ChiselTrace: Typed Behavioral Debugging in Chisel Through Signal Dependency Tracing

Jarl Brand*, Casper Cromjongh*, H. Peter Hofstee*†, Zaid Al-Ars*
*Delft University of Technology (Delft, The Netherlands), †IBM Infrastructure (Austin, TX, US)
j.y.k.brand@student.tudelft.nl, c.cromjongh@tudelft.nl, hofstee@us.ibm.com, z.al-ars@tudelft.nl

2025 IEEE Nordic Circuits and Systems Conference (NorCAS) | 979-8-3315-1501-0/25/$31.00 ©2025 IEEE | DOI: 10.1109/NORCAS66540.2025.11231292

***Abstract*—While modern HDLs such as Chisel (Constructing Hardware In a Scala Embedded Language) significantly improve the process of design entry, debugging these designs is often problematic, because the tools that aid debugging operate on translated code rather than the original HDL. Furthermore, engineers often resort to manual waveform debugging, undermining productivity gains promised by such a language. We present ChiselTrace, an open-source tool for Chisel that is capable of (dynamic) program slicing and automatic signal dependency tracing, allowing faults to be more easily traced back to their root cause. Where prior work focuses on data-flow analysis at the (compiled) Verilog level, ChiselTrace functions at the Chisel source level. Contributions include: modifications to the Chisel library to enable post-simulation analysis; a library capable of dynamic program slicing and dependence graph generation; and a front-end dependency graph viewer. We demonstrate debugging capabilities by tracing an injected fault in the ChiselWatt processor back to the source. We observe that using ChiselTrace's dynamic program dependence graph, the number of lines of code relevant to the fault path is reduced significantly.**

Project repository: https://github.com/jarlb/chiseltrace

***Index Terms*—source-level debugging, Chisel, program dependency graph, Tywaves, program slicing**

I. Introduction

Since the inception of hardware description languages (HDLs), hardware designs have significantly grown in complexity. While traditional HDLs, such as Verilog and VHDL, remain the standard, interest in new languages, often referred to as "hardware construction languages" (HCLs) or "hardware generator languages" (HGLs), has grown in recent years. These languages, such as Clash [1], MyHDL [2], and Chisel [3], offer high-level abstractions often only found in software languages, such as functional programming, meta-programming, and polymorphism [4]. Furthermore, they typically compile to traditional HDLs, bringing productivity gains while retaining compatibility with existing tool chains. Among these languages, Chisel has been identified as the most popular [5].

While traditional HDLs benefit from a long history of commercial and open-source debugging tooling, applying these tools to HGL-generated code brings many challenges. Generated HDL is often not easily human-readable and obfuscates the high-level abstractions of the source language. Flattened signals, optimized logic, and the lack of high-level constructs make it difficult to relate bugs in a design back to the source language. This undermines the productivity gains of using HGLs and creates a critical gap: the lack of source-level debugging solutions for modern HGLs.

ChiselTrace aims to address this gap by enabling automated dependency tracing through time and program slicing of behavioral designs at the Chisel source level. With this functionality, ChiselTrace seeks to reduce the time spent manually debugging by tracing wrong values back to their source in the waveform viewer. Contributions of this work include:

1) A Chisel library extension facilitating probe insertion and extraction of program dependence graphs (PDGs) and control flow graphs (CFGs).
2) A dynamic slicing library that reconstructs control flow and is capable of static/dynamic program slicing and dynamic PDG (DPDG) generation.
3) An interactive debugging interface that visualizes a DPDG along with simulation data and source code, replacing manual signal tracing with dependency-driven design exploration.

To the authors' knowledge, ChiselTrace is the first open-source tool to provide automated dependency tracing for modern HGLs, such as Chisel. Furthermore, ChiselTrace differs from existing, proprietary data-flow analysis tools by operating at the HGL level, instead of the Verilog level, providing a representation that is closer to the source circuit.

We demonstrate the functionality of ChiselTrace on a real-world design, ChiselWatt [6], a soft-core processor using the OpenPOWER ISA [7] that is implemented in Chisel.

In Section II, background information regarding Chisel, program slicing, and Tywaves, a typed waveform viewer, is presented. Work relating to both HDL and HGL debugging is explored in Section III. Section IV states the design choices of the various components of ChiselTrace, while Section V presents implementation details. In Section VI, a real-world example of ChiselTrace is shown, and Section VII summarizes and concludes the findings.

II. Background

In this section, context is provided about the Chisel language, namely its compilation and simulation flows. Furthermore, related concepts from Tywaves and program slicing are presented.

A. Chisel

Chisel [3] is an HGL built on the Scala programming language. Where traditional HDLs focus on low-level modeling of circuits, Chisel aims to bring higher-level software concepts such as Functional Programming, parametrization, and type safety to hardware design. Furthermore, being embedded in Scala, it fosters the reuse of components, similar to software libraries.

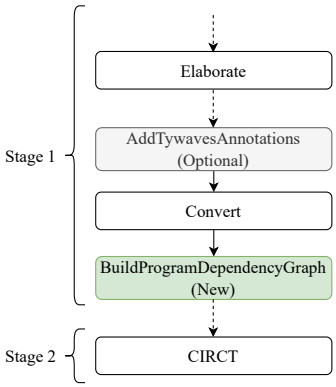

Fig. 1: Overview of the most important phases in the Chisel compilation pipeline. The `AddTywavesAnnotations` and `BuildProgramDependencyGraph` phases do not belong to the Chisel library, but are required for Tywaves and ChiselTrace, respectively. Dotted arrows indicate one or more phases.

B. Tydi

Tydi [8] is an open specification that defines a type system for variable-sized structures with compound datatypes and provides a way to transport them over hardware streams. Tydi-Lang [9] is a language for defining Tydi components, which is compiled to Tydi-IR [10], before VHDL is generated. Tydi-Chisel [11], [12] introduces a transpiler that converts Tydi-Lang into Chisel. Tywaves [13] is a typed waveform viewer that originated from the Tydi ecosystem.

C. Chisel compilation chain

Chisel, being an HGL, is compiled to a traditional HDL such as Verilog. Internally, this process is done by applying *phases* to the Chisel circuit, as shown in Figure 1.

Two main stages can be distinguished: translation to FIRRTL [14] (1), and compilation to Verilog (2). First, the Chisel circuit is converted to a FIRRTL representation. FIRRTL (Flexible Intermediate Representation for RTL) is an IR (intermediate representation). During the `Elaborate` and `Convert` phases, meta-programming is executed, and many high-level Scala and Chisel constructs are converted to primitives, while retaining constructs such as vector and bundle types and conditionals. This has the goal of simplifying and standardizing the input for the `CIRCT` compilation phase.

In the second stage, the generated FIRRTL circuit is compiled to Verilog by CIRCT [15]. In addition to the generated Verilog, the compiler may generate debug information in the form of HGLDD files. These files contain structured JSON-like information about source-language mappings for the generated Verilog.

D. FIRRTL

FIRRTL [14], being an IR, is a highly simplified HDL that still retains the capability to represent any Chisel circuit. A FIRRTL circuit can contain multiple modules, one of which is the top-level module. Moreover, modules can instantiate other modules. Module I/O is defined by input and output statements.

Modules are constructed from components, providing both combinatorial and sequential logic. Combinatorial logic is handled using `node`, `wire`, and I/O statements. Of these, `node`s may not be reassigned. Registers (`reg`, `regreset`) and memory statements are responsible for sequential logic. Registers may or may not have an explicit reset value. Lastly, sub-modules are also components. All components have a type. Aside from ground types, FIRRTL offers compound datatypes. Two types of particular interest are the `Bundle` and the `Vector` types, which are akin to structs and arrays in other programming languages. These two types allow for arbitrarily complex datatypes. In `Bundle`s, fields may be marked as `flipped` to reverse the connection direction.

Data flow is handled through connect statements. The only restriction is that two connected signals should have the same type. When connecting to a register signal, the update at the rising clock edge is implicit. Furthermore, multiple connections can assign to the same signal. Only the last connection will be used. Multiplexers are another data flow construct that may be used for conditional assignments. Control flow in FIRRTL is almost exclusively handled by the `when` statement, which is a simple `if` conditional statement. Lastly, annotations can be added to the circuit that add information relating to components of the circuit.

E. Chisel simulations & Tywaves

ChiselSim is a simulation and testing framework built into the Chisel library. It allows the user to write unit tests for circuit components in Scala, as opposed to manual testbenches in Verilog. For simulations, the Chisel circuit is first compiled to Verilog, then a simulator such as Verilator is used to carry out the user-provided test.

The Tywaves [13] project brings waveform debugging at the source-language level to Chisel, which ChiselSim lacks. The Chisel-to-Tywaves compilation and simulation chain is shown in Figure 2. First, annotations containing the source language types are added to the nodes of the Chisel circuit graph, which are then passed on to the FIRRTL statements as FIRRTL annotations. The FIRRTL circuit is then compiled by a modified version of CIRCT, which adds the type information to the generated HGLDD files.

The modified HGLDD files are used by the library `tywaves-rs` to rewrite the VCD (value change dump) file generated during simulation, packing signals according to source hierarchy (i.e., all signals that belong to a compound datatype in the source language are packed into one bitstring). Using this rewritten VCD file and the information available in the HGLDD files, the library can reconstruct a source-level view of the signal values. Finally, a custom ChiselSim simulator is used to launch a modified version of Surfer [16] that presents a source-level view of the waveform.

F. Program slicing

Program slicing [17] is a technique that allows for finding a subset of statements (a slice) that can affect a criterion. The criterion consists of a statement and a list of variables. To compute a static slice, first, a program dependency graph is

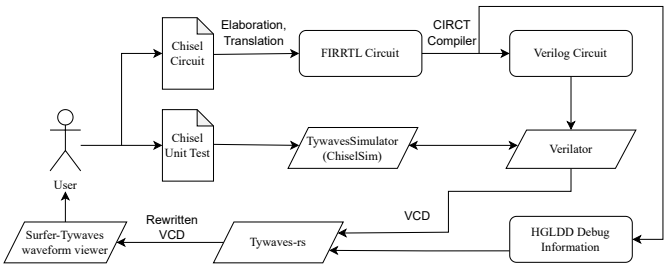

Fig. 2: Chisel-to-Tywaves simulation flow. The Chisel circuit is compiled to Verilog. ChiselSim executes the unit test by giving instructions to Verilator. The resulting VCD file is processed by `tywaves-rs` along with the debug information, and finally, the typed waveform is shown to the user.

formed. In this graph, statements that depend on each other are connected by directed edges. Then this graph is traversed from the criterion, marking all visited nodes as included in the slice [18].

The narrower a slice is, the more useful it is for debugging purposes. By using runtime data, the slice can be reduced. This is called dynamic program slicing [19]. One method that is of particular interest for hardware debugging constructs a DPDG based on execution history. Each occurrence of a statement is turned into a node in this graph. From this graph, a dynamic slice can be created by finding the latest occurrence of the criterion, then, again, traversing the graph from that point.

One important distinction between software and hardware program slicing is the fact that software execution is on a line-by-line basis, while in hardware designs, many statements execute in parallel. To allow for program slicing, the circuit can be seen as an infinite loop [20]. Furthermore, most HDLs offer semantics for determining the active signal driver. Namely, the last statement to assign to a signal will be the driving statement. This, when combined with a CFG, allows for reconstruction of dynamic dependencies.

HDLs offer features that significantly complicate determining the control flow, such as processes with sensitivity lists and wait statements. FIRRTL [14], however, does not support these features, making analysis easier.

Figure 3 shows program slicing and DPDG construction applied to a Chisel circuit. The criterion in this example is the output signal (statement 0). As is shown, the simulation is run for three time steps: initial and two clock cycles. This means that statement 3 is never executed. When looking at the static slice, it can be seen that all statements are present. This is because, based on static data, all statements in the circuit may contribute to the value of the output signal. In the DPDG, however, it can be seen that statement 3 is not included. As a result, statement 3 is also not included in the dynamic slice of this example circuit.

III. Related work

A. Traditional HDL debugging

The concept of back-tracing signal dependencies is not new. Tools for automatic signal driver tracing and data-flow analysis exist within commercial tools such as Synopsys Verdi [21], Cadence Xcelium [22], and Siemens Questa Sim [23]. ChiselTrace differs from these tools by enabling dependency analysis at the Chisel source level. This is an important distinction because the generated Verilog that these tools operate on is often dissimilar to the Chisel source code, making it difficult to relate these tools' output to the original representation.

In [20] and [24], static program slicing methods for VHDL and Verilog have been proposed, respectively. Furthermore, there has been academic effort to use program slicing for hardware debugging purposes. [25] introduces a debugging method with automatic error correction for HDLs that uses static program slicing as one method to reduce the search space. Dynamic program slicing on HDLs is explored in [26], where dynamic slices are calculated by combining static slices with coverage information and are used to accelerate fault injection. Lastly, program slicing and dynamic coverage analysis have been used for automatic fault localization [27].

All of the aforementioned works, however, are built for traditional HDLs, such as Verilog, and are therefore ill-suited for debugging Chisel designs due to their lack of source mappings. In addition, they lack the high-level type system that is used by Tydi and Tywaves.

B. Debugging modern HGLs

In the context of more modern HGLs, such as Chisel, source-level debugging is an active topic of research. Tywaves [13] introduces a waveform viewer with source-level representation of signals. [28] introduces HGDB, a breakpoint-style debugger for HGLs that is capable of breakpoint emulation and reverse-debugging.

ChiselTrace aims to enhance the HGL debugging experience even further by automating the tedious task of manual signal tracing in waveform viewers using DPDGs. Furthermore, it provides an open-source tool to view waveforms as a dependency graph representation.

IV. Design approach

In this section, we discuss the design choices behind ChiselTrace. As explained in Section I, the end goal is to be able to perform static/dynamic program slicing on Chisel designs and allow users to debug their design using a dependency graph representation of their waveform. As seen in Section II, to perform program slicing and construct a DPDG, a PDG and execution history must be available. This calls for the following three main components:

1) Chisel extension to extract a PDG and CFG to allow for reconstruction of the execution history.
2) Library capable of reconstructing FIRRTL execution history, program slicing, and mapping the created graph back to Chisel representation.
3) Graph viewer that can be integrated with the aforementioned library.

A high-level overview of the components can be seen in Figure 4.

A. PDG extraction and execution history reconstruction

ChiselTrace requires a circuit PDG and execution history reconstruction. The required analysis is performed on FIRRTL

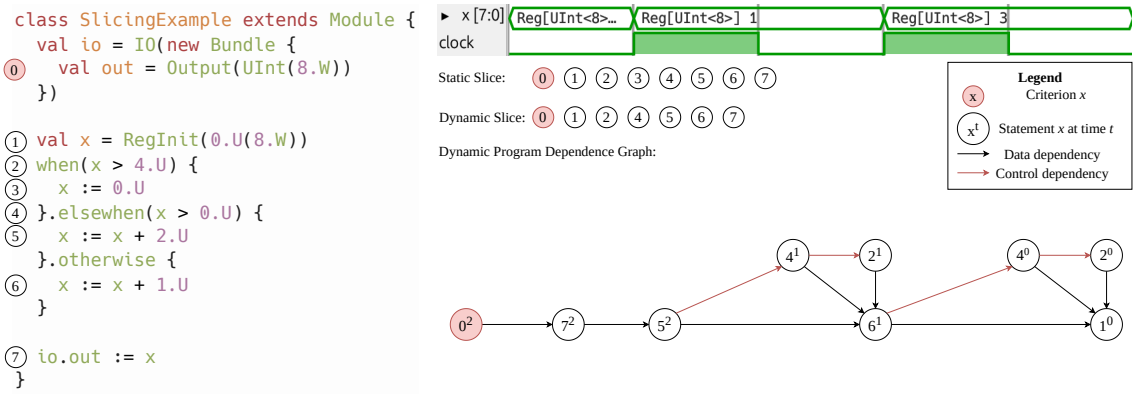

Fig. 3: Program slicing and DPDG construction of an example Chisel circuit.

instead of Chisel or Verilog, due to the simplicity of the IR. Furthermore, the Chisel library and CIRCT compiler guarantee that the FIRRTL circuit is functionally equivalent to the other two representations, making analysis performed on the FIRRTL circuit valid for Chisel as well. Lastly, FIRRTL contains source locators for each statement that allow reconstruction of the Chisel representation from graphs generated on the FIRRTL representation.

Information extraction in Chisel was chosen over CIRCT, because the analysis can be easily integrated as a phase after the `Convert` phase (see Figure 1). Furthermore, it avoids the need to maintain a CIRCT compiler fork.

For dependency analysis and PDG construction, compound-type symbols are split into atomic symbols. To supplement static dependency analysis with dynamic data and allow reconstruction of execution history, probe signals are used that allow the downstream library to reconstruct vector indexing, MUX predicates, and conditional predicates.

B. External DPDG building library

To generate a DPDG and program slices from the input VCD, PDG/CFG, and HGLDD files, an external library is implemented in Rust (`chiseltrace-rs`), which was chosen for performance considerations and compatibility with `tywaves-rs`. This makes it reusable and allows it to work with any FIRRTL-based language that implements the required data extraction phase.

Since Chisel circuits are often synchronous, the dynamic dependency reconstruction part of the library assumes a single global clock to process the dependencies in chunks of one clock cycle. While this simplifies the dependency tracking, it limits support for multiple clock domains and asynchronous designs. Future work could extend this to handle clock-domain crossings.

C. Front-end

The front-end must visualize the dependency graph with nodes grouped into clock-cycle time slots and support interactive node placement. A web-based solution is used for portability and a fast development cycle. The Tauri GUI framework is used, which integrates a Rust back-end with a web front-end. For the UI, Svelte (JavaScript framework) and vis.js (graph visualization) are used.

V. Implementation

This section presents the implementation details of the different components introduced in Section IV.

A. Modifications to Chisel

A new phase, `BuildProgramDependencyGraph`, is added after the `Convert` phase (see Figure 1). It functions as follows. First, the FIRRTL circuit graph is traversed, flattening the module hierarchy in the process. For each statement, symbol IDs are determined for any symbols the statement provides or depends on. The IDs contain the hierarchical path to the symbols module, as well as the path within the compound type, where applicable. For a bundle `io`, with subfield `addr` in module `top`, the ID would be `top.io.addr`.

Probe signals are inserted into the circuit that replace any indexing or predicate expression. The original expression then gets replaced with a reference to the probe, of which the name is an ID. Furthermore, these signals are marked with a `DontTouchAnnotation`, disallowing compiler optimizations.

After extracting statement dependencies, statements can have or provide four kinds of dependencies: Data, Conditional, Declaration, and Index. ID-based dependency matching is used for PDG construction, which is implemented using a hash map. The vector index and MUX probes are added as conditions to nodes and edges of the PDG, allowing downstream analysis to discard unrelated dependencies based on simulation data. Additionally, a CFG is formed. The names of the generated predicate probe signals are embedded into the CFG to allow execution history reconstruction. Finally, these two graphs are exported in the JSON format.

B. DPDG building library

The Chisel phase produces the PDG in a two-list format: nodes and edges. For processing, this format is converted to a linked memory structure. Static program slicing is achieved using the technique described in Section II. The criterion is found in the PDG, and then it is traversed to determine all contributing statements.

DPDG generation and dynamic slicing are handled differently. In Figure 4, the DPDG generation loop is highlighted in gray. For each clock cycle, signal changes are read, after which

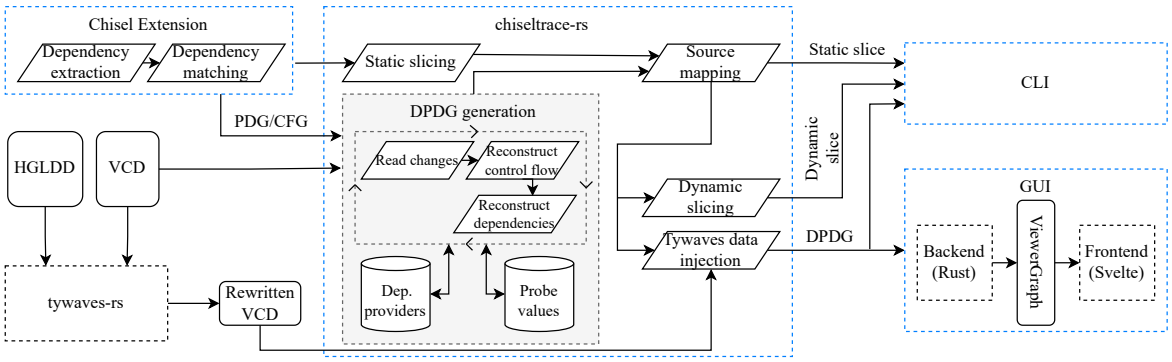

Fig. 4: Overview of the components involved in ChiselTrace. Novel contributions are marked in blue.

the CFG is combined with the predicate probes to reconstruct activated statements. This way, the library keeps track of the latest dependency provider for each symbol. While resolving dependencies, the index probes are used to discard unrelated dependencies.

After building the DPDG, the linked structure is converted back to a list of nodes and edges. It is then converted to Chisel representation by grouping nodes based on their source mappings. At this point, a dynamic slice can be constructed by listing all unique nodes in the DPDG. Alternatively, the `tywaves-rs` library can be used to reconstruct source-level simulation data where available, which is attached to the nodes of the exported DPDG.

C. Graph viewer

A timeline view is achieved by creating a scrollable collection of timeslots. Each timeslot contains its related nodes. Nodes are automatically placed, after which the user can manually position the nodes. The front-end dynamically obtains the relevant graph data from the back-end based on timeslots that are in view. This keeps the front-end responsive and allows it to scale to graphs with many clock cycles.

Figure 6 shows the graph viewer on part of ChiselWatt in comparison with a traditional waveform viewer containing Tywaves data (5). ChiselTrace presents the waveform as a graph of connected statements, spread across time slots of one clock cycle. Furthermore, the kinds of dependencies and their typed values are visible in the viewer (in Figure 6, only data dependencies are shown). Any dependencies that fall out of view are shown in separate containers at the top of a time slot.

VI. Case-study: ChiselWatt

The functionality of ChiselTrace and the advantages compared to manual waveform inspection are now demonstrated by debugging the ChiselWatt soft-core processor.

A fault is injected into the XOR instruction, which makes this instruction perform an OR operation instead. The test program in Listing 1 is used. This program loads the values 5 and 3 into registers 3 and 4. Then, an XOR and ADDI 2 are performed on this data. A correctly functioning processor should produce the value 8 in register 6, but value 9 is observed due to the injected fault.

```
1 li 3, 0b0101
2 li 4, 0b0011
3 xor 5, 3, 4 # Fault will cause 7 in register 5
4 addi 6, 5, 2 # Will result in 9 in register 6
5 blr
```

Listing 1: ChiselWatt assembly test program

Figures 5 and 6 show the more conventional Surfer-Tywaves as well as the ChiselTrace view, respectively. In the example, control flow and index flow have been disabled for clarity. In Figure 5, the order of signals that need to be inspected to find the fault is indicated. The criterion is the output of the adder module. It is shown that ChiselTrace is capable of tracing signal dependencies through multiple hierarchical levels back to the fault in `logical.scala` at line 29. Furthermore, it can be seen that waveform debugging requires significant design understanding to select the relevant signals to analyze. ChiselTrace, on the other hand, is able to automatically determine the relevant signals. Lastly, it is shown that some of the edges in the ChiselTrace view do not have value annotations. These values are not present in the VCD produced by the simulator. This shows that ChiselTrace is capable of tracking signal relations, even if not all signals are present in the simulation data. These signals are still present in the ChiselTrace view because of the control and data flow reconstruction.

On average, ChiselTrace adds 8.2s (50%) to the Chisel compilation/simulation time on this example. The DPDG construction takes 846.4 ms on average. The results are measured on an Intel i7-9750H processor.

Using ChiselTrace, 19 statements were inspected to trace the fault back to the source. In contrast, 27 statements were inspected during manual analysis. Each manual statement inspection requires: locating the signal in the source code; locating the signal in the waveform, where applicable; and determining which dependencies to analyze next. This results in 81 actions with high cognitive overhead, as opposed to 19 simple dependency-following actions using ChiselTrace (a reduction of 76.5%).

The number of lines of code in the DPDG relevant to the fault path was analyzed in comparison with the total number of lines in relevant modules calculated from the PDG (298 lines). ChiselTrace reduced this amount to 62 lines (a reduction of 79.2%), of which 17 were in the fault path.

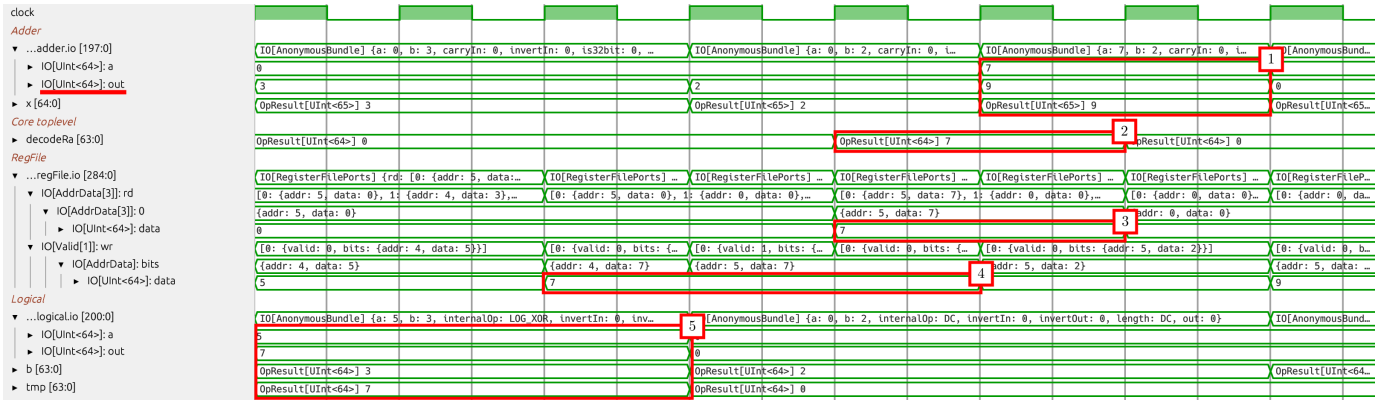

Fig. 5: Tywaves waveform of the ChiselWatt simulation. Relevant signals have been highlighted, and irrelevant signals have been removed from the hierarchical view. The criterion signal is underlined. The order of signal inspection required to locate the fault is indicated by the numbering.

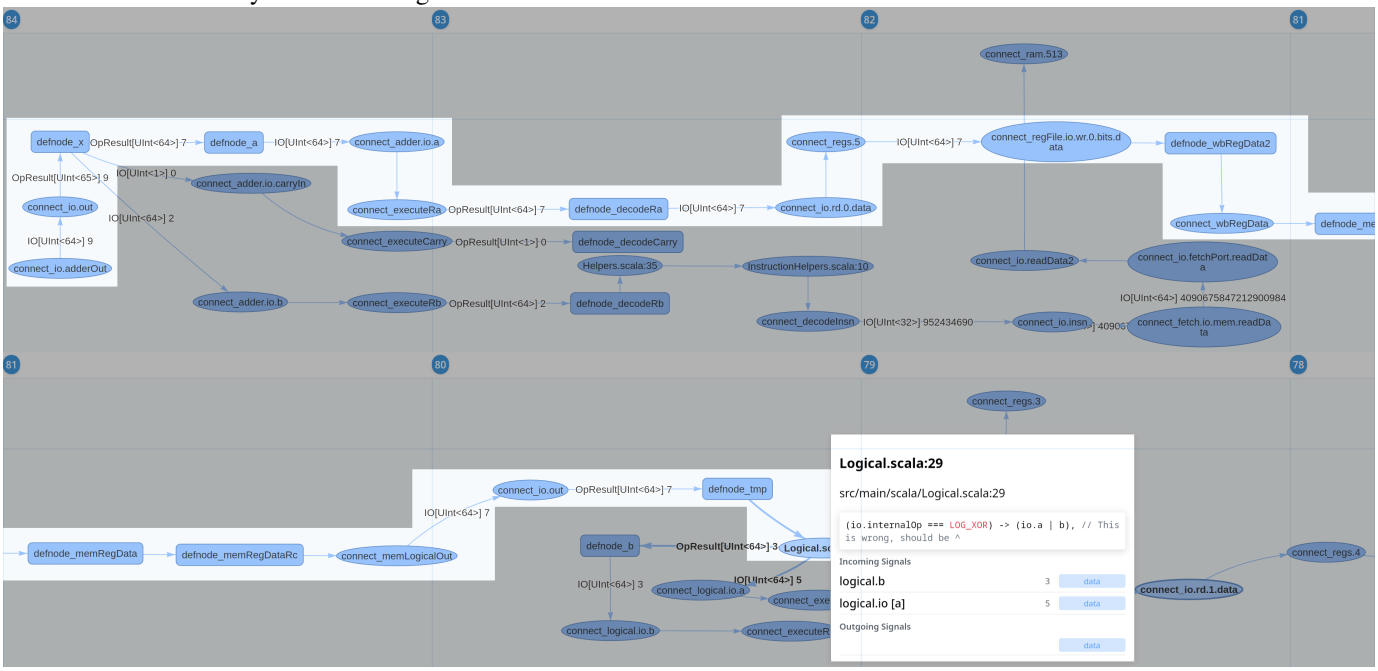

Fig. 6: ChiselTrace view of the ChiselWatt simulation. Only data flow (no control flow or index flow) is shown for clarity. Post-processing has been applied to highlight the relevant data path.

[29] provides a more detailed discussion of ChiselTrace, the ChiselWatt usecase, and how ChiselTrace is used with ChiselSim.

VII. Conclusion

Modern HGLs lack tools to debug designs at the source level, forcing engineers to debug at the generated HDL level. In this paper, ChiselTrace was introduced to address this gap. ChiselTrace is a novel, open-source, source-level debugging tool for Chisel, building on the Tywaves project.

ChiselTrace is capable of program slicing and dependency tracing. Multiple parts were introduced, such as a Chisel extension for PDG/CFG extraction and probe insertion, a program slicing and DPDG generating library, and a dependency graph viewer.

The functionality of ChiselTrace was demonstrated by debugging a faulty instruction in the ChiselWatt processor, where the benefit of ChiselTrace over a waveform viewer was shown. Waveform debugging required a better design understanding, and ChiselTrace was capable of tracing dependencies, even in the absence of simulation data for relevant signals. ChiselTrace reduced the amount of debugging actions by 76.5% and the number of lines of code to be inspected, relevant to the fault, by 79.2%.

In this paper, ChiselTrace was shown to work on Chisel. Due to the decoupled design of `chiseltrace-rs` and the graph viewer, these parts of the library may be reused, allowing extension to any FIRRTL language that implements the information extraction back-end.

Project repository: https://github.com/jarlb/chiseltrace

Acknowledgment

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

REFERENCES

[1] Christiaan Baaij, "Clash: From haskell to hardware," University of Twente, Tech. Rep., Dec. 2009.

[2] Jan Decaluwe, "Myhdl: A python-based hardware description language," *Linux J.*, vol. 2004, no. 127, p. 5, Nov. 2004, ISSN: 1075-3583.

[3] 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 *DAC Design Automation Conference 2012*, 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[4] 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 Comput. Surv.*, vol. 55, no. 5, Dec. 2022, ISSN: 0360-0300. DOI: 10.1145/3532989. [Online]. Available: https://doi.org/10.1145/3532989.

[5] Matti Käyrä and Timo D. Hämäläinen, "A survey on system-on-a-chip design using chisel hw construction language," in *IECON 2021 – 47th Annual Conference of the IEEE Industrial Electronics Society*, 2021, pp. 1–6. DOI: 10.1109/IECON48115.2021.9589614.

[6] Anton Blanchard. "Chiselwatt." (), [Online]. Available: https://github.com/antonblanchard/chiselwatt (visited on 05/06/2025).

[7] "Openpower instruction set architecture," OpenPOWER Foundation. (2024), [Online]. Available: https : / / openpowerfoundation . org / specifications/isa/ (visited on 05/02/2025).

[8] 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*, vol. 40, no. 4, pp. 120–130, 2020. DOI: 10.1109/MM.2020.2996373.

[9] 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*, ser. SC-W '23, Denver, CO, USA: Association for Computing Machinery, 2023, pp. 521–529, ISBN: 9798400707858. DOI: 10.1145/3624062.3624539. [Online]. Available: https://doi.org/10.1145/3624062.3624539.

[10] 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," English, *CEUR Workshop Proceedings*, vol. 3462, 2023, Joint Workshops at the 49th International Conference on Very Large Data Bases, VLDBW 2023 ; Conference date: 28-08-2023 Through 01-09-2023, ISSN: 1613-0073.

[11] 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)*, 2023, pp. 1–7. DOI: 10 . 1109 / NorCAS58970 . 2023 . 10305451.

[12] Casper Cromjongh, Yongding Tian, H. Peter Hofstee, and Zaid Al-Ars, "Hardware-Accelerator Design by Composition: Dataflow Component Interfaces With Tydi-Chisel," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 32, no. 12, pp. 2281–2292, Dec. 2024, ISSN: 1557-9999. DOI: 10 . 1109 / TVLSI . 2024 . 3461330. [Online]. Available: https : / / ieeexplore . ieee . org / document / 10705101 (visited on 04/07/2025).

[13] Raffaele Meloni, H. Peter Hofstee, and Zaid Al-Ars, "Tywaves: A typed waveform viewer for chisel," in *2024 IEEE Nordic Circuits and Systems Conference (NorCAS)*, 2024, pp. 1–6. DOI: 10 . 1109 / NorCAS64408.2024.10752465.

[14] Patrick S. Li, Adam M. Izraelevitz, and Jonathan Bachrach, "Specification for the firrtl language," Tech. Rep. UCB/EECS-2016-9, Feb. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html.

[15] S. Eldridge, P. Barua, A. Chapyzhenka, A. Izraelevitz, J. Koenig, C. Lattner, A. Lenharth, G. Leontiev, F. Schuiki, R. Sunder, A. Young, and R Xia, "Mlir as hardware compiler infrastructure," in *Workshop on Open-Source EDA Technology (WOSET)*, Oct. 2021.

[16] Frans Skarman, Lucas Klemmer, Daniel Große, Oscar Gustafsson, and Kevin Laeufer, "Surfer — an extensible waveform viewer," in *Computer Aided Verification*, Ruzica Piskac and Zvonimir Rakamarić, Eds., Cham: Springer Nature Switzerland, 2025, pp. 392–404, ISBN: 978-3-031-98685-7.

[17] Mark Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984. DOI: 10 . 1109 / TSE.1984.5010248.

[18] Karl J. Ottenstein and Linda M. Ottenstein, "The program dependence graph in a software development environment," *SIGPLAN Not.*, vol. 19, no. 5, pp. 177–184, Apr. 1984, ISSN: 0362-1340. DOI: 10.1145/390011.808263. [Online]. Available: https://doi.org/10.1145/390011.808263.

[19] Hiralal Agrawal and Joseph R. Horgan, "Dynamic program slicing," *SIGPLAN Not.*, vol. 25, no. 6, pp. 246–256, Jun. 1990, ISSN: 0362-1340. DOI: 10.1145/93548.93576. [Online]. Available: https://doi.org/10.1145/93548.93576.

[20] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum, "Program slicing for vhdl," *International Journal on Software Tools for Technology Transfer*, vol. 4, pp. 125–137, 1 Oct. 2002, ISSN: 14332779. DOI: 10.1007/s100090100069.

[21] Synopsys, *Verdi*. [Online]. Available: https : / / www . synopsys . com / verification/debug/verdi.html.

[22] Cadence, *Simvision*. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html.

[23] Siemens, *Questa sim*. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/.

[24] Jen-chieh Ou, Daniel G. Saab, and Jacob A. Abraham, "Hdl program slicing to reduce bounded model checking search overhead," in *2006 IEEE International Test Conference*, 2006, pp. 1–7. DOI: 10 . 1109 / TEST.2006.297665.

[25] Bijan Alizadeh, Payman Behnam, and Somayeh Sadeghi-Kohan, "A scalable formal debugging approach with auto-correction capability based on static slicing and dynamic ranking for rtl datapath designs," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1564–1578, 2015. DOI: 10.1109/TC.2014.2329687.

[26] Ahmet Cagri Bagbaba, Maksim Jenihhin, Jaan Raik, and Christian Sauer, "Accelerating transient fault injection campaigns by using dynamic hdl slicing," in *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, 2019, pp. 1–7. DOI: 10.1109/NORCHIP.2019.8906932.

[27] Jian Hu and Zhenlei Liu, "Context aware deep learning-based fault localization for hardware design code," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2025. DOI: 10.1109/TCAD.2025.3543426.

[28] 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*, ser. DAC '22, San Francisco, California: Association for Computing Machinery, 2022, pp. 1171–1176, ISBN: 9781450391429. DOI: 10 . 1145 / 3489517 . 3530603. [Online]. Available: https : / / doi . org / 10 . 1145 / 3489517 . 3530603.

[29] Jarl Brand, "Chiseltrace: Typed behavioural debugging in modern typed hdls through signal dependency tracing," M.S. thesis, Delft University of Technology, 2025. [Online]. Available: https://resolver.tudelft.nl/uuid:6e5b0f6c-b0ef-4e76-b498-00fb313dee15.