JSON and return a sequence of field-value pairs. Also reduce the number of lanes to 1 for the fields, and 4 for any strings.
`wordGrouper`	$Stream(c = 5, d = 1, e =$ $Group(count:$ `Int`$, word:$ $Stream(t = 4, d = 1, e =$ `Char`$)))$	Combine the field-value pairs for a word into one single item with a substream for the word.
`wordGrader`	$Stream(c = 5, d = 1, e =$ `Int`$)$	Calculate the score of each word, multiplied by its occurance.
`scoreAccumulator`	$Stream(c = 5, e =$ `Int`$)$	Sum the scores of each word per file.
`scorePrinter`	`Maybe Int`	Output the stream of summed scores as a `Maybe` signal.

Streams are giving in Tydi notation. Unless otherwise specified, streams take the default parameters $t = 1$, $d = 0$, $u = Null$, $r = Forward$, $s = Sync$, and the complexity is inherited from the parent stream.

19.3.1. Memory reader

For testing purposes, the files are hardcoded. The `memoryReader` contains a small state machine that loops over the different files, and transmits their contents in chunks. After all files have been read, the reader stops transmission.

19.3.2. JSON parsing

The `jsonParser` module is by far the most complicated module. Not only does it parse the JSON, it also reduces the data to fewer lanes. A state machine update function is applied to all lanes of the input. Based on state transitions, four values are generated: data for the outer stream containing the field, sequence termination bits for this stream, data for the word characters substream, and sequence termination bits for this substream.

These values are compacted separately for both streams, and divided into blocks that can be transmitted at once over one stream. A transfer may not have more data items than the number of data lanes, and last bits must occur at the end of the transmission block. Combining consecutive compatible last bits is not supported for simplicity. The transfer at the input is only completed when all data for both streams has been transmitted to the next module successfully.

An example of this process can be seen in Fig. 19.2.

19.3.3. Word grouping

The `wordGrouper` module groups together field-value pairs into word-count pairs. Any count field with an integer value is stored. When a word field with a string value arrives, the module sends an item containing the count, and transmits the word string as a substream. For this, it is important that the count field arrives before the word field. The count defaults to `1`.

19.3.4. Word grading

The `wordGrader` looks up the Scrabble score for each letter in the word, and multiplies it with the word count. Once a last bit is received for the innermost sequence, indicating the word has fully been processed, the total score is transmitted. Any messages that do not contain data, but contain a last bit for the whole file, directly create a transfer with the same last bit at the output.

19.3.5. Accumulator

The `scoreAccumulator` simply sums up the scores in each file, and transmits the total once a transfer contains a last bit indicating the file is completed. This also resets the score.

19.3.6. Output

The `scorePrinter` ultimately functions as the sink of the whole pipeline. It is always ready to receive, and any total scores received are sent to the output in the form of a simple `Maybe` value.

19.4. Results

The example Tydi-Clash project, consisting of 600 lines of code, was compiled into VHDL (6000 lines) and Verilog (5000 lines). This might seem like a large improvement, but this is largely just due to the way the Clash compiler works. It would be possible to write a design in these languages in fewer lines, possibly as little as 1000. However, this code would be harder to write and understand due to the lack of abstractions.

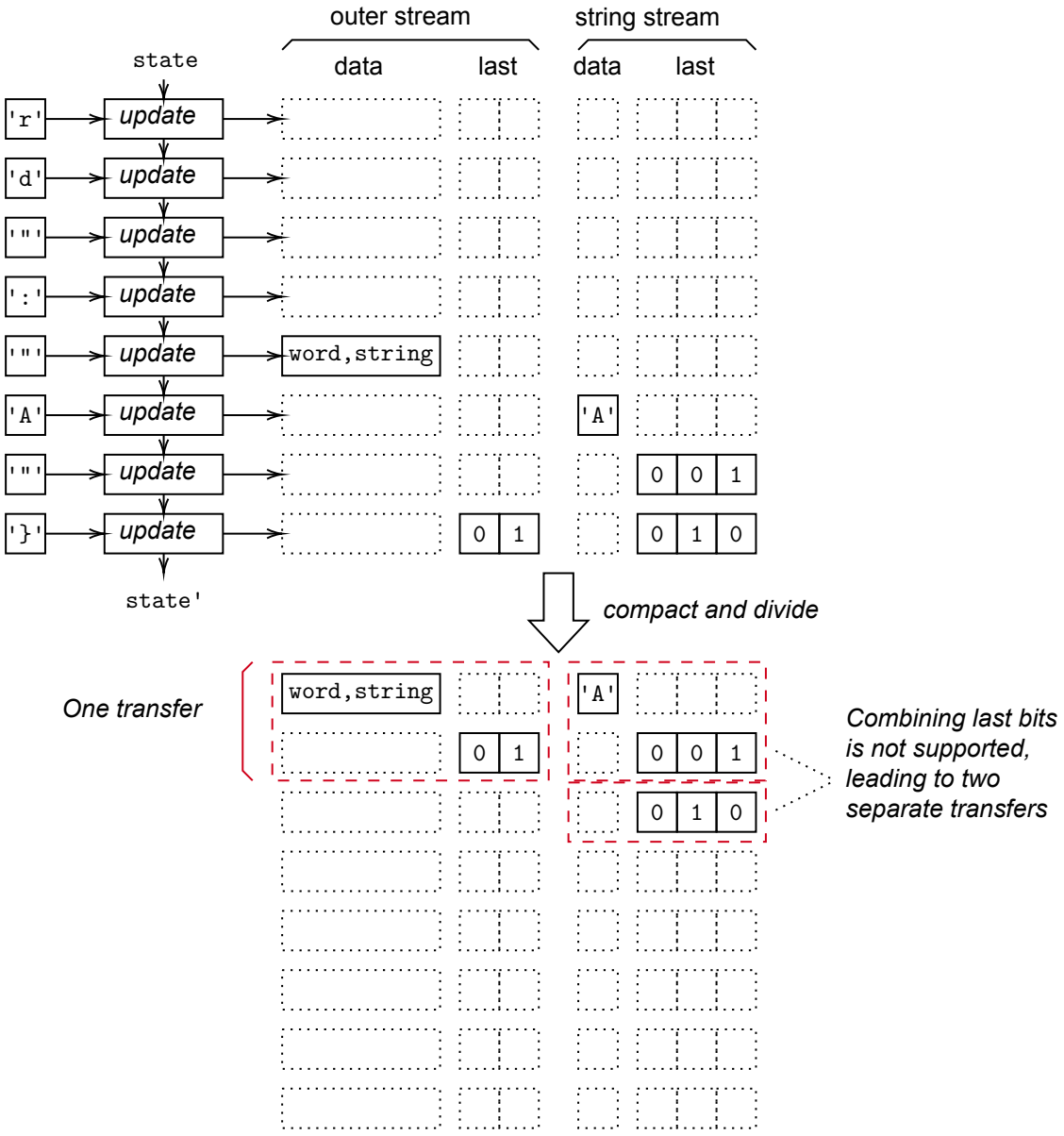

Figure 19.2: Data processing of the `jsonParser` module.

20

Discussion

All parts of Tydi-Clash were succesfully implemented, and the example implementation demonstrated a subset of the functionality.

In writing the example implementation, it quickly became apparent that the Tydi specification suffers from the existance of several edge cases that complicate interface design. For example, at the complexity level used (5), last bits may arrive with or after data, complicating the implementation of modules that only operate on the innermost sequence. This is not necessarily because of bad interface design, but rather the inevitable result of the complexity supported by Tydi. In fact, reducing the complexity of the interface would increase source complexity, only moving the problem.

The encapsulating data types acted as a double-edged sword. Though it certainly made it harder to use the interface incorrectly, by preventing incorrect data access, it did little to ease correct usages. Particularly, the intertwined nature of case detection and data access made the code harder to read. A cleaner implementation style using `Maybe` for case detection was thought up to improve code quality (see Appendix D), although much of the low-level work remains. To ease implementation of simple modules, directly exposing the inner signals may be sensible.

Here, it is important to remember the goals of Tydi: it is not merely to make complex streams possible, but to provide a standard that can be built upon. One could figure streaming out on a case-to-case basis, and for each case come up with a different, more optimal solution, but this would induce a lot of repeated effort. Currently, the basis of Tydi is there, but there is a shortcomning in the lack of standard implementations of common functions.

Some functionality could be achieved by integrating Tydi-Clash with Clash Protocols, allowing the use of Clash Protocols machinery to connect streamlets. Tydi-Clash can also be combined with Tydi-lang to improve the high level design process - it may even be possible to combine the two. Additionally, Tydi-Clash needs a collection of transformations and wrappers that take care of common functions and edge cases at streamlet implementation level. For example, a wrapper might take care of lower dimension information for a module that only operates on innermost data sequences, and much of the complexity in Subsection 19.3.2 could be reduced by using high complexity output, coupled with a lane and complexity reducer module. Such methods could greatly reduce the implementation effort, and reduce the codebase size.

Finally, though it initially had some bugs, debugging the design was much easier due to the Shockwaves integration. Without the integration, it would have been nigh impossible to find the bugs in the desig. The largest signal used while debugging was 5117 bits long, and would not have had any value to the programmer. This clearly demonstrates the value of Shockwaves.

21

Summary

Tydi is a streaming specification allowing associated multi-lane streams of multi-dimensional typed data. Tydi-Clash is a Clash library for the full Tydi specification.

Tydi types have isomorphism with Haskell types. Conversion functions are in place to switch between Haskell data types and different Tydi representations.

Tydi's physical streams could be implemented directly, but the direct access to undefined data is frowned upon. Therefore, the internal signals are primarily accessible through algebraic and abstract data types. These types ensure that only data that is defined according to the Tydi specifications can be accessed or written.

Several tests are provided to ensure the phsyical streams meet behavioral specifications. These were designed to support empty sequences, which so far have not been explicitly defined in the Tydi specification.

Logical streams are a bit different. Instead of flattening and splitting up the structure, the synthesis produces a hierarchical structure, as this allows the stream bundle to be manipulated more freely.

Shockwaves instances were implemented for Tydi-Clash types to cleanly display Tydi streams and data types in the waveform viewer. This was essential while debugging a simple example implementation.

The example implementation unfortunately highlighted several drawbacks in using Tydi-Clash. Due to its complexity, Tydi is inherently prone to having edge cases that are difficult to cleanly resolve. The encapsulation of data into safer data types makes it harder, though not impossible, to separate case detection from data operations. Great care is required to keep the code clean. On the other hand, the data encapsulation did protect the physical streams from incorrect usage, demonstrating the value of capturing interface relations in types. Standardized modules, potentially also defined at Tydi specification level, could resolve some of the implementation complexity. Finally, integration with Clash Protocols seems very valuable, and would be a logical next step in the development. This might be combined with integration with Tydi-lang.

Finally, the development of Tydi-Clash drew attention to the currently mostly overlooked effects of empty sequences on several interface requirements. Tydi-Clash fully supports these empty sequences, but further analysis at the Tydi level is encouraged.

Part III

Closure

22

Conclusion

This work details two systems meant to help raise abstraction of different aspects of the Haskell-based HDL Clash: Shockwaves for typed waveforms, and Tydi-Clash for Tydi streams.

Shockwaves is a typed waveform viewing system for Clash. Unlike some other typed waveform solutions, Shockwaves performs translation of values in the Haskell runtime and stores the results in a lookup table, rather than translating them in the viewer itself. This gives the programmer full control over the representation of data from inside the code, without needing to update Shockwaves itself, but also introduces potential performance issues for large projects. Shockwaves has a replacement module for Clash's tracing library, which is fully functional, as well as code for translating values in designs compiled to Verilog. The latter was partially implemented, but required compiler modification beyond the scope of the project to fully work. Nevertheless, the system is functional and valuable, and the approach used demonstrates the benefits of giving the programmer control over the waveform representation from within their designs. Next steps would include performance optimizations, as well as compiler modifications to support type annotation in compiled designs.

The Tydi-Clash implementation includes Tydi data types, physical streams, and logical streams, as well as modules for type conversion and behavioral verification. Correct usage of the physical streams is encouraged through encapsulated interface data types, which provide a new look on Tydi streams. Furthermore, the behavioral verifications address the possibility of empty sequences - something not covered by the current Tydi documentation. Since Tydi's complexity and inherent edge cases can make it difficult to write correct code despite these protective measures, the library would benefit greatly from a set of standardized modules that take care of some of the details when using Tydi. The value of a such collection would not be confined to Tydi-Clash, but extend to Tydi as a whole. Additionally, more work will need to be done to integrate Tydi with Clash's Protocols library as well as Tydi-lang to integrate it into both the Clash and Tydi ecosystems.

References

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

[2] S. Marlow *et al.*, “Haskell 2010 language report,” 2010. [Online]. Available: https://www.haskell.org/definition/haskell2010.pdf.

[3] C. P. R. Baaij, “Digital circuits in CλaSH: Functional specifications and type-directed synthesis,” Ph.D. dissertation, University of Twente, Enschede, Jan. 2015.

[4] QBayLogic. “Clash: A modern, functional, hardware description language.” (2025), [Online]. Available: https://clash-lang.org/ (visited on 03/27/2025).

[5] F. Skarman, L. Klemmer, K. Laeufer, and O. Gustafsson, *Surfer 0.2.0*, version 0.2.0, Jun. 2024. DOI: 10.5281/zenodo.11447243.

[6] F. Skarman, G. Sörnäs, and O. Gustafsson, *Spade 0.12.0*, Jan. 2025. DOI: 10.5281/zenodo.14623297.

[7] R. Meloni, H. P. Hofstee, and Z. Al-Ars, “Tywaves: A typed waveform viewer for chisel,” in *2024 IEEE Nordic Circuits and Systems Conference (NorCAS)*, 2024, pp. 1–6. DOI: 10.1109/NorCAS64408.2024.10752465.

[8] J. Bachrach, H. Vo, B. Richards, *et al.*, “Chisel: Constructing hardware in a Scala embedded language,” in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 1216–1225. DOI: 10.1145/2228360.2228584.

[9] LF Projects LLC. “Chisel: Software-defined hardware.” (2025), [Online]. Available: https://www.chisel-lang.org/ (visited on 03/27/2025).

[10] A. Izraelevitz, J. Koenig, P. Li, *et al.*, “Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations,” in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2017, pp. 209–216.

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

[12] A. Lenharth and C. Lattner, “CIRCT: Lifting hardware development out of the 20th century,” 2021. [Online]. Available: https://llvm.org/devmtg/2021-11/slides/2021-CIRCT-LiftingHardwareDevOutOfThe20thCentury.pdf (visited on 12/08/2024).

[13] J. Decaluwe, “MyHDL: A python-based hardware description language,” *Linux Journal*, vol. 2004, p. 5, Jan. 2004.

[14] *GTKWave*. [Online]. Available: https://gtkwave.sourceforge.net/.

[15] QBayLogic, *Clash.Signal.Trace*, version 1.8.2, 2025. [Online]. Available: https://hackage-content.haskell.org/package/clash-prelude-1.8.2/candidate/docs/Clash-Signal-Trace.html.

[16] W. Snyder, P. Wasson, D. Galbi, *et al.*, *Verilator*, version 5.001, 2022. [Online]. Available: https://verilator.org.

[17] C. Shao and R. Eisenberg. “Haskell dark arts, part I: Importing hidden values.” (2021), [Online]. Available: https://www.tweag.io/blog/2021-01-07-haskell-dark-arts-part-i/ (visited on 01/16/2024).

[18] S. Meyers, “The most important design guideline?” *IEEE Software*, vol. 21, no. 4, pp. 14–16, 2004. DOI: 10.1109/MS.2004.29.

[19] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, “Lime: A Java-compatible and synthesizable language for heterogeneous architectures,” *SIGPLAN Not.*, vol. 45, no. 10, pp. 89–108, Oct. 2010. DOI: 10.1145/1932682.1869469.

[20] J. Thomas, P. Hanrahan, and M. Zaharia, “Fleet: A framework for massively parallel streaming on fpgas,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 639–651. DOI: 10.1145/3373376.3378495.

[21] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, “A survey on the evolution of stream processing systems,” *The VLDB Journal*, vol. 33, no. 2, pp. 507–541, 2024.

[22] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, “A survey of distributed data stream processing frameworks,” *IEEE Access*, vol. 7, pp. 154 300–154 316, 2019.

[23] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, “Optimus: Efficient realization of streaming applications on FPGAs,” in *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2008, pp. 41–50. DOI: 10.1145/1450095.1450105.

[24] Apache Software Foundation, *Apache Kafka*, 2024. [Online]. Available: https://kafka.apache.org/1.

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

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

[27] J. Pontes, R. Soares, E. Carvalho, F. Moraes, and N. Calazans, “SCAFFI: An intrachip FPGA asynchronous interface based on hard macros,” in *2007 25th International Conference on Computer Design*, IEEE, 2007, pp. 541–546.

[28] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, “LLHD: A multi-level intermediate representation for hardware description languages,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 258–271.

[29] J. W. Cutler, C. Watson, E. Nkurumeh, *et al.*, “Stream Types,” *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024. DOI: 10.1145/3656434.

[30] J. Peltenburg, J. Van Straten, M. Brobbel, Z. Al-Ars, and H. P. Hofstee, “Tydi: An open specification for complex data structures over hardware streams,” *IEEE Micro*, vol. 40, no. 4, pp. 120–130, 2020. DOI: 10.1109/MM.2020.2996373.

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

[32] Y. Tian, M. Reukers, Z. Al-Ars, *et al.*, “Tydi-lang: A language for typed streaming hardware,” in *Proceedings of the SC ’23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 521–529. DOI: 10.1145/3624062.3624539.

[33] C. Cromjongh, Y. Tian, P. Hofstee, and Z. Al-Ars, “Tydi-Chisel: Collaborative and interface-driven data-streaming accelerators,” English, in *Proceedings of the 2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*, IEEE, 2023. DOI: 10.1109/NorCAS58970.2023.10305451.

[34] A. Gundry, A. Löh, A. Rybczak, and O. Grenrus, *Optics*, version 0.4.2.1, May 2024. [Online]. Available: https://hackage.haskell.org/package/optics-0.4.2.1.

[35] QBayLogic, *Clash Protocols*, version 0.1, 2025. [Online]. Available: https://github.com/clash-lang/clash-protocols (visited on 03/03/2025).

Appendix

A

Repositories

A.1. Shockwaves

Main Shockwaves repository: `https://github.com/The-Redstar/shockwaves`

Surfer fork: `https://gitlab.com/The-Redstar/surfer-shockwaves`

Clash compiler fork: `https://github.com/The-Redstar/clash-shockwaves-compiler`

A.2. Tydi-Clash

Tydi-Clash: `https://github.com/The-Redstar/tydi-clash`

Example system: `https://github.com/The-Redstar/tydi-clash-demo`

B

JSON Format

This appendix contains examples of the JSON lookup tables for Shockwaves.

B.1. Signal Type Table

```
{
    "DUT.counter.x": "Clash.Sized.Internal.Unsigned 5",
    "DUT.count": "GHC.Types.Bool"
}
```

B.2. Value Translation Table

In the actual data, `"kind": "Normal"` and `"subfield":[]` are omitted, and all fields have been shortened to a single letter (for example, `"Compound"` becomes `"C"`).

```

{
  "(Clash.Sized.BitVector.Bit,Clash.Sized.BitVector.Bit)": [
    { "Compound": {
        "subfields": [
          ["0","String"],
          ["1","String"]
        ]
      }
    },
    {
      "10": {
        "val": {"String": "(1,0)"},
        "kind": "Normal",
        "subfields": [
          {
            "name": "0",
            "result": {
              "val": {"String": "1"},
              "kind": "Normal",
              "subfields": []
            }
          },
          {
            "name": "1",
            "result": {
              "val": {"String": "0"},
              "kind": "Normal",
              "subfields": []
            }
          }
```

```
        ]
      },
      "00": {
        "val": {"String": "(0,0)"},
        "kind": "Normal",
        "subfields": [
          {
            "name": "0",
            "result": {
              "val": {"String": "0"},
              "kind": "Normal",
              "subfields": []
            }
          },
          {
            "name": "1",
            "result": {
              "val": {"String": "0"},
              "kind": "Normal",
              "subfields": []
            }
          }
        ]
      },
      "01": {
        "val": {"String": "(0,1)"},
        "kind": "Normal",
        "subfields": [
          {
            "name": "0",
            "result": {
              "val": {"String": "0"},
              "kind": "Normal",
              "subfields": []
            }
          },
          {
            "name": "1",
            "result": {
              "val": {"String": "1"},
              "kind": "Normal",
              "subfields": []
            }
          }
        ]
      }
    }
  ]
}
```

C

Shockwaves Supported Types

The following types are implemented without subsignals:
`()`, `Bool`, `Char`, `Bit`, `Int`, `Int8`, `Int16`, `Int32`, `Int64`, `Ordering`, `Word`, `Word8`, `Word16`, `Word32`, `Word64`, `CUShort`, `Signed`, `Unsigned`, `Double`, `Float`, `Fixed`, `SNat`, `Proxy`

Types with standard derived implementations:
tuples of up to 15 elements, `Complex`, `Down`, `Identity`, `Const`, `Product`, `Sum`, `Compose`

These types have custom implementations of `Display` and/or `Split`:
`Maybe`, `Either`, `Vec`, `BitVector`, `RTree`, `Zeroing`, `Wrapping`, `Saturating`, `Overflowing`, `Erroring`

D

Tydi Coding Styles

This appendix provides three different ways to write Tydi-Clash code.

D.1. Direct Control

The most direct way is to control all signals directly. While in many cases this results in fairly simple code, there is no protection against incorrect usage. Case detection can be separated from data handling by defining boolean values.

Since the inner signals of physical streams are private, this method of interfacing is currently not actually possible in Tydi-Clash.

```
-- case detection using booleans
dataTransfer = inStream.valid && inStream.strb
lastTransfer = inStream.valid && not inStream.strb && any inStream.last
outTransfer = outStream.valid && outReady==Ready

-- output dependent on case
outStream = PStream
            { valid = dataTransfer || lastTransfer
            , dat = out
            , last = inStream.last
            }

inReady = convertReady outReady

(state', out) = go state inStream.dat
newState = if outTransfer then state'
                          else state
```

D.2. Intertwined Data

With the data encapsulated in special types, the data only becomes accessible with the data detection. Unfortunately, mixing case detection and data generation results in less readable code.

```
-- combined case detection and output generation
(outStream,inReady,state',outTransfer) = case inStream of
  Transfer tf | Just d <- getDataSliced tf -> ( fromSliced (Just out)
                                                           (getLast tf)
                                                           ()
                                              , readyConvert outReady
                                              , state''
                                              , outStream==Ready)
    where (state'',out) = statemachine state d
  Transfer tf | any $ getLast tf           -> ( fromSliced Nothing
                                                           (getLast tf)
                                                           ()
```

```
                                                , readyConvert outReady
                                                , state
                                                , outStream==Ready)
  _                                           -> (NoTransfer,Ready,state,False)

newState = if outTransfer then state'
                          else state
```

D.3. Case Detection Using Maybe

Instead of booleans, cases can be stored as `Maybe` values containing their relevant data. This style was thought up as a solution to the implementation difficulties while writing the example implementation.

```
-- case detection
dataTransfer = case inStream of
  Transfer tf | Just d <- getDataSliced tf -> Just ( statemachine state d
                                                        , getLast tf )
  _                                        -> Nothing
lastTransfer = case inStream of
  Transfer tf | l <- getLast tf, any l -> Just l
  _                                    -> Nothing

-- output generation
inReady = if | Just _    <- dataTransfer -> readyConvert outReady
             | Just last <- lastTransfer -> readyConvert outReady
             | otherwise                 -> Ready

outStream = if | Just ((_,d'),last) <- dataTransfer -> fromSliced (Just d')
                                                                  last
                                                                  ()
               | Just last          <- lastTransfer -> fromSliced Nothing
                                                                  last
                                                                  ()
               | otherwise                          -> Ready

newState = if | ((s',_),_) <- dataTransfer, outReady==Ready -> s'
              | otherwise                                   -> state
```