Towards a Highly Interactive Design-Debug-Verification Cycle

Lucas Klemmer Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
lucas.klemmer@jku.at, daniel.grosse@jku.at

***Abstract*—Taking a hardware design from concept to silicon is a long and complicated process, partly due to very long-running simulations. After modifying a *Register Transfer Level* (RTL) design, it is typically handed off to the simulator, which then simulates the full design for a given amount of time. If a bug is discovered, there is no way to adjust the design while still in the context of the simulation. Instead, all simulation results are thrown away, and the entire cycle must be restarted from the beginning.**

In this paper, we argue that it is worth breaking up this strict separation between design languages, analysis languages, verification languages, and simulators. We present *virtual signals*, a methodology to inject new logic into existing waveforms.

Virtual signals are based on WAL, an open-source waveform analysis language, and can therefore use the capabilities of WAL for debugging, fixing, analyzing, and verifying a design. All this enables an interactive and fast response design-debug-verification cycle. To demonstrate the benefits of our methodology, we present a case-study in which we show how the technique improves debugging and design analysis.

I. Introduction

In today's rapidly moving industry, design speed and fast iterations are key to getting and staying competitive. When it comes to agility and speed, no industry can keep up with the software industry. But software also relies on fast and efficient hardware as its foundation, and thus hardware companies must find ways to fasten their development cycle to keep up with the demand. In reality, however, hardware development is inherently more complex than software development due to a number of reasons [1]: (1) much higher risk compared to software development, (2) a much more complex process (compiling a binary vs. synthesis, place and route, floor-planning, ...), (3) a far greater dependency on lab time (a single person can write revolutionary new software), (4) far slower and fewer iterations during development (simulations running for multiple days vs. hot-reloading of code in modern software frameworks).

Some of these points are just inherent to hardware design and can hardly be improved (high risk), but there are others that can indeed be improved. In this paper, our focus is on addressing the long delays between changing an *Register Transfer Level* (RTL) design, waiting for the simulation to stop, and verifying the newly introduced changes. We aim to achieve this by reducing the frequency of new simulation runs and providing an integrated environment in which **all** steps of the hardware design process can be performed with fast response times and no clean-cut separations between development, simulation, and verification.

In particular, we want to limit the number of times redundant work is done by providing a fine-grained method for updating existing waveforms. We achieve this by injecting new signals into already existing waveforms that were produced by hardware simulators or by formal tools. This way, only the newly injected signal has to be simulated and not the whole design, dramatically reducing simulation runtime and increasing the iteration speed.

We call these signals **virtual signals**, as they are not included in the original design but are added later, yet they look and behave exactly like the original signals. With virtual signals, existing waveforms can be enhanced, making complicated analysis problems significantly easier to express and enabling developers to iteratively approach their design either for fixing bugs or adding extra logic. Our virtual signals are implemented as an extension to the the open-source *Waveform Analysis Language* (WAL) [2]. By using WAL as a basis, virtual signals can be conveniently defined in a fully fledged programming language. They are therefore even more powerful than traditional signals, for example, they can hold complex data structures like hashmaps, or they can call library functions.

To demonstrate virtual signals, we present two detailed case-studies in which we show how virtual signals can be used to try out bug fixes without performing new simulations after modifying the logic driving an erroneous signal. We leverage the full potential of WAL, by compiling temporal properties – specified as *System Verilog Assertions* (SVA) [3] – to WAL code, thus using them for bug detection and extremely fast verification of our bug fixes, all while still having access to the full simulation data.

II. Related Work

The wish for higher productivity and better development tools is shared by many in industry and academia [4]. Recently, a lot of progress has been achieved by taking modern ideas from software development and transferring them into the hardware domain.

This has been done, probably most famously, by the Chisel *Hardware Description Language* (HDL) [5], a modern hardware generator embedded into Scala. It allows lifting the abstraction of HDL design without loosing fine-grained control over every wire and register. Chisel is not the only such new HDL, many other HDLs emerged in the last few years, each bringing its own set of ideas to the table. Notable examples

for this are CλaSH [6] which is embedded into Haskell from which it draws many inspirations, Spade [7] which provides specialized constructs for pipelining designs and a build tool inspired by Rust's cargo, and Migen/Amaranth [8].

In the context of modern HDLs and RTL code generation, a range of intermediate languages emerged. These include FIRRTL [9], an RTL intermediate representation that allows, among other things, performing device-specific optimizations and RTL generation, and CIRCT [10], another intermediate representation with related tools.

Previous work into hot-reloading of RTL simulations has been presented in [11] and [12]. LiveSim is a framework that allows to update running simulations after changes have been made to the RTL code. The goal of LiveSim is to reduce the edit-run-debug delay, which is the time it takes for a change in the RTL code to become visible in the simulation results. LiveSim achieves a very high simulation performance and reacts rapidly on design changes. In contrast to our proposed virtual signals, however, LiveSim is a very large system on its own and requires adaptions to the development process to integrate LiveSim as a new simulator. Further, it is focused on simulating typical RTL languages. In contrast, virtual signals are not limited to synthesizable constructs or the restrictions of RTL languages. Virtual signals can hold arbitrary data structures, and every virtual signal is in itself a full program. Additionally, virtual signals are tightly integrated into WAL allowing them to interact with other WAL programs. For example, in this paper, we use an SVA-to-WAL compiler to detect bugs and verify that the suggested fixes are correct. All this is possible within the same environment, and it can be easily extended with other WAL programs, for example to analyze the performance impact of the changes or to generate visualizations of the changed design. Finally, virtual signals seamlessly integrate into the development process and existing toolchains without requiring any modifications. This is because they solely rely on waveforms, which are already generated at various stages of the hardware design process.

III. Virtual Signals

This section intoduces the concept of *virtual signals*. Since, virtual signals are implemented on top of WAL, we first provide a short introduction to WAL. Next, we introduce the general idea of virtual signals, show how virtual signals can be defined, how they integrate with other WAL functions, and how they can be used to build higher abstraction layers that allow a more convenient use.

A. Waveform Analysis Language

WAL [2], [13]–[15] is a programming language for debugging and analyzing waveforms created by hardware simulators or formal tools. WAL is based on the idea that **signals from waveforms can be used like variables**, and that **simulation time and design hierarchy are fundamental parts** of the language. Thus, accessing signals in WAL is similar to accessing variables, with the difference that the value returned depends on the loaded waveform and the time at which the signal is accessed.

WAL's syntax is based on Symbolic expressions (abbrev. as S-expressions) which makes the language flexible and easily extendable via powerful macros. For example, counting how often a state machine `tb.dut.state` is in state 1 can be done using the WAL program `(count (= tb.dut.state 1))`. Here, the expression is evaluated at every timestamp of the waveform and the number of timestamps at which it evaluates to true is returned. Note, that the value of the signal `tb.dut.state` is automatically read from the waveform at the correct time by WAL. Now, we can extend this program to count how often the state machine progressed from state 4 to state 1: `(count ( && (= tb.dut.state@-1 4) (= tb.dut.state 1))`. To achieve this we access the previous value of the signal using the `@` operator.

This paper focuses on extending WAL with virtual signals and thus we like to refer to [2] for a more thorough introduction to WAL.

B. General Idea

The core idea behind virtual signals is, that they are additional signals which can be injected into the waveform after simulation time. Compared to "real" signals, the value of a virtual signal is not defined by the loaded waveform but by a WAL expression that is evaluated when the signal is accessed. This means that virtual signals are expressions with the added context of the simulation time. They can be used in other WAL expressions using all methods that are available to regular signals (e.g., relative evaluation, grouping, scoping) and completely behave like other "real" signals that are contained within waveforms.

To implement virtual signals we extended the available open-source implementation of the WAL interpreter[1].

C. Defining Virtual Signals

WAL virtual signals can be defined using the `defsig` function. This function expects two arguments, first the name of the signal that is created and secondly at least one WAL expression that computes the value of the new signal. Please note, that the expressions can be any valid WAL expressions and that virtual signals can hold any valid WAL value (e.g., a signal holding a list of all transactions up until this timestamp). Listing 1 shows how some exemplary virtual signals are created. The first example on Line 1 defines a virtual signal whose value is always equal to the value of the "real" signal *counter* incremented by one. This virtual signal definition corresponds to the Verilog concurrent assignment `assign cnt_plus_one = counter + 1;`. Next, a virtual signal that inverts the *reset* signal to generate an active-low reset is shown on Line 2. Finally, a signal that always holds the index at which a write-enable signal (*tb.we*) was high the last time is shown on Line 3. This signal is defined recursively, if the write-enable is currently high, it's value is the current index, else it's value is the value this signal had at the previous index. This recursive chain

[1] https://github.com/ics-jku/wal

```
1 (defsig cnt-plus-one (+ counter 1))
2 (defsig resetn (- 1 reset))
3 (defsig last-we (if tb.we INDEX last-we@-1))
```

Listing 1: Defining virtual signals.

repeats until the write-enable signal is high or the start or end of the waveform is reached. Virtual signals are evaluated lazily, and their values are cached, and thus this recursive evaluation does not add significant runtime cost.

D. Integration of Virtual Signals into WAL

In this section, we describe how virtual signals are integrated into WAL using the example of a simple bus interface that will be extended for debugging purposes.

Suppose a design contains a simple bus using a ready-valid handshaking scheme. To give developers a better overview of the bus activity, we are going to extend each such bus interface in the design with a new virtual signal holding a list of all transactions that have been sent so far. For each transaction, we will store an *address* and *data* tuple inside an additional virtual signal called *packets*. This signal will hold a growing list of transactions, that have happened on this bus until the index at which the signal is evaluated. If the *rst* signal is high, the virtual signal will be set to hold an empty list (Line 4). If the reset is low, it appends the current transaction to its value if the bus is sending (Lines 5-6), else it keeps its old value (Line 7).

The number of such interfaces in a design can vary depending on the design configuration (e.g., a configuration with 2 vs. a configuration with 3 devices) and this affects the number of virtual signals we have to inject. Doing this manually quickly becomes tedious and therefore, virtual signals should be well integrated with the generic programming features of WAL. In general, from the perspective of WAL programs virtual signals should behave like "real" signals that are loaded from a waveform. The benefit of this is, that they can be directly used by all other WAL features which are already present for "real" signals. This means that they should be available via the `SIGNALS` special variable, they should compose with the grouping and scoping features of WAL, and they should follow the WAL timing semantics (i.e., `step`, and relative evaluation).

Our virtual signal implementation satisfies these requirements, and hence we can utilize WAL's full generic programming features to automate the virtual signal creation. This allows us to scan the design for all instances of the bus using the `groups` function[2] (Listing 2 Line 1) and create new virtual signals at all discovered locations using the `in-groups` function. Inside the body of the `in-groups` function, signals that start with `#` and virtual signals are expanded to an absolute name, depending on the currently visited group. Therefore, by creating the virtual signals inside the `in-groups` function, they are automatically name-extended and placed at the correct locations. and thus we can automate the task of creating virtual signals. After defining the virtual signals, we go over the complete waveform once for each interface with the `whenever`

[2]For a further explanation of the utilized WAL functions, please see [2].

```
1  (in-groups (groups clk rst rdy vld addr data)
2    (defsig packets
3      (if #rst
4        '()
5        (if (&& (rising #clk) #rdy #vld)
6            (append #packets@-1 (list #addr #data))
7            #packets@-1)))
8    (print CG ":") ;; print Current Group
9    (whenever (&& (rising tb.clk)
10                 (! (stable #packets)))
11     (printf "%2d:␣%s\n" INDEX #packets)))
12
```

Listing 2: Extending bus interfaces with virtual signals.

```
1 tb.c1.:
2 12: ((27 232))
3 20: ((27 232) (65 236))
4 tb.c2.:
5  4: ((179 48))
6 12: ((179 48) (185 52))
7 16: ((179 48) (185 52) (194 54))
```

Listing 3: The transactions on two bus interfaces.

function. At each timestamp, we check if a new transaction is added (Line 9-11) to the virtual signal and, if this is the case, print the current index and the *packets* signal (Listing 3).

E. Reg and Wire Macros

Using WAL's macro system, users can extend the language to their needs. In this section, we present two macros that provide a convenient way to define virtual signals. The first macro, `wire`, allows defining new combinational signals using a `(wire name expr)` style similar to Verilog. Signals in the expression of a `wire` definition always refer to the signal value at the current index. In Listing 4, a combinational virtual signal is created to detect if some clearing operation has to be performed.

The second macro, `reg`, allows defining new sequential signals. Similar to sequential logic in e.g. Verilog, signal names in the expression of a `reg` definition refer to the signal value at the *previous* clock edge. Sequential virtual signals require three additional arguments: the clock to which they are sensitive, the reset signal, and the reset value. In Listing 5, a counter register is defined which wraps around after the value 5 and resets to 0.

IV. Debugging With Virtual Signals

In this section, we introduce our design-debug-verification workflow based on virtual signals using a simplified but not far-fetched case-study. Suppose one of the components of a new system contains a counter that, according to the specification, should wrap around after the value 5. However, during development, a member of the design team misread the specification and implemented the wrap around after the value 4, the nearest power of two. These types of errors are commonly referred to as *off-by-one* errors, and they can

```
1 (wire clear (&& cmd_valid (= cmd[1:0] 3)))
```

Listing 4: Defining a combinational signal using the `wire` macro.

```
1 (reg counter [clk [rst 0]]
2   (if (= counter 5)
3       0
4       (+ counter 1)))
```

Listing 5: Defining a sequential signal using the `reg` macro.

occur with remarkable ease. Since bugs of this kind are very common, they are one of the classic bugs in both, software development and hardware design.

In the following case-study, we will present our methodology based on virtual signals in three steps: 1) finding the bug in the waveform using an SVA property, 2) injecting a fix for the bug into the waveform, and 3) verify the bug fix using the same SVA property.

To find the bug inside the waveform we compile the SVA property that uncovered it originally to a WAL function. We then inject a new virtual signal into the waveform that fixes the previously uncovered bug. Next, we simulate this, **and only this** signal, and check, that it did not invalidate the waveform data by deviating from the original signal before the time at which the SVA property first fails. Finally, we run the same SVA property again to verify that our fix indeed is correct. Please note that all of this happens in the same debugging environment and using one unified language, WAL. The verification engineer does not have to use multiple tools or even simulate the whole design again multiple times during debugging. Only after the engineer came up with a fix that passes the SVA property on the waveform, a new full simulation or other kind of regression run is needed. This way, **time-consuming and workflow breaking simulation is reduced to a minimum**. The WAL shell session containing all steps of this case-study is shown in Listing 7.

Step 1: Finding the Bug

We start the bug fixing process with the knowledge that a bug was uncovered by one of the SVA properties that have been checked during regression testing. The first step in fixing a bug using virtual signals is locating the time when the bug first appears in the waveform. For this, we want to utilize the same SVA property that already uncovered the bug. Listing 6 shows the SVA property that failed for our design during simulation. The assertion checks if the counter value correctly overflows after it reached the value 5. However, this is not the case at some point in the waveform generated by the simulator.

As WAL is a fully fledged programming language, it is possible to translate an SVA assertion into WAL a program that checks if the assertion holds on a given waveform. We translated the SVA property into a WAL program using a conversion program. This program generates a WAL file containing a function definition *fp* for each property *p* in the SVA file. With this generated WAL file, checking if property *p* holds on the currently loaded task is as simple as calling the function *fp* by evaluating `(fp)`. To use the generated WAL functions, we first have to import them into the running WAL session. In WAL this is done by evaluating `(require assertions)`, assuming the assertion file was translated into *assertions.wal*. After loading the waveform (Line 1 in Listing 7) and the file containing the assertions (Line 2) we can call the *check_overflow* function (Line 3; representing the SVA property check_overflow) to get a list of all timestamps at which the property fails (this list contains tuples containing the time of the first and last matches of the failing assertion).

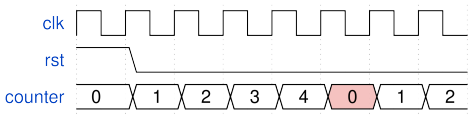

Fig. 1: Waveform containing the wrong counter value (red).

```
1 check_overflow: assert property(
2   @(posedge clk)
3   disable iff (rst)
4   counter == 0 |-> $past(counter) == 5;
5 );
```

Listing 6: The SVA property that detected the bug.

A waveform containing the wrong *counter* value (highlighted in red) is shown in Fig. 1.

Step 2: Injecting the Bugfix

Having spotted the first timestamp at which the assertion fails, we can analyze the bug. In this case, evaluating the assertion function returns a non-empty list, signaling that the assertion failed at some point. Every entry in the list is a tuple describing the failing assertion. The first element of the tuple is the timestamp at which the assertion first matched, and the second element is the timestamp at which the assertion last matched the trace. Since the *check_overflow* assertion checks only one timestamp, both values of returned tuple are the same. In this example, the assertion returns a list with one element pointing to a failure at timestamp 110 (Line 4 in Listing 7).

Next, we print all values of the *counter* signal to get an overview of the context in which the bug occurs (Lines 5-13). Here we can see the wrong *counter* value on Line 11, which is 4 due to the bug in the counter, but should be 5. We will now fix this bug by injecting *counter/new*, a new signal that models the counter value with the correct overflow value. For this, we use the previously created *reg* macro that creates a new sequential signal which is sampled on each rising clock edge (Lines 14-17). By creating a new signal *counter/new* with a different name than *counter* we still have access to the old counter value from the waveform. At this point, we can print the values of both the old and new signals side by side to gain a deeper understanding of the behavior exhibited by the new signal (Lines 28-27). The new virtual signal is also shown in Fig. 2 with the correct maximum value of the counter highlighted in blue.

Step 3: Verifying the Bugfix

At a first glance, the new implementation seems to fix the bug. However, now we need to make sure that our bugfix passes the SVA assertion and that it did not corrupt the other signal values. In general, a corruption can occur if the value

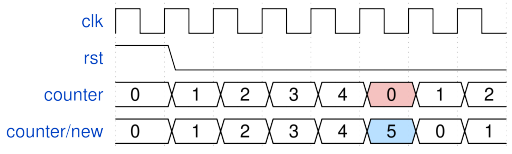

Fig. 2: Waveform containing the wrong counter value (red) and the corrected value of the virtual signal (blue).

```
1  >-> (load "counter.vcd")
2  >-> (require assertions)
3  >-> (check_overflow)
4  ((110 110))
5  >-> (whenever (&& (rising clk) (< TS 140))
6        (printf "%3d: %d\n" TS counter))
7    10: 0
8    30: 1
9    50: 2
10   70: 3
11   90: 4
12  110: 0
13  130: 1
14 >-> (reg counter/new [clk (rst 0)]
15      (if (= counter/new 5)
16          0
17          (+ counter/new 1)))
18 >-> (whenever (&& (rising clk) (< TS 140))
19       (printf "%3d: %d %d\n" TS
20                               counter
21                               counter/new))
22  10: 0 0
23  30: 1 1
24  50: 2 2
25  70: 3 3
26  90: 4 4
27 110: 0 5
28 130: 1 0
29 >-> (check-integrity counter counter/new 110)
30 >-> (alias counter counter/new)
31 >-> (check_overflow)
32 ()
```

Listing 7: Injecting a new counter signal into the waveform. "`>->`" is the WAL shell prompt and `TS` the current timestamp.

of the virtual signal is different than the value of the original signal at any time except the place where the bug occurs. This check is essential, since a mismatch between both signals can invalidate the values of the other signals. This is the case, since we only simulate the new virtual signal, therefore signals that depend on this value are not updated. In cases in which such a corruption occurs, the simulation data is not valid, and we can make no statement about the correctness of our bugfix. In our counter example, this integrity check is not needed since the counter signal is a top-level output of the counter module and no other signal depends on it. However, to show the general principle, we include the check in this example.

Finally, we can check the correctness of the virtual signal using the SVA assertion. Since the SVA assertion checks the correctness of the *counter* signal, we first have to overlay the *counter* signal by the new virtual signal *counter/new*. This is done by introducing an alias for the virtual signal (Line 30 in Listing 7). Aliases always have precedence over real or virtual signals, therefore evaluating *counter* from now on returns the value of *counter/new*. With the old signal overlaid, we can run the SVA property again to see if the new signal corrects the behavior of the old signal (Line 31) on the complete waveform. This time, the assertion returns an empty list, meaning that it did not fail at any timestamp.

V. Case Study: RISC-V Hazard Unit

In this section, we repair faulty forwarding logic inside the hazard unit of a pipelined RISC-V processor. Due to a mistake that happened during the wiring of the hazard unit module the *forwardae* signal, which is a select signal on a multiplexer driving one of the ALU inputs, *srca*, is undefined

```
1 check_forwarding: assert property(
2   @(posedge clk)
3   disable iff (reset)
4   (rs1e == $past(rde)) && regwritem && (rs1e != 0)
5     |-> (srca == aluresultm)
6 );
```

Listing 8: The SVA property that detects the wrong forwarding signal value.

```
1  >-> (load "bug.vcd" t)
2  >-> (require assertions)
3  >-> (check_forwarding)
4  ((90000000 90000000))
5  >-> (step-to-ts 90000000)
6  >-> forwardae
7  0
8  >-> (wire forwardae/new
9        (cond
10         [(&& (= rs1e rdm) regwritem rs1e) 2]
11         [(&& (= rs1e rdm) regwritew rs1e) 1]
12         [else 0]))
13 >-> (alias forwardae forwardae/new)
14 >-> forwardae
15 2
16 >-> (check_forwarding)
17 ((90000000 90000000))
18 >-> (wire srca/new
19       (case forwardae
20         [0 rd1e]
21         [1 resultw]
22         [2 aluresultm]))
23 >-> (alias srca srca/new)
24 >-> (check_forwarding)
25 ()
```

Listing 9: WAL shell session documenting the analysis and repair of the forwarding logic.

in one specific case. If the bug is triggered, the ALU operates on an outdated value and thus can produce the wrong result.

During the verification of the processor, the bug was detected using the SVA property shown in Listing 8. This property checks that the *srca* ALU input is the same as the previous ALU result if the register number *rs1* of ALU operand *a* of the instruction currently in the *execute* stage is the same as the result register *rd* of the previous instruction and the previous instruction produces a result that is written to a register other than *x0*. Like in Section IV, we translated the SVA property to WAL using a compiler.

Listing 9 documents a shell session in which the faulty forwarding logic is debugged and fixed. First, the waveform with the bug (Line 1) and the compiled SVA properties are loaded into WAL (Line 2). Next, the SVA property *check_forwarding* is checked against the waveform (Line 3) and a list of the locations at which the property fails is returned (Line 4). Using the *step-to-ts* function we step forward to the time at which the property fails (Line 5). Here, we check the value of the *forwardae* control signal and see that it is 0 (Line 7). However, since the instructions that are currently in the pipeline produce a hazard, this control signal should be set to 2.

On Lines 8-12 the new logic for the *forwardae/new* signal is defined using the `wire` macro. This logic can detect two kinds of hazards (i.e., possible conflicts with the two previous instructions) and sets the *forwarde* signal accordingly. Now, the original *forwardae* signal is overlaid with the *forwardae/new* signal (Line 13). We can now observe the correct value of *forwardae* (Line 15), however, the SVA property still fails

at the same timestamp (Line 17). This is because the SVA property checks not the *forwardae* but the *srca* signal. The *srca* signal is driven by a simple 3-Mux that uses *forwardae* as the select signal. Hence we can easily inject a new virtual signal to propagate the bugfix to *srca* (Line 18-23). Finally, the SVA property is checked again, this time without failing (Line 25).

VI. Discussion

In this paper, we try to take the first step in breaking up the strict separation between design time, simulation time, and verification time. For this, we presented virtual signals which allow injecting new logic into simulation waveforms after the simulation is finished. By utilizing virtual signals, a novel and highly interactive debugging approach emerges, enabling new possibilities for investigation and analysis. Instead of the need to exit the development environment for every minor code change, virtual signals enable the immediate observation and verification of the changes' impact. Likewise, the reverse scenario holds true: in the event of a bug discovered within a waveform, virtual signals facilitate the analysis and resolution of the bug within the same environment.

However, presently, our virtual signal workflow only works under some assumptions. For example, if a signal is overlaid with a virtual signal, in most cases the virtual signal must behave exactly the same as the original signal for all timestamps up to where the SVA property fails. Additionally, after the values of the two signals diverge, the other values of signals that depend on the changed signal become invalid, as they too have to be now simulated again. We plan to address this in the future by analyzing the cone of influence of a signal. Then, signals that are affected by a virtual signal can be converted into additional virtual signals. Ideally, these steps would have to be performed automatically, for example by a synthesis step that compiles signals of the original design into virtual signals.

Another direction of future work concerns how virtual signals are defined. WAL is specifically designed to make expressing hardware problems as easy as possible. However, the S-expression and prefix notation make it hard for people not familiar with Lisp languages to pick up WAL. On the other hand, the same S-expression notation also significantly simplifies transpiling other languages to WAL. Alternative frontends for WAL are already available: Including WAWK, which is a waveform analysis language inspired by AWK, and WSVA, the SystemVerilog Assertions frontend we used in Section IV and Section V. Both, AWK and SVA are already widely used by hardware engineers, providing them an easy entry into the WAL system. Adding a new frontend that translates Verilog or VHDL to WAL would make virtual signals available to hardware engineers without any knowledge about the underlying technology. This also highlights a major strength of using WAL as the backend for virtual signals: all projects that utilize WAL are automatically compatible with each other. Further, as WAL is centered around waveforms, which are produced in large quantities during simulation and formal verification, it is completely supplementary to existing toolchains and workflows. In future work, we will investigate how virtual signals can be made permanent by propagating them back up into the original design.

We believe that improvements in productivity are necessary to effectively meet the ever-increasing demands placed on domain-specific hardware [16]. One approache to achieve this objective is by minimizing simulation overhead and streamlining the development process. Even though virtual signals still limitated in their current form, we are confident that they are a promising new direction towards a modern and more interactive hardware development flow.

VII. Conclusion

In this paper, we presented virtual signals, a novel methodology that allows injecting new logic into existing waveforms. Virtual signals allow an interactive development flow similar to what is known as hot-reloading in the software domain. By showcasing a simple but compelling example, we presented how virtual signals can be used to try out bugfixes and see their effect immediately without running a full simulation. Since virtual signals are based on WAL, they have the power of a full-fledged programming language and can benefit from other, for example, verification related, WAL projects.

Acknowledgments

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

[1] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, 2010.

[2] L. Klemmer and D. Große, "WAL: a novel waveform analysis language for advanced design understanding and debugging," in *ASP-DAC*, 2022, pp. 358–364.

[3] *IEEE Std 1800-2017: IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*. IEEE.

[4] A. Rautakoura and T. Hämäläinen, "Does soc hardware development become agile by saying so: A literature review and mapping study," *ACM Transactions on Embedded Computing Systems*, 2023.

[5] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC*, 2012, p. 1216–1225.

[6] C. Baaij, M. Kooijman, J. Kuper, W. Boeijink, and M. Gerards, "Cλash: Structural descriptions of synchronous hardware using haskell," in *DSD*. IEEE, 2010, pp. 714–721.

[7] F. Skarman and O. Gustafsson, "Spade: an expression-based HDL with pipelines," in *Proc. Workshop Open-Source Des. Automat.*, 2023.

[8] "Amaranth HDL," https://github.com/amaranth-lang/amaranth, 2023.

[9] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *ICCAD*, 2017, pp. 209–216.

[10] CIRCT contributors, "CIRCT / Circuit IR Compilers and Tools," https://github.com/llvm/circt/tree/main/, 2023.

[11] H. Skinner, R. Trapani Possignolo, S.-H. Wang, and J. Renau, "Livesim: A fast hot reload simulator for hdls," in *ISPASS*, 2020, pp. 126–135.

[12] S.-H. Wang, R. T. Possignolo, H. B. Skinner, and J. Renau, "Livehd: A productive live hardware development flow," *IEEE Micro*, vol. 40, no. 4, pp. 67–75, 2020.

[13] L. Klemmer and D. Große, "Waveform-based performance analysis of RISC-V processors: late breaking results," in *DAC*, 2022, pp. 1404–1405.

[14] L. Klemmer, E. Jentzsch, and D. Große, "Programmable analysis of RISC-V processor simulations using WAL," in *DVCon Europe*, 2022.

[15] F. Skarman, L. Klemmer, O. Gustafsson, and D. Große, "Enhancing compiler-driven HDL design with automatic waveform analysis," in *FDL*, 2023.

[16] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, p. 48–60, 2019.