Surfer — An Extensible Waveform Viewer

Frans Skarman[1(✉)], Lucas Klemmer[2], Daniel Große[2], Oscar Gustafsson[1], and Kevin Laeufer[3]

[1] Linköping University, Sweden
frans.skarman@liu.se
[2] Johannes Kepler University Linz, Austria
[3] Cornell University, USA

Abstract. The waveform viewer is one of the most important tools in a hardware engineer's toolbox. It is the main interface used to track down design bugs found by simulation or formal verification. In this paper, we present Surfer, a modern waveform viewer designed to integrate with the broader hardware design ecosystem. It supports translation from bit vectors to semantically meaningful values, integration with simulation and verification tools, and lays the groundwork for interactive simulation in the open-source ecosystem.

1 Introduction

The computer-aided verification community has excelled at generating ideas and tools that automatically find bugs in system designs. However, once a verification tool finds a trace that demonstrates the violation of an invariant, then the real work for the system engineer begins. They need to debug their system to understand what is going wrong and how the issue can be addressed. When the system in question is a digital hardware design for a microchip, the tool of choice for investigating the buggy behavior is the waveform viewer. A waveform viewer visualizes the signals in a circuit over time. We present Surfer, an open-source[4] waveform viewer that is designed from scratch to be easily customizable and embeddable. It has enabled new research around hardware description languages [19] and verification languages [15], been used to teach digital hardware debugging[5], and has been considered for integration in commercial hardware verification products.

A lot of engineering work goes into building a capable waveform viewer and thus, until recently, there has only been one fully-featured open-source implementation of such a tool. GTKWave [10] has been a staple in the teaching and open-source community for years. GTKWave supports many input formats and the development team has pioneered the FST format, the most-space efficient

[4] https://gitlab.com/surfer-project/surfer/

[5] We are aware of classes at Cornell, MIT, UC Santa Cruz, and Johannes Kepler University Linz recommending Surfer to their students.

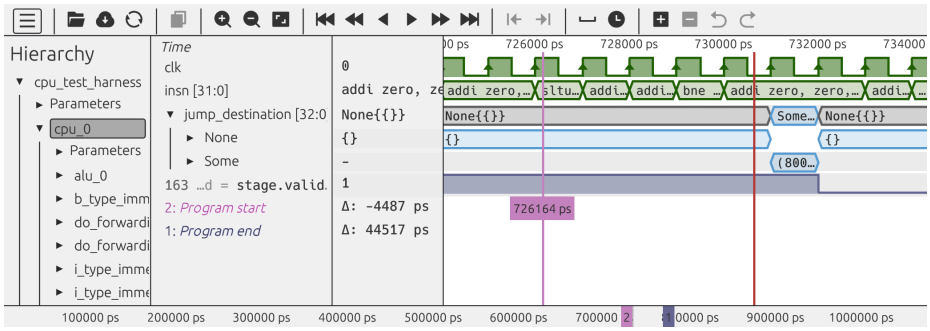


Fig. 1: A screenshot of the Surfer waveform viewer debugging a RISC-V core.

waveform format with an open specification. However, it has proven to be rather difficult to extend GTKWave beyond its current functionality, as the code base is quite old and appears to not have been written with extensibility in mind. Surfer was built from the ground up to address these issues and enable new workflows.

While existing waveform viewers excel at debugging hardware designs written in SystemVerilog or VHDL, there is no support for newer hardware languages like Chisel [3] or any of the other new languages [2, 4, 20, 26]. These languages generally feature more advanced type systems, but none of that rich meta-data is considered when viewing signal traces in existing viewers. Without seeing the native representation of signal values, the user has a bad debugging experience. This is a significant hurdle that prevents these new languages from being adopted. Surfer addresses this problem by including an extensible translator system that can be used to decode semantically meaningful values from raw bit vectors debug traces (Section 2).

Besides new languages, there is also a need for a flexible waveform viewer that can be used to design new high-level debugging and analysis tools. Surfer can be easily embedded in web-applications and features a novel remote control protocol. It is also the first open-source viewer to support direct integration with a running simulator (Section 3). A custom waveform backend quickly loads VCD, FST or GHW files, taking advantage of modern multicore CPUs while minimizing user-facing latency and memory use and avoiding exploitable memory bugs (Section 4). Fig. 1 shows a screenshot of the Surfer waveform viewer.

2 Extensible Translator System

An important job of a waveform viewer is to transform the raw bit vectors emitted by the simulator into semantically meaningful values. Most waveform viewers have some support for doing this for numeric values, for example allowing the user to show numbers as signed, unsigned, or hexadecimal versions. However, real-world signals often have much richer semantic meanings.

Currently, Surfer primarily deals with two sources of these semantically rich formats. The first are modern Hardware Description Languages (HDLs) with more expressive type systems compared to VHDL and Verilog. Thus, effective debugging requires seeing native values directly rather than raw bit vectors. Surfer also has extensive support for decoding RISC-V, LA64, and MIPS instructions, which are commonly found in modern hardware designs. Less common formats which might be specific to a particular project can be incorporated with a custom translator written in Rust or Python.

2.1 Hardware Description Language Support

VHDL and SystemVerilog are not longer the only options available to designers when designing hardware. Chisel [3] has seen widespread adoption in industry for both design and verification [9]. VexRiscV perhaps the most commonly used soft core RISC-V is written in SpinalHDL [26], and several ASIC and FPGA designs are being developed in Amaranth [1], BlueSpec [21] and Clash [2]. In addition, there are a very large number of languages in development that have yet to see widespread adoption, including Spade [25] PipelineC, [11] Silice, [16] SUS [28], Filament [20], RHDL [4], and many others. Many of these languages include features that make traditional waveform viewers difficult to use. These languages often have more advanced type systems than Verilog. Most allow defining product types like structs or tuples. Some languages like Spade, SUS, and Clash and RHDL take this one step further and allow full algebraic data types.

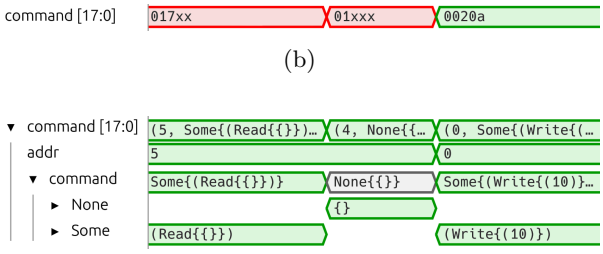
Currently, most of these languages are compiled to Verilog or VHDL for synthesis and simulation, and in this process, the high-level type information is lost. After the simulation is done, the user is left with a trace that consists only of bit vectors that they then need to interpret. To exemplify this problem, Listing 2a contains the definition of a Spade type modelling commands to a memory module. A short simulation trace of a signal of this type is shown in Fig. 2b. Without knowledge of the internal representation of Spade types, this trace is very difficult to interpret, and even with such knowledge, doing the translation manually requires significant mental effort.

Surfer's translator system makes it possible for a specialized Spade translator to use type information to recover the hierarchical data from the signal. Fig. 2c contains the same trace as Fig. 2b but with hierarchical translation enabled. From the root `translated` trace, the user quickly gets an overview of the full value of the signal. By expanding the individual field, it is easy to tell at a glance when commands are present, and which commands are active at different timestamps.

Translation is currently implemented for the Spade language as well as Chisel via the Tywaves [19] research project. There is also in progress work to support Clash [2] and RHDL [5] demonstrating that the translator system is extensible to support multiple different languages.

```
enum Option<T> {
  Some{val: T},
  None
}
enum Op {
  Write{val: uint<8>},
  Read
}
struct MemCommand {
  addr: uint<8>,
  command: Option<Op>
}
```

(a)



(b) (c)

Fig. 2: Example of a Spade type (a), a trace with a raw bit vector of that type (b), and a trace with a translated version of the same signal.

2.2 Instruction decoding

Another common source of semantically meaningful values are processor instructions. To help work with these, Surfer includes a dedicated system for decoding instructions[6]. This system lets users define the structure of their instruction set in a TOML based configuration format. Surfer ships with definitions for all officially ratified RISC-V instructions, LongArch64, and MIPS instructions and can thus translate them out of the box.

2.3 Project-Specific Python Translators

In addition to supporting translation of common formats like instructions and HDL types, Surfer also features support for user defined custom translators, allowing easy translation of project-specific signal formats. Currently, users can write simple python scripts to implement their translations. In the future, we will explore the use of plugins via WebAssembly to allow faster translators written in any language. WebAssembly being sandboxed also means that these user plugins can be distributed safely without the risk of malicious code affecting the viewer or the system it is running on.

3 Integrating Surfer with the EDA Ecosystem

The open-source EDA ecosystem has many excellent tools that do one particular job very well. However, compared to commercial EDA tools, there is significantly less integration. Most integration consists of workflow specific one-off scripts, which presents a hurdle to adopting these tools. Recently, Visual Studio Code has become a hub for more integrated HDL workflows, with plugins such as

[6] https://github.com/ics-jku/instruction-decoder

YoWASP [8] and TerosHDL [27] enabling synthesis and simulation without leaving the editor. However, neither approach integrates a waveform viewer, instead relying on the user to install GTKWave manually on their local machine.

Besides Integrated Development Environments (IDEs) for developers, many online teaching tools also rely on simulation and need some sort of waveform viewer to show simulation results to students. Projects like TinyTapeout [29] require users to use locally installed waveform viewers in an otherwise fully in-browser flow, presenting a hurdle for novices. Platforms like MakerChip [22] and quicksilicon [6] have used ad-hoc waveform viewers that are built specifically for running in the web. However, since there are limited resources available for building such a tool, those end up being very basic compared to the tools used in real design work. Being able to leverage a solid waveform viewer like Surfer would make these tools more effective.

Hardware verification tools can often benefit from having an integrated waveform viewer that is powerful and easy to use. For example, Silogy [24] and LubisEDA [18] run simulations or formal checkers in cloud infrastructure and then generate a web-based report. Showing these results in an interactive waveform viewer right on the web browser is much more convenient than requiring users to download VCD files for offline viewing as it accelerates the process, and doesn't require having a waveform viewer available on the machine on which the results are viewed. A deep integration of a waveform viewer would allow users to seamlessly navigate from the textual report to the signals and time at which the problem manifests, significantly reducing setup costs for debugging.

3.1 Embedding Surfer in Web Applications and Visual Studio Code

A common thread across these three integration areas is the heavy reliance on web technology. Visual Studio Code is a Chromium-based editor, and both teaching tools and cloud compute reports are usually built on web technology. Web integration requires the tool to either be built in JavaScript, or be compiled to WebAssembly (WASM) a portable compilation target for programming languages which allows it to be run in any web-based client. Surfer is written in Rust and compiled to WASM, which allows web-integration without sacrificing performance in native builds. Besides embedding, this also allows Surfer to be used directly in a web browser without an installation.

3.2 Controlling Surfer from a Third Party Tool

Web technology solves many problems associated with *embedding* the waveform viewer into a bigger application. However, we also need to be able to *control* the viewer from the application Surfer is embedded into. In debugging and verification tools, this can be used to point the user to specific problematic signals and time stamps. As Surfer supports drawing annotations, it can also be used to show more complex things like relationships between values of different signals. Teaching tools benefit in similar ways, being able to control the viewer to highlight interesting timestamps or signals can be very beneficial.

We developed a Waveform Control Protocol (WCP) that can be used to control a waveform viewer (server) from an embedding application (client). The protocol is heavily inspired by the Language Server Protocol (LSP) which is used to build language and editor-agnostic IDEs. Like the LSP, the WCP defines a set of JSON commands and responses that a client can send to control the waveform viewer, or query it about the currently viewed signals. Being JSON-based means that the protocol can be both language, and transport agnostic. Like the LSP, the WCP can be sent over network sockets, standard IO, or in the case of web integration, by passing full JSON messages between web views. While Surfer is the first implementor of the protocol, the intention is to allow other waveform viewer-like projects to also adopt this protocol to allow broader interoperability between debugging tools.

Besides the limited but stable WCP interface, Surfer also exposes a much lower level, unstable API to integrators. The reactive architecture of Surfer, allows external programs to directly inject messages to control Surfer in a manner very similar to how Surfer reacts to mouse and keyboard commands from the user. In Surfer, the callback function that reacts to a user pressing a particular button does not directly modify program state, but instead generates a message, for example, `AddSignal(...)`. This message is added to a central queue which is processed at the end of each frame. All state changes go through this system, whether they are user interactions like adding signals, events like `WaveformDataLoaded(...)` or messages injected by external integration. This brings huge advantages for integrators because the source of these messages is irrelevant to the program, and messages can easily be serialized in a way that simple function calls cannot. This means that the messages can be injected from an integrator via any source that can communicate serialized strings such as sockets, standard input, or JavaScript. Fig. 3 illustrates this architecture, which also powers Surfer's VSCode-inspired command palette and snapshot testing infrastructure.

3.3 Interactive Simulation

In the open-source ecosystem, simulation is currently exclusively done in batch mode. This means that the simulator runs a test bench to completion while generating a signal trace. Only after the simulation is done, the trace is loaded into a waveform viewer for inspection. Alternatively, in an interactive workflow, the simulation is controlled through a GUI. The user can run the simulation for a few clock cycles, inspect the result right away, change some signal values and continue simulation. Previously, interactive simulation was only available with commercial tools, benefiting from tight integration between simulator and waveform viewer. In the open-source space, the user is generally free to mix and match their choice of simulator with their choice of waveform viewer, making it more difficult to support interactive simulation for the large number of possible combinations.

Instead of a tight integration between one company's simulator and viewer, the open-source approach requires standardizing a common interface. Recently,

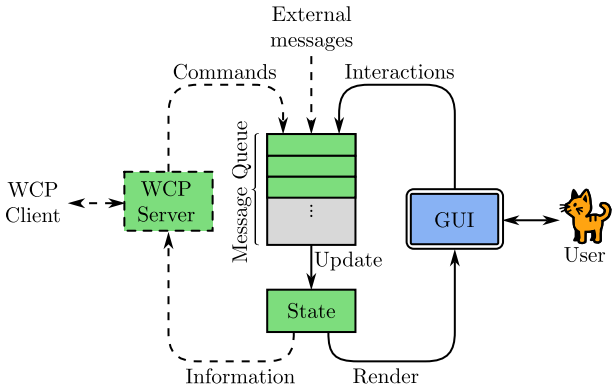


Fig. 3: The Surfer architecture. Solid lines denote the main program. Interfaces used by external integrators are dashed.

the CXXRTL Debug Server Protocol (CDSP) [7] has been introduced to support exactly this use-case. Like the WCP discussed previously, it is a JSON-based protocol that allows a client (waveform viewer) to connect to a server (simulator) to receive signal values and control the simulation. Surfer is the first waveform viewer to support this new open standard. Based on this support, we are currently integrating Surfer with the RTL-Debugger project which allows for an integrated simulation, debugging, and development environment[7].

While WCP and the CDSP are similar in implementation, their use case is quite different, which is why a both protocols are needed. CDSP is intended for communication between a simulator and a waveform viewer, with the simulator acting as the host. Through it, the waveform viewer can control the simulation, receive information about events such as failed assertions, and query the simulator for waveform data. WCP on the other hand has the waveform viewer acting as the host, and allows clients to control what the waveform viewer is currently showing. The primary use case is to allow clients integrating surfer to add signals of interest, show the user specific timestamps and draw additional information on the waveform to highlight important details. WCP also includes facilities for adding additional user interactions to the viewer, for example, allowing the user to jump to the source code location of a signal in their text editor.

3.4 Tools Integrating Surfer

We are aware of several tools that take advantage of the fact that Surfer is easy to embed and control, as described in the previous sections. The WSVA system [12] evaluates SystemVerilog Assertions on waveform traces and uses Surfer to visualize failed assertions by creating coursers pointing out the start and end of the property through the WCP interface. The WAL language runtime

[7] `https://github.com/amaranth-lang/rtl-debugger/pull/5`

integrates Surfer to display results of waveform analysis programs written in WAL [13–15]. It uses CXXRTL Debug Protocol [7] to inject analysis results as signals into Surfer.

Several commercial and open-source projects have already integrated Surfer as part of their projects. Two commercial verification platforms LubisEDA [18] and Silogy [24] have successfully used Surfer as an embedded waveform viewer to show waveforms that reproduce errors found during verification. Currently, the products are still in the prototype phase, and have not been released to the public.

Surfer has also seen adoption in numerous teaching tools. An example of this is SonicRV[8], a tool which lets students explore how processors execute each instruction. The primary interface in it shows the current state of the processor graphically and the user can step through execution, or click on individual instructions to see how they flow through the pipeline. The Surfer integration lets the user dive deeper to see the full state of the processor using the remote control functionality. This view also allows for interacting with instructions and seeing the relevant signals and timestamps in the waveform viewer.

Since the primary means of debugging digital designs is the waveform viewer, teaching platforms that teach HDLs also need to have an integrated waveform viewer for students to see their results. Quicksilicon [6] is such a platform and now uses Surfer as the primary waveform viewer. MakerChip [22], a tool used primarily to teach the TL-Verilog language [23] also has the option of using Surfer as the waveform viewer, though they still maintain their own viewer which has integration with TL-Verilog that Surfer currently lacks. TinyTapeout [29] is an educational project that makes it straightforward to get small designs manufactured on real chips. Their workflow is fully browser-based, with the full synthesis and simulation flow being run in the cloud to let users play with chip design without installing any tools. Once the simulation is complete, users, of course, need to inspect the results if problems occurred. By using a version of Surfer that is embedded in the TinyTapeout flow, this is now possible without having to install software, reducing a hurdle to people getting their designs working and taped out.

4 A Performant Waveform Backend

Surfer displays signal traces from VCD, FST and GHW files. Our file parsing code was designed from the ground up with three important requirements in mind: (1) To keep memory consumption in check, we need to only load the signals that a user has selected. Most signals in a waveform are never displayed and should not consume unnecessary memory. (2) To improve responsiveness, the file metadata and signal names must be available as soon as possible. (3) To support re-use and embedding in various applications, Surfer needs to be able to work with untrusted input files.

[8] `https://sonic-rv.ics.jku.at/`

VCD file: on-disk size	2.9 GiB
GTKWave: time to load / total application memory use	16s / 800 MiB
Surfer: time to load / compressed signal size (1 thread)	7.5s / 74 MiB
Surfer: time to load / compressed signal size (8 threads, 4-core CPU)	2.0s / 74 MiB
Surfer: average time to decompress a signal for viewing	271 µs

Tab. 4: Speed and memory comparisons for a single example VCD file.

To render the values of a signal, Surfer needs to be able to efficiently query the most recent value of the signal at a given time and the duration until the next change. This lookup is implemented in $O(\log_2(n))$ by performing a binary search on an array of changes sorted by the time at which the change occurs. This requires all values to be stored in a fixed size, $O(1)$-accessible, manner which takes a lot of space as it prohibits most compression schemes. To keep memory consumption in check, Surfer only stores signals that have been selected by the user in this uncompressed representation. The FST file format makes this task relatively simple, since it allows for efficient access to individual signals. VCD and GHW, on the other hand, require the complete file to be read and parsed, even if only a single signal is selected.

We developed an interesting middle ground between parsing all signals from a VCD into the uncompressed representation and reparsing the whole VCD every time a signal is added to the view. In our implementation, we parse the VCD once and then store all signals in a compressed in-memory representation, using many ideas for the FST on-disk format. When a signal is selected, it can be quickly decompressed without any file I/O. The compression rate achieved this way is high enough that this approach works, even for large VCD files. Besides allowing fast access with reduced memory usage, this intermediate storage layer also enables parallel file parsing. We can split up a single VCD file and parse each chunk on a separate CPU core. Performance results for a single example VCD are shown in Tab. 4.

A typical workflow for Surfer consists of the user loading a file and then browsing the signal hierarchy to select signals of interest. Thus, we first parse the header and metadata, including the signal names and hierarchy. For most input files, this can be done in much less than one second, and thus the user is presented with a signal hierarchy almost instantaneously. We continue parsing the actual signal value data in the background while the user is making their signal selection. This has allowed Surfer to avoid a loading screen.

Many of the applications described in Section 3.4 require running Surfer on files that were not generated by the user. Unfortunately, file parsers are notorious for containing exploitable bugs. Our parsing code exclusively uses the safe subset of Rust and therefore Surfer can be used on untrusted input files without having to worry about exploitable memory bugs. Maintaining the modular de-

sign philosophy of Surfer, our parsing library is available under the name `wellen` on GitHub[9]. It has been used outside Surfer to read waveform traces for power analysis[10] and as part of the VaporView VSCode plugin [17]. There is also a python wrapper available. Thus, our focus on modularity has already paid off, enabling the community to develop new analysis and debugging tools.

4.1 Remote Servers and Continuous Integration

Simulation of large designs is often performed on dedicated compute servers rather than the developer's local computer. To view simulation results, developers traditionally have to download the whole trace file to open it up in their local waveform viewer. This transfer takes time and generates a lot of data traffic, with VCD files often measuring in the tens of gigabytes in size. Surfer offers a server mode in which it opens a waveform file on a remote machine and then allows a local instance of Surfer to access the data on demand. As soon as the local viewer receives the meta-data and signal names for the server, the user can browse them and make their selection. Once a signal is selected, the server sends the compressed value change data to the local instance for displaying. Since this feature re-uses our in-memory signal compression, it drastically reduces the amount of data sent over the network while also reducing the user-facing latency since signal data is transferred lazily.

5 Software Project

Surfer is developed as an open-source project[11] under the EUPL license. It was created at Linköping university and has since seen several major contributions from the community. For longevity, the project is now stewarded by the FOSSi foundation. We have recently received funding from the NLnet foundation[12] to implement several highly requested features including signal grouping, drawing of analog signals, and plugin support.

6 Conclusion

Surfer is a new open-source waveform viewer that is built to be extensible and embeddable to enable new workflows in hardware design and verification. A major goal of Surfer is to integrate more high-level information from modern hardware type systems, analysis and verification tools to simplify debugging. Surfer is also the first open-source tool to support interactive simulation. Surfer has been used inside several bigger tools, primarily web-based verification and teaching platforms. We would like to invite the reader to consider what they

[9] `https://github.com/ekiwi/wellen`
[10] `https://github.com/antmicro/trace2power`
[11] `https://gitlab.com/surfer-project/surfer/`
[12] `https://nlnet.nl/`

want to build with Surfer. We like to look back at the project where an M.Sc. student was trying to work on a waveform viewer with support for Chisel types for his thesis. Building such a viewer from scratch would be impossible, but enhancing Surfer with his idea turned out to be a realistic goal [19]. Whether you want to use Surfer for teaching, or as a graphical user interface for your next verification tool, please get involved. We are happy to help you get started.

Acknowledgements The authors are grateful to all external contributors who have helped continuously improving Surfer. An updated list can be found in the repository, but currently these are (full name or GitLab user name):

Alejandro Tafalla	Hugo Lundin	Remi Marche
Andreas Wallner	Jacob Urbanczyk	Robin Ole Heinemann
Ben Mattes Krusekamp	James Connolly	Theodor Lindberg
Christian Dattinger	Kacper Uminski	Todd Strader
ecstrema	Kaleb Barrett	Tom Verbeure
Felix Roithmayr	Lars Kadel	TopTortoise
Francesco Urbani	Lukas Scheller	Verneri Hirvonen
Greg Chadwick	Matt Taylor	Yehowshua Immanuel
Gustav Sörnäs	Ondrej Ille	Øystein Hovind

This work has been partially supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria, and by NSF award number 2426764

References

1. Amaranth contributors: Amaranth HDL (2022), https://github.com/amaranth-lang/amaranth
2. Baaij, C.: Digital circuits in ClaSH. PhD. Thesis, University of Twente (Jan 2015). https://doi.org/10.3990/1.9789036538039
3. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., Asanović, K.: Chisel: Constructing hardware in a Scala embedded language. In: Proc. Des. Automat. Conf. pp. 1212–1221 (Jun 2012). https://doi.org/10.1145/2228360.2228584
4. Basu, S.: Rust as a hardware description language. In: Workshop on Languages, Tools, and Techniques for Accelerator Design (2024)
5. Basu, S.: RHDL: Rust as a hardware description language. In: Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE) (Apr 2025)
6. Behl, R.: QuickSilicon, https://quicksilicon.in/
7. Catherine "Whitequark": CXXRTL debug server concepts, https://cxxrtl.org/protocol.html
8. Catherine "Whitequark": YoWASP - unofficial WebAssembly-based packages for Yosys, nextpnr, and more, https://yowasp.org/
9. Dobis, A., Petersen, T., Damsgaard, H.J., Hesse Rasmussen, K.J., Tolotto, E., Andersen, S.T., Lin, R., Schoeberl, M.: ChiselVerify: An open-source hardware verification library for Chisel and Scala. In: Proc. Nordic Circuits Syst. Conf. pp. 1–7. IEEE (Oct 2021). https://doi.org/10.1109/norcas53631.2021.9599869

10. GTKWave contributors: GTKWave, https://gtkwave.sourceforge.net/
11. Kemmerer, J.: PipelineC: Easy open-source hardware description between RTL and HLS. In: Proc. Workshop Open-Source EDA Technol. (2022), https://woset-workshop.github.io/PDFs/2022/17-Kemmerer-poster.pdf
12. Klemmer, L., Große, D.: WSVA: a SystemVerilog assertion to WAL compiler. In: Workshop on Open-Source Design Automation (2024), https://ics.jku.at/files/2024OSDA_WSVA.pdf
13. Klemmer, L., Große, D.: Versatile Hardware Analysis Techniques – From Waveform-based Analysis to Formal Verification. Springer (2025). https://doi.org/10.1007/978-3-031-83093-8
14. Klemmer, L., Große, D.: WAVING goodbye to manual waveform analysis in HDL design with WAL. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **43**(10), 3198–3211 (Oct 2024). https://doi.org/10.1109/tcad.2024.3387312
15. Klemmer, L., Skarman, F., Gustafsson, O., Große, D.: Surfer: a waveform viewer as dynamic as RISC-V. In: RISC-V Summit Europe (Jun 2024), https://ics.jku.at/files/2024RISCVSummit_Surfer.pdf
16. Lefebvre, S.: Silice (Nov 2022), https://github.com/sylefeb/Silice/tree/5003ec72
17. Lramseyer: VaporView, https://github.com/Lramseyer/vaporview
18. Lubis EDA: Lubis EDA, https://lubis-eda.com/
19. Meloni, R., Hofstee, H.P., Al-Ars, Z.: Tywaves: A typed waveform viewer for Chisel. In: Proc. Nordic Circuits Syst. Conf. pp. 1–6. IEEE (Oct 2024). https://doi.org/10.1109/norcas64408.2024.10752465
20. Nigam, R., de Amorim, P.H.A., Sampson, A.: Modular hardware design with timeline types. Proc. ACM Program. Lang. (PLDI) (Jun 2023)
21. Nikhil, R.: Bluespec SystemVerilog: efficient, correct RTL from high-level specifications. In: Proc ACM/IEEE Int. Conf. Formal Methods and Models for Co-Design. pp. 69–70. IEEE (2004). https://doi.org/10.1109/memcod.2004.1459818
22. Redwood EDA: Makerchip, http://makerchip.com/
23. Redwood EDA: TL-Verilog (Oct 2022), https://web.archive.org/web/20221006080731/https://www.redwoodeda.com/tl-verilog
24. SiLogy Technologies: Silogy, https://silogy.io/
25. Skarman, F., Gustafsson, O.: Spade: an expression-based HDL with pipelines. In: Proc. Workshop Open-Source Des. Automat. (Apr 2023)
26. SpinalHDL contributors: SpinalHDL (2022), https://github.com/SpinalHDL/SpinalHDL
27. TerosHDL: TerosHDL - an open source toolbox for ASIC/FPGA (Oct 2024), https://terostechnology.github.io/terosHDLdoc/
28. Van Hirtum, L., Plessl, C.: Latency counting in the SUS language. In: Workshop on Languages, Tools, and Techniques for Accelerator Design (2024), https://capra.cs.cornell.edu/latte24/paper/4.pdf
29. Venn, M.: Tiny Tapeout: A shared silicon tape out platform accessible to everyone. IEEE Solid-State Circuits Mag. **16**(2), 20–29 (2024). https://doi.org/10.1109/mssc.2024.3381097