Tywaves

A Typed Waveform Viewer for Chisel HDL with typed circuit components and Tydi streams

Raffaele Meloni

Technische Universiteit Delft

Tywaves

A Typed Waveform Viewer for Chisel HDL with typed circuit components and Tydi streams

by

Raffaele Meloni

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday August 2, 2024 at 11:00 AM.

Student number:	5845106	
Project duration:	November 20, 2023 – August 2, 2024	
Thesis committee:	Dr. Zaid Al-Ars	TUDelft, supervisor
	Prof. Dr. H. Peter Hofstee	TU Delft, IBM
	Dr. Soham Chakraborty	TU Delft

TUDelft

Abstract

Modern hardware design languages introduce high-level constructs to considerably improve design capabilities. The adoption of software language features and strong type systems contribute to expressing complex designs with cleaner and more robust code, facilitating the translation of software algorithms for hardware accelerators. Despite these advantages, their mainstream adoption is often discouraged by the lack of debugging tools that support the same level of abstraction. The usage of standard tools implies inspecting automatically generated RTL code, dissimilar from the source, which leads to a convoluted debugging experience.

This thesis presents Tywaves, a new kind of type-centered waveform viewer for the Chisel hardware language with typed circuit components and Tydi streams. Contributions to both the Chisel library and CIRCT MLIR compiler are described. Type information for debugging is extracted from the source language and linked with the target Verilog. A frontend waveform viewer is updated with the functionality to interpret and associate type information with values dumped from an RTL simulator and reconstruct the source language view. Finally, a Chisel API has been implemented to enable Tywaves from a high-level testbench.

The Tywaves project aims to enhance the debugging experience of modern hardware languages by reducing the gap between the source code and waveforms. It provides a new type-centered debugging format that helps to bring the same level of abstraction of new languages into waveform viewers.

Preface

When this project was proposed to me, I was excited to create something new that could help and simplify the testing of hardware code. In my past experience, during my bachelor's study, I created software APIs to facilitate hardware development. In the same way, creating a new type of waveform viewer for modern hardware languages can make life easier for designers, reducing the effort and speeding up debugging. This gave me strong motivation to start and complete the project.

The implementation of this project for Tydi and the Chisel hardware language was not trivial. It required an in-depth study of large tools implemented in different programming languages. This experience helped me to improve my skills in contributing to large projects, integrating new functionalities, and combining multiple tools together. In addition, it introduced me for the first time to the interesting field of programming languages and compilers.

I would like to thank my supervisors who introduced me to the topic and supported me during this journey, and the team of Chisel, CIRCT and Surfer who gave me useful advice on how to contribute to their projects and also gave me the opportunity to present my project at their public meetings. Furthermore, I would like to thank people who previously worked on Tydi and Chisel for their contributions which revealed fundamentals for Tywaves.

Finally, I am grateful to my girlfriend, my friends in Delft and back home, and my family for all the support, good times and fun moments they have given me over the past two years.

I hope Tywaves will have a long future and encourage people to try it and continue its development to further improve the debugging of modern HDLs.

Raffaele Meloni
Delft, July 2024

Contents

1 | Introduction

The current hardware design domain has been facing an explosion of new hardware description languages (HDLs) and hardware generator languages (HGLs) that introduce new levels of abstraction to reduce development time and design effort. This trend has marked the beginning of a new golden age for hardware languages [53, 62].

Despite this progress, less work has been done on improving the debugging infrastructure. To keep compatibility with pre-existing designs and synthesis tools, most of these languages usually target classic HDLs. This often requires designers to make use of old testing frameworks that reflect an automatically generated code, usually dissimilar from the source, leading to a convoluted debug experience. Thus, hardware testing and debugging tools should be at the same level as languages to benefit from the new levels of abstraction.

1.1. Context

The end of Moore's law has led to increased research into new specialized chips and accelerators to overcome the performance limitations of general-purpose processors [32]. Over the last decade, this claim has been confirmed by a greater number of publications in favor of specialized hardware accelerators. As stated by Mahmoud [47], the rising trend of using new specialized processors, observed between 2013 and 2021, will continue over the next years, making their development more prevalent.

Hardware accelerators usually integrate into existing software applications to support and speed up portions of their computation. Often their development process starts from a software version of the algorithm which is subsequently adapted for execution on an ASIC or FPGA. Nevertheless, the effort for porting, writing, and testing a specific functionality on HDL is higher than the one needed in conventional programming languages. This challenge slows the mainstream adoption of HDLs. It is much more evident while translating complex typed data structures into a hardware representation, due to the lack of the same declarative flexibility of classic HDLs.

The current progress in hardware development and debugging can be summarized as follows:

- **Software abstractions in hardware.** Adoption of common software functionalities in hardware languages.
- **Type system.** Definition of a strong type system in hardware for complex data structures and easier translation from software.
- **Intra-cycle debugging.** Porting of the software breakpoint debugger concept for hardware simulation.
- **Waveforms.** Inspection of multiple signal values over time to highlight inter-cycle signal changes.

Each point is discussed in more detail below.

Software abstractions in hardware. New hardware languages introduce common software programming features in digital design such as polymorphism, functional programming, and custom complex types definitions to enable reusable and cleaner hardware code [35, 58]. Modern HDLs such as Chisel, Clash, and Spade [10, 11, 55] showed that they are able to maintain similar performances to Verilog and VHDL while adding abstraction value.

Type system. Although these languages allow the definition of custom types to encode complex data structures of fixed size, there is no inherent support to represent and use dynamically sized data structures in hardware. Exchanging, describing, and representing variable-sized data in hardware implies mapping them onto streams. As a consequence, an additional challenge arises when translating them into corresponding hardware. This challenge is addressed by the Tydi (Typed dataflow interface) specification and Tydi-lang [49, 61, 63] defining a protocol and a domain-specific language (DSL) with abstractions to specify typed streaming hardware minimizing the programmer's effort and the number of lines of code required. Tydi-Chisel [23] ported the Tydi constructs and concept into Chisel, allowing Tydi-lang to use the same testing infrastructure of Chisel. With Tydi, Chisel will not only offer the strongly typed system that is described in [16, *section: Data Types Overview*] but also abstractions for exchanging those typed data over streams.

Intra-cycle debugging. Recently, progress has been made in improving the debugging experience with new HDLs. In 2022, Zhang et al. presented a novel breakpoint debugger (HGDB) for intra-cycle reverse debugging [65, 66] bringing the same concept used in software programs also for assessing hardware behavior. However, in an HDL, variables updated in different parts of the code change value in the same clock cycle (considered the time unit of registers), while in HGDB the value changes are shown sequentially. Hence, a visualization of value changes of multiple variables over time is more suitable for inter-cycle inspection, like the one implemented by waveform viewers.

Waveforms. Waveform viewers are widely used programs able to read values dumped by hardware simulators and provide a graphical representation of signal changes in a circuit. This visual aspect is crucial in understanding interactions, correlations, and behavior over time in a digital design. The timing aspect becomes particularly relevant when developers seek to inspect performance metrics, such as the number of cycles needed to complete a task, throughput, and signal delays. Yet, viewers may encounter a challenge in maintaining abstraction when dealing with modern HDLs, especially in the open-source domain. These visualizers work tightly coupled with the low-level RTL simulators used for the languages. Therefore, if the simulators lack support for the language or the compilers do not produce the proper debug information, rendering a waveform representation reflecting the source language is nearly impossible.

This thesis reports on improving the debug experience for modern HDLs/HGLs by implementing a new type-based waveform viewer for typed circuit components and Tydi streams, called Tywaves [48]. The custom type system introduced by Tydi-lang but also by Chisel and other languages is lost once they are compiled into classic HDLs and then simulated. The Tywaves project aims to reduce the gap between the source and the waveforms displayed by increasing the level of abstraction of waveform debugging with a type-based visualization. Developers continue to choose classic HDLs over new languages because of the lagging support of the required features in testing tools. A new type of viewer can help to speed up the mainstream adoption of any of these languages. Targeting all existing languages at once is impossible due to their differing characteristics. Therefore, to increase the chance of having a bigger impact on reusability, the Tywaves project integrates with Chisel and CIRCT [19] a novel compiler that targets multiple languages with the same infrastructure.

1.2. Challenges

The primary objective of Tywaves is to provide a new debugging format that displays custom data-type information next to values of signals in a waveform viewer and customize value representation based on these types. However, associating type information in the context where HGLs are compiled and simulated with current testing tools for classic HDLs might not be trivial. In fact, transformations and optimizations are performed during the compilation, leading to the loss of abstraction. Reverse engineering of a compiled output to reconstruct the source view is difficult due to the complexity of these operations within the compiler.

Furthermore, the compilation and simulation pipelines implemented for a language likely involve multiple tools where the language is transformed into various intermediate representations. The challenge here is related to the complexity of modifying multiple programs and creating consistency between them without breaking other functionalities.

Moreover, the same classic HDL is the target of multiple compilers and languages. For a viewer that receives information only about the target, foreseeing and knowing any generic type is challenging. Therefore, it should receive information about how to access or build the original view.

1.3. Problem statement and research question

This thesis addresses a crucial problem in the current hardware development field. Nowadays multiple modern HDLs and HGLs have been created to reduce design time and effort. However, there has been limited progress in developing new debugging tools, especially within the open-source domain.

Simulating these languages comprises compilations to traditional HDLs which might lead to the loss of abstraction, resulting in an awkward debugging experience. Although some novel debugging methods have been proposed to address this issue, waveform viewers remain one of the preferred tools for hardware designers but the progress in their development remains insufficient in the open-source community. Therefore, this thesis focuses on creating a type-based waveform viewer to increase the abstraction level to debug hardware in a similar manner to debugging software by including a strong type system. In order to address this issue, the main research question to motivate the work is:

Main research question: *How can a type-based waveform viewer be effectively developed to raise the abstraction level while debugging modern HDLs and HGLs, specifically reducing the gap between the waveform visualization and source code?*

Following the challenges addressed within this thesis, we address the following subtopics:

1. How can types be associated with values output from simulations and the source language be reconstructed from a compiled output not matching the source?

2. What are the necessary steps to improve multiple compilers and tools involved in simulation of modern HDLs in order to create debug information for a waveform viewer?

3. How can types be displayed in a graphical user interface and how show the source view?

4. How should types and, more in general, debug information be generated, encoded, and propagated throughout the compilation and simulation pipelines?

5. Due to the multiplicity and diversity of modern HDLs, can a method be found and defined to support multiple languages or, possibly, to extend support easily by re-utilizing much of the infrastructure so as to have a greater impact in the open-source community?

1.4. Contributions

The Tywaves project involves multiple contributions aimed at enhancing the hardware debugging experience. This thesis focuses specifically on implementing a type-based waveform viewer for Chisel HDL with typed circuit components and Tydi streams.

First, an overview of Tydi-Chisel, a description of Chisel simulation flow, Chisel-CIRCT compilation, and a definition of typed circuits are provided (Chapter 2). The overview also describes how Tydi is integrated into the Chisel compilation pipeline, how the user interacts with the tools, and the progress until now to highlight missing components necessary to make Tywaves feasible.

Second, updates in the Chisel library are introduced to generate extra debug information and pass it to the underlying CIRCT compiler (Section 4.1). This information includes the source language types, eventual parameters, and other info like enum variants value mapping. In addition, it contributes to creating a methodology for passing generic debug information from Chisel to CIRCT.

Third, CIRCT is updated to read and process this new information (Section 4.2). The previous debug flow permitted to emitting a debug file for reconstructing the view of the intermediate representation generated by Chisel which showed only hierarchies. Tywaves adds the functionality to enable linking Chisel types information to Verilog and a new debug file format for representing it.

Moreover, support for Chisel is added to the Surfer waveform viewer (Section 5.1). A Rust library, tywaves-rs, is implemented to process the debug information emitted from CIRCT and convert it into a more generic and efficient data structure. In fact, following the concept of CIRCT and Surfer to support extensibility for other languages, interfacing the viewer with the debug info through an intermediate data structure allows to add support for new inputs without changing the rest of the program.

Finally, a Chisel API is defined to easily use the updated simulation flow and tools (Section 5.2). High-level Chisel Scala simulators are implemented to hide the complexity of the underlying tools and prevent the users from the tedious process of manually calling multiple programs to generate the debug information, like the compilers and waveform viewer.

Example of Tywaves output

To give an illustration of the overall contribution, Figure 1.1 shows an example of the final output of Tywaves. The figure mainly highlights how the types in the circuit are preserved from the source code.

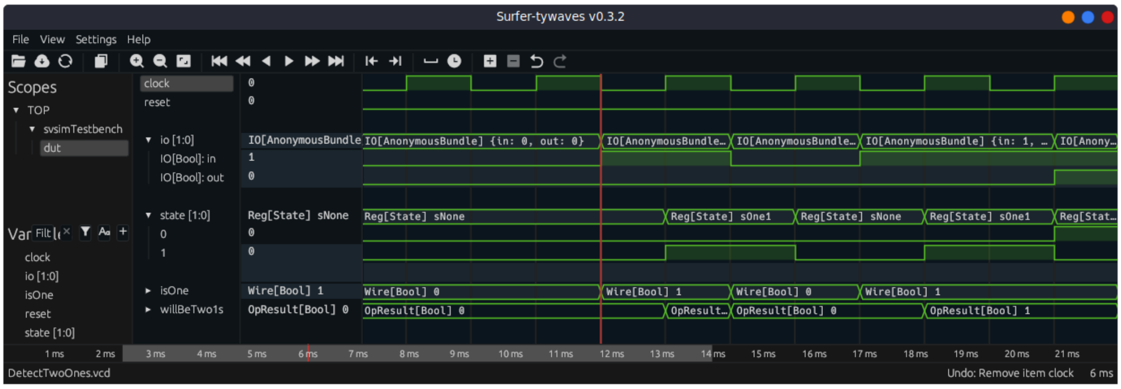


Figure 1.1: Output example of Tywaves

1.5. Outline

The remainder of this thesis is organized as follows:

- First, Chapter 2 provides a background description of Tydi-Chisel, how the Chisel compilation and simulation flow works, and an overview of the current debug information generated by the Chisel-CIRCT compiler. A definition of typed circuits in Chisel is provided and a brief overview of how modern HDL/HGL simulators and waveforms can be used is presented. Related works on both traditional and contemporary testing and debugging tools are presented, with a particular emphasis on tools that are closely related to Chisel.

- Chapter 3 analyses the implementation requirements and alternatives for generating waveforms for typed Chisel circuits. Then, it describes the main components involved, namely a backend for debug information generation and a frontend for waveform visualization.

- Changes to the Chisel and CIRCT compilation pipelines are discussed in Chapter 4. The changes are necessary for reconstructing type information from an external tool. In addition, justifications on the debug format emitted for the viewer and means employed for exchanging information between Chisel and CIRCT are given.

- Chapter 5 reports the updates in the waveform viewer, the tywaves-rs library, and an API created for easy adoption of the features into existing chisel simulations. Furthermore, a rationale for the specific open-source viewer extended is provided.

- Chapter 6 shows an evaluation of Tywaves results. Comparisons between the standard waveform visualization and Tywaves output are given. The focus of this chapter is to illustrate whether Tywaves have or not an added value while debugging Chisel and Tydi-Chisel circuits.

- Finally, Chapter 7 summarizes the work done in the thesis and outlines the future work.

2 | Background

This chapter provides a general background on the technologies and tools involved in the project. The Tydi-Chisel library integrates within the Chisel compilation and simulation flow. ChiselSim provides a high-level interface between the low-level RTL simulators and the CIRCT compiler. Moreover, CIRCT implements initial support to reconstruct the FIRRTL-level view from the compiled Verilog code. Tydi and Chisel introduce new paradigms for typed circuit components. Finally, current HGL/HDL simulators can be used to extract trace files for waveform viewers.

Figure 2.1 summarizes how Tydi [23, 49] is related to Chisel, the CIRCT backend compiler, and ChiselSim simulation infrastructures [11, 14, 19]. Within the workflow, the initial interaction of the user occurs with Tydi-lang [61], where they define the data types involved in the circuits, describe the streaming interface characteristics, and declare the components involved along with stream connections. This code is translated into Chisel boilerplate code, enabling users to add the actual functionality of the components which are subsequently elaborated to FIRRTL [33] and later to Verilog by CIRCT. Finally, the user interacts with ChiselSim to simulate the circuit through a specific testbench, interfacing with a low-level Verilog simulator like Verilator [57].

The remainder of the chapter goes into a more detailed description of the parts depicted in the figure. The Tydi specification and its integration into Chisel are first introduced in Section 2.1. Second, Chisel and particularly the simulation flow are described in Section 2.2, while Section 2.3 presents the CIRCT compiler. Then, a definition of circuits with typed components accompanied by a concrete example is provided in Section 2.4. Finally, Section 2.5 gives a brief overview of HDL simulators and waveform viewers. and Section 2.6 presents related work.

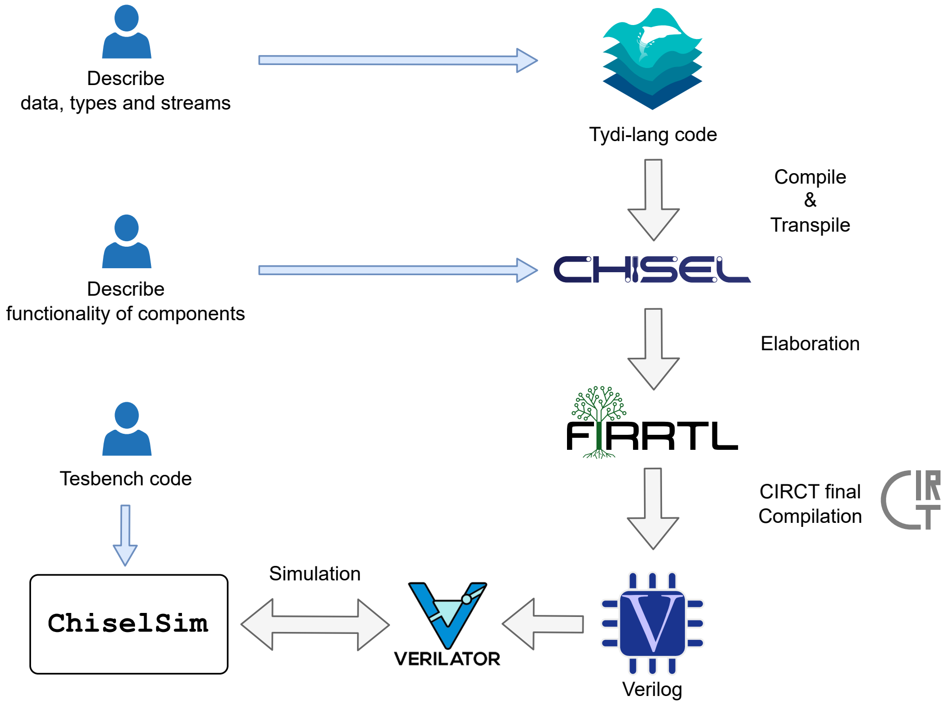


Figure 2.1: Relationship between Tydi, Chisel, and user interaction with the compiling/simulation infrastructure

2.1. Tydi, Tydi-Chisel and Tydi-lang

Tydi [49] is an open specification that defines a methodology for representing typed dynamically sized data structures and a protocol for exchanging them over hardware streams. Tydi-Chisel [21, 23] is a Scala library that ports the protocol into the Chisel language and implements abstractions for using the typed hardware streams introduced by the specification. Thus, users who want to use Tydi as a communication interface can use the library similarly to any other existing component in Chisel. On the other hand, Tydi-lang [61] is a DSL with a specific syntax for simplifying the definition of tydi components, data types, stream types, and connections between streams. Listing 2.1 and 2.2 report an example retrieved from [23] implementing a pipeline to filter a stream of numbers with timestamps and emit some statistics. Numbers, timestamps, and statistics (min, max, sum, and avg) are defined with Tydi hardware types that matches software data types, closing the gap between the two implementations. Thus, the code snippets clearly show the advantages of Tydi for defining hardware components from a software implementation. Before Tydi-Chisel, the Tydi toolchain comprised TIL (Tydi Intermediate Language) [51] to emit VHDL instead of Chisel. However, using the Chisel language as Tydi backend was revealed as a better option to further simplify the development and help the mainstream adoption of Tydi streams.

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

Listing 2.1: Example of Spark code. From *Tydi-Chisel: Collaborative and Interface-Driven Data-Streaming Accelerators* [23, *Listing 1*]

```
#### package pack0;
UInt_64_t = Bit(64); // UInt<64>
SInt_64_t = Bit(64); // SInt<64>

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

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

// Rest of the code: specification of
    streams and components
// ...
```

Listing 2.2: Corresponding Tydi-lang code snippet. From *Tydi-Chisel: Collaborative and Interface-Driven Data-Streaming Accelerators* [23, *Listing 2*]

When Tydi-lang is used together with a transpiler [22], it generates Chisel boilerplate code containing all definitions, types, and interfaces translated one-to-one from the Tydi-lang code, further simplifying user's work (see Appendix A). Tydi-lang is not an HDL, so it does not have the expressivity to define logical behavior, therefore the functionality of components should be added through Chisel as introduced by Figure 2.1.

The combination of these components contributed to the definition of the flow shown in Figure 2.1 which uses Chisel as development backend. Relying on Chisel as a host HDL for the library allows Tydi to exploit a full existing development infrastructure, testing pipeline, and easier integration in other Chisel projects. As a consequence, any other tool that utilizes or tests Chisel can be also used for circuits operating with Tydi, including waveform viewers. Tywaves would not need any extra information from Tydi-lang because Chisel has full expressivity for custom data types by extending its basic types as reported in [37, *p. 37*] and [11, *Sections 2.1-3*].

2.2. Chisel and simulation flow

Chisel [11] (Constructing Hardware In a Scala Embedded Language) is a new but already broadly adopted HDL that brings object-oriented and functional programming, and type-safety to hardware design. Chisel falls into the category of HGLs, namely programs that generate a classic HDL. Specifically, it is a Scala library internally implementing a compiler frontend that transforms Chisel into FIRRTL [33] (Flexible Internal Representation for RTL) which is consequently compiled to Verilog by a FIRRTL compiler. From Figure 2.1 we can observe that Chisel exploits existing RTL simulators for behavioral simulation of components. The compilation to Verilog also ensures compatibility with other vendors and open-source tools.

Although simulating the output Verilog is certainly possible, writing testbench in such a low-level language would not be convenient in an HGL context, since all the abstractions introduced by Chisel would be lost. In addition, even the testbench compatibility would not be guaranteed because the compiled code may change unexpectedly when the source is updated. Therefore, the Chisel library provides high-level simulation components to write tests and run simulations directly from Scala.

Initially, Chisel used ChiselTest [40], an external testing framework based on the Scala FIRRTL Compiler (SFC) [8], to support the simulation of circuits. However, starting from the release of Chisel 5, the team switched to the CIRCT project [19, 44] for FIRRTL compilation. This made supporting ChiselTest for future versions difficult and a new, officially maintained and actively integrated component (ChiselSim), has been built as a testing framework replacement [14].

Figure 2.2 shows a generalized flow diagram of the approach for testing HGLs through existing simulators and writing tests directly in the source language rather than the target HDL. Such a pipeline is used in ChiselTest but partially implemented in ChiselSim. Yet, unlike the former, the new simulation framework does not provide abstractions to emit traces for waveform viewers[1]. The flow follows the classic rules of any other HDL simulation with the exception that there are two levels of simulators rather than one, plus a compiler from HGL to HDL. Unfortunately, this simulation flow does not enable source-level waveform debugging. The simulation is executed by a low-level Verilog simulator which outputs a trace file reflecting its representation. The consequence is the absence of information to reconstruct a representation that matches what the Chisel's user writes.

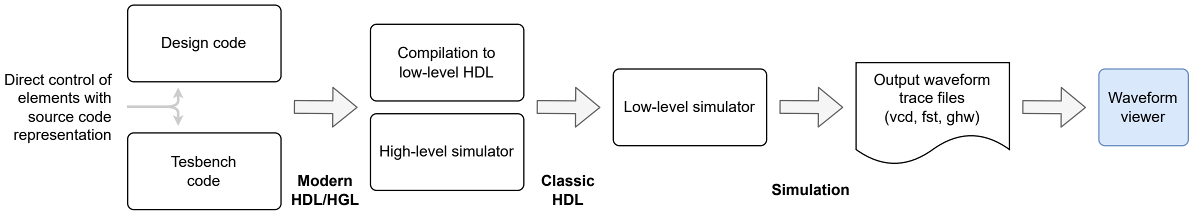

Design code
Direct control of elements with source code representation
Tesbench code
Modern HDL/HGL
Compilation to low-level HDL
High-level simulator
Classic HDL
Low-level simulator
Output waveform trace files (vcd, fst, ghw)
Simulation
Waveform viewer

Figure 2.2: Generalized HGL simulation approach used in ChiselTest and partially used in the current version of Chiselsim

2.3. CIRCT

The previous section mentions that Chisel has embraced the CIRCT project to execute the final compilation phase. This compiler infrastructure applies MLIR and LLVM development methodologies [42, 43] to the domain of hardware design tools, including a shared and reusable compiling infrastructure with interoperability with other HDLs and HLS [26, *p. 8*]. This removes the need to create a compiler for every new language and also opens opportunities to combine different input languages in a single target output and reuse all the tools compatible with that [19].

MLIR provides a methodology to abstract individual languages handled by the compiler through unique namespaces called *dialects*, referred to also as intermediate representations (IR) of MLIR. Each dialect groups together MLIR operations, attributes, and types representing characteristics of languages or information in a compiler step. Through the compilation, dialects are lowered and transformed into common dialects independent from source languages, enabling a progressive and more organized compilation towards an optimized target output.

Operations in MLIR (Ops) are the semantic unit of an intermediate representation and model every component of a language. An operation can receive *operands* and output *results* which store *values* and associated *types* used to represent data at runtime and compile-time. Operation results can be passed as operands of other operations, enabling the representation of data and control flow properties of a language. This follows the idea of static single assignment (SSA) [24] which is used in MLIR. Moreover, additional compile-time information related to a language component can be represented and manipulated through *attributes* (later referred to also as *fields* of MLIR Ops) of dialect operations. Finally, dialects and operations can be defined using a declarative syntax provided by TableGen in LLVM [45]. This latter is a DSL built to speed up and simplify the definition of components in LLVM

[1] *ChiselSim improvements to close the gap to ChiselTest.* Issue `#4203` and partially solved in `#4201` in [16]

compilers since, with few lines of code, it allows the automatic generation of all the common functionalities needed to manipulate operations like C++ classes, getters, and setters.

CIRCT achieves Chisel compilation through `firtool` which parses a FIRRTL input to a specific *firrtl* dialect, performs lowering steps and transformations into language-independent CIRCT dialects (referred to as *core* dialects), and emits output formats for standard EDA tools as shown in Figure 2.3. In addition, CIRCT offers custom options to select different optimization levels or the output target. *Lowering* is a semantic technique used in compilers that consists of rewriting complex constructs in terms of simpler ones in the same language. For instance, `structs` in terms of their fields or software `while` loops and `foreach` statements as `for` loops. This technique is essential for optimizations in a compiler.

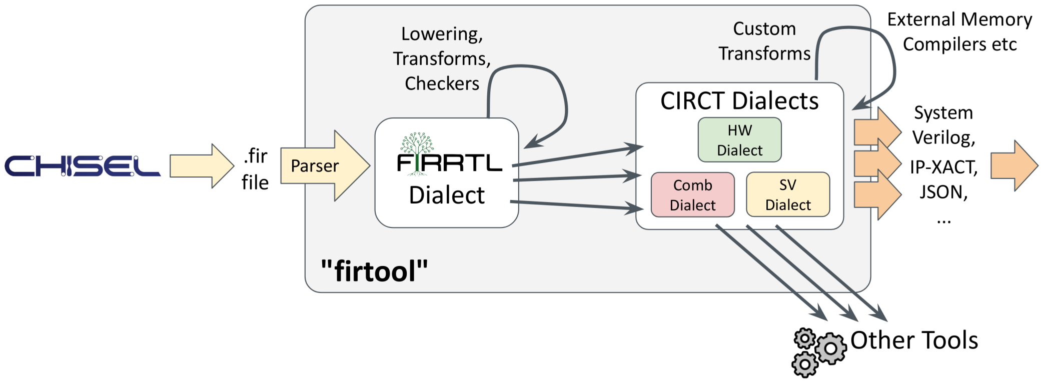


Figure 2.3: `firtool` - a CIRCT implementation of FIRRTL compiler in replacement to SFC. From *"CIRCT: Lifting hardware development out of the 20th century"* [44]

2.3.1. CIRCT debug dialect

Figure 2.3 highlights the difference between core and input-language dialects; nonetheless, it does not provide a complete overview of all dialects involved in CIRCT. In addition to the ones in the figure, firtool integrates a *debug* dialect providing MLIR Ops to track the relationship between values, types, and hierarchy of an input language and the target compiled output [18]. This allows external testing tools to reconstruct a source language view for more straightforward debugging. It is part of the CIRCT dialects and it can be used with any other dialect when proper MLIR transformation passes are created.

The debug dialect implements operations that firtool uses to generate debug information (DI) from the rest of the compilation process. Specifically, it is composed of four operations that can be combined to represent and reconstruct the original hierarchical view of variables that might be flattened or even optimized out during the compilation:

- *dbg.variable*: represents a named variable declared in the source code.
- *dbg.struct* and *dbg.array*: create aggregates from lists of named and indexed values respectively. Preserve the hierarchical representation of the variable independently from the optimizations used for the final output.
- *dbg.scope*: define a scope in the source code. It creates a namespace to group variables and other scopes, including module instances.

Following the concept of producing standard formats for external tools, firtool processes the DI of the debug dialect and emits a new open standard file format called HGLDD (Hardware Generator Language Debug Data). It is based on JSON and has been defined in collaboration with Synopsys for an alpha version of VCS/Verdi, as presented by the Chisel's team during the Latch-Up 2024 conference [29, 38]. HGLDD is an open standard that can be used, extended, and emitted also by non-CIRCT, non-Chisel projects and other tools, including a waveform viewer.

Even though the debug dialect operations permit the reconstruction of an input language source view, they can store only values, types, and hierarchies of a dialect effectively defined in CIRCT, namely FIRRTL. Chisel interfaces with CIRCT through FIRRTL and, for this reason, information about it is not available to the debug dialect. Hence, the debug dialect and HGLDD lack inherent support for Chisel features and abstractions that are lost during the lowering and transformation phases. Furthermore, the latter is a recent file format (2024) and still misses formal documentation, making its adoption in the open-source community more difficult.

For these reasons, a new file format or an updated and documented HGLDD might be required to include types or any other source language information. On the other hand, the independence from a specific language suggests that using and improving the debug dialect might lead to possible extensions of the Tywaves project to other HGLs rather than only Chisel.

2.3.2. Integration of CIRCT in ChiselSim

ChiselSim is composed of two core components:

- **Svsim**: a low-level library for compiling and interfacing SystemVerilog simulators from Scala code, providing maximal portability and backend-independent test harness and simulation.

- **PeekPoke API**: a set of basic operations built in Scala to control a testbench. It includes methods to advance the simulation, assert signals with expected values, and peek and poke IO signals.

Figure 2.4 shows a high-level diagram of ChiselSim and the integration with user code, Verilog simulators, and firtool. The `PeekPoke` API and `svsim` can be combined in high-level simulators to abstract and hide details of low-level simulation, expose the control of IO signals, and enable more advanced functionalities. This combination translates into a complete testing framework with a straightforward and intuitive user front-end.

Internally, `svsim` invokes `ChiselStage` (a Scala class implementing *"a User-facing API for invoking Chisel"* [37, *p. 17*]) to prepare and analyze Chisel code for FIRRTL-Verilog compilation and it later passes the circuit emitted by firtool to the hardware simulator backend (currently supported, Verilator and VCS [57, 59]).

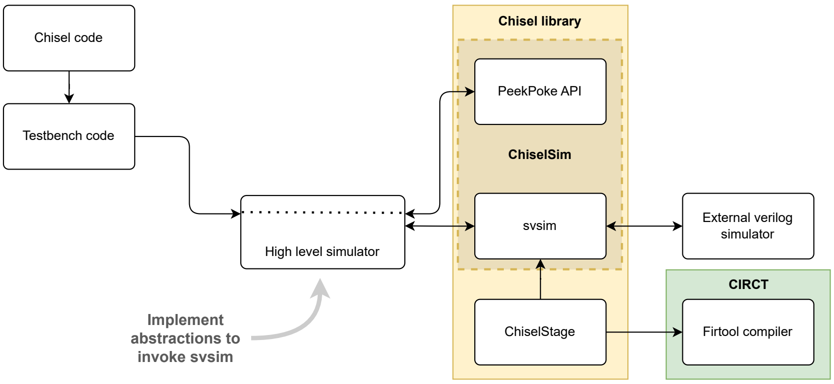


Figure 2.4: The current ChiselSim architecture and interface to Firtool

The translation to FIRRTL and firtool invocation within `ChiselStage` is internally performed by several mathematical transformations ($f : A \to A'$), referred to as phases, whose interface is reported in Listing 2.3. Each `Phase` implements an annotation-transformation mechanism shown in Figure 2.5, where each annotation is a Scala case class storing some information about the circuit. These phases are handled by the `PhaseManager` class which is responsible for storing, executing, and checking the dependencies and order between multiple elaboration steps. Therefore, `ChiselStage` provides the FIRRTL output to CIRCT and collects a SystemVerilog design available to `svsim`.

```
/** A polymorphic mathematical transform (f: A ->
    A')  */
trait TransformLike[A] extends LazyLogging {
  /** A mathematical transform on some type */
  def transform(a: A): A
}

trait Phase extends TransformLike[AnnotationSeq]
    with DependencyAPI[Phase] {
  lazy val name: String = this.getClass.getName
}
```

Listing 2.3: `Phase` trait in Scala

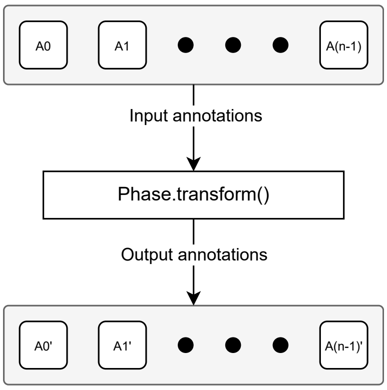


Figure 2.5: Mathematical transformation implemented by a phase: $f : A \rightarrow A'$

2.4. Typed hardware circuits

The concept of a typed waveform viewer originated from the type system introduced by Tydi, Tydi-lang, and Tydi-Chisel library (`tydilib`). As mentioned in Section 2.1 and Appendix A, Chisel is able to fully express the definitions of typed components at a high level of abstraction and to implement the behavior of the design.

The `tydilib` hides all the implementation details of the specification. In a typed streaming system, the data structures defined are internally ”serialized” at the bit-level for the actual transmission, but thanks to the abstractions this aspect is hidden from the user. Therefore, from their perspective, components are transferring complex data structures as whole ”packets” and not ordinary bits or bit vectors. Moreover, in the context of Chisel, a user likely chooses to use or define a new specific type for a signal rather than another one because they want to manipulate and associate that type of information with the actual value of the variable. Likewise, for the same reason, developers declare variables as `char`, `string`, `int`, arrays, etc. When software variables are inspected in debuggers, they expect that the value is associated with the respective type, i.e. a `char`, an array of 8 `float` or `int` rather than a mere byte, 4 words of 8 bytes. If software debuggers would show simple bytes, all the abstraction introduced in favor of assembly would be lost.

The same concept of typed elements can be extended from Tydi to Chisel through typed circuits where each signal has a high-level type rather than being a bit vector, and with Tydi this is further strengthened. Yet, types cover an important role in Chisel as well, Chisel includes internal, ”standard”, and user-defined types. Considering this type-system [16, *section: Data Types Overview*], displaying Chisel types alongside signal hierarchies in waveforms is essential for bridging the gap between writing and debugging code. Furthermore, it helps to distinguish signals clearly. Different types, such as `Bool`, `SInt`, `Enum`-like or a `Bundle` and user-defined, have distinct characteristics, which makes their differentiation even more important.

This section illustrates an example of a typed circuit in Chisel (Subsection 2.4.1). Specifically, it helps to understand the difference between hardware and Scala/Chisel types and what the main characteristics of a signal that Tywaves addresses are. Since FIRRTL has been implemented as an IR for Chisel it shares some concepts on types, such as the aggregated signals or signed and unsigned integers. However, Chisel and FIRRTL are distinct, and Subsection 2.4.2 clarifies how they differ.

2.4.1. Example of a typed-circuit in Chisel

While writing a Chisel circuit, each signal has a clearly defined Scala type. Listing 2.4 and Table 2.1 help to understand what a typed circuit in Chisel is. In this kind of circuit there are three main characteristics of a signal to consider:

1. The hardware type, like IO;
2. The Scala/Chisel type, consisting of the class name and constructor’s parameters;
3. The value representation and range of values.

The example uses both standard types like `Bool`, `UInt` and `SInt`, and more advanced types such as vectors, and user-defined enums and bundles. All of them have different properties that do not exist in Verilog. Therefore, once transformed, it becomes tricky to distinguish their characteristics due to the fewer types available and optimizations performed by the compiler. For instance, a Chisel operation between an `SInt` and `UInt` is not allowed without explicit casting. However, this is not the case after the Verilog transformation since they are both translated as `logic`. Additionally, aggregates might be flattened into parallel signals and their inspection as a grouped variable might be very difficult. Finally, enumeration variants might be translated as bit values and, as a consequence, their named value lost.

Next to signals, also modules represent reusable circuit blocks with their Scala types. In the example considered, there are two module instances: both are instantiated using `MyModule` but their types are different since their instances use different parameters.

```
// User-defined aggregate with named fields
class MyBundle(val n: Int) extends Bundle {
   val a: Bool = Bool()
   val b: SInt = SInt(n.W)
   val nested  = new Bundle { val x: UInt = UInt(8.W) }
   val v       = Vec(n, SInt(32.W))
}

// User-defined enum
class MyState extends ChiselEnum {
    val IDLE, A, B, C, Other = Value
}

// User-defined module
class MyModule(val n1: Int, val n2: Int, val n3: Int) extends Module {
    val inBundle  : MyBundle = IO(Input(new MyBundle(n= n1)))
    val wireBundle: MyBundle = Wire(new MyBundle(n2))
    val outBundle : MYBundle = IO(Output(new MyBundle(n3)))

    val state = RegInit(MyState.IDLE))

    // ...
}

class TopCircuit extends Module {
    val mod1 = Module(new MyModule(10, 7, 9))
    val mod2 = Module(new MyModule(1, 1, 1))
    // ...
}
```

Listing 2.4: An example of Chisel typed hardware circuit

Variable	Type	Values
inBundle	`IO[MyBundle(n: 10)]`	Aggregated named signals in `MyBundle`
wireBundle	`Wire[MyBundle(n: 7)]`	Aggregated named signals in `MyBundle`
outBundle	`IO[MyBundle(n: 9)]`	Aggregated named signals in `MyBundle`
inBundle.a	`IO[Bool]`	$[0; 1]$
inBundle.b	`IO[SInt<10>]`	$[-2^9; +2^9 - 1]$
inBundle.nested	`IO[AnonymousBundle]`	Hierarchical representation with `x` as field
inBundle.nested.x	`IO[UInt<8>]`	$[0; 2^8 - 1]$
inBundle.v	`IO[SInt<32>[10]]`	Indexed `SInt` elements
state	`MyState`	`IDLE, A, B, C, Other`
...	...	...

Table 2.1: Types and possible values of signals defined in Listing 2.4

In conclusion, a Chisel type is the type of the Scala variable representing a signal or a module in the source Chisel code. This Scala type includes the name of the class and the names, types and values of the constructor parameters. This information, together with signal hierarchies, is missing from the standard waveform viewers.

2.4.2. Differences between Chisel and FIRRTL types

FIRRTL is the intermediate representation used to bridge Chisel with Verilog [33]. The main objective of an IR is to reduce the complexity of the compilation process from a high-level to a low-level language. As stated by Chow in [17], an IR enables splitting the compilation into multiple phases and the portability of the sample compiler flow to multiple languages. Therefore, an IR should be both closer to the target output than the source language and more generic. At the same time, it may preserve some common characteristics of the input languages.

In the case of FIRRTL, it preserves some characteristics of Chisel such as some of the basic types that can be obtained from all the others [9]. Table 2.2 reports the respective FIRRTL types of some variables from the example of Subsection 2.4.2. It is worth noticing that not all the Chisel types have a corresponding type in the IR. This is because some Chisel types are defined to give an additional level of abstraction to the user which is not needed when compiling to Verilog since some of them represent special cases of more generic types. For instance, `Bool` can be represented as `UInt<1>` or a user-defined bundle as an anonymous bundle in FIRRTL. Even though this simplifies the optimizations and transformations made by the compiler, it does not fully preserve the source language information which would be an added value when debugging Chisel code.

Chisel Type	Firrtl type
`Bool`	`UInt<1>`
`UInt(n.W)`	`UInt<n>`
`SInt(n.W)`	`SInt<n>`
`new Bundle {val x: UInt = UInt(8.W)}`	`{x: UInt<8>}`
`Vec(n, SInt(32.W))`	`SInt<32>[n]`
`class MyState extends ChiselEnum { ... }`	`UInt<2>`
`MyBundle(3)`	`{a: UInt<1>, b: SInt<3>, ...}`
...	

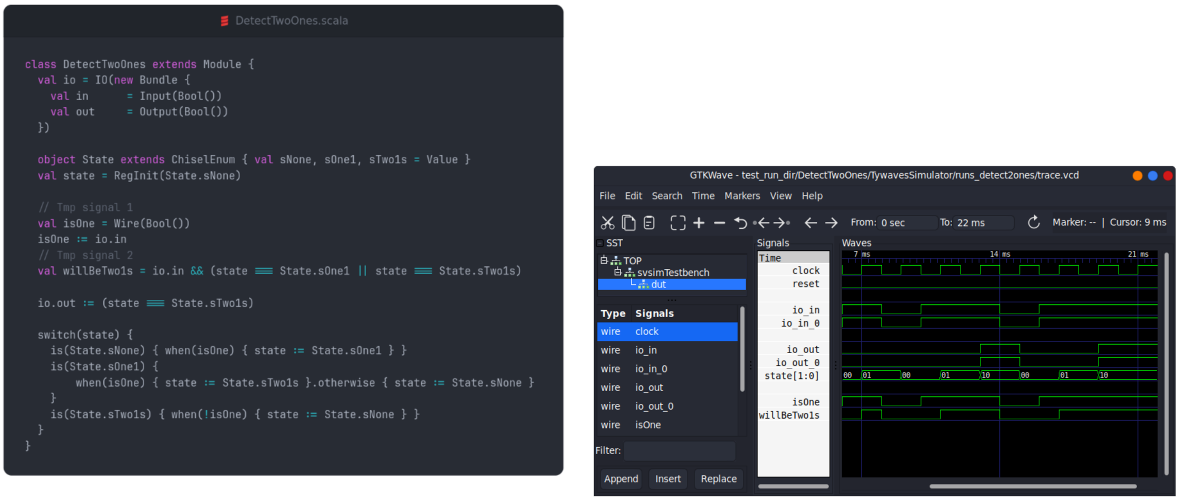
Table 2.2: Examples of Chisel types transformed to corresponding FIRRTL types [9]

2.5. HDL/HGL simulators and waveform viewers

Waveform viewers load values dumped into a trace file generated through an RTL simulation. Widely used and supported standard open source formats are VCD (Value Change Dump), GHW (GHDL Waveform), and FST (Fast Signal Trace) [6, 12, 30]. All of these open standards can only handle Verilog, SystemVerilog, and VHDL types. Thus, in the domain of modern HDLs, they focus on storing the signal values from a low-level simulation rather than how they should be linked to the source language.

Figure 2.2 showed how modern HDLs and HGLs can be simulated and tested with existing tools for classic HDLs. The main advantage is to provide immediate support and compatibility for simulation without any additional work. As mentioned earlier, Chisel is compiled into Verilog, simulated with low-level tools like Verilator, and the output trace is directly passed to the viewer without any extra information. However, this method does not offer source-level debugging but rather target-HDL-level debugging. The simple example in Figure 2.6 highlights a first issue with the current flow. The waveforms do not correspond to the hierarchies, names, and types of the corresponding circuit (Figure 2.6a). For instance, `io` is not displayed as a grouped signal, some unwanted additional artifacts such as `io_in_0` are generated, and the values of the `State` enum are converted into bit vectors. Furthermore, there is no type distinction between the signals (Figure 2.6b, all of them are represented as simple wires although it is clear to see that `io` is a `IO[Bundle]`, `io.in` is an `IO[Bool]` and `state` is a `Reg[State]` in the source code.

A slight improvement could be observed by using ChiselTest to output FST instead of VCD. It distinguishes wires from ports and also the direction of signals is shown. However, the main problem still remains.



DetectTwoOnes.scala

```scala
class DetectTwoOnes extends Module {
  val io = IO(new Bundle {
    val in      = Input(Bool())
    val out     = Output(Bool())
  })

  object State extends ChiselEnum { val sNone, sOne1, sTwo1s = Value }
  val state = RegInit(State.sNone)

  // Tmp signal 1
  val isOne = Wire(Bool())
  isOne := io.in
  // Tmp signal 2
  val willBeTwo1s = io.in && (state === State.sOne1 || state === State.sTwo1s)

  io.out := (state === State.sTwo1s)

  switch(state) {
    is(State.sNone) { when(isOne) { state := State.sOne1 } }
    is(State.sOne1) {
        when(isOne) { state := State.sTwo1s }.otherwise { state := State.sNone }
    }
    is(State.sTwo1s) { when(!isOne) { state := State.sNone } }
  }
}
```

GTKWave - test_run_dir/DetectTwoOnes/TywavesSimulator/runs_detect2ones/trace.vcd

From: 0 sec To: 22 ms Marker: -- | Cursor: 9 ms

Type	Signals
wire	clock
wire	io_in
wire	io_in_0
wire	io_out
wire	io_out_0
wire	isOne

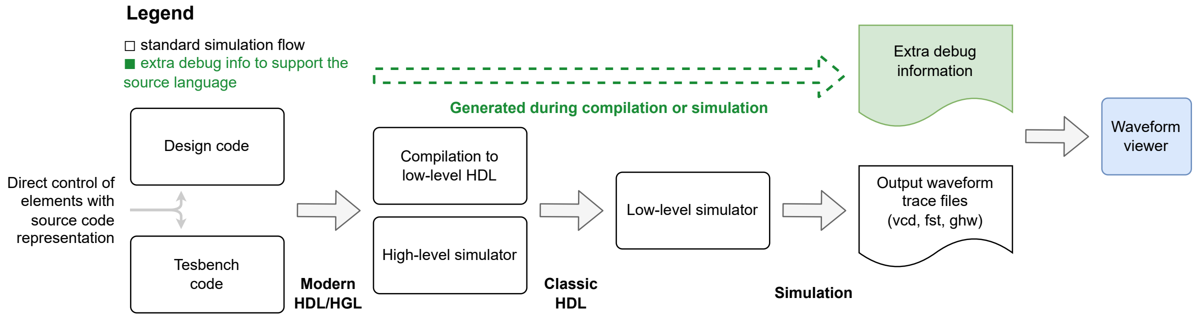
(a) Example of an FSM in Chisel with enums and grouped signals

(b) Screenshot of GTKWave with only VCD from simulation uploaded

Figure 2.6: Example of Chisel code and waveforms with the simulation approach of Figure 2.2

A representation that reflects a compiled code would not be a big deal for simple cases like the one above. The example provided implements a trivial circuit with few signals, so it is straightforward to understand what the signal inspected refers to. However, with larger and more complex designs like CPUs or hardware accelerators, the way signals are represented might lead to a tedious debugging experience due to the explosion of signals in the compiled Verilog. As mentioned in Section 2.3 the Chisel and MLIR compilers perform a set of transformations from Chisel-FIRRTL to Verilog. Among these, optimizations may be executed and temporary signals created, carrying out an omission of declared signals or an explosion of additional artifacts.

To overcome this, the generation of waveforms supporting a high-level language should follow the steps depicted in Figure 2.7 where extra debug information is generated during compilation or simulation and passed to a viewer accordingly. This approach would allow external programs to process and associate source HGL debug information with values dumped into normal trace files. For instance, Synopsys followed this flow to realize a first alpha version of a viewer for HGL/Chisel [38] through the HGLDD introduced in Subsection 2.3.1.



Legend
□ standard simulation flow
■ extra debug info to support the source language
Generated during compilation or simulation
Extra debug information
Waveform viewer
Design code
Direct control of elements with source code representation
Tesbench code
Modern HDL/HGL
Compilation to low-level HDL
High-level simulator
Classic HDL
Low-level simulator
Output waveform trace files (vcd, fst, ghw)
Simulation

Figure 2.7: A proposed generalized HGL simulation flow to enable source-level waveforms

2.6. Related work

In the context of hardware development, we can distinguish between tools for testing and debugging. The former involves the simulation of designs and verification of whether components operate as intended, while debugging is the process of identifying and removing eventual errors. Both techniques can be enhanced and contribute to the general improvement of hardware development with new tools that make the process faster and more user-friendly.

A notable tool for testing HDLs is cocotb [7] which helps productivity by allowing the definition of testbench and functional RTL verification in Python and providing a single interface platform for commonly used RTL simulators. However, cocotb works with classic HDLs like Verilog and VHDL. As mentioned in Section 2.2, ChiselTest does a similar job for Chisel and ChiselVerify [27, 28] is a library that enables functional verification from Scala.

WAL (Waveform Analysis Language) [36] is a DSL that can be used for the automated analysis of waveforms, allowing to perform arbitrary actions on waveforms such as the comparisons of values output from a simulation with pre-computed values.

Furthermore, there are various waveform viewers available from open-source options like GTKWave [31] to proprietary software such as Virtuoso, Modelsim, Vivado, and Verdi [13, 54, 60, 64]. These tools provide waveform visualization by supporting the most common trace formats such as VCD, FST, FSDB etc. Additionally, vendor programs provide more comprehensive functionalities in a unique platform like formal verification and synthesis tools.

Recent developments have introduced three new tools that are particularly relevant to the study. These include the HGDB debugger [65, 66], the Surfer viewer [56] for the Spade language [55], and an alpha version of Verdi. The following subsections delve deeper into these tools.

2.6.1. Hardware Generator Debugger

The Hardware Generator Debugger (HGDB) is a debugging tool that presents a novel methodology for testing the behavior of hardware languages. It allows to perform breakpoint debugging for inspecting intra-cycle signal values. Breakpoints enable inspection of value changes within the same clock cycle and this helps to identify possible errors that may occur during an intermediate calculation. Waveforms would not show the values calculated by intermediate operations during the same clock cycle but rather the final value that is assigned to signals. Nevertheless, HGDB brings source-level debugging by discussing a different problem. When designers need to inspect multiple signals at once, check value changes between cycles, and also show the behavior of a circuit graphically, waveform viewers still reveal a better choice.

2.6.2. Surfer for the Spade language

As introduced in Chapter 1 multiple new hardware languages have been created to improve the development experience. Among them, the Spade language [55] stands out from the others because its syntax and high-level constructs are natively supported by Surfer. Skarman et al. created this viewer because they found that other standard waveform viewers like GTKWave do not provide a good experience for debugging modern HDLs. For instance, Spade has an `Option<T>` type enum whose values are `Some(T)` and `None` similar to Rust. With these high-level values, the actual bit representation of a signal is hidden from designers; therefore, it would be nice for a viewer to display the values of a simulation in their source representation rather than their raw bits. Surfer does this specifically for Spade and supports also Verilog and VHDL. Although it is built for extensibility it does not offer native support for other HDLs yet and it misses the type information from the view.

2.6.3. Synopsys Verdi HGL viewer

Synopsys Verdi is a vendor waveform viewer with many advanced functionalities and it is tightly coupled with the VCS simulator for functional verification. As anticipated in Subsection 2.3.1, Synopsys started working on a new alpha version of Verdi and cooperated with CIRCT to define the HGLDD debug format. By combining HGLDD and the FSDB trace file, Verdi manages to support an HGL/HDL waveform visualization. Since HGLDD provides information only about FIRRTL [33] as mentioned earlier, in this prototype, Verdi provides a waveform visualization at a FIRRTL level rather than Chisel.

2.6.4. Conclusion

Recent years have seen the development of tools to improve hardware design with old and new HDLs. New testing frameworks and tools have been created to facilitate the creation of testbench, the usage of multiple simulators with a common interface API, and the employment of functional verification at a higher level of abstraction.

Other debugging tools have been proposed to bring source-level debugging for HGLs and modern HDLs. Most significantly, HDGD introduces a novel breakpoint debugging technique as an alternative to waveform viewers for intra-cycle inspection instead of inter-cycle evaluation. Surfer presents a modern waveform viewer for another language and classic HDLs but it does not provide a type-based visualization and native support for other HDLs yet. Whereas, Verdi is a proprietary program that provides some support for Chisel through FIRRTL waveform visualization.

On the other hand, Tywaves aims to provide a new kind of open-source waveform visualization by showing source-level data types and support for Chisel and Tydi-Chisel. The integration within CIRCT would imply a new methodology for propagating information through an existing toolchain and it suggests that would contribute to the extension of other languages.

3 | Generating waveforms for typed Chisel circuits

Chisel inherently lacks a proper waveform visualization. First, as noted from Section 2.2, the current version of ChiselSim does not have native abstractions to emit any trace file or control the firtool compiler[1]. Controlling the available options directly from a testbench would relieve users from the tedious task of calling several programs for a single simulation, such as invoking firtool to get debug information. In addition, customizing the compilation mode for a simulation would give control over the optimization level of the circuit. Second, although ChiselTest is a more stable and mature framework than ChiselSim, it still enables a waveform visualization of an output Verilog view rather than Chisel. Third, in CIRCT the debug dialect and HGLDD include only information related to FIRRTL as described in Subsection 2.3.1. This means that in the current version, HGLDD cannot be used to have a pure Chisel-level visualization due to the differences with FIRRTL presented in Subsection 2.4.2. Finally, although some work for supporting Chisel has been done for VCS and Verdi, the community still misses an open-source waveform viewer for Tydi, Chisel, and CIRCT.

This chapter analyzes the implementation requirements and possible designs, proposing an update to the current simulation flow of Chisel to answer the main research question.

3.1. Implementation requirements

Since Tydi is integrated within Chisel and its types and constructs are defined in a library for Chisel (Section 2.1), creating a typed waveform viewer is equivalent to creating a tool able to render signal values of Chisel and CIRCT circuits while preserving the source code representation. This main goal leads to the following implementation requirements which follow the generic flow of Figure 2.7:

1. Collect high-level debug information (DI) from Chisel;
2. Elaborate the DI in order to associate it with the values in a trace from a simulator. In other words, an association with the signals and modules output of a Chisel-CIRCT compilation;
3. Emit a well-defined file format that an external viewer can read and associate easily to a trace file to properly display the waveforms;
4. Define compatibility and portability with ChiselSim.

3.2. Implementation alternatives

In this section, two alternative approaches to achieving the established goals are briefly presented and discussed. The first consists of creating a library external to Chisel, while the second solution proposes an integration within the Chisel-CIRCT compilation pipeline described in Section 2.2 and 2.3.

External tool. An *external tool* would need to get intermediate data structures used to represent the different intermediate representations throughout the pipeline, namely Chisel, FIRRTL, and Verilog. Then, it would require joining the signals together and identifying an ID that is unique for different signals but shared between the representations of the same signal in the three IRs. Finally, it should use this information to emit a symbol table for an external viewer.

Unfortunately, `ChiselStage` [37, *p. 17*] does not publicly expose the internal IRs used to represent Chisel and FIRRTL in the `PhaseManager`, making them subject to potential changes. This highlights the main drawback of an external approach because there would not be the opportunity to guarantee a unique ID for each signal, the maintainability over time would be even more difficult, and the implementation complicated and less reliable.

Integrated solution. A *solution integrated* into the official Chisel-CIRCT repositories, instead, would not be subject to the problem of finding a unique ID and, at the same time, would improve the

[1] Verified until Chisel v6.4.0.

maintainability. The idea is to generate the information directly when the transformation from one IR to another is done, namely when Chisel is translated to FIRRTL during the `PhaseManager`, pass this information to firtool, process it and emit the file for the viewer, either an updated HGLDD or a new format.

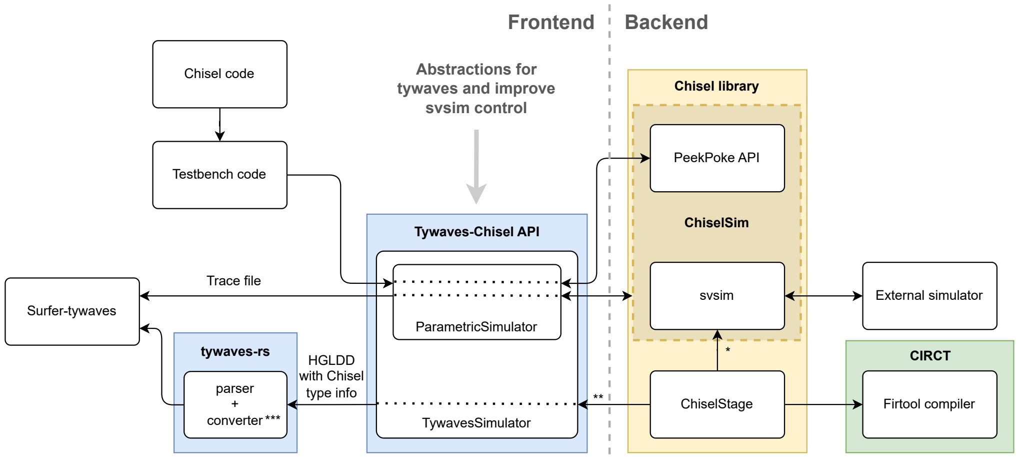
The two solutions present complementary advantages and disadvantages. Although the first one would give total control of the choices and how to represent the code internally and no need to deeply study how the Chisel internals work, the challenge for maintainability and defining a unique ID suggests a great preference in favor of an integrated solution.

3.3. Updated simulation flow

As addressed at the beginning of this chapter, the current flow of ChiselSim has two main issues: the debug information that firtool can generate is not available to testbench users and the whole pipeline lacks inherent support for generating and propagating the type information presented in Section 2.4. Therefore, both Chisel-CIRCT compilation stages and ChiselSim necessitate an update to implement Tywaves and provide necessary information to an external viewer. The diagram in Figure 3.1 provides an overview of the software architecture of the project and shows how the Tywaves frontend integrates into the Chisel compilation pipeline to achieve the objectives established. Specifically, this implies an update to the simulation flow and its implementation requires:

- An update to `ChiselStage` and `PhaseManager` to collect the type information during the transformation phase of Chisel to FIRRTL and propagate it to firtool.
- Processing the extra information in firtool through the debug dialect. Handling these new features led to an update of both the dialect and HGLDD format.
- Choosing and extending an existing waveform viewer to render a Chisel representation. This also includes developing a library for parsing HGLDD and converting it into a manageable and efficient data structure capable of linking abstract representations with an attached trace file.
- An API implementing two high-level PeekPoke simulators with more advanced functionalities (see Appendix B for an overview of high-level functionalities in the current Chiselsim) to control firtool, emit trace files, and link all the components.

The implementation can be subsequently divided into two main parts: a *backend* comprising of necessary changes in the Chisel-CIRCT compilation pipeline and a *frontend* that processes the information emitted by the backend to properly render the signals and gives, at the same time, an intuitive interface to the user. The next chapters explain how it is achieved. Section D.1 presents a sequence diagram that highlights the inner steps of how Chisel invokes and cooperates with the other tools.



* ChiselStage with **standard** elaboration settings
** ChiselStage with the **extra debug info** enabled for Tywaves
*** HGLDD is parsed and converted to an internal data structure for Surfer

Figure 3.1: Tywaves software architecture

4 | Tywaves backend

This chapter details the implementation of the backend introduced in Figure 3.1. The generation of extra debug information integrates within the official Chisel compilation pipeline as explained in Chapter 3. The backend is divided into two components: the Chisel library and the CIRCT firtool compiler.

First, the methodology for collecting the type information of Chisel elements is presented. It includes the addition of a special transformation phase to `ChiselStage` that is executed only in a new debug elaboration mode and the definition of a FIRRTL annotation [19, *'firrtl' dialect: FIRRTL Annotations*] to encode the type information. This annotation system is intrinsic to the intermediate representation and it is subsequently used to pass the data to firtool. Then, this new debug info is consumed in CIRCT and associated with the final Verilog format. The debug dialect and HGLDD are updated accordingly to manage and represent types. Moreover, additional details about the algorithms and steps to extract types from the code, process and transform them into a cleaner format are provided in the subsections respective to each component.

The final output of the backend is a debug file containing enough information to reconstruct the hierarchical view of signals and associate them with their original Scala types. Since CIRCT and FIRRTL are not restricted to Chisel, the features introduced can potentially be used with other languages using the same compiler.

4.1. Collecting and passing the type information to CIRCT through FIRRTL

Subsection 2.3.2 introduced the `ChiselStage` class that implements a bridging interface with the CIRCT compiler and internally executes a sequence of phases (Figure 2.5) through a `PhaseManager` class. In this sequence, two phases are specifically responsible for the transition of Chisel to a FIRRTL equivalent representation: the `Elaborate` and `Convert` phases. The former executes and elaborates the body of a module into a circuit graph stored in a `ChiselCircuitAnnotation` while the latter converts it into a FIRRTL equivalent hardware graph stored in a `FirrtlCircuitAnnotation`. This, in turn, is passed to a final phase that calls the actual CIRCT compilation from Scala.

4.1.1. Updated phases to generate source language type information

The current `PhaseManager` pipeline permits firtool to access only information intrinsic to a FIRRTL representation, missing details about the source language that generated it. FIRRTL contains a standardized representation of Chisel circuits. It is able to preserve the structure and signal hierarchies, but it cannot express any Scala abstraction and associate its elements with original types as shown in Subsection 2.4.2. For instance, user-defined bundles are always converted to anonymous FIRRTL bundles, making their identification difficult as highlighted in Listing 4.1 and 4.2.

```scala
class Foo extends RawModule {
    // Anonymous bundle
    val io = IO(new Bundle {
        val a: UInt = Input(UInt(4.W))
        val b: Bool = Output(Bool())
    })
    // User defined bundle
    class BarAggr extends Bundle {
        val a: UInt = Input(UInt(4.W))
        val b: Bool = Output(Bool())
    }
    val bar: BarAggr = IO(new BarAggr)
    // ...
}
```

Listing 4.1: User-defined and anonymous bundles in Chisel

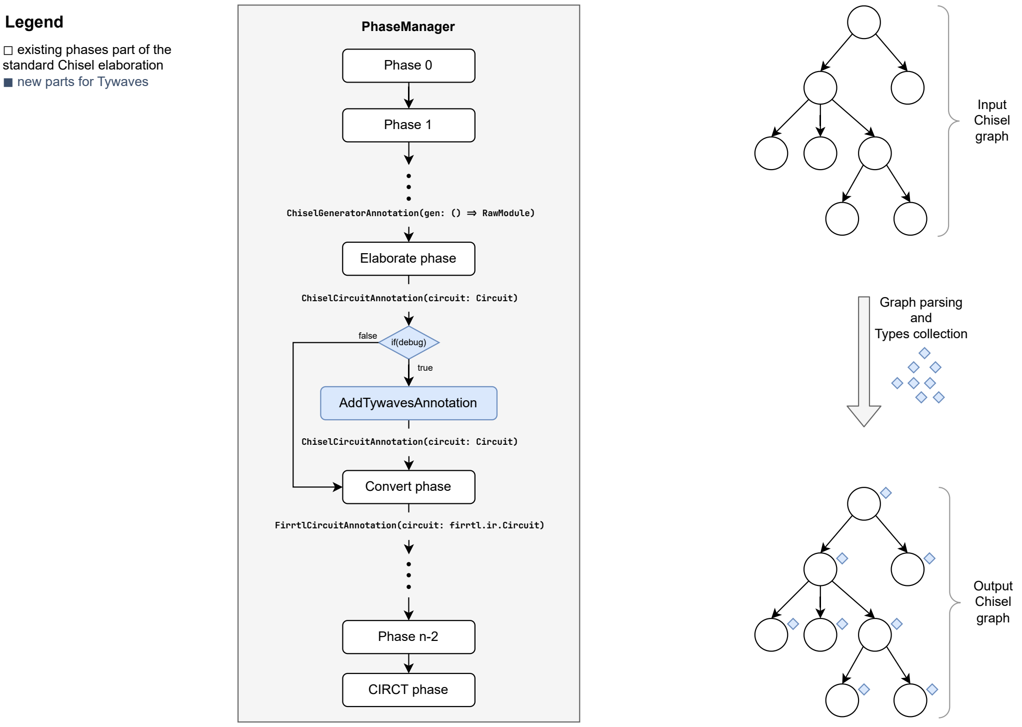
```
circuit Foo :
  module Foo :
    output io : {
        flip a : UInt<4>,
        b      : UInt<1>
    }

    output bar : {
        flip a : UInt<4>,
        b      : UInt<1>
    }

    ; ...
```

Listing 4.2: User-defined and anonymous bundles in FIRRTL

This limitation arises because FIRRTL is generated once the Scala meta-programming is executed, and there is no support for reconstructing it yet. In fact, during the `Convert` phase, any non-relevant information for FIRRTL is skipped. Although this simplifies the optimization process for lowering in CIRCT, it runs against the idea of debugging at Chisel source level. To overcome this, the information of Scala types needs to be passed to FIRRTL without changing its output functional code, thus, keeping compatibility with the existing pipeline.

An update to the existing `PhaseManager` to preserve types is proposed and Figure 4.1 reports the insertion of the changes made within the pipeline. A new phase, `AddTywavesAnnotation`, is added in between the `Elaborate` and `Convert` phases. It parses the circuit graph stored in `ChiselCircuitAnnotation` and annotates each of its nodes (modules and signals) with its respective type information and constructor parameters through a `TywavesAnnotation` (Subsection 4.1.2). The annotated circuit is then passed to the `Convert` phase for FIRRTL transformation.

Finally, this extra phase is executed only when a specific flag is set to indicate that the circuit should be elaborated in a ”debug compilation mode”. Hence, this addition will not affect the execution time of normal compilations. Debug information is not always necessary, there are several cases when this applies, for instance, when the logic is compiled for synthesis or when the designer wants to rapidly check if the code compiles without testing behavior. Similarly, the debug flag of the GCC compiler [1] can be set when it is used with the GDB debugger [2] to provide useful information for catching errors in the source code. When the code is compiled in release mode, the compiler passes to collect debug information do not run, resulting in a faster and lighter executable.


Legend
□ existing phases part of the standard Chisel elaboration
■ new parts for Tywaves
PhaseManager
Phase 0
Phase 1
ChiselGeneratorAnnotation(gen: () ⇒ RawModule)
Elaborate phase
ChiselCircuitAnnotation(circuit: Circuit)
false
if(debug)
true
AddTywavesAnnotation
ChiselCircuitAnnotation(circuit: Circuit)
Convert phase
FirrtlCircuitAnnotation(circuit: firrtl.ir.Circuit)
Phase n-2
CIRCT phase
Input Chisel graph
Graph parsing and Types collection
Output Chisel graph

(a) Updated PhaseManager to include type information in FIRRTL

(b) Associate type information to the Chisel hardware graph

Figure 4.1: New phase to generate the type information from Chisel

4.1.2. Tywaves Annotation: encoding types in FIRRTL

The `TywavesAnnotation` definition is reported in Listing 4.3. It is a case class with two fields, `typeName` and `params`, to encode respectively the name and the list of parameters of the variable type. Parameters of a Scala class are defined by a name, a Scala-type, and a value. Each of these case class fields is implemented as strings allowing to cover the representation of any possible type and value.

The annotation class implements the `SingleTargetAnnotation` trait which provides a serialization API to JSON in the form of a FIRRTL annotation. During the serialization to FIRRTL, annotation definitions are inserted at the top of a fir file, making the information they contain accessible to compilers and available for specific transformations such as debug operations in the case of Tywaves. Specifically, the FIRRTL annotations are a mechanism used to associate arbitrary metadata with zero or more target objects (i.e. signals, modules, etc...) of a FIRRTL circuit [20, *FIRRTL Annotations*].

```scala
/** Store constructor parameters of a Scala class */
case class ClassParam(name: String, typeName: String, value: Option[String])

/** Store types of a variable and its [[ClassParam]]s */
private[chisel3] case class TywavesAnnotation[T <: IsMember](
    target:   T,                        // The Chisel-FIRRTL target element
    typeName: String,                   // The name of the type
    params:   Option[Seq[ClassParam]] // Optionally emitted - not every class has
        parameters
  ) extends SingleTargetAnnotation[T] {
    // ...
}
```

Listing 4.3: Scala code of `TywavesAnnotation` which encodes the extra type information for Tywaves and extends a serialization API targeting FIRRTL files

As can be seen from Listing 4.4, the new additions of Figure 4.1 reflect minimal changes for using `ChiselStage`. The generation and serialization of `TywavesAnnotation` is automatically handled and abstracted by the `PhaseManager` execution. Each variable and module is recursively associated with its type name and parameters, so the info is created also for sub-variables and sub-modules. To give an illustration of how it is represented, Listing 4.5 shows a snippet with the resulting serialization of the annotations for the anonymous and user-defined bundles of the circuit `Foo` (Listing 4.1). The two targets are now decorated with their respective type names: `IO[AnonymousBundle]` for `io` and `IO[BarStruct]` for `bar`. The Scala-type can be now read by firtool making feasible the rest of the pipeline and with a compiler potentially able to distinguish their original types of components.

```scala
val chiselStageOpt = new ChiselStage                    // Classic compilation mode
chiselStageOpt.execute(Array("--target", "chirrtl"),
    Seq(ChiselGeneratorAnnotation(() => new Foo())))

val chiselStageDbg = new ChiselStage(withDebug = true) // Generate extra debug mode
chiselStageDbg.execute(Array("--target", "chirrtl"),
    Seq(ChiselGeneratorAnnotation(() => new Foo())))
```

Listing 4.4: Generating FIRRTL through `ChiselStage` abstractions

```
circuit Foo :%[[
  {
    "class":"chisel3.tywaves.TywavesAnnotation",
    "target":"~Foo|Foo>io",
    "typeName":"IO[AnonymousBundle]"
  },
  ; ... OTHER ANNOTATIONS
  {
    "class":"chisel3.tywaves.TywavesAnnotation",
    "target":"~Foo|Foo>bar",
    "typeName":"IO[BarStruct]"
  }
]] ; ... REST OF THE CIRCUIT
```

Listing 4.5: Serialization of Tywaves annotations in FIRRTL of Listing 4.1

4.1.3. An alternative to the FIRRTL annotation

Annotations are a mechanism deriving from the old SFC and represent an actual extension of the FIRRTL language. In other words, each new annotation translates into a new feature of the language that is required to be supported explicitly by any FIRRTL compiler.

The CIRCT compiler recently added a new option to add meta-data to FIRRTL without extending the language, called *intrinsics* [19, *'firrtl' dialect: Intrinsics*]. In contrast to annotations, this new construct represents an innate characteristic of the compiler rather than a language. This removes the need to update other potentially existing FIRRTL compilers. Moreover, it better matches the MLIR mechanism of operations, including strict definitions and type checking. On the Chisel side, intrinsics can be seen as blocks with pre-defined functionality that are intrinsic to the compiler and not implementable by hand [15, *Intrinsics*].

Although after switching to the CIRCT compiler intrinsics are considered a better mechanism to extend FIRRTL more suitable for MLIR, intrinsics were still a work in progress with great risk of being modified at the moment of realizing the implementation of Tywaves. Therefore, the project uses a more stable format like annotations and leaves intrinsics as an optimization for future improvements.

A possible implementation using intrinsics would not change many parts of the backend. The changes would involve the creation of an intrinsic expression for Chisel and the corresponding MLIR operation. Thus, differences would be only in the interface between Chisel and CIRCT.

4.1.4. Details about the collection of the type information

Previous subsections introduced the changes made to Chisel to access, transform, and pass the type information to the underlying CIRCT compiler. This section focuses on details about the actual algorithm and methods used to traverse the hardware circuit graphs and retrieve Scala compile-time information.

The IR data structures stored in `ChiselCircuitAnnotation` and `FirrtlCircuitAnnotation` are trees where each node represents a component in a circuit and may have multiple children nodes of the same or different type. After the `Elaborate` phase each Chisel construct like signals, module instances, control statements, temporary values, etc., creates a node entry of the circuit data structure. The nodes and respective children involved in Tywaves are summarized in Table 4.1. The new pass added by Tywaves traverses and updates the `ChiselCircuit` tree using the Depth First Search algorithm (DFS) [39]. The variant used is the pre-order traversal that processes each visited node before its children.

Node	Children
Circuit	Component (DefModule)
DefModule	Port, Command/Definition
Port	Data
Command/Definition	Command/Definition, Node, Data
Node	Data
Data	

Table 4.1: Nodes and respective children of `ChiselCircuitAnnotation` involved in the `AddTywavesAnnotation` phase

According to the definition of typed-circuits from Section 2.4, a `TywavesAnnotation` (Listing 4.3) is created for module definitions and signals, where signals are defined through the `Data` class of Table 4.1. To create the annotation, the type and parameters need to be extracted from the nodes considered. Both of them are compile-time information while the `PhaseManager` is executed during runtime. Therefore, their information is obtained using Scala reflection [5] which allows to access compile-time data during the runtime of the program.

Finally, Section 2.4 claimed that named values of enum variants also have an important role in debugging a Chisel circuit. In comparison to types though, enum variants already have their own annotation in FIRRTL. Thus, a specific update for them in Chisel has not been necessary. The debug information of an enum is represented in FIRRTL by a set of three annotations (Table 4.2), storing the definitions and associating each enum signal to the respective definition. However, the type name of an enum signal is not handled by these annotations, and this makes Tywaves an added value also for `ChiselEnum`.

Enum Annotation	Description
EnumDefAnnotation	Contains a unique ID of an enum type and a map of the variants with raw integer values
EnumComponentAnnotation	Associate a FIRRTL target with an enum definition using the ID
EnumVecAnnotation	Associate a FIRRTL vector with an enum definition using the ID. When the vector has aggregate types it contains a list of which fields have the enum type

Table 4.2: Enum annotations already implemented in Chisel

To give an illustration of how a circuit is represented in a Chisel IR graph together with its `Tywaves` annotations Figure 4.2 depicts the `ChiselCircuitAnnotation` of `Detect2Ones` example.

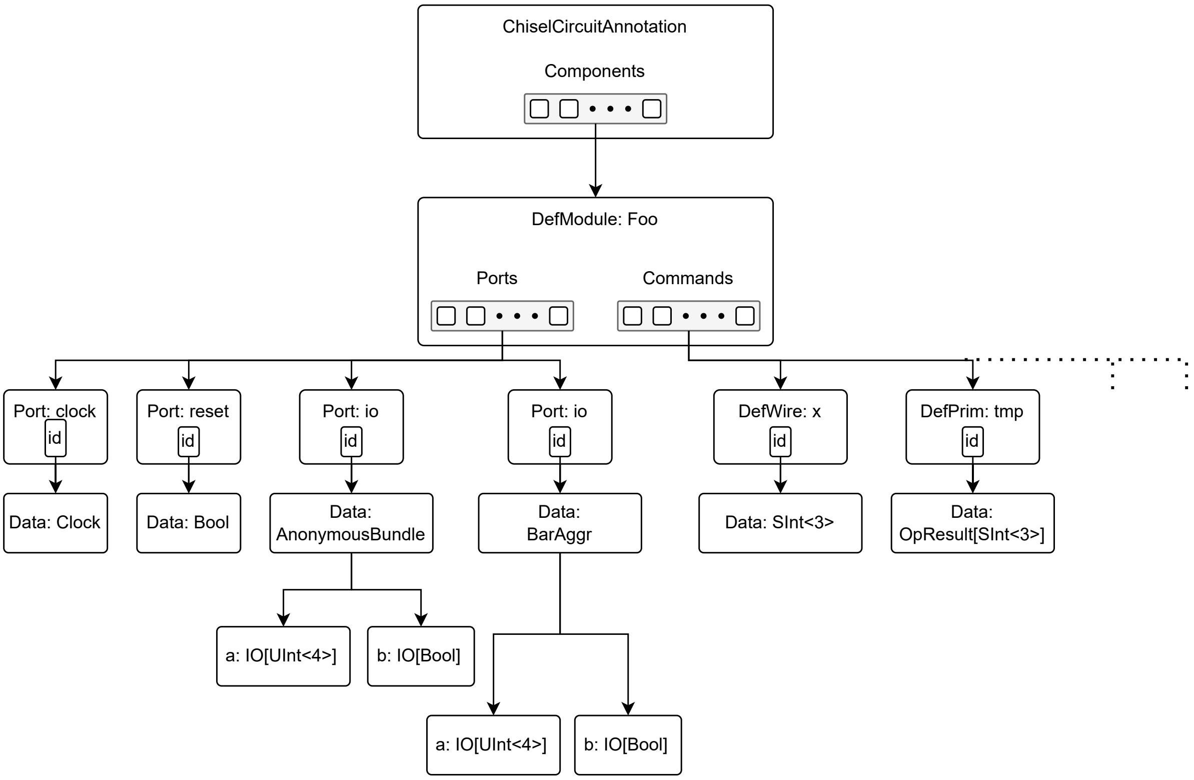


Figure 4.2: Example of Chisel circuit graph

4.2. Associate original source code info with dumped traces

Previous sections described how the annotation mechanism can be used to encode the type information in FIRRTL. This extra information needs to be consumed inside firtool and elaborated to new specific HGLDD fields for reconstructing the view of Chisel from the generated Verilog. The debug dialect and HGLDD already implement a structure to track the correlation between values, types, and hierarchy of the IRs internal to CIRCT implemented as other dialects. Nevertheless, as mentioned in Subsection 2.3.1, only FIRRTL has an MLIR dialect in the compiler whereas Chisel does not, precluding any opportunity to get Chisel types into the debug dialect and limiting viewers to merely enable FIRRTL waveforms at the current state. Although FIRRTL keeps the same hierarchies and variable names of Chisel, it does not express any Scala meta-programming information, i.e. the Scala types.

This section explains how CIRCT has been updated to consume the new `TywavesAnnotation`, how it materializes the information in an updated debug dialect, and transforms the changes of the IR into respective new JSON entries of HGLDD to make the new features usable by external programs without breaking the existing behavior.

Operation	Operands	Results	Attributes
dbg.variable	value Opt<scope> ***Opt<enumDef>***	-	name ***Opt<typeName>*** ***Opt<params>***
dbg.scope	scope	ScopeType	instanceName moduleName
dbg.array	element	ArrayType	-
dbg.struct	fields	StructType	-
dbg.subfield	***value*** ***Opt<scope>*** ***Opt<enumDef>***	***SubFieldType***	***name*** ***Opt<typeName>*** ***Opt<params>***
dbg.moduleinfo	-	-	***Opt<typeName>*** ***Opt<params>***
dbg.enumdef	***Opt<scope>***	***EnumDefType***	***enumTypeName*** ***id*** ***variantsMap***

Table 4.3: Debug dialect operations. In ***bold-italic*** the changes made.

4.2.1. Consuming the Tywaves annotations in CIRCT

According to Figure 2.3, once a FIRRTL source is parsed and translated into the respective dialect, some lowering operations, transformations, and checks are executed before diving into the core dialects. Specifically, one of the first steps executed by CIRCT is the `LowerAnnotation` pass which parses the JSON representation of annotations in the file header, discards unsupported annotations, and processes the others to compute custom operations. Each annotation stores different meta information (debug, optimization, formal verification, etc.) and performs a distinct transformation. In the case of this project, the `TywavesAnnotation` presented in Subsection 4.1.2 is pushed as an entry of the annotations field of the MLIR operation correlated with its FIRRTL target [19, *’firrtl’ dialect*].

4.2.2. Updated debug dialect

As mentioned earlier MLIR dialects enable the integration of different levels of abstraction and computations. Specifically, Subsection 2.3.1 anticipated that the four debug dialect operations do not have native support for Chisel source information yet. Tywaves updates the debug dialect through TableGen [45] to reconstruct type names and parameters from MLIR operations. TableGen provides a straightforward DSL to define dialects and prototype new features, in fact, it automatically generates all the C++ classes and functions to manipulate the operations. Table 4.3 summarizes the debug dialect and underlines the additions made to handle Tywaves.

According to the proposed pipeline of Figure 4.1 the `TywavesAnnotation` is not always emitted, in such a case the type Chisel information might not be available in FIRRTL. To handle this possibility, all operations and attributes of the updated debug dialect related to the source language type must be declared as optional MLIR fields (see Appendix C for the full definition of the new debug dialect). The next subsections provide more details and reasons for the changes made.

4.2.2.1. Storing enum type definitions in dbg.enumdef

One of the current problems of the debug dialect is the inability to reconstruct named values of enum variants. To combat this issue, a new dbg.enumdef operation is derived from the enum annotations of Table 4.2 and its attributes store data to reconstruct the named variants of an enum type. More precisely, variants are internally implemented as a `DictionaryAttr` : $(Int \rightarrow String)$. This kind of map can be directly converted to a hash map by a viewer to access the variant with a time complexity of $O(1)$. In contrast, an array attribute of strings cannot be used since the user might select the order and the single raw values of the variants.

Finally, since enum creates mapped values for signals, the dbg.enumdef needs to be accessed from other operations. Thus, it returns a result that can be pushed as an operand of other debug Ops.

Listing 4.6 contains the definition corresponding to the `State` enum of the `Detect2Ones` example.

```
%0 = dbg.enumdef ”DetectTwoOnes$State”, id 0,
     {sNone = 0 : i64, sOne1 = 1 : i64, sTwo1s = 2 : i64}
```

Listing 4.6: The dbg.enumdef operation from the FSM example of Figure 2.6a

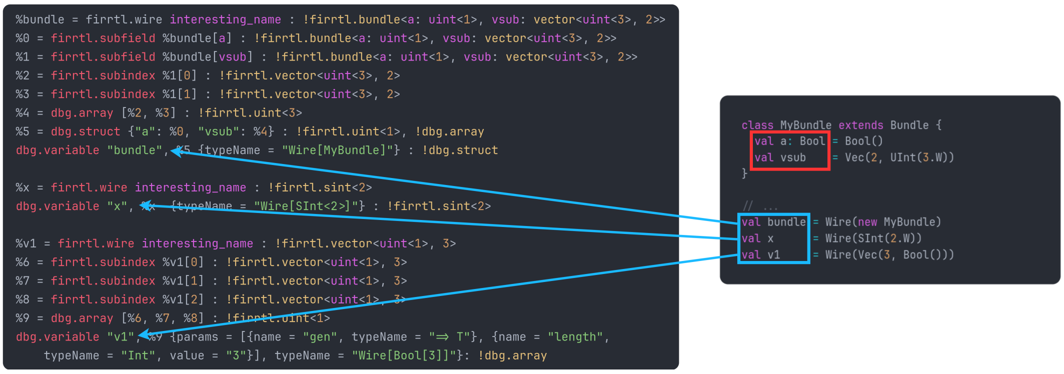
4.2.2.2. New dbg.variable attributes

Two new MLIR attributes have been added to the dbg.variable for representing the name and parameters of the source type of a variable declared in the source code. In TableGen, these two new MLIR fields need to contain the string serialization of `typeName` and `ClassParam` of the Scala annotation Listing 4.3 and they are therefore declared as an LLVM `StringAttr` and an `ArrayAttr` of a `DictAttr` respectively. On the other hand, a variable can be created from an enum signal. Hence an optional operand is added to accept the result of a dbg.enumdef operation. This allows to map the values of a variable with the corresponding named variants.

Notably, Tywaves information might not be generated, and this suggests that also the `typeName` and `params` attributes should be optional (see Appendix C).

4.2.2.3. The dbg.variable is not enough for subfields of aggregates

A dbg.variable explicitly captures direct declarations of variables in a module but it does not relate fields of aggregated values as shown in Figure 4.3. On the other hand, dbg.array and dbg.struct are obviously responsible for only maintaining the hierarchical structure of their respective aggregates.



```
%bundle = firrtl.wire interesting_name : !firrtl.bundle<a: uint<1>, vsub: vector<uint<3>, 2>>
%0 = firrtl.subfield %bundle[a] : !firrtl.bundle<a: uint<1>, vsub: vector<uint<3>, 2>>
%1 = firrtl.subfield %bundle[vsub] : !firrtl.bundle<a: uint<1>, vsub: vector<uint<3>, 2>>
%2 = firrtl.subindex %1[0] : !firrtl.vector<uint<3>, 2>
%3 = firrtl.subindex %1[1] : !firrtl.vector<uint<3>, 2>
%4 = dbg.array [%2, %3] : !firrtl.uint<3>
%5 = dbg.struct {"a": %0, "vsub": %4} : !firrtl.uint<1>, !dbg.array
dbg.variable "bundle", %5 {typeName = "Wire[MyBundle]"} : !dbg.struct

%x = firrtl.wire interesting_name : !firrtl.sint<2>
dbg.variable "x", %x  {typeName = "Wire[SInt<2>]"} : !firrtl.sint<2>

%v1 = firrtl.wire interesting_name : !firrtl.vector<uint<1>, 3>
%6 = firrtl.subindex %v1[0] : !firrtl.vector<uint<1>, 3>
%7 = firrtl.subindex %v1[1] : !firrtl.vector<uint<1>, 3>
%8 = firrtl.subindex %v1[2] : !firrtl.vector<uint<1>, 3>
%9 = dbg.array [%6, %7, %8] : !firrtl.uint<1>
dbg.variable "v1", %9 {params = [{name = "gen", typeName = "⇒ T"}, {name = "length",
    typeName = "Int", value = "3"}], typeName = "Wire[Bool[3]]"}: !dbg.array
```

```
class MyBundle extends Bundle {
  val a: Bool = Bool()
  val vsub    = Vec(2, UInt(3.W))
}

// ...
val bundle = Wire(new MyBundle)
val x      = Wire(SInt(2.W))
val v1     = Wire(Vec(3, Bool()))
```

Figure 4.3: The dbg.variable reconstructs only signal top declarations (light-blue). The subfields (red) miss an associated dbg Op.

Consequently, the current operations on their own cannot fully reconstruct the source types of subfields, even though they are capable of retrieving the original hierarchies from the generated Verilog. This is further confirmed when we take a look at the official documentation of the Ops:

”1) The dbg.variable operation is useful to represent named values in a source language. For example, ports, constants, parameters, variables, nodes, or name aliases can all be represented as a variable... 2) The dbg.struct operation allows for struct-like source language values to be captured in the debug info. This includes structs, unions, bidirectional bundles, interfaces, classes, and other similar structures... 3) The dbg.array operation allows for array-like source language values to be captured in the debug info. This includes arrays, ...” [18].

The first statement does not suggest any reference to subfields for a dbg.variable but rather the cited operations cover only the declarations of FIRRTL types in a module [20, *Operation Definitions – Declarations*]. On the other hand, fields of aggregates have specific subfields FIRRTL operations [20, *Operation Definitions – Expressions*] also shown in the figure. Therefore, a *value* of a debug variable Op can only have FIRRTL declarations as operands. Additionally, dbg.variable cannot be even used as the operand of structs and arrays since it does not return any result type and, because of that, it cannot be passed to other Ops[1].

The left side of Figure 4.3 shows how the debug dialect, after the first update, can reconstruct the source level hierarchy independently from FIRRTL and highlights three things. First, the variable does

[1] In MLIR operations return results and these results can be passed to other operations as operands.

not return any result; second, there has been only one variable for the declaration of `bar` and no one for its named fields; finally, the results of the operations are cascaded between operations.

The observations above advise three possible solutions to store and manipulate type information for subfields through the debug dialect:

1. *Using dbg.variable as an operand in dbg.struct/dbg.array*. The first possible solution to the problem would consist of adding a return result for a variable Op such that it can be passed as an operand to the debug aggregate Ops.

2. *Update structs and arrays with attributes for type information*. An alternative approach is to add the extra information next to the operations of the aggregates operations directly. For instance it would be possible to obtain something similar to
   `%2 = dbg.struct ”a”: ”value”: %0, typeName = ”Wire[Bool]”, ”vsub”: ”value”: %4, typeName = ”Wire[UInt<3>[2]]”, params = [ ... ].`

3. *Create a new distinct debug operation for subfields*. A third possibility is to define a separate operation specific to elements of aggregates.

The first two solutions seem to be able to solve the issue addressed, but they both have disadvantages compared to the last one. First of all, some updates to existing operations not only imply changes in the Op definitions but may also require adapting their usage in the compiler, likely leading to more difficult integration. This is exactly the case of the first and second proposals. For instance, although changing the definitions in TableGen is trivial, updating already implemented usages might be an onerous task due to the size of the CIRCT project. Moreover, with the first approach, it would be unclear whether the variable refers to a top declaration or a subfield of another declaration. In the second option, the information is not part of the subfield value but is instead associated with the operand.

Taking into account these thoughts, the solution is obvious. Declaring a new debug operation specific for managing extra debug info of subfields allows keeping the code using other Ops in the compiler untouched. Then, it would also be possible to clearly distinguish variables and fields. Separating data of declarations, aggregates, and sub-elements is another key point for a better organization of the information, already implemented in other dialects, and following the principles of MLIR [43, *p. 3*].

4.2.2.4. The dbg.subfield operation

This operation enables tracking debug information for subfields of aggregates separately from the parent variable. The dbg.subfield Op (Table 4.3) has the same attributes as the dbg.variable to store the value, source language name, type name, and type parameters but, unlike the latter, it also returns a `SubFieldType` result which can be passed as operand of dbg.struct and dbg.array operations as demonstrated in Listing 4.7. Since a field value is always a descendant of another value, the dbg.subfield does not have a scope operand, as opposed to the dbg.variable.

```
%0 = firrtl.subfield %bundle[a] : !firrtl.bundle<a: uint<1>,
  vsub: vector<uint<3>, 2>>
%1 = dbg.subfield ”bundle.a”, %0 {typeName = ”Wire[Bool]”} :
  !firrtl.uint<1>
%2 = firrtl.subfield %bundle[vsub] : !firrtl.bundle<a: uint<1>,
  vsub: vector<uint<3>, 2>>
%3 = firrtl.subindex %2[0] : !firrtl.vector<uint<3>, 2>
%4 = dbg.subfield ”bundle.vsub[0]”, %3 {typeName = ”Wire[UInt<3>]”} :
  !firrtl.uint<3>
%5 = firrtl.subindex %2[1] : !firrtl.vector<uint<3>, 2>
%6 = dbg.subfield ”bundle.vsub[1]”, %5 {typeName = ”Wire[UInt<3>]”} :
  !firrtl.uint<3>
%7 = dbg.array [%4, %6] : !dbg.subfield
%8 = dbg.subfield ”bundle.vsub”, %7 {params = [ ... ],
  typeName = ”Wire[UInt<3>[2]]”} : !dbg.array
%9 = dbg.struct {”a”: %1, ”vsub”: %8 : !dbg.subfield, !dbg.subfield
dbg.variable ”bundle”, %9 {typeName = ”Wire[MyBundle]”} : !dbg.struct
```

Listing 4.7: The dbg.subfield operation for the `bundle` variable from Figure 4.3

4.2.2.5. Encoding generic module type information in dbg.moduleinfo

As stated in Section 2.4, types cover modules and instances as well as signals of a circuit. Hence, Tywaves defines dbg.moduleinfo operation which stores this information for a module and makes it available in the compiler. The Op is just declarative and does not accept operands or return results. Unlike signals, module hierarchies and dependencies are preserved characteristics during the compilation process and explain the choice made. An example is reported in Listing 4.8. Although `firrtl.circuit` and `firrtl.module` seem to contain the same name, they are not part of the debug information.

```
module {
  firrtl.circuit ”MyModule” {
    firrtl.module @MyModule() {
      // Other Ops ...
      dbg.moduleinfo typeName = ”MyModuleType”
    }
  }
}
```

Listing 4.8: The dbg.moduleinfo operation

4.2.3. Materializing the debug dialect

After the `LowerAnnotation`, the meta-data contained in the FIRRTL annotations is added to the respective attribute of FIRRTL operations. Despite this, the information is not associated with debug operations, yet it may be subject to further optimizations. To overcome this potential issue, the annotations should be converted to debug operations, free from compiler optimization steps.

The `MaterializeDebugInfo` is a compiler pass responsible for looking at the modules, ports, and wires of a FIRRTL circuit and generating the corresponding tracking operations such that the FIRRTL perspective is preserved through the transformation pipeline.

According to the new operations introduced previously, the pass is updated to follow the pseudocode of Algorithm 1, 2 and 3. When the algorithm implemented in `MaterializeDebugInfo` pass is applied the debug dialect operations presented in the previous section are created. Specifically, all the operations declared in a module are processed to preserve the hierarchical structure of aggregates and associate to variables and fields the type information. Finally, the algorithms of the pass implement the following features:

- Extract all the enum definitions, create the respective dbg.enumdef operations, and cache each of them in a hash map for later reuse.
- Create a dbg.variable operation for each declaration of ports and non-IO signals in the module.
- Unpack all the aggregates and repack them with debug aggregates and subfield operations.
- For each declared variable and subfield FIRRTL operations: extract the Tywaves information and enum reference definition from the annotation list of the processed Op/SubOp; eventually lookup the enum definition operations cache map and, if any, insert the Op as an operand of the dbg.variable or dbg.subfield; insert the type name and parameters as attributes.

Figure 4.4 shows a dependency graph example of debug operations. The ”`%`” values in MLIR are the operations’ results and represent their return values. Therefore, they are used as operands of other operations (i.e. in the figure, `%1` and `%3` are the results of the two dbg.subfield Ops and operands of the dbg.struct Op).

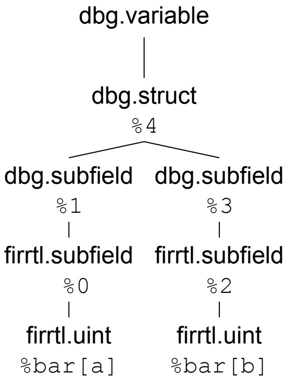


Figure 4.4: Dependency graph of results of operations

Algorithm 1 `MaterializeDebugInfo` pass pseudocode

```
procedure MaterializeDebugInfo($Module$, $OpBuilder$)
    $Ops \leftarrow Module.getElements()$ // Get all the operations in a module (ports, wires, nodes, regs)
    $EnumDefCache \leftarrow \emptyset$ // Cache all the enum definitions in a module
    for all $op \in Ops$ do
        for all $Anno \in AnnoList$ do
            if $Anno$ is EnumDefAnnoType then
                $id \leftarrow$ createNextId
                $enumDefOp \leftarrow OpBuilder.createDbgEnumDefOp(Anno.extractFields, id)$
                $EnumDefCache.insert(enumDefOp.name, enumDefOp)$
    for all $op \in Ops$ do
        $AnnoList \leftarrow op.getAllAnnotations$
        $(Value, TypeInfo, EnumDefRef) \leftarrow$ CreateDebugAggregates($op$, $OpBuilder$, $AnnoList$,
        $EnumDefCache$) // Get the value for the dbg variable
        return $OpBuilder.createDbgVariableOp(Value, VarAnno, TypeInfo.typeName,$
        $TypeInfo.params, EnumDefRef)$
```

Algorithm 2 `GetTywaves`: extract type info and eventual enum definition from the list of annotations of a module

```
procedure GetTywaves($AnnoList$, $Op$, $EnumDefCache$)
    $T \leftarrow \emptyset$
    $E \leftarrow \emptyset$
    for all $a \in AnnoList$ do // Filter the annotations and extract the fields from the targets in a better format
        if $a$ is TywavesAnnoType then
            $T.push$(extractFields($a$))
        if isGround($Op$) & $a$ is EnumCompAnnoType then
            if $E \neq \emptyset$ then
                // Search the definition, the EnumCompAnnoType stores only the name, not the enumDefOp
                $E \leftarrow EnumDefCache.lookup(a.name)$
    return $(T, E)$
```

4.2.4. Emitting a debug file for external programs

According to Figure 2.7 the compiler infrastructure should output and pass extra debug information about the source language to an external viewer. All the MLIR dialects are used internally in the compiler and their output serialization is not meant to be parsed by external tools. Therefore, the debug dialect cannot be used directly by a waveform viewer and the information in it needs to be converted to a file format independent of the MLIR syntax.

As introduced in Subsection 2.3.1, the current state of the CIRCT compiler and the debug dialect implements the HGLDD debug file format, a JSON-based format defined in collaboration with Synopsys. This implementation does not support specific Chisel source type information and a new format needs to be specified for the waveform viewer that this project targets. However, it seems that an update to this existing file format would simplify the work done, compared to a new specification, because

Algorithm 3 `CreateDebugAggregates`: redefine all the FIRRTL operations with debug operations. Unpack all the FIRRTL aggregates and repack them. Add also the associated type and enum def information.

```
procedure GenerateResult($OpBuilder$, $Op$, $result$, $AnnoList$, $EnumDefCache$)
    if isDeclaration($Op$) then
        return ($result$, ∅, ∅)
    else
        ($tywavesInfo, enumDefRef$) ← GetTywaves($AnnoList$, $Op$, $EnumDefCache$)
        return ($OpBuilder.createDbgSubFieldOp(result, tywavesInfo.typeName,$
        $tywavesInfo.params, enumDefRef)$, $tywavesInfo$, $enumDefRef$)

procedure CreateDebugAggregates($Op$, $OpBuilder$, $AnnoList$, $EnumDefCache$)
    // Extract type and enumDef information
    if $Op$ is FirrtlBundle then
        // Collect subfields of the struct
        $fields$ ← ∅
        for all $subOp$ ∈ GetSubOps($Op$) do
            $subFieldOp$ ← $OpBuilder.createStructSubFieldOp(subOp)$
            $val$ ← CreateDebugAggregates($subFieldOp$, $OpBuilder$, $AnnoList$, $EnumDefCache$) // Recursive pass
            $fields.push(val)$

        // Create and return the result
        $result$ ← $OpBuilder.createDbgStruct(Op, fields)$
        return GenerateResult($OpBuilder$, $Op$, $result$, $AnnoList$, $EnumDefCache$)
    else if $Op$ is FirrtlVector then
        // Create indexed elements of the vector
        $elements$ ← ∅
        for $idx = 0, ...,$ GetNumSubOps($Op$) do
            $subOp$ ← $OpBuilder.createSubIndexOp(Op.value, idx)$
            $val$ ← CreateDebugAggregates($subOp$, $OpBuilder$, $AnnoList$, $EnumDefCache$) // Recursive pass
            $elements.push(val)$

        // Create and return the result
        $result$ ← $OpBuilder.createDbgArray(Op, elements)$
        return GenerateResult($OpBuilder$, $Op$, $result$, $AnnoList$, $EnumDefCache$)
    else if $Op$ is FirrtlGround then
        return GenerateResult($OpBuilder$, $Op$, $Op$, $AnnoList$, $EnumDefCache$)
    else
        // Return nothing
        return (∅, ∅, ∅)
```

it would allow the re-usage of most of the emitted code. Therefore, an extension to the HGLDD format is defined in this section (Section D.3 contains the UML class diagram of the new format).

Nevertheless, overwriting the current emitter in firtool to output a new version with updated or new fields might break the file from Synopsys's side. Because of that, the updated format presented in this thesis is emitted separated from and in addition to the current HGLDD, and it is reported in Table 4.4.

Extended HGLDD

Every HGLDD file contains a header with generic information about the source and output files and a list of "objects" representing the module and aggregates definitions used in the HGL circuit that is compiled. A few fields (Listing 4.9 and 4.10), are added to the format in order to represent Tywaves type information and maps for the values of enum types. First, the "`source_lang_type_info`" is added correspondingly to each module definition and variable/subfield instance entry in HGLDD. Second, each enum definition is encoded with a map ($EnumDef : RawValue \rightarrow Name$) and inserted in an "`enum_defs`" list present in module definitions. This last list is implemented as a map ($list : EnumId \rightarrow EnumDef$) to enable fast lookup. Finally, the variables of enum types have an additional entry storing the reference ID to the respective enum definition.

Node	**Children**		**Description**
	Node	**JSON entry name**	
Root	Header Object	HGLDD objects	The whole file. Contains a header and a list of objects.
Header	-		Header of the file. Contains version of the format and info about the source files
Object	ObjectKind Location Variable Instance SourceLangType `Map<int<int, string>>`	kind hgl_loc, hdl_loc port_vars children source_lang_type_info enum_defs	An object in in the HGLDD file. It can be a module definition or a struct-like type. It has also a name (hgl and hdl) and a flag in the case of an external module.
ObjectKind			An enum variant to indicate the type of an Object. It can be module or struct.
Variable	Location Expression TypeName PackedRange UnpackedRange SourceLangType int	hgl_loc, hdl_loc value_expr type_name packed_range unpacked_range source_lang_type_info enum_def_ref_id	A variable in HGLDD. It has a name and information indicating the location in source (hgl) and target file (hdl), the reference in hdl (value_expr).
SourceLangType	ContructorParams	params	The serialization of Tywaves type info in HGLDD format.
ContructorParams	-	-	The parameters of a source lang type.
Instance	Location Variable Instance	hgl_loc, hdl_loc port_vars children	The instance of a module. Its content is similar to a module definition. Each Instance is usually defined in another HGLDD file. Each module definition has its own file.
Expression	-	-	An expression can refer to a signal in the target language or a constant value or an operator that combines other expressions/variables.
PackedRange			Indicates the dimensions of a variable in the target language (i.e. logic [7;0] x in verilog)
UnpackedRange	-	-	Dimensionality of a variable in the target language (i.e. logic x [1:0][3:0] in verilog). It is associate to a vec-like variable.
TypeName	-	-	The type name in the target language.
Location	-	-	A location of a variable, instance in a file.

Table 4.4: Fields and nodes in HGLDD file format.

```
"source_lang_type_info": {
    "type_name": "<The type of the variable or module>",
    "params": [
        {
            "name": "<The name of the parameter>",
            "typeName": "<The source language type of
                the parameter>",
            "value": "<The value actually used>"
        }
    ]
}
```

Listing 4.9: Example of `"source_lang_type_info"` to serialize type names and constructor parameters in HGLDD

```
"enum_defs": {
  "0": {
    "0": "sNone",
    "1": "sOne1",
    "2": "sTwo1s"
  },
  "1": {
    "1": "A",
    "3": "B"
  }
}
```

Listing 4.10: Example of the enum definition maps in HGLDD

5 | Tywaves frontend

The waveform visualization and testbench abstractions play a significant role in the project since they represent the interaction with the user. The former implements the actual debug tool that designers use to check the behavior of a digital design through the inspection of signal values while the latter allows them to execute simulations and tests on a specific design. Therefore, referring to the views of Figure 3.1 this chapter presents the frontend of Tywaves.

This portion of the implementation can be divided into two main parts: the waveform GUI and the Tywaves-Chisel API. First of all, a rationale for the choice of extending the Surfer waveform viewer [56] is given in this chapter. Next, a detailed explanation of the changes to the chosen UI and the implementation of a library to interface the waveform with the backend outputs and manipulate the debug information is provided. To conclude, the end of the chapter describes an intuitive ChiselSim API to use the Tywaves project for testing Chisel circuits.

5.1. Extending Surfer

As can be deducted from the software architecture diagram shown in Figure 3.1, Tywaves involves several tools and updates to achieve type-based signal visualization. For this reason, the optimal solution for creating a graphical user interface (GUI) is to extend one of the existing open-source waveform viewers. This has the advantage of leveraging established and tested functionalities and so is reducing development time and the potential for errors compared to realizing a new interface from scratch.

Several open-source alternatives exist such as GTKWave, Surfer, VaporView, WaveTrace, Sootty, and simview [4, 25, 31, 46, 50, 56]. Among them, GTKWave is one of the oldest and most used. However, it does not offer the opportunity to easily integrate new extensions to its current features, i.e. through plugin systems. On the contrary, Surfer is built with a focus on extensibility and provides a more straightforward method to add new functionalities such as the support of a new language or a different kind of signal rendering as in the case of this project. Thus, the frontend uses and extends Surfer to support a type-based representation of Chisel circuits.

Surfer is a new viewer written in Rust with an active community open to updates and extending support to other HDLs, including Chisel. It has been implemented with an emphasis on extensibility; in fact, it natively supports the customization of signal representations. The value rendering system is based on a `Translator` trait which translates a given raw value associated with a signal with a custom representation.

Figure 5.1 shows a high-level diagram of Surfer and how the Tywaves functionality is integrated. As can be seen, trace files are loaded and converted to an intermediate data structure by Wellen [41] providing a format-independent interface with the rest of the system. A waveform container, in Surfer, loads and preserves the state of the intermediate data structure while the `Translator` trait defines common functions to render a raw bit vector value with a custom representation. Moreover, the trait system ensures modularity since it can be implemented by multiple Rust structs. On the other hand, the Tywaves-rs library bridges debug information with trace files. It follows the same logic as Wellen by converting multiple formats into a common intermediate data structure (TyVcd). This way the translator can eventually interface with multiple debug file formats without changing its functionality. Additionally, a generic structure facilitates the extension to other languages. Finally, the wave container extracts the values associated with a signal and queries the current translator for the rendering of such a value. Section D.2 illustrates sequence diagrams of how a trace file and debug information are loaded in Surfer and highlights the interaction with the user.

The following subsections provide specific details about each added and updated component present in the figure.

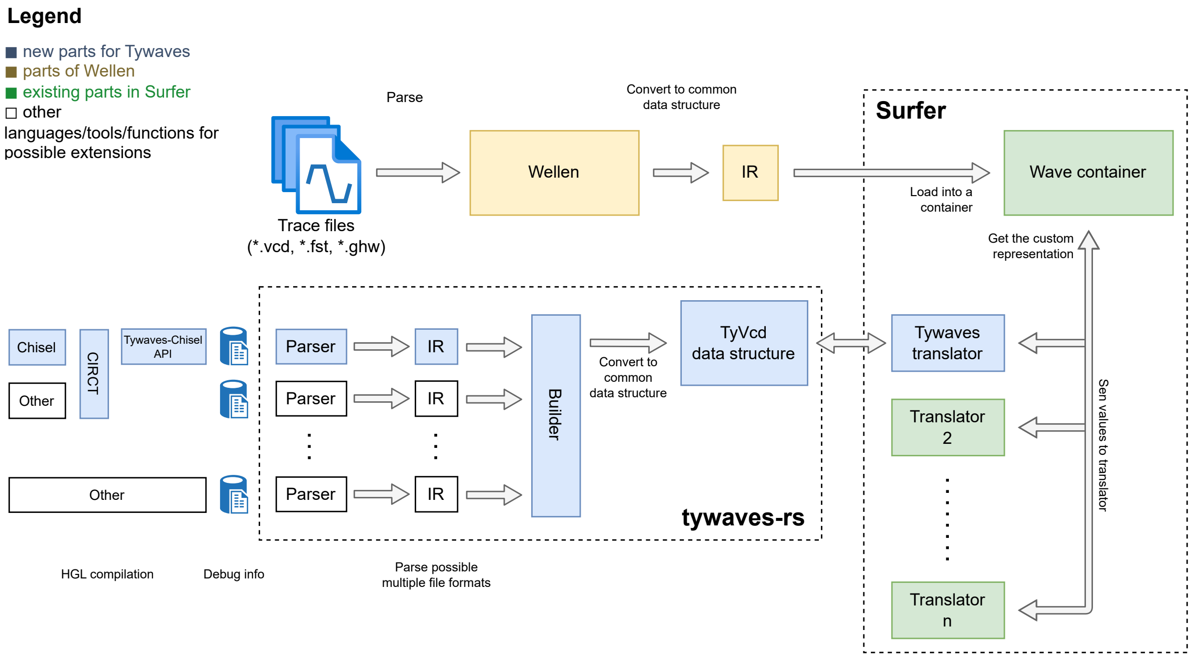


Figure 5.1: Tywaves integration within the Surferwaveform viewer

5.1.1. Tywaves-rs: bridging Tywaves information in Rust

Tywaves-rs is a Rust library that provides a programming interface with the debug file and information emitted by CIRCT as shown in Figure 3.1. It specifically provides functions to parse the file and process the debug information to build an internal data structure independent from the constructs of the input (Section D.2 shows a sequence diagram that highlights the exact steps followed in Surfer). Tywaves-rs is divided as follows:

- TyVcd: an intermediate data structure that links source language representation with the respective trace files.
- A parser or multiple parsers for reading and converting one or multiple debug files into a Rust data structure.
- TyVcd builder: a generic struct that provides an interface to populate TyVcd from any other data structure, i.e. a data structure representing HGLDD.
- VCD rewriter: a construct that rewrites VCD entries of aggregate variables such that all the values are collected and concatenated in a unique entry of the output.

Its internal structure is organized such that the common interface can be built from multiple debug file formats. For this project specifically, a parser and builder for HGLDD have been implemented and presented.

5.1.1.1. TyVcd intermediate data structure

As mentioned in Section 5.1, Surfer is an extensible waveform viewer, and following this idea tywaves-rs should provide a format-agnostic interface for Surfer. TypedVcd (TyVcd) is an efficient and standardized Rust data structure with support for multithreading, not restricted to Tydi, Chisel, or HGLDD (see Section D.4 for the full UML class diagram).

Table 5.1 summarizes the nodes of the data structures. The root of TyVcd stores all the scope definitions of a design that can be modules, instances, etc. These definitions are stored in a shared atomic reference (`Arc<RwLock<T>>`) to enable thread-safety and to allow linking scope hierarchies using shared references. In this way, a scope definition can have a field pointing to its children subscopes without memory overhead. In contrast, variables are unique to a scope definition, and they can be stored without Rust reference counters. Then an abstraction to the generic hierarchical kind of a variable is implemented by VariableKind. This can be a ground, struct-like, vec-like, or external kind. The latter is used by TyVcd when none of the other kinds correspond to any kind of source language. The other three kinds implemented are used by most of the languages for the ”topology” of signals. For instance, Chisel has ground types (bits), bundles, and vectors to represent different hierarchies, and also VHDL and System Verilog have similar constructs to indicate a the shape.

On the other hand, the type information as introduced in this thesis in Section 2.4 refers to the actual displayed type in the source language rather than the hierarchical structure of the signal. This data is represented in TyVcd through TypeInfo and ConstructorParams; two structs that reflect the definitions of Listing 4.3.

Similarly to scope definitions, enum definitions are also handled through a shared reference because they can be associated with one or more variables. As anticipated in Subsubsection 4.2.2.1 the map of the named variants of an enum can be implemented as `HashMap` to enable fast lookup, not as a vector because the named variants might not have contiguous values. This way, when the wave container passes an integer value of the trace associated, the enum variants can be accessed in $O(1)$. Finally, depending on how many signals are defined as enums in the design and on the number of variants, a definition can become huge and a shared reference helps to reduce the memory cost.

Node	Children	Description
TyVcd	ScopeDef	The root of the whole data structure. It stores a list of scope definitions.
ScopeDef	ScopeDef, Variable, TypeInfo	Represent a single scope definition. It has a name, a path (from the root), type information associated, and it may have zero or more subscopes/variables.
Variable	TypeInfo, VariableKind, EnumValMap	Represent a variable of any type (declaration, subfield...). It has a name, type information, a generic kind, and optionally a link to an enum definition.
TypeInfo	ConstructorParams	Represents the tywaves type information. It therefore has a name and a list of constructor parameters.
ConstructorParams	-	Stores parameters associated with a twaves type.
VariableKind	-	An Enum that represents a kind of a Variable. It can be Ground, Struct, Vec or External. Struct and Vec may have links to other variables.
EnumValMap	-	A hasmap: `HashMap<i64, String>` following the definition of enum def in the debug dialect.

Table 5.1: Nodes and respective children of the TyVcd intermediate data structure

5.1.1.2. TyVcd builder

Tywaves-rs implements `TyVcdBuilder<T>`, a parametric struct that provides methods to build `TyVcd` from another input type `T` (the IRs in Figure 5.1). To define the methods of a Rust structs the `impl` keyword should be used. In the case of a struct with generics, it can be implemented for a generic type and for a specific concrete type. The first case happens when `impl TyVcdBuilder<T>` is used, while the second happens with `impl TyVcdBuild<MyStruct1>` or `impl TyVcdBuild<MyStruct2>` and we refer to these cases as struct specializations for that specific types. Since the purpose of this builder is to bridge only supported debug formats with TyVcd, the builder is not implemented to handle the generic case.

To improve the robustness and coherence of different `TyVcdBuiler` specializations, Tywaves-rs defines a `GenericBuilder`trait which provides a fixed interface for every builder specialization. It is reported in Listing 5.1 together with an example of `TyVcdBuilder` specialization for generating `TyVcd` from HGLDD.

```rust
pub trait GenericBuilder {
    fn build(&mut self) -> Result<()>;
    fn get_ref(&self) -> Option<&TyVcd>;
    fn get_copy(&self) -> Option<TyVcd>;
}

/// A concrete builder for the TyVcd object.
pub struct TyVcdBuilder<T> {
    // The input list of objects from which the TyVcd object will be built
    origin_list: Vec<T>,
    // The target TyVcd object
    tyvcd: Option<TyVcd>,

    // Cache the enum definitions
    enum_def_map: EnumDefMap,
}

impl GenericBuilder for TyVcdBuilder<hgldd::Hgldd> {
    // Implementation ...
}
```

Listing 5.1: The `GenericBuilder`

Example of building Tyvcd from HGLDD

To give an illustration of how a specialized builder can be implemented, in this case, to enable Tydi-Chisel waveforms, the `TyVcdBuilder<hgldd::Hgldd>` is defined in Tywaves-rs. As introduced by Figure 5.1 the output debug information of CIRCT is passed as input and parsed using the `serde` library which allows to easily serialize and deserialize Rust data structures. In the specific case of HGLDD the JSON module is used to automatically deserialize it. Moreover, CIRCT outputs one HGLDD data structure per module definition either in a single or multiple files, therefore the HGLDD parser in Tywaves-rs is able to read over multiple files and convert them into a single intermediate representation.

HGLDD is deserialized into a `hgldd::Hgldd` data structure that reflects the layout of the file. The specialized TyVcdBuilder reads through the deserialized HGLDD and extracts all useful information needed to populate TyVcd. Since the module definitions and struct definitions are stored in the same list, a DFS approach would complicate the work. In fact, in HGLDD, these definitions are pointed by a variable or instance declared in other modules but the definition itself does not point back to its module since it can be used in multiple places. Therefore, at the first iteration, all module and struct definitions are cached into a map and then traversed. This way, any time a variable or instance is encountered, the id name can be used to access the respective definition with $O(1)$ complexity. Then, the respective TyVcd entry is created following the meanings in Table 5.1. For instance, all the module definitions and instances are defined as ScopeDef in TyVcd, similarly, variables and struct definitions are defined through Variable and VariableKind.

5.1.1.3. Vcd rewriter

Surfer loads simulation values from a trace file using Wellen [41], a library that provides an efficient interface to read multiple trace formats. As shown in Figure 5.1, these signals are converted into a Rust data structure and loaded into a wave container by Surfer. After that, when a signal is selected by a user from the GUI, the wave container passes the meta-data of the signal, stored in the IR, to the selected translator for rendering its raw value. This flow might cause some issues when the fields of aggregated variables are flattened like in the case of Chisel-CIRCT compilation. Flattening occurs when subfields of struct-like or vec-like variables are rewritten as multiple independent signals as shown by Listing 5.2 and 5.3 which show the Verilog representation and corresponding VCD header of `wireBundle` from Listing 2.4.

```
// wireBundle.a
logic wireBundle_a;
// wireBundle.b
logic [6:0] wireBundle_b;
// wireBundle.nested.x
logic [7:0] wireBundle_nested_x;
// wireBundle.v
logic [31:0] wireBundle_v_0,
    wireBundle_v_1, wireBundle_v_2,
    wireBundle_v_3, wireBundle_v_4,
    wireBundle_v_5, wireBundle_v_6;
```

Listing 5.2: The flattened fields of `wireBundle` from Listing 2.4

```
$var wire 1 # wireBundle_a [0:0] $end
$var wire 7 ) wireBundle_b [6:0] $end
$var wire 8 ( wireBundle_nested_x [7:0] $end
$var wire 32 / wireBundle_v_0 [31:0] $end
$var wire 32 & wireBundle_v_1 [31:0] $end
$var wire 32 % wireBundle_v_2 [31:0] $end
$var wire 32 $ wireBundle_v_3 [31:0] $end
$var wire 32 ? wireBundle_v_4 [31:0] $end
$var wire 32 " wireBundle_v_5 [31:0] $end
$var wire 32 ! wireBundle_v_6 [31:0] $end
```

Listing 5.3: The corresponding var entries dumped in VCD

In this case, Surfer loads each subfield as a separate value and rendering the whole `wireBundle` with preserved hierarchy is impossible. That said, a workaround consists of rewriting the input VCD file such that the value entries corresponding to subfields of aggregates are concatenated and dumped again as a single VCD variable. This way, the value of the whole aggregate can be passed to the translator, and the concatenation is decoded based on the information contained in TyVcd.

It is worth noticing that this solution is not optimized for performance because it needs one additional file write/read operation. A better implementation would consist of changing the flow for accessing the values from a trace file in order to allow the user to select the signals based on TyVcd and query the traces to get the raw values, in other words, the opposite way of the current flow. However, this necessitates either changes in Wellen to support TyVcd or in the internals of the wave container. Moreover, it would break the translator system implemented in Surfer requiring additional work. Since the main objective of the thesis is to provide a more intuitive debugging experience closer to the source language rather than the fastest signal rendering, this second solution is left for future improvements.

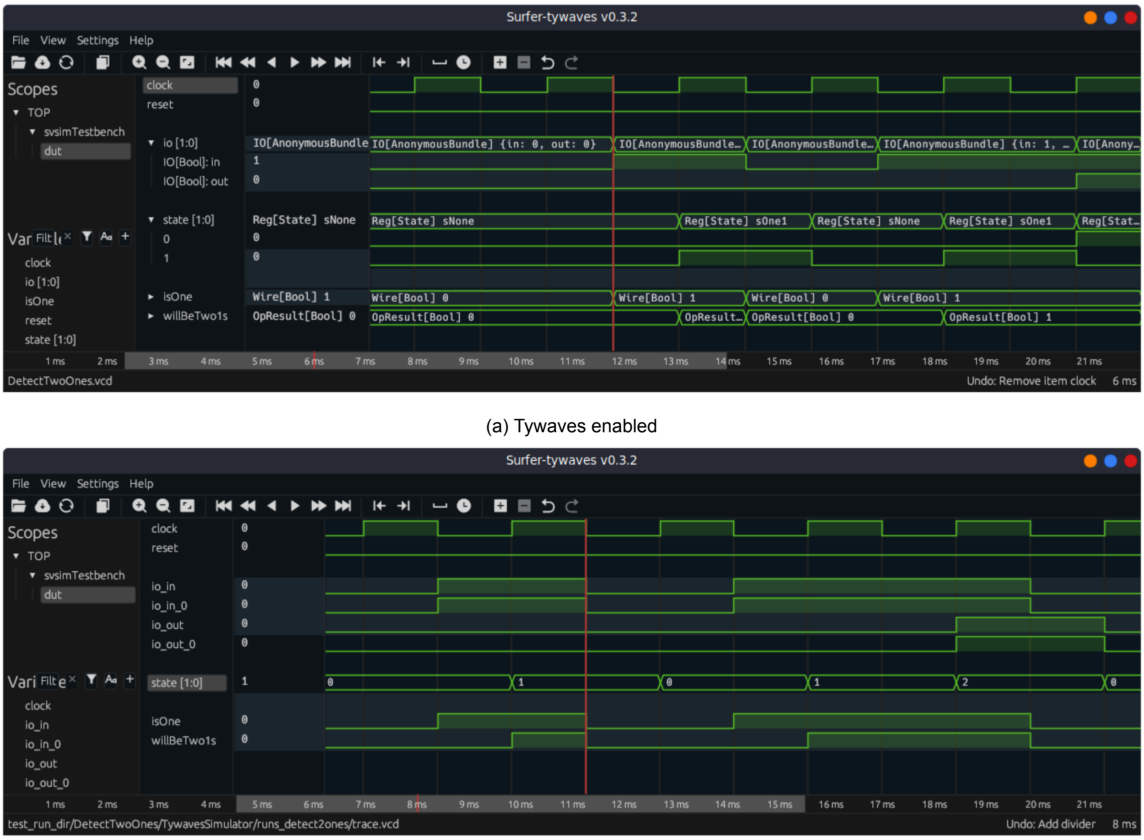
5.1.2. Tywaves translator

The final component that is implemented to give support for a type-based representation in Surfer is presented in this subsection. As introduced at the beginning of this section, the translator system controls the rendering of the raw values of signals. Specifically, multiple translators can be defined to show different customized visualizations of signals, i.e. integer or character values instead of a bit-vector. Each of them extends the same `Translator` trait providing a common interface for the wave container. As depicted by Figure 5.1, the `TywavesTranslator` is responsible for accessing the TyVcd data structure and processing the value in order to visualize signals with source-level types and hierarchies. The translator unpacks the concatenated values, rebuilds the hierarchies of aggregates, and shows the type information defined by Tywaves in Section 2.4.

This project focuses on Chisel and Tydi-Chisel, therefore an example of the output result and comparison with the former visualization are illustrated in Figure 5.2. As can be seen when the debug information is loaded and the `Tywaves` enabled, the hierarchies of the signals are kept and types are displayed next to the values. Thus, the variable info can be retrieved immediately from the viewer. The visualization of Figure 5.2a reduces the gap of the waveforms with the source code of `Detect2Ones` compared to Figure 5.2b. Named values of enum variants are displayed instead of a number, relieving the user from the tedious task of manually interpreting the raw value. In addition, the translator provides multiple representations for a value. This allows to distinguish between bits, integers, characters, float, and more representations for a value while still preserving the hierarchies and type information.

5.2. Tywaves-Chisel API

The last component highlighted in the general software architecture of the Tywaves project of Figure 3.1 is a ChiselSim API that hides the details for using together the updated compilation backend and Surfer for Tywaves and completes the generalized HGL simulation flow of Figure 2.7 for Tydi-Chisel. As initially shown by Figure 2.4, the two core components of ChiselSim (svsim and the PeekPoke API) can be combined to create high-level simulators that developers can use to write testbenches and launch tests. As mentioned, the PeekPoke API of ChiselSim [14] allows an easy transition from ChiselTest [40]



(a) Tywaves enabled

(b) Only VCD loaded

Figure 5.2: Comparison of waveforms when only VCD is loaded and when Tywaves is enabled for the FSM of Figure 2.6a

and implements methods to control the simulation, feed and read IO signals. However, the state of the art of ChiselSim implements only simple `EphemeralSimulator` for ephemeral simulations (see Appendix B). This simulator only enables running a simulation and it does not keep any information about the test as the name suggests, like emitting a trace. Moreover, no control settings for the underlying CIRCT compiler are available with high-level abstractions, but they can only be used by manually using svsim and firtool. Thus, none of the features introduced in Chapter 4 are available in ChiselSim yet.

This section presents two new simulators, shown in Figure 3.1 to improve the current abstractions of ChiselSim:

- **Parametric simulator**: a simulator for parametrizing the ChiselSim simulations through a set of predefined parameters, referred to as simulator settings.
- **Tywaves simulator**: an extension of the previous simulator that calls the tools in order to generate and process Tywaves information and launch the updated waveform viewer directly from a ChiselSim testbench.

While `TywavesSimulator` is tightly coupled with the proposed debug compilation pipeline and updates to Surfer, `ParametricSimulator` is more generic and it cannot access Tywaves features. Table 5.2 sums up the simulator settings available. Two interesting settings are `WithFirtoolArgs` and `WithTywavesWaveforms` since they allow to customize the compilation of the simulated Verilog coded and generate debug information with updated Tywaves features directly from the testbench.

Running a Tywaves simulation

To conclude the section, an example of usage of the API to enable simulation and a code comparison with the existing `EphemeralSimulator` in ChiselSim and ChiselTest are provided. Both ChiselSim and ChiselTest can be integrated into the ScalaTest framework, thus providing good IDE support for launching unit tests.

Setting	Description	SupportedSimulator
`VcdTrace`	Enable the VCD output optimizing out signals starting with an underscore (_) in the final verilog	Both
`VcdTraceWithUnderscore`	Enable the VCD output (including ”underscored” signals)	Both
`SaveWorkdir`	Save the workdir of ‘svsim‘ with a name based on the current timestamp of the simulation	Both
`SaveWorkdirFile(name: String)`	Save the workdir with a specific name	Both
`NameTrace(name: String)`	Save the VCD trace with a custom name	Both
`WithFirtoolArgs(args: Seq[String])`	Pass arguments to the CIRCT compiler under the simulation	Both
`WithTywavesWaveforms(runWaves: Boolean)`	Enable the generation of extra debug information (to fully exploit the tywaves project) and (optionally *runWaves=true*) launch the waveform viewer directly once the simulation has been completed	Only TywavesSimulator
`WithTywavesWaveformsGo(runWaves: Boolean)`	Same as `WithTywavesWaveforms` but without blocking sbt if `runWaves` is `true`	Only TywavesSimulator

Table 5.2: Simulator settings for Parametric and Tywaves simulators

Listing 5.5, 5.6 and 5.7 reports specific code examples for testing `Detect2Ones` of Figure 2.6a with the `TywavesSimulator`, `EphemeralSimulator` and ChiselTest. As can be noticed, the three examples have a really similar code and they are all compatible with the same peek poke test function of Listing 5.4. Therefore, switching between testing frameworks or simulators is trivial since, apart from a few differences in the syntax, the code is almost identical.

```scala
// Define a test function using the PeekPoke API. It can be used both in ChiselSim
    and ChiselTest without any changes
def runTest(fsm: DetectTwoOnes) = {
  // Inputs and expected results
  val inputs   = Seq(0, 0, 1, 0, 1, 1, 0, 1, 1, 1)
  val expected = Seq(0, 0, 0, 0, 0, 1, 0, 0, 1, 1)

  // Reset and run
  fsm.io.in.poke(0)
  fsm.clock.step(1)
  for (i <- inputs.indices) {
      fsm.io.in.poke(inputs(i))
      fsm.clock.step(1)
      fsm.io.out.expect(expected(i))
  }
}
```

Listing 5.4: The testbench function using peek/poke can be used both in ChiselSim and ChiselTest testbench

```scala
import tywaves.simulator._
import tywaves.simulator.simulatorSettings._
import TywavesSimulator._
// Scala test
class DetectTwoOnesTest extends AnyFunSpec with Matchers {
  describe(”TywavesSimulator”) {
    it(”runs DetectTwoOnes correctly”) {
      simulate(
        new DetectTwoOnes(),
        settings = Seq(VcdTrace, WithTywavesWaveforms(true)), // List of simulator
            settings
        simName = ”runs_detect2ones”
      ) { dut => runTest(dut) }
    }
  }
}
```

Listing 5.5: Testing `Detect2Ones` in ChiselSim with `TywavesSimulator` to enable source-level waveform debugging

```
import chisel3.simulator.EphemeralSimulator._
// Scala test
class DetectTwoOnesTest extends AnyFunSpec with Matchers {
  describe("EphemeralSimulator") {
    it("runs DetectTwoOnes correctly") {
      simulate(new DetectTwoOnes()) { fsm => runTest(fsm) }
    }
  }
}
```

Listing 5.6: Testing `Detect2Ones` in ChiselSim with `EphemeralSimulator` to simply run a test

```
import chiseltest._
// Scala test
class DetectTwoOnesTest extends AnyFunSpec with Matchers with ChiselScalatestTester
    {
  describe("ChiselTest") {
    it("runs DetectTwoOnes correctly") {
      test(new DetectTwoOnes())
        .withAnnotations(Seq(WriteVcdAnnotation)) { fsm => runTest(fsm) }
    }
  }

}
```

Listing 5.7: `Detect2Ones` in ChiselTest to generate VCD traces (only at Verilog level)

Finally, the integration and compatibility with ScalaTest enables abstractions to launch and divide tests in Scala test classes and test cases to further improve organization and readability. Moreover, this guarantees the support of some IDE functionalities, i.e. running a single test or multiple by pressing a button from the interface of IntelliJ IDEA [34] or using the command line sbt `test` or `testOnly` options [3].

6 | Results

The updates in the Chisel and CIRCT repositories, the tywaves-rs and the Tywaves-Chisel API discussed in the previous chapters contribute to the development of Tywaves: a type-based waveform viewer. Specifically, the implementation is intended to be used for testing real Chisel designs. Therefore, this chapter contains an evaluation of whether and how the output reflects the source code representation and a comparison with the standard visualization. However, since the main focus of a viewer is to show signal values, the designs used as evaluation metrics do not need to implement any fancy or complicated behavior. According to this, three tests are made:

- The Chisel constructs used in the presentation paper [11] are evaluated in Section 6.1.
- The Tydi representation is inspected using the example of Tydi-Chisel implementing a simple pipeline in [23] in Section 6.2.
- A critical case of conflicting Verilog names.

6.1. Evaluating Chisel constructs

Chisel's paper [11] presented the highlighting features when the language was introduced. The *Data Types Overview* section in its repository [15] and the documentation of the official main page [16] have introduced updates to the data types. Among them, the data types evaluated here are:

- Basic data types: `UInt`, `SInt` and `Bool`;
- Aggregate data type: `Bundles` and `Vec` also nested;
- Enumerations;
- Temporary signals or values.

6.1.1. Behavior of the design

Since the main purpose of this section is not to test the behavior of a specific circuit but rather to show the difference in waveforms, we decided to reproduce a design that is easy to understand. A FIFO (First-In-First-Out) implements a queue where elements that are pushed first are also the first to be processed. In other words, a FIFO is an ordered memory block where elements are inserted (enqueue) on one side and taken (dequeue) from the other side of the memory. Listing 6.1 implements the code of the module retrieved from the book *"Digital Design with Chisel"* [52, *p. 159-171*]. The circuit is simple; writing and reading interface types are declared at the top of the code as `WriterIO` and `ReaderIO`, and a `Buffer` module with these interfaces is created to store a single element, representing a memory block. Finally, the top module `BubbleFifo`, implementing the same IO interface, instantiates and connects multiple buffers.

Internally, each buffer preserves a state that tells whether it is full or empty, thus if the register can be written or read. This state is consequently "exposed" through the IO interfaces. As a consequence, when a writing or reading operation is requested (`io.enq.write` and `io.deq.read` true), either externally or from another connected buffer, the `dataReg` can be written or read when the state is empty or full respectively.

Behaviorally, when a write operation is requested, the value is inserted in the first buffer and shifted to the next every cycle. Insertion and shift operations can be executed if the buffers to be written are empty. With this kind of FIFO, an output is available after `depth` cycles, where `depth` is the size of the FIFO.

For completeness of the types coverage, the `BubbleFifo` is wrapped into another module shown in Listing 6.2 (`Collector`) which implements the same IO interface and collects all the values inserted in the FIFO in a 2D Chisel vector to cache the history of the FIFO.

```scala
class WriterIO[T <: Data](private val gen: T) extends Bundle {
  val write = Input(Bool())
  val full  = Output(Bool())
  val din   = Input(gen)
}

class ReaderIO[T <: Data](private val gen: T) extends Bundle {
  val read  = Input(Bool())
  val empty = Output(Bool())
  val dout  = Output(gen)
}

// Store an element of the FIFO
class Buffer[T <: Data](gen: T) extends Module {
  val io = IO(new Bundle {
    val enq = new WriterIO(gen)
    val deq = new ReaderIO(gen)
  })

  object StateBuff extends ChiselEnum { val EMPTY, FULL = Value }

  val stateReg = RegInit(StateBuff.EMPTY) // Flag
  val dataReg  = Reg(gen) // Store the data of the buffer

  when(stateReg === StateBuff.EMPTY) {
    // Temporary signal
    val nextState = Mux(io.enq.write, StateBuff.FULL, StateBuff.EMPTY)
    when(io.enq.write) {
      dataReg := io.enq.din
    }
    stateReg := nextState
  }.elsewhen(stateReg === StateBuff.FULL) {
    when(io.deq.read) {
      stateReg := StateBuff.EMPTY
      dataReg  := DontCare // just to better see empty slots in the waveform
    }
  }.otherwise { /* There should not be an otherwise state */ }
  io.enq.full  := (stateReg === StateBuff.FULL)
  io.deq.empty := (stateReg === StateBuff.EMPTY)
  io.deq.dout  := dataReg
}

class BubbleFifo[T <: Data](gen: T, depth: Int) extends Module {
  val io = IO(new Bundle {
    val enq = new WriterIO(gen)
    val deq = new ReaderIO(gen)
  })
  val buffers = Array.fill(depth)(Module(new Buffer(gen)))
  for (i <- 0 until depth - 1) {
    buffers(i + 1).io.enq.din   := buffers(i).io.deq.dout
    buffers(i + 1).io.enq.write := ~buffers(i).io.deq.empty
    buffers(i).io.deq.read      := ~buffers(i + 1).io.enq.full
  }
  // Connect head and tail
  io.enq <> buffers(0).io.enq
  io.deq <> buffers(depth - 1).io.deq
}
```

Listing 6.1: `BubbleFifo` example from *"Digital Design with Chisel"* [52, *p. 159-171*]

```
class Collector[T <: Data](gen: T, depth: Int) extends Module {
  val io = IO(new Bundle {
    val enq = new WriterIO(gen)
    val deq = new ReaderIO(gen)
  })
  val fifo = Module(new BubbleFifo(gen, depth))
  io.enq <> fifo.io.enq
  io.deq <> fifo.io.deq

  val history = Reg(Vec(depth, Vec(depth, gen)))
  val readCounter = new Bundle { val i, j = Counter(depth) }

  when(io.enq.write && !io.enq.full) {
    history(readCounter.i.value)(readCounter.j.value) := io.deq.dout
    readCounter.j.inc()
    when(readCounter.j.value === (depth - 1).U) {
      readCounter.j.reset()
      readCounter.i.inc()
    }
  }
}
```

Listing 6.2: `Collector` example implementing a 2D Chisel vector

6.1.2. Waveforms

The FIFO design example uses all the constructs listed at the beginning of this section. Specifically, it uses both user-defined (`WriterIO` and `ReaderIO`) and anonymous bundles, a chisel enumeration to represent the state of a buffer and a temporary value result of an operation (`nextState`). Moreover, `io` anonymous bundle represents a nested structure in the code. As shown throughout the previous chapters, this information is lost once Chisel is compiled into Verilog, either due to the compiler optimizations or limitations of the classic HDL.

To evaluate the waveforms generated when Tywaves is enabled, this section compares the Tywaves output of a testbench of the `Collector` example with the case when only VCD is loaded.

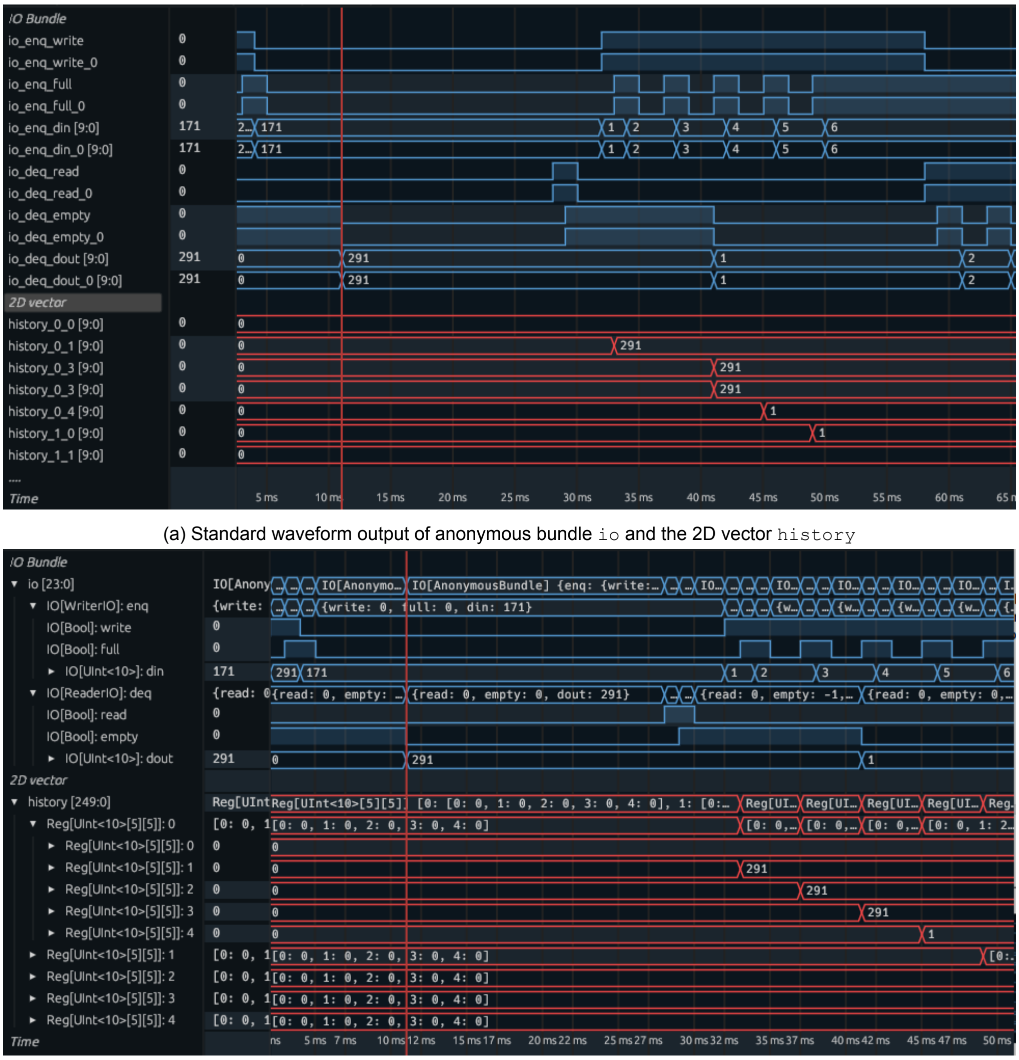
6.1.2.1. Basic types and aggregates

First of all, the basic and aggregate data types are analyzed in Figure 6.1. As can be seen, the Tywaves waveforms (Figure 6.1b) keep the same appearance as the original code. The fields of the bundles in `io` (blue signals) are grouped together as a tree, this way the structure as well as the field names match the original representation. Moreover, as addressed by the research questions of Section 1.3, types are successfully displayed next to the signal values. This helps to make a clear distinction between types. For instance, `io` is marked as an `AnonymousBundle` while `io.enq` and `io.deq` are shown as `WriterIO` and `ReaderIO`. Similarly, the red signals represent the 2-dimensional vector `history` and reflect the indexed view.

On the other hand, Figure 6.1a reflects the target Verilog view without abstractions. In fact, no Scala-type information is available, `io.deq.empty` is not shown as `Bool` and `io.deq.dout` is not a `UInt<10>` to give an example. Fields of `io` and elements of `history` are represented as parallel signals without any grouping. This view, not only does not correspond with the source code, but it also may cause an explosion of signals, making the analyses of the values difficult especially with huge vectors, i.e. in the case of importing a whole vector, each signal must be imported separately and it cannot be collapsed. Whereas, Tywaves view allows to collapse and expands both struct-like and vec-like aggregates. Finally, additional artifacts generated by the compiler and emitted in Verilog and VCD are displayed when Tywaves is not enabled, Tywaves removes them from the view.

6.1.2.2. Hardware type and binding

Section 2.4 introduced a distinction between hardware and Scala type. In the waveforms, this is shown as the binding of the type (`Bind[Type]`). To give an illustration, the full types `IO[Bool]` and `Reg[UInt<10>]` from Figure 6.1b make a clear distinction also about the hardware type of a



(a) Standard waveform output of anonymous bundle `io` and the 2D vector `history`

(b) Tywaves output of anonymous bundle `io` and the 2D vector `history`

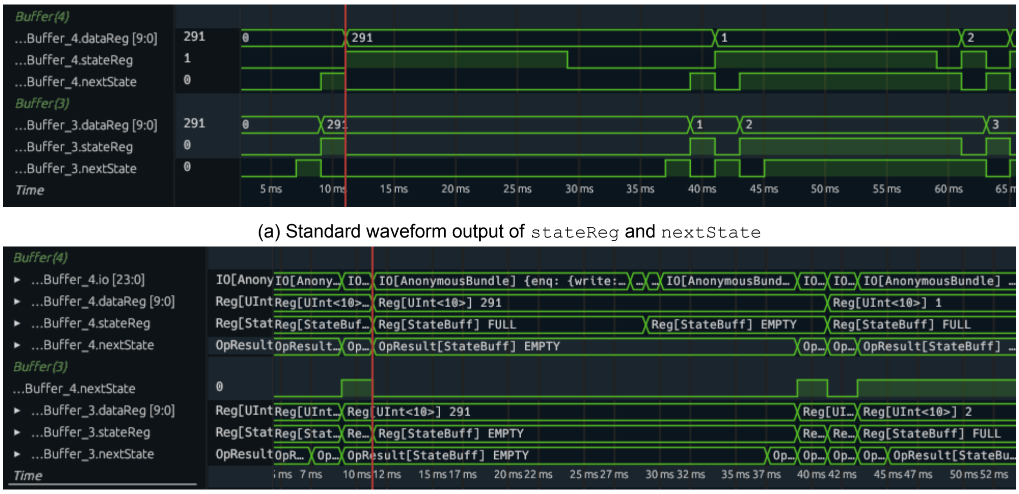
Figure 6.1: Comparison of Tywaves and standard waveform outputs for the basic data types and aggregate types of `Collector` (Listing 6.2)

signal. Such information is not available in a VCD file, either because of transformations and optimizations performed by the CIRCT compiler or because the specific hardware type is missing from the VCD format.

Sometimes, users assign temporary results to Scala `val` like shown in Listing 6.1 where `nextState` is declared inside a `when` statement and assigned from an operation directly. This kind of operation is translated to a signal in Verilog. As a consequence, it is shown using `wire` type. However, Chisel has a different representation for this binding called `OpResult`. This difference is highlighted by Figure 6.2 and Figure 6.2b proves that Tywaves is able to retrieve and display both custom Scala data types, hardware types, and custom bindings.

6.1.2.3. Enumerations

Finally, Figure 6.2 shows that Tywaves is able to successfully render the named variant of enumeration. The figure reports the case of `stateReg` in `BufferFifo` which is of type `StateBuff` whose values are `EMPTY` and `FULL`. When the raw values are displayed instead, as in Figure 6.2a, designers must manually map the number to the actual state name by looking at the order of the states in the ChiselEnum declaration. Although it does not seem a problem with small enums, this association may not be trivial for large enumerations.



(a) Standard waveform output of `stateReg` and `nextState`

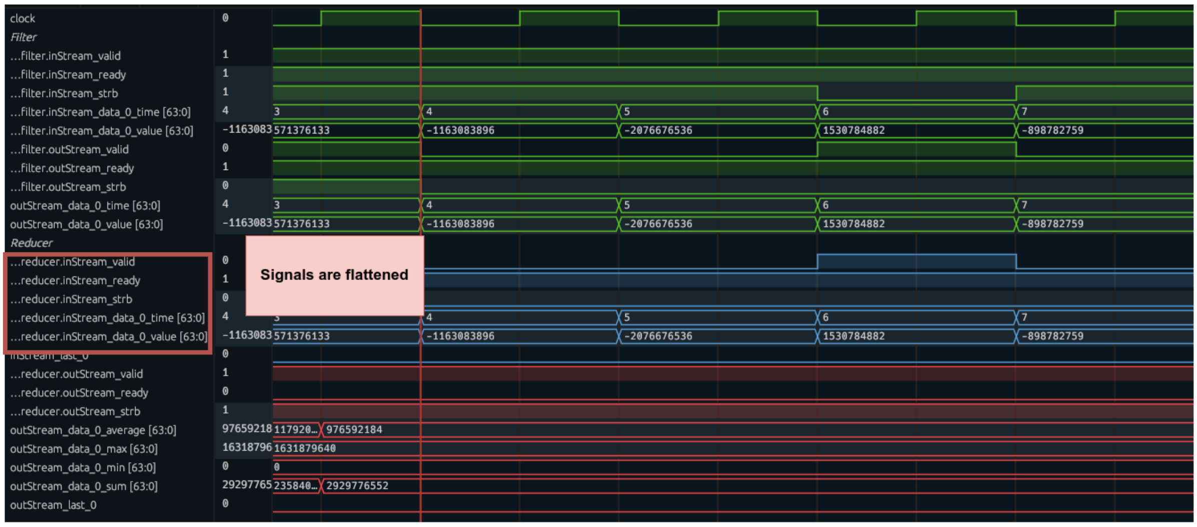
(b) Tywaves output of `stateReg` and `nextState`

Figure 6.2: Comparison of Tywaves and standard waveform outputs for enumeration and temporary values in `BufferFifo` (Listing 6.1)

6.2. Evaluating Tydi-Chisel example

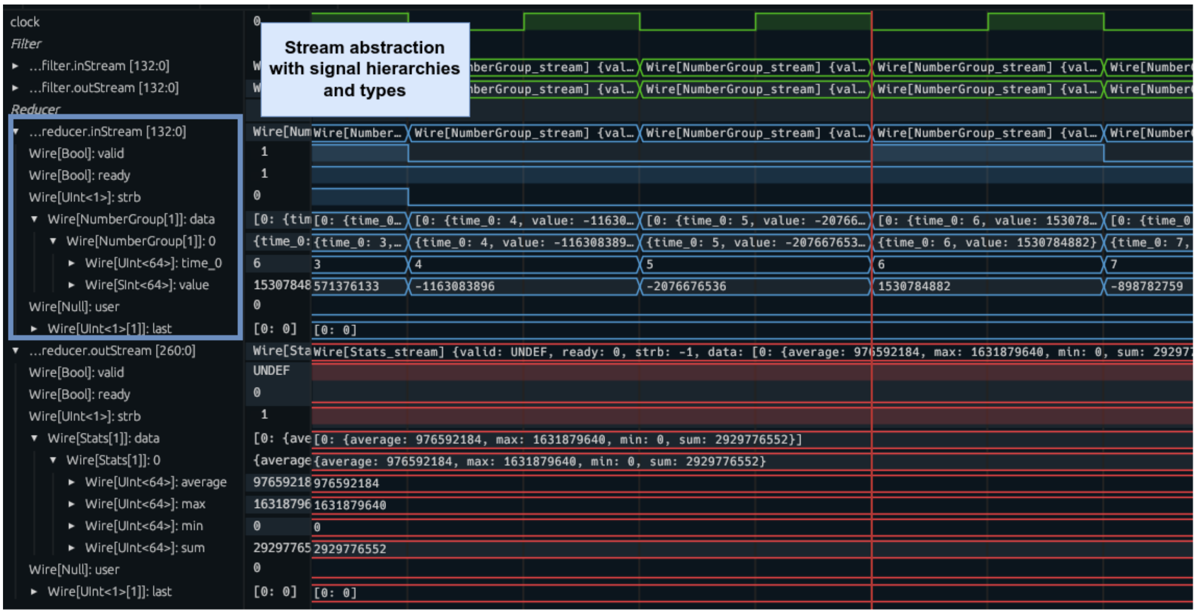
As introduced by Section 2.1 and Section 2.4 types cover a crucial role in Tydi circuits and streams. Figure 6.3 the waveform results (VCD only and Tywaves) of the `PipelineExample` implementing the Spark code of Listing 2.1. From a comparison of the two outputs, it is clear that there is no actual distinction between the two types of stream in the Reducer module, highlighted in blue and red in Figure 6.3a. Although the difference is clearly visible from the Tydi-lang and Chisel code (full code in Appendix A), the designer can get it only from the signal names.

In Tywaves instead, streams encapsulate the control signals and data buses in a single collapsable namespace. This result is expected after the results of Section 6.1 since Tydi-Chisel implements streams as extensions of Chisel bundles. All signals have their own type and the data bus of the stream reports the actual type of the value exchanged over the stream. For instance, `reducer.inStream.data` is represented as a `NumberGroup`, including its hierarchical view while in the classic waveforms, all the signals contained in the stream are flattened, causing an explosion of parallel signals. Tywaves avoids the issue of creating confusion by representing these signals in a grouped and formatted manner.


clock
Filter
...filter.inStream_valid
...filter.inStream_ready
...filter.inStream_strb
...filter.inStream_data_0_time [63:0]
...filter.inStream_data_0_value [63:0]
...filter.outStream_valid
...filter.outStream_ready
...filter.outStream_strb
outStream_data_0_time [63:0]
outStream_data_0_value [63:0]
Reducer
...reducer.inStream_valid
...reducer.inStream_ready
...reducer.inStream_strb
...reducer.inStream_data_0_time [63:0]
...reducer.inStream_data_0_value [63:0]
inStream_last_0
...reducer.outStream_valid
...reducer.outStream_ready
...reducer.outStream_strb
outStream_data_0_average [63:0]
outStream_data_0_max [63:0]
outStream_data_0_min [63:0]
outStream_data_0_sum [63:0]
outStream_last_0
Signals are flattened
571376133
-1163083896
-2076676536
1530784882
-898782759
976592184
1631879640
2929776552

(a) Standard waveform output of a Tydi-Chisel stream

Figure 6.3: Comparison of Tywaves and standard waveform outputs for Tydi streams using the example retrieved from Listing 2.1 and Appendix A



(b) Tywaves output of a Tydi-Chisel stream

Figure 6.3: Comparison of Tywaves and standard waveform outputs for Tydi streams using the example retrieved from Listing 2.1 and Appendix A

6.3. Circuit with conflicting names in the final Verilog

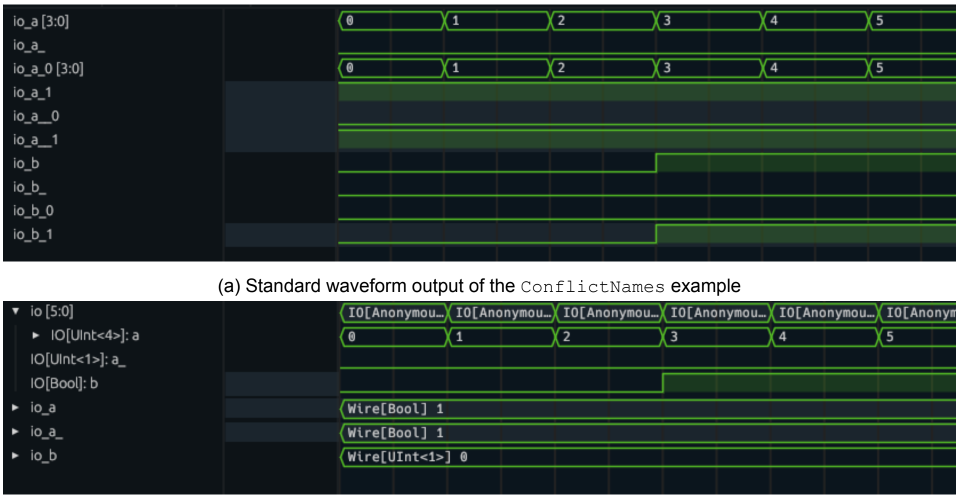
Finally, one critical case of debugging Chisel circuits with standard waveforms is when the serialization of source code names creates conflicting names in the output Verilog. For instance, this is something that happens with the code of Listing 6.3.

```
class ConflictNames extends Module {
    val io = IO(new Bundle {
      val a  : UInt = Input(UInt(4.W))
      val a_ : UInt = Input(UInt(1.W))
      val b  : Bool = Output(Bool())
    })

    // Variables causing conflicts
    val io_a  = Wire(Bool())
    val io_a_ = Wire(Bool())
    val io_b  = Wire(UInt(1.W))

    io_a  := true.B
    io_a_ := true.B
    io_b  := 1.U

    io.b := a > 2.U
}
```

Listing 6.3: Chisel example which leads to conflicting names when serialized to verilog

The Chisel compiler tries to serialize io.a and io_a with the same name but they conflict. Therefore, it appends an additional underscore character at the end of the signal name, resulting in io_a and io_a_. In the example, this later leads to a new conflict between io.a_ and io_a_ both tried to be serialized with the same name. This process is repeated by the compiler until all the conflicts are solved. Unfortunately, the user cannot know to what source variable the names io_a_, io_a__0, io_a__1 and so on refer to and they may not know at all how the compiler works. Thus, the view of standard waveforms in Figure 6.4a might create confusion while debugging. On the contrary, Figure 6.4b shows the same waveforms with Tywaves and proves how it is able to reconstruct the source view also in this case.

This example has been created on purpose to show that Tywaves solves this issue, but it is something that might happen in real-case scenarios. Since the source code allows to have those situations a waveform viewer for Chisel should handle it.



(a) Standard waveform output of the `ConflictNames` example

(b) Tywaves output of the `ConflictNames` example

Figure 6.4: Comparison of Tywaves and standard waveform outputs in the case of conflicting names (Listing 6.3)

6.4. Drawbacks

The results above proved how Tywaves is able to reduce the gap with the Chisel source code by keeping the same signal hierarchies, transmitting and showing the type information, properly displaying enumeration variants and assessing custom hardware bindings. However, the frontend is not able yet to display any type information for the module types type although this information is available to the viewer and successfully propagated as explained in Chapter 4.

Furthermore, the viewer does not allow to load subfields of aggregates separately from the others. This is due to the current limitation of Surfer which reads the values from the VCD input and asks the selected translator for a custom rendering. As explained in Subsubsection 5.1.1.3 a VCD rewrite has been implemented to permit the custom representation of signals. Updating Wellen would have required additional changes, shifting the focus of the project from finding a way of propagating debug information though the compilation pipeline.

Finally, even though Tydi streams have benefited from a type-based representation compared to the classic waveforms, better abstractions could be used in Surfer to represent Tydi streams. For instance, showing only what is passing over a stream without details about the control logic.

7 | Summary, conclusions and recommendations

7.1. Summary

This thesis presents a novel kind of type-based waveform viewer to raise the abstraction level for debugging modern HDLs with typed circuit components. A new debug format for reconstructing the source level view and displaying custom data types is defined to reduce the gap between the source code and waveforms with Chisel and Tydi-Chisel circuits.

In Chapter 2, we discussed the Tydi specification and its integration within Chisel for defining and exchanging custom data types between components and over hardware streams. This led to the definition of typed circuit components in the Chisel language, identifying modules and signals as typed constructs. The simulation and compilation flow of Chisel is described and the current work for generating debug information through the CIRCT debug dialect and the HGLDD file format is presented. Moreover, we gave a brief overview of how HGL/HDL simulators work and how they can be combined with waveform viewers. On top of that, a generalized simulation flow for high-level hardware languages is discussed. Finally, the end of the chapter presents previous and current developments in testing and debugging tools for classic and modern hardware languages.

The implementation requirements to collect and elaborate type information, to pass the information to a waveform viewer, and to keep compatibility with the Chisel simulation library were presented in Chapter 3. We examined two different alternatives that could contribute to the final functionality. The observations made in the chapter suggested that a solution integrated within the Chisel-CIRCT compilation was preferable over an external tool to target maintainability, stability, and performance. Therefore, the updated simulation flow is described and the Tywaves software architecture is outlined.

Integrating the proposed functionality within the compiler is challenging. It requires a deep understanding of the internal implementations and finding proper methods to exchange extra information. Chapter 4 illustrates how this is achieved for Tywaves. The Chisel library is updated first to generate type information from the source and associate it with the IR input of CIRCT through the FIRRTL annotations. Then, the updates to CIRCT are implemented to integrate the new information in the existing MLIR debug dialect and to emit a new file format, originating from HGLDD, to associate the source type view with the final output traces. Throughout the chapter, an alternative to FIRRTL annotations for future improvements is explored.

In Chapter 5, we discussed how Surfer is extended to handle the debug information emitted by Chisel and CIRCT. The chapter also illustrates the tywaves-rs library which is created to decouple the internal functionality of the viewer from the debug file. The library is built such that it can be extended to support other input formats. The debug input is parsed and converted to an internal data structure that is independent from any language. The end of the chapter presents a Chisel API both to improve the current high-level simulators of ChiselSim and use the Tywaves functionality from Scala.

At the end of the thesis, the visualization results and comparisons with standard waveform visualization are provided in Chapter 6. Two circuits of Chisel and Tydi-Chisel are tested to highlight the differences with classic waveforms and the advantages introduced by Tywaves. Finally, a critical case that creates conflicting names in the compiled HDL is tested. This last example illustrates how standard waveforms are not suitable for modern HDLs and proves the robustness of Tywaves.

7.2. Conclusions

The main research question was formulated starting from the challenges addressed at the beginning of the thesis and it asked: *How can a type-based waveform viewer be effectively developed to raise the abstraction level while debugging modern HDLs and HGLs, specifically reducing the gap between the waveform visualization and source code?*

To answer this main question, multiple subtopics have been defined:

1. *How can types be associated with values output from simulations and the source language be reconstructed from a compiled output not matching the source?*
   Subsection 4.1.1 and 4.1.4 accomplish the generation and collection of type information from Chisel. This information is subsequently elaborated in CIRCT through an updated MLIR dialect (debug dialect in Subsection 2.3.1) linking the compiled output with source types.

2. *What are the necessary steps to improve multiple compilers and tools involved in simulation of modern HDLs in order to create debug information for a waveform viewer?*
   This work presented updates in the simulation flow of Chisel (Section 3.3), targeting multiple phases and tools of Chisel compilation. Both the Chisel library and CIRCT compiler were updated to integrate new debug information into the existing debug dialect and HGLDD output. The sections also show what exact information is exchanged between the tools to emit a debug format (Subsection 4.2.4) for a waveform viewer which contributed to improve the overall simulation flow.

3. *How can types be displayed in a graphical user interface and how to show the source view?*
   The translator mechanism offered by the Surfer waveform viewer allows associating custom data types of signals with their values as explained in Section 5.1. The tywaves-rs library implemented an interface with the HGLDD file format for the reconstruction of Chisel source view from Surfer, including the types of the signals.

4. *How should types and, more in general, debug information be generated, encoded, and propagated throughout the compilation and simulation pipelines?*
   In Tywaves, type information is generated directly from the compilers used for Chisel. Then, it is encoded in FIRRTL IR, subsequently passed to CIRCT and re-emitted in the form of a documented debug file (HGLDD) to conclude the debug compilation (Chapter 4). Encoding information within the IR exchanged through the compilers ensured the communication of types within the pipeline.

5. *Due to the multiplicity and diversity of modern HDLs, can a method be found and defined to support multiple languages or, possibly, to extend support easily by re-utilizing much of the infrastructure so as to have a greater impact in the open-source community?*
   In the work presented in this thesis, the integration within CIRCT and Surfer guarantees the opportunity for future extensions with other languages. Both tools are open-source and widely used in the field, which contributes to having a bigger impact on the community. CIRCT implements the MLIR and LLVM design methodologies to embed multiple sources in the same compiler infrastructure. Surfer is thought for extensibility and its community is open to support new languages. Decoupling the various steps in the Tywaves pipeline enables easier integration of language pipelines and file formats with re-usage of parts of the tool.

7.3. Recommendations for future work

To conclude, based on the work done in this thesis and the results obtained, the future work and recommendations are presented below:

- Use intrinsics as a communication method to propagate type information between Chisel and CIRCT to replace the FIRRTL annotations used. Intrinsics are a new and more robust method to associate metadata in FIRRTL, more suitable with the MLIR operation mechanism. Moreover, this would not require changes in the debug dialect and file format but only in the reading interface which creates the debug operations.
- Create a new debug file format free from the code patterns and limitations of VCS-internals. HGLDD uses some names and methods that overcomplicate the format in the general case.
- Extend and test the infrastructure of Tywaves to support other languages that are integrated into CIRCT. As said, the whole backend can be re-used without changes with the exception that the language should generate the same FIRRTL intermediate representation. Otherwise, in the case of another IR, it could use the same updated debug dialect presented in this thesis.

- A performance improvement for tywaves-rs can be achieved by providing a parallelized version of the `TyVcdBuilder`. `TyVcd` already supports its employment in a multithreaded environment since it uses internally `Arc<RwLock<T>>`.
- A second performance improvement consists of removing the VCD rewriter step. It would require the integration of `TyVcd` within Wellen such that also the wave container in Surfer can access the generic HDL view also when hierarchical signals are flattened by compilers.
- The Tywaves pipeline provides type information also about module definitions of chisel circuits, following the type definition introduced in this thesis. A feature improvement consists of displaying this type in the UI since it is not currently enabled.
- Automatically choosing how to render a value based on the type name could be an interesting improvement. For instance, any type that may represent an integer like `UInt`, `unsigned`, `long` could have an integer representation automatically selected. Another approach could be defining an API that allows designers to select how to render a value, similarly to a `toString()` method used in many software languages.
- Enabling switching between multiple representations in the viewer would contribute in simplifying more the debugging experience. For instance, providing a functionality to switch from Chisel to FIRRTL to Verilog representation from the same Surfer session.
- Finally, another future work point could consist of supporting specific visualization abstractions for hardware streaming interfaces. This would help to make debugging Tydi streams simpler.

Bibliography

[1] GCC, the GNU Compiler Collection - GNU Project. URL `https://gcc.gnu.org/`. Accessed: 2024-07-25.

[2] GDB: The GNU Project Debugger. URL `https://sourceware.org/gdb/`. Accessed: 2024-07-25.

[3] Sbt Reference Manual — Testing. URL `https://www.scala-sbt.org/1.x/docs/Testing.html`. Accessed: 2024-07-21.

[4] WaveTrace VCD. URL `https://www.wavetrace.io/`. Accessed: 2024-07-25.

[5] Scala documentation: Scala reflection overview. URL `https://docs.scala-lang.org/overviews/reflection/overview.html`. Accessed: 2024-07-25.

[6] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 653–673, February 2018. doi: 10.1109/IEEESTD.2018.8299595. URL `https://ieeexplore.ieee.org/document/8299595`.

[7] Cocotb, 2024. URL `https://www.cocotb.org/`. Accessed: 2024-07-25.

[8] CHIPS Alliance. GITHUB chipsalliance/firrtl. CHIPS Alliance, December 2023. URL `https://github.com/chipsalliance/firrtl`. Accessed: 2024-07-25.

[9] CHIPS Alliance. Specification for the FIRRTL Language: Version 3.2.0, sep 2023. URL `https://github.com/chipsalliance/firrtl-spec/releases/latest/download/spec.pdf`.

[10] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. C?aSH: Structural Descriptions of Synchronous Hardware Using Haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pages 714–721, September 2010. doi: 10.1109/DSD.2010.21. URL `https://ieeexplore.ieee.org/document/5615430`.

[11] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225, San Francisco California, June 2012. ACM. ISBN 978-1-4503-1199-1. doi: 10.1145/2228360.2228584. URL `https://dl.acm.org/doi/10.1145/2228360.2228584`.

[12] Anthony Bybell. Appendix F: Implementation of an Efficient Method for Digital Waveform Compression. In *GTKWave 3.3 Wave Analyzer User's Guide*, pages 137–145. November 2020. URL `https://gtkwave.sourceforge.net/gtkwave.pdf`.

[13] Cadence. Virtuoso ADE Suite. URL `https://www.cadence.com/en_US/home/tools/custom-ic-analog-rf-design/circuit-design/virtuoso-ade-suite.html`. Accessed: 2024-07-25.

[14] Chisel. Migrating from ChiselTest | Chisel, . URL `https://www.chisel-lang.org/docs/appendix/migrating-from-chiseltest`. Accessed: 2024-06-30.

[15] Chisel. Home | Chisel, . URL `https://www.chisel-lang.org/`. Accessed: 2024-07-25.

[16] Chisel. Chipsalliance/chisel. CHIPS Alliance, January 2024. URL `https://github.com/chipsalliance/chisel`. Accessed: 2024-07-25.

[17] Fred Chow. Intermediate Representation: The increasing significance of intermediate representations in compilers. *Queue*, 11(10):30–37, October 2013. ISSN 1542-7730. doi: 10.1145/2542661.2544374. URL `https://doi.org/10.1145/2542661.2544374`.

[18] CIRCT. Debug Dialect - CIRCT. URL https://circt.llvm.org/docs/Dialects/Debug/#dbgscope-circtdebugscopeop.

[19] CIRCT. CIRCT - Circuit IR Compilers and Tools, 2024. URL https://circt.llvm.org/. Accessed: 2024-07-25.

[20] CIRCT. 'firrtl' Dialect - CIRCT, 2024. URL https://circt.llvm.org/docs/Dialects/FIRRTL/. Accessed: 2024-07-25.

[21] Casper Cromjongh. Ccromjongh/Tydi-Chisel, December 2023. URL https://github.com/ccromjongh/Tydi-Chisel. Accessed: 2024-06-30.

[22] Casper Cromjongh. Ccromjongh/tydi-lang-2-chisel, September 2023. URL https://github.com/ccromjongh/tydi-lang-2-chisel.

[23] Casper Cromjongh, Yongding Tian, Peter Hofstee, and Zaid Al-Ars. Tydi-Chisel: Collaborative and Interface-Driven Data-Streaming Accelerators. In *2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*, pages 1–7, Aalborg, Denmark, October 2023. IEEE. ISBN 9798350337570. doi: 10.1109/NorCAS58970.2023.10305451. URL https://ieeexplore.ieee.org/document/10305451/.

[24] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. ISSN 0164-0925. doi: 10.1145/115372.115320. URL https://doi.org/10.1145/115372.115320.

[25] Benjamin Darnell. Ben1152000/sootty, July 2024. URL https://github.com/Ben1152000/sootty. Accessed: 2024-07-25.

[26] John Demme, Fabian Schuiki, Mike Urbach, and Andrew Young. Charting CIRCT: The present and future landscape. URL https://llvm.org/devmtg/2021-11/slides/2021-ChartingCIRC-TThePresentAndFutureLandscape.pdf.

[27] Andrew Dobis, Tjark Petersen, Hans Jakob Damsgaard, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. ChiselVerify: An Open-Source Hardware Verification Library for Chisel and Scala. In *2021 IEEE Nordic Circuits and Systems Conference (NorCAS)*, pages 1–7, Oslo, Norway, October 2021. IEEE. ISBN 978-1-66540-712-0. doi: 10.1109/NorCAS53631.2021.9599869. URL https://ieeexplore.ieee.org/document/9599869/.

[28] Andrew Dobis, Kevin Laeufer, Hans Jakob Damsgaard, Tjark Petersen, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. Verification of Chisel Hardware Designs with ChiselVerify. *Microprocessors and Microsystems*, 96:104737, February 2023. ISSN 0141-9331. doi: 10.1016/j.micpro.2022.104737. URL https://www.sciencedirect.com/science/article/pii/S0141933122002666.

[29] FOSSi Foundation. Latch-Up 2024: April 19-21, 2024 in Cambridge, MA, USA. URL https://fossi-foundation.org/latch-up/2024. Accessed: 2024-07-17.

[30] Tristan Gingold. GHDL Waveform (GHW) - 4.0.0-dev. URL https://ghdl.github.io/ghdl/ghw/index.html. Accessed: 2024-07-25.

[31] GTKWave. GTKWave documentation. URL https://gtkwave.github.io/gtkwave/index.html. Accessed: 2024-07-25.

[32] John Hennessy and David Patterson. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–29, June 2018. doi: 10.1109/ISCA.2018.00011. URL https://ieeexplore.ieee.org/document/8416813.

[33] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 209–216, Irvine, CA, November 2017. IEEE. ISBN 978-1-5386-3093-8. doi: 10.1109/ICCAD.2017.8203780. URL http://ieeexplore.ieee.org/document/8203780/.

[34] JetBrains. IntelliJ IDEA – the Leading Java and Kotlin IDE. URL https://www.jetbrains.com/idea/. Accessed: 2024-07-25.

[35] Nachiket Kapre and Samuel Bayliss. Survey of domain-specific languages for FPGA computing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–12, Lausanne, Switzerland, August 2016. IEEE. ISBN 978-2-8399-1844-2. doi: 10.1109/FPL.2016.7577380. URL http://ieeexplore.ieee.org/document/7577380/.

[36] Lucas Klemmer and Daniel Große. WAVING Goodbye to Manual Waveform Analysis in HDL Design With WAL. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2024. ISSN 1937-4151. doi: 10.1109/TCAD.2024.3387312. URL https://ieeexplore.ieee.org/document/10496480.

[37] Jack Koenig. CCC22 - Chisel Breakdown 3, . URL https://docs.google.com/presentation/d/1gMtABxBEDFbCFXN_-dPyvycNAyFROZKwk-HMcnxfTnU.

[38] Jack Koenig. Latch-Up Conference presentation of ”Chisel 6 and beyond” - Jack Koenig (2024) - YouTube, . URL https://youtu.be/A5iz6mnPNW4?si=wudOacyveSVYMv9U.

[39] Dexter C. Kozen. *Depth-First and Breadth-First Search*, pages 19–24. Springer New York, New York, NY, 1992. ISBN 978-1-4612-4400-4. doi: 10.1007/978-1-4612-4400-4_4. URL https://doi.org/10.1007/978-1-4612-4400-4_4.

[40] Kevin Laeufer. Ucb-bar/chiseltest. UC Berkeley Architecture Research, June 2024. URL https://github.com/ucb-bar/chiseltest.

[41] Kevin Laeufer and Oscar Gustafsson. ekiwi/wellen: v0.9.14, July 2024. URL https://doi.org/10.5281/zenodo.12774825.

[42] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, March 2004. doi: 10.1109/CGO.2004.1281665. URL https://ieeexplore.ieee.org/document/1281665.

[43] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, February 2021. doi: 10.1109/CGO51591.2021.9370308. URL https://ieeexplore.ieee.org/abstract/document/9370308.

[44] Andrew Lenharth and Chris Lattner. CIRCT: Lifting hardware development out of the 20th century. *LLVM Developer Meeting*, 2021. URL https://llvm.org/devmtg/2021-11/slides/2021-CIRCT-LiftingHardwareDevOutOfThe20thCentury.pdf.

[45] LLVM. TableGen Overview — LLVM 19.0.0git documentation. URL https://llvm.org/docs/TableGen/. Accessed: 2024-06-30.

[46] Lramseyer. Lramseyer/vaporview, July 2024. URL https://github.com/Lramseyer/vaporview. Accessed: 2024-07-25.

[47] Dina Mahmoud. Hardware Acceleration. In Valentin Mulder, Alain Mermoud, Vincent Lenders, and Bernhard Tellenbach, editors, *Trends in Data Protection and Encryption Technologies*, pages 109–114. Springer Nature Switzerland, Cham, 2023. ISBN 978-3-031-33386-6. doi: 10.1007/978-3-031-33386-6_20. URL https://doi.org/10.1007/978-3-031-33386-6_20.

[48] Raffaele Meloni. Rameloni/tywaves-chisel-demo: A repository that implements Tywaves: Enabling a type-based waveform debugging for Chisel and Tydi-Chisel. Mapping from Chisel level code to values dumped by simulators is now possible thanks to Tywaves! URL https://github.com/rameloni/tywaves-chisel-demo.

[49] Johan Peltenburg, Jeroen Van Straten, Matthijs Brobbel, Zaid Al-Ars, and H. Peter Hofstee. Tydi: An Open Specification for Complex Data Structures Over Hardware Streams. *IEEE Micro*, 40(4): 120–130, July 2020. ISSN 0272-1732, 1937-4143. doi: 10.1109/MM.2020.2996373. URL https://ieeexplore.ieee.org/document/9098092/.

[50] pieter3d. Pieter3d/simview, February 2024. URL https://github.com/pieter3d/simview. Accessed: 2024-07-25.

[51] Matthijs A Reukers, Yongding Tian, Zaid Al-Ars, Peter Hofstee, Matthijs Brobbel, Johan Peltenburg, and Jeroen van Straten. An Intermediate Representation for Composable Typed Streaming Dataflow Designs.

[52] Martin Schoeberl. ChselBook: Digital Design with Chisel. URL http://www.imm.dtu.dk/~masca/chisel-book.pdf. Accessed: 2024-06-20.

[53] Ofer Shacham, Omid Azizi, Megan Wachs, Wajahat Qadeer, Zain Asgar, Kyle Kelley, John Stevenson, Stephen Richardson, Mark Horowitz, Benjamin Lee, Alexandre Solomatnikov, and Amin Firoozshahian. Rethinking Digital Design: Why Design Must Change. *Micro, IEEE*, 30: 9–24, January 2011. doi: 10.1109/MM.2010.81.

[54] Siemens. ModelSim HDL simulator. URL https://eda.sw.siemens.com/en-US/ic/modelsim/. Accessed: 2024-07-25.

[55] Frans Skarman and Oscar Gustafsson. Spade: An HDL Inspired by Modern Software Languages. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 454–455, August 2022. doi: 10.1109/FPL57034.2022.00075. URL https://ieeexplore.ieee.org/document/10035162.

[56] Frans Skarman, Lucas Klemmer, Kevin Laeufer, and Oscar Gustafsson. Surfer 0.2.0. Zenodo, June 2024. URL https://doi.org/10.5281/zenodo.11447243.

[57] Wilson Snyder. Veripool: Verilator. URL https://www.veripool.org/verilator/. Accessed: 2024-07-25.

[58] Emanuele Del Sozzo, Davide Conficconi, Alberto Zeni, Mirko Salaris, Donatella Sciuto, and Marco D. Santambrogio. Pushing the Level of Abstraction of Digital System Design: A Survey on How to Program FPGAs. *ACM Computing Surveys*, 55(5):106:1–106:48, December 2022. ISSN 0360-0300. doi: 10.1145/3532989. URL https://dl.acm.org/doi/10.1145/3532989.

[59] Synopsys. VCS Functional Verification Solution | Synopsys Verification, . URL https://www.synopsys.com/verification/simulation/vcs.html. Accessed: 2024-07-25.

[60] Synopsys. Verdi Automated Debug System | Synopsys Verification, . URL https://www.synopsys.com/verification/debug/verdi.html. Accessed: 2024-07-25.

[61] Yongding Tian, Matthijs Reukers, Zaid Al-Ars, Peter Hofstee, Matthijs Brobbel, Johan Peltenburg, and Jeroen Straten. 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*, pages 521–529, Denver CO USA, November 2023. ACM. ISBN 9798400707858. doi: 10.1145/3624062.3624539. URL https://dl.acm.org/doi/10.1145/3624062.3624539.

[62] Lenny Truong and Pat Hanrahan. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. *LIPIcs, Volume 136, SNAPL 2019*, 136:7:1–7:21, 2019. ISSN 1868-8969. doi: 10.4230/LIPICS.SNAPL.2019.7. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SNAPL.2019.7.

[63] twoentartian. Twoentartian/tydi-lang-2, November 2023. URL https://github.com/twoentartian/tydi-lang-2. Accessed: 2024-06-30.

[64] Xilinx. Vivado Overview. URL https://www.xilinx.com/products/design-tools/vivado.html. Accessed: 2024-07-25.

[65] Keyi Zhang. Hardware Generator Debugger. URL https://hgdb.dev/. Accessed: 2024-06-20.

[66] Keyi Zhang, Zain Asgar, and Mark Horowitz. Bringing source-level debugging frameworks to hardware generators. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 1171–1176, San Francisco California, July 2022. ACM. ISBN 978-1-4503-9142-9. doi: 10.1145/3489517.3530603. URL https://dl.acm.org/doi/10.1145/3489517.3530603.

A | Tydi-lang to Tydi-Chisel: a one-to-one translation

Tydi-lang is a domain-specific language (DSL) designed to reduce disparities between hardware and software implementations. It provides a syntax to express custom data types similarly to software development. For instance, software structs can be defined with full flexibility as Tydi `Group` and integers can be defined as `UInt_t = Bit(32)`, `SInt_t = Bit(32)` or `long = Bit(64)` and so on. This allows to make the translation from a software program to hardware more straightforward for developers, especially software developers. These latter do not have the same broad knowledge and experience about classic HDLs as hardware engineers. Thus, Tydi-lang abstractions become essential to facilitate developers' initial approach and push Tydi between those high-level languages to speed up the accelerators' implementations and, as a consequence, to combat the performance limitations of software programs [32].

This appendix reports an example of such a case, provided by Cromjongh et al. in [23]. As shown by Figure 2.1 and the Tydi-Chisel toolchain flow in [23, Figure 3], software specifications and data types can be easily defined through the Tydi-lang constructs (Listing A.1 and A.2). Then, the compiler and transpiler transform the Tydi code into Chisel boilerplate code with all data types and connections already defined in Scala. The Chisel data type hierarchy [37, *p. 37*] ensures enough flexibility to translate exactly the Tydi types previously defined as shown in Listing A.3. At this point, the user makes use of the Tydi-Chisel library and the Chisel language and pipeline to design and test the behavior of the accelerator or, more generically the circuit.

As can be observed by the listings, each data type in tydi-lang defined from the original spark code is translated one-to-one with a Chisel data type. Serving as an example, `UInt_64_t = Bit(64)` is defined in Chisel as `class UInt_64_t extends BitsEl(64.W)`. This exact mapping further justifies the choice of using the Chisel testing infrastructure and the fact that no extra information from Tydi-lang is needed to implement Tywaves.

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

Listing A.1: Example of Spark code. From *Tydi-Chisel: Collaborative and Interface-Driven Data-Streaming Accelerators* [23, *Listing 1*]

```
#### package pack0;
UInt_64_t = Bit(64); // UInt<64>
SInt_64_t = Bit(64); // SInt<64>

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

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

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

#### package pack1;
use pack0;

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

impl NonNegativeFilter of NumsFilter_interface {}

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

impl Reducer of NumsToStats_interface {}

impl PipelineExample of NumsToStats_interface {
    instance filter(NonNegativeFilter);
    instance reducer(Reducer);

    filter.std_out => reducer.std_in;
    reducer.std_out => self.std_out;
    self.std_in => filter.std_in;
}
```

Listing A.2: Corresponding Tydi-lang code. From *Tydi-Chisel: Collaborative and Interface-Driven Data-Streaming Accelerators* [23, *Listing* 2]

```
package Pack1
import nl.tudelft.tydi_chisel._
import chisel3._
import chisel3.experimental.ExtModule

object MyTypes {
    def UInt_64_t: UInt = UInt(64.W)
    assert(this.UInt_64_t.getWidth == 64)
    def SInt_64_t: SInt = SInt(64.W)
    assert(this.SInt_64_t.getWidth == 64)
}

class UInt_64_t extends BitsEl(64.W)
class SInt_64_t extends BitsEl(64.W)

class NumberGroup extends Group {
    val time = MyTypes.UInt_64_t
    val value = MyTypes.SInt_64_t
}

class Stats extends Group {
    val average = MyTypes.UInt_64_t
    val max = MyTypes.UInt_64_t
    val min = MyTypes.UInt_64_t
    val sum = MyTypes.UInt_64_t
}
```

```
class Stats_stream extends PhysicalStreamDetailed(e=new Stats, n=1, d=1, c=1,
    r=false, u=Null())
object Stats_stream {
    def apply(): Stats_stream = Wire(new Stats_stream())
}

class NumberGroup_stream extends PhysicalStreamDetailed(e=new NumberGroup, n=1,
    d=1, c=1, r=false, u=Null())
object NumberGroup_stream {
    def apply(): NumberGroup_stream = Wire(new NumberGroup_stream())
}


class NumsFilter_interface extends TydiModule {
    protected val inStream: PhysicalStreamDetailed[NumberGroup, Null] =
        NumberGroup_stream().flip
    val in: PhysicalStream = inStream.toPhysical

    protected val outStream: PhysicalStreamDetailed[NumberGroup, Null] =
        NumberGroup_stream()
    val out: PhysicalStream = outStream.toPhysical
}

class NonNegativeFilter extends NumsFilter_interface {
    inStream  := DontCare
    outStream := DontCare
}

class NumsToStats_interface extends TydiModule {
    protected val inStream: PhysicalStreamDetailed[NumberGroup, Null] =
        NumberGroup_stream().flip
    val in: PhysicalStream = inStream.toPhysical

    protected val outStream: PhysicalStreamDetailed[Stats, Null] = Stats_stream()
    val out: PhysicalStream = outStream.toPhysical
}

class Reducer extends NumsToStats_interface {
    inStream  := DontCare
    outStream := DontCare
}

class PipelineExample extends NumsToStats_interface {
    // Fixme: Remove the following line if this impl. contains logic. If it just
        interconnects, remove this comment.
    inStream := DontCare
    // Fixme: Remove the following line if this impl. contains logic. If it just
        interconnects, remove this comment.
    outStream := DontCare

    // Modules
    private val filter = Module(new NonNegativeFilter)
    private val reducer = Module(new Reducer)
    // Connections
    reducer.in := filter.out
    out := reducer.out
    filter.in := in
}
```

Listing A.3: Corresponding Tydi-Chisel boilerplate code

B | EphemeralSimulator: The default high-level simulator in ChiselSim

ChiselSim is the new simulation front-end for Chisel. Its adoption has been recommended since Chisel 5, when the development team switched from the Scala FIRRTL Compiler to the CIRCT MLIR compiler [14]. As depicted in Figure 2.4, ChiselSim interfaces with low-level RTL simulators and is composed of two components which can be combined to create a complete testing framework similar to ChiselTest [40]: `svsim` and `PeekPoke` API. Currently, the Chisel library implements one high-level simulator, called `EphemeralSimulator`, providing basic functionality of the PeekPoke API without any control of the underlying CIRCT compiler.
Both svsim and backend simulators, such as Verilator, support emitting trace files, but users cannot exploit this feature through the `EphemeralSimulator` provided. Furthermore, CIRCT integrates different options, enabling different grades of optimizations and allowing the emission of some debug information linking firrtl and Verilog as introduced in Section 2.3.
Listing B.1 reports an example testbench code using the `EphemeralSimulator`. The `simulate` method does not accept any parameter, preventing any possibility for customization of the simulation. This is confirmed by Listing B.2 which illustrates the signature of `simulate`[1].

```
import chisel3._
import chisel3.simulator.EphemeralSimulator._
import org.scalatest.flatspec.AnyFlatSpec

class MyModuleSpec extends AnyFlatSpec {
  behavior of ”MyModule”
  it should ”do something” in {
    simulate(new MyModule) { c =>
      c.io.in.poke(0.U)
      c.clock.step()
      c.io.out.expect(0.U)
      c.io.in.poke(42.U)
      c.clock.step()
      c.io.out.expect(42.U)
      println(”Last output value : ” + c.io.out.peek().litValue)
    }
  }
}
```

Listing B.1: ChiselSim testbench using the `EphemeralSimulator`

```
def simulate[T <: RawModule](module: => T)(body: (T) => Unit): Unit = {
    makeSimulator.simulate(module)({ module => body(module.wrapped) }).result
}
```

Listing B.2: `simulate` method implemented by `EphemeralSimulator` in Chisel 6.4.0

Allowing the emission of signal traces is an essential feature for using open-source waveform viewers in ChiselSim. In addition, abstractions for accessing the options of the underlying compiler are necessary for supporting the extension of generating and dumping extra debug information.

[1] `EphemeralSimulator` in Chisel 6.4.0 `https://github.com/chipsalliance/chisel/blob/v6.4.0/src/main/scala/chisel3/simulator/EphemeralSimulator.scala`

C | TableGen of debug dialect updated

```
def ScopeOp : DebugOp<”scope”> {
  let summary = ”Define a scope for debug values”;
  let arguments = (ins
    StrAttr:$instanceName,
    StrAttr:$moduleName,
    Optional<ScopeType>:$scope
  );
  let results = (outs ScopeType:$result);
  let assemblyFormat = [{
    $instanceName `,` $moduleName (`scope` $scope^)? attr-dict
  }];
}


def VariableOp : DebugOp<”variable”, [AttrSizedOperandSegments]> {
let summary = ”A named value to be captured in debug info”;
  let arguments = (ins
    StrAttr:$name,
    AnyType:$value,
    OptionalAttr<StrAttr>:$typeName,
    OptionalAttr<ArrayAttr>:$params,
    Optional<EnumDefType>:$enumDef,
    Optional<ScopeType>:$scope
  );
  let assemblyFormat = [{
    $name `,` $value (`scope` $scope^)? attr-dict (`enumDef` $enumDef^)?
        `:` type($value)
  }];
}

def SubFieldOp : DebugOp<”subfield”> {
  let summary = ”A named value to be captured in debug info which is a
    subfield of an aggregate”;
  let arguments = (ins
    StrAttr:$name,
    AnyType:$value,
    OptionalAttr<StrAttr>:$typeName,
    OptionalAttr<ArrayAttr>:$params,
    Optional<EnumDefType>:$enumDef
  );
  let results = (outs SubFieldType:$result);
  let assemblyFormat = [{
    $name `,` $value attr-dict (`enumDef` $enumDef^)? `:` type($value)
  }];
}

def StructOp : DebugOp<”struct”, [
  Pure,
  PredOpTrait<”number of fields and names match”,
    CPred<”$fields.size() == $names.size()”>>
]> {
  let summary = ”Aggregate values into a struct”;
  let arguments = (ins Variadic<AnyType>:$fields, StrArrayAttr:$names);
  let results = (outs StructType:$result);
  let hasCustomAssemblyFormat = 1;
}


def ArrayOp : DebugOp<”array”, [Pure, SameTypeOperands]> {
  let summary = ”Aggregate values into an array”;
  let arguments = (ins Variadic<AnyType>:$elements);
  let results = (outs ArrayType:$result);
  let hasCustomAssemblyFormat = 1;
}


def ModuleInfoOp : DebugOp<”moduleinfo”> {
  let summary = ”Define extra debug information for a module”;
  let arguments = (ins
    StrAttr:$typeName,
    OptionalAttr<ArrayAttr>:$params
  );
  let assemblyFormat = [{ attr-dict }];
}

def EnumDefOp : DebugOp<”enumdef”> {
  let summary = ”Define the value variants of an enumeration”;
  let arguments = (ins
    StrAttr:$enumTypeName,
    I16Attr:$id,
    DictionaryAttr:$variantsMap,
    Optional<ScopeType>:$scope
  );
  let results = (outs EnumDefType:$result);
  let assemblyFormat = [{
    $enumTypeName `,` `id` $id `,` $variantsMap (`scope` $scope^)? attr-dict
  }];
}
```

Listing C.1: Updated TableGen code for the debug dialect to handle new high-level type information for debugging

D | UML diagrams

D.1. Sequence diagram of Chisel side

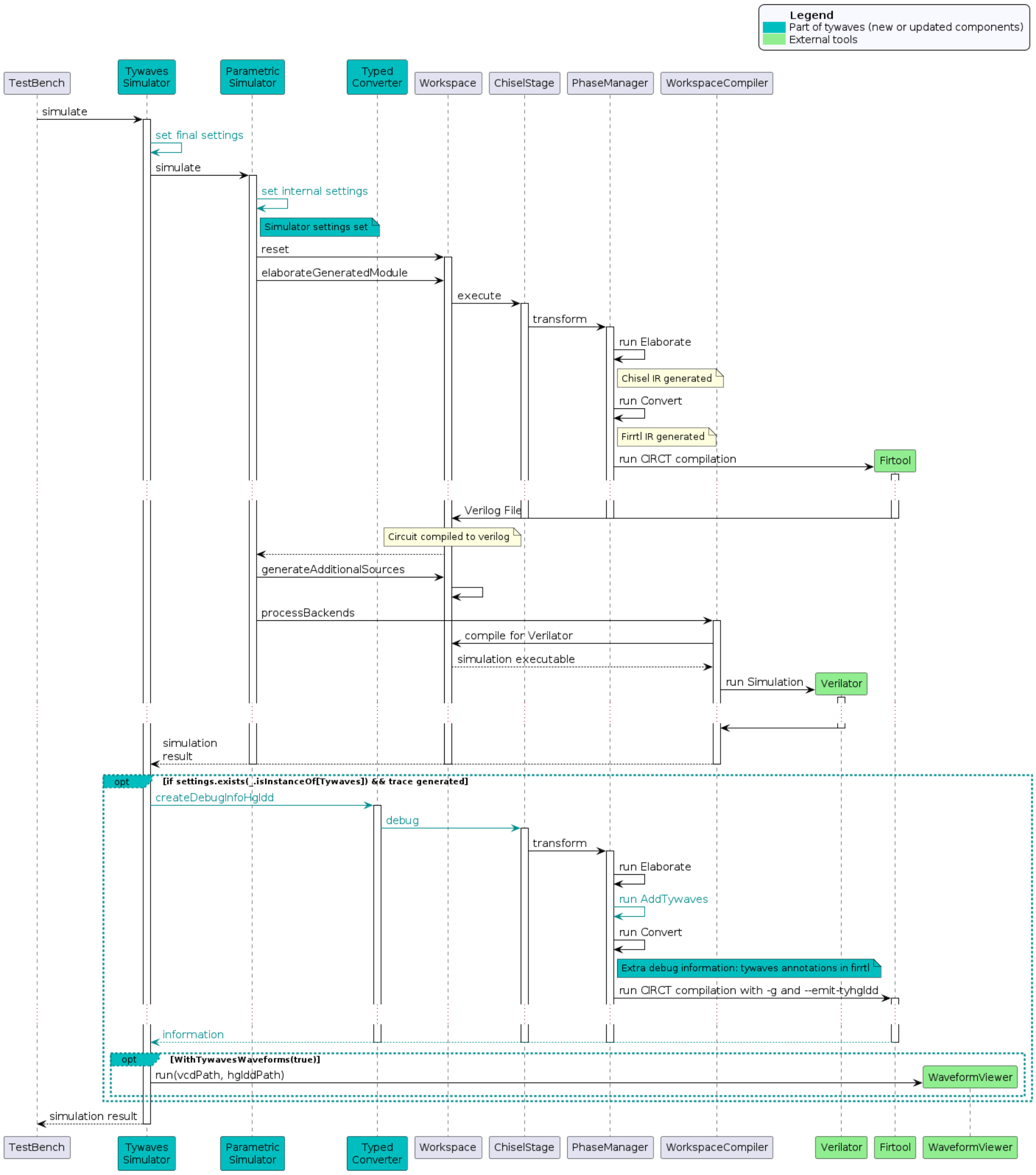


Figure D.1: Sequence diagram of the full call stack

D.2. Sequence diagram of Surfer side

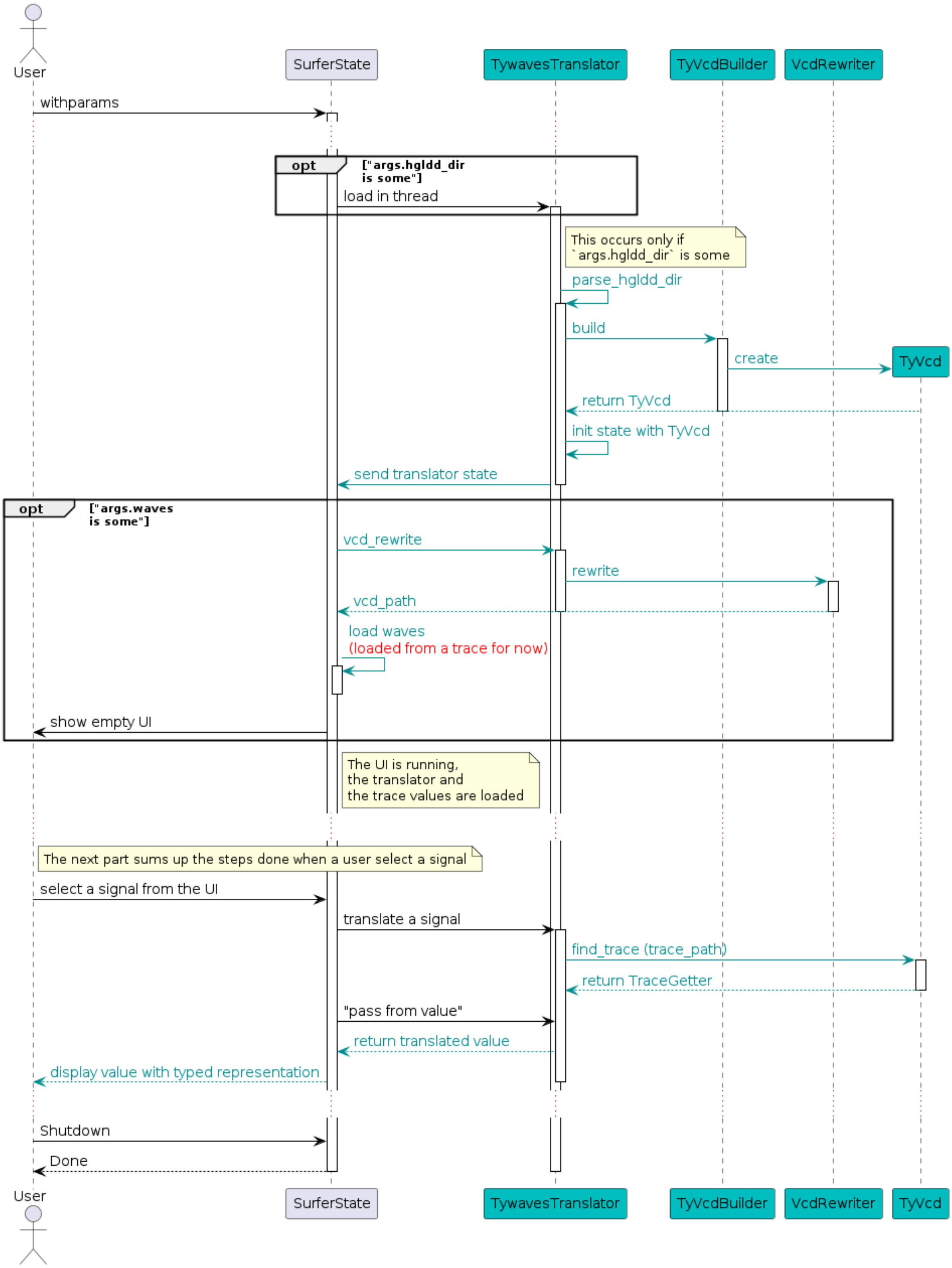


Figure D.2: Sequence diagram of Surfer

D.3. Class diagram of Hgldd

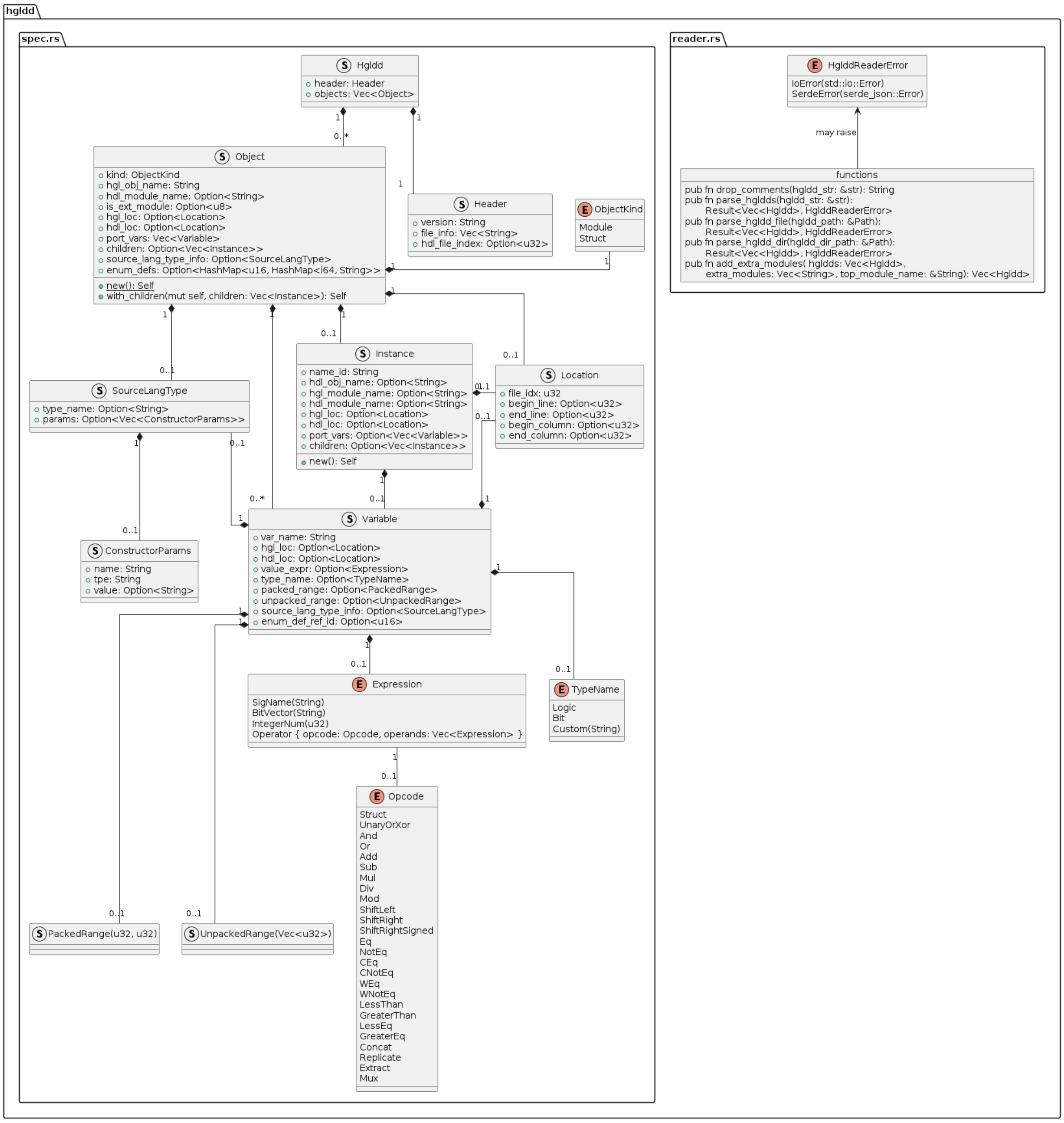

hgldd
spec.rs
S Hgldd
header: Header
objects: Vec<Object>
1
0..*
1
1
S Object
kind: ObjectKind
hgl_obj_name: String
hdl_module_name: Option<String>
is_ext_module: Option<u8>
hgl_loc: Option<Location>
hdl_loc: Option<Location>
port_vars: Vec<Variable>
children: Option<Vec<Instance>>
source_lang_type_info: Option<SourceLangType>
enum_defs: Option<HashMap<u16, HashMap<i64, String>>
new(): Self
with_children(mut self, children: Vec<Instance>): Self
S Header
version: String
file_info: Vec<String>
hdl_file_index: Option<u32>
E ObjectKind
Module
Struct
1
1
1
1
1
1
0..1
0..1
0..1
S Instance
name_id: String
hdl_obj_name: Option<String>
hgl_module_name: Option<String>
hdl_module_name: Option<String>
hgl_loc: Option<Location>
hdl_loc: Option<Location>
port_vars: Option<Vec<Variable>>
children: Option<Vec<Instance>>
new(): Self
S Location
file_idx: u32
begin_line: Option<u32>
end_line: Option<u32>
begin_column: Option<u32>
end_column: Option<u32>
0..1
0..1
S SourceLangType
type_name: Option<String>
params: Option<Vec<ConstructorParams>>
1
0..1
0..*
1
0..1
1
0..1
1
S ConstructorParams
name: String
tpe: String
value: Option<String>
S Variable
var_name: String
hgl_loc: Option<Location>
hdl_loc: Option<Location>
value_expr: Option<Expression>
type_name: Option<TypeName>
packed_range: Option<PackedRange>
unpacked_range: Option<UnpackedRange>
source_lang_type_info: Option<SourceLangType>
enum_def_ref_id: Option<u16>
1
1
1
1
0..1
0..1
E Expression
SigName(String)
BitVector(String)
IntegerNum(u32)
Operator { opcode: Opcode, operands: Vec<Expression> }
E TypeName
Logic
Bit
Custom(String)
1
0..1
E Opcode
Struct
UnaryOrXor
And
Or
Add
Sub
Mul
Div
Mod
ShiftLeft
ShiftRight
ShiftRightSigned
Eq
NotEq
CEq
CNotEq
WEq
WNotEq
LessThan
GreaterThan
LessEq
GreaterEq
Concat
Replicate
Extract
Mux
0..1
0..1
S PackedRange(u32, u32)
S UnpackedRange(Vec<u32>)
reader.rs
E HglddReaderError
IoError(std::io::Error)
SerdeError(serde_json::Error)
may raise
functions
pub fn drop_comments(hgldd_str: &str): String
pub fn parse_hgldds(hgldd_str: &str):
Result<Vec<Hgldd>, HglddReaderError>
pub fn parse_hgldd_file(hgldd_path: &Path):
Result<Vec<Hgldd>, HglddReaderError>
pub fn parse_hgldd_dir(hgldd_dir_path: &Path):
Result<Vec<Hgldd>, HglddReaderError>
pub fn add_extra_modules(hgldds: Vec<Hgldd>,
extra_modules: Vec<String>, top_module_name: &String): Vec<Hgldd>

Figure D.3: Class diagram of the Hgldd data structure in Tywaves-rs

D.4. Class diagram of TyVcd

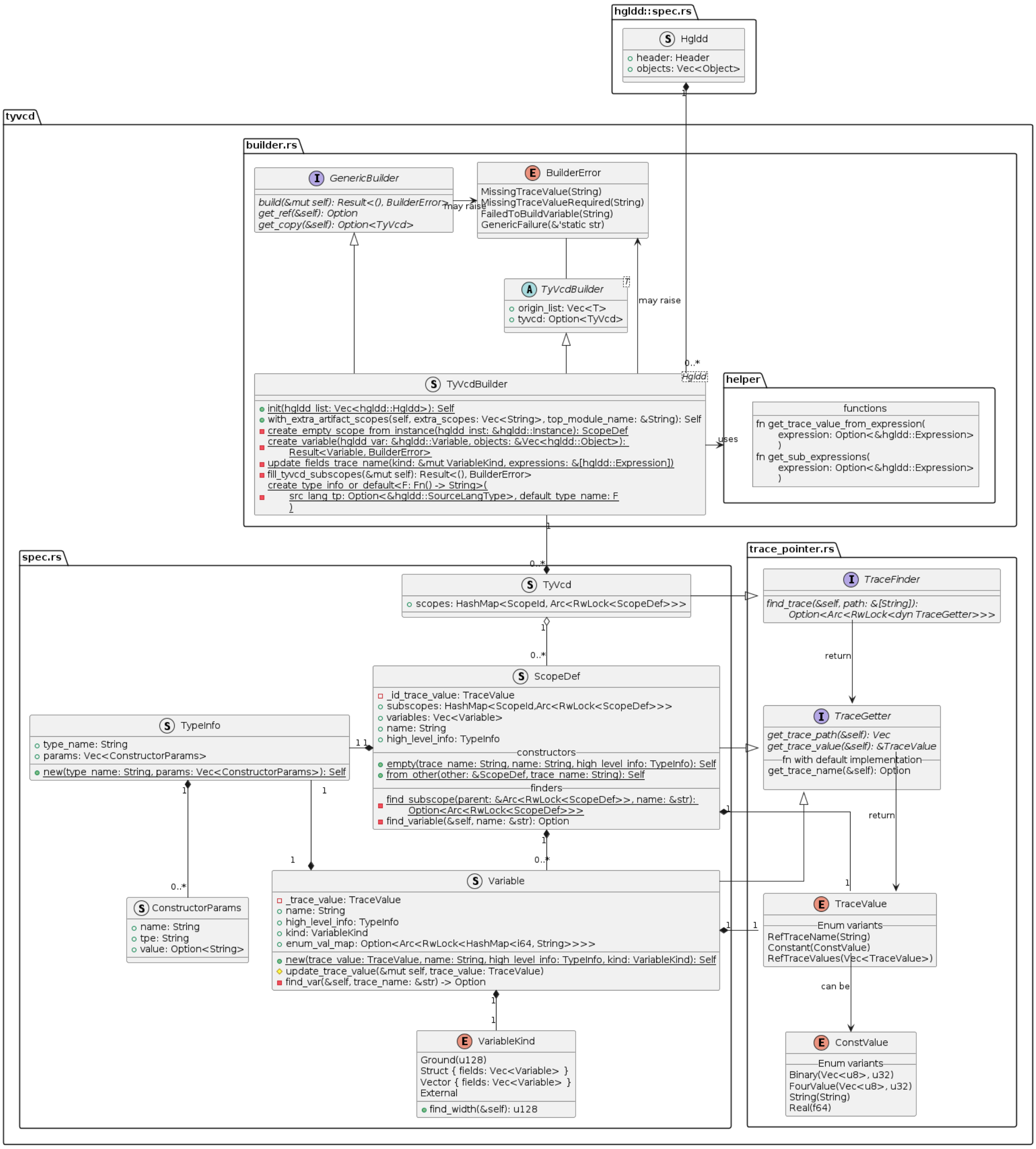

hgldd::spec.rs
S Hgldd
header: Header
objects: Vec<Object>
1
tyvcd
builder.rs
I GenericBuilder
build(&mut self): Result<(), BuilderError>
get_ref(&self): Option
get_copy(&self): Option<TyVcd>
may raise
E BuilderError
MissingTraceValue(String)
MissingTraceValueRequired(String)
FailedToBuildVariable(String)
GenericFailure(&'static str)
A TyVcdBuilder
T
origin_list: Vec<T>
tyvcd: Option<TyVcd>
may raise
0..*
Hgldd
S TyVcdBuilder
init(hgldd_list: Vec<hgldd::Hgldd>): Self
with_extra_artifact_scopes(self, extra_scopes: Vec<String>, top_module_name: &String): Self
create_empty_scope_from_instance(hgldd_inst: &hgldd::Instance): ScopeDef
create_variable(hgldd_var: &hgldd::Variable, objects: &Vec<hgldd::Object>):
Result<Variable, BuilderError>
update_fields_trace_name(kind: &mut VariableKind, expressions: &[hgldd::Expression])
fill_tyvcd_subscopes(&mut self): Result<(), BuilderError>
create_type_info_or_default<F: Fn() -> String>(
src_lang_tp: Option<&hgldd::SourceLangType>, default_type_name: F
)
uses
helper
functions
fn get_trace_value_from_expression(
expression: Option<&hgldd::Expression>
)
fn get_sub_expressions(
expression: Option<&hgldd::Expression>
)
1
spec.rs
0..*
S TyVcd
scopes: HashMap<ScopeId, Arc<RwLock<ScopeDef>>>
1
0..*
S ScopeDef
_id_trace_value: TraceValue
subscopes: HashMap<ScopeId,Arc<RwLock<ScopeDef>>>
variables: Vec<Variable>
name: String
high_level_info: TypeInfo
constructors
empty(trace_name: String, name: String, high_level_info: TypeInfo): Self
from_other(other: &ScopeDef, trace_name: String): Self
finders
find_subscope(parent: &Arc<RwLock<ScopeDef>>, name: &str):
Option<Arc<RwLock<ScopeDef>>>
find_variable(&self, name: &str): Option
S TypeInfo
type_name: String
params: Vec<ConstructorParams>
new(type_name: String, params: Vec<ConstructorParams>): Self
1 1
1
1
1
1
0..*
0..*
S ConstructorParams
name: String
tpe: String
value: Option<String>
S Variable
_trace_value: TraceValue
name: String
high_level_info: TypeInfo
kind: VariableKind
enum_val_map: Option<Arc<RwLock<HashMap<i64, String>>>>
new(trace_value: TraceValue, name: String, high_level_info: TypeInfo, kind: VariableKind): Self
update_trace_value(&mut self, trace_value: TraceValue)
find_var(&self, trace_name: &str) -> Option
1
1
E VariableKind
Ground(u128)
Struct { fields: Vec<Variable> }
Vector { fields: Vec<Variable> }
External
find_width(&self): u128
trace_pointer.rs
I TraceFinder
find_trace(&self, path: &[String]):
Option<Arc<RwLock<dyn TraceGetter>>>
return
I TraceGetter
get_trace_path(&self): Vec
get_trace_value(&self): &TraceValue
fn with default implementation
get_trace_name(&self): Option
return
1
1
1
1
E TraceValue
Enum variants
RefTraceName(String)
Constant(ConstValue)
RefTraceValues(Vec<TraceValue>)
can be
E ConstValue
Enum variants
Binary(Vec<u8>, u32)
FourValue(Vec<u8>, u32)
String(String)
Real(f64)

Figure D.4: Class diagram of the TyVcd data structure in Tywaves-rs