Hardware Deserves a REPL

Karl Hallsby
Northwestern University
Evanston, IL, USA
kgh@u.northwestern.edu

Peter Dinda
Northwestern University
Evanston, IL, USA
pdinda@northwestern.edu

Abstract

Hardware design languages at the register transfer level have seen a resurgence in research popularity. Despite the tremendous amounts of work put into both traditional HDLs and modern alternatives, none offer a development velocity that can match software's. We believe that the root cause of this slowdown is the latency in designer feedback: hardware languages and their tools are not interactive.

We argue that hardware design languages could have development velocities matching those of software if they take advantage of interactive REPL-based development. But why leave the language itself as the only interactive component? We further argue that in addition to the language being interactive, so should the build system, the testing of the designed hardware with simulators, and the debugging of the hardware simulation.

1 Introduction

There has been significant progress in Register-Transfer Level (RTL) hardware design languages (HDLs) from both academia and industry, yet hardware development remains a slow and arduous process *relative to software*. Chisel [5] is an Embedded Domain Specific Language (eDSL) that tackled this problem by allowing users to easily write type-safe reusable hardware generators. Alternatively some companies are eschewing traditional and modern HDLs and High-Level Synthesis (HLS) in favor of developing their own solution. Google's XLS [7] can lower a design all the way from a software-like specification with behavioral/structural specifications to a statically-scheduled Verilog design under ideal-timing constraints.

We believe part of hardware design's slowdown comes down to the inability to *quickly* test ideas in a low-stakes environment. Software has had a tool for this exact task for eons, the REPL (Read-Evaluate-Print Loop). Software developers are familiar with using a REPL to quickly test an idea before committing to it in a file. REPLs allow users to hack away at ideas, completely ignoring build systems, compilation ordering, etc. The notebook system popularized by Python and Jupyter Notebooks [12] is the modern graphical interpretation of a REPL.

eDSL hardware languages happen to include a REPL by virtue of being embedded in a software language that has a REPL, but do not focus on the REPL or leverage its features. Cλash (Clash) [3, 4], Amaranth [1], and Kratos [18] all behave this way. While all of these languages are impressive efforts in their own rights, none were designed with interactivity in mind. We believe that designing an HDL and its surrounding tooling with interactivity in mind will:

- Shorten iteration time by testing ideas quickly.
- Allow designing new tests without large infrastructure.
- Enable interactive and time-travel debugging.
- Offer deep cross-referencing capabilities between designs, tests, and simulation outputs.

We have been developing a brand new hardware design ecosystem designed from the ground-up for seamless hardware-software interaction and tight REPL development called Chil. Chil is an all-encompassing project whose lofty vision includes several goals: an embedded domain-specific HDL in Common Lisp, a build system that treats compilation and elaboration artifacts as records in a database and provides a query language to work with it, a compilation driver tool to make complex tasks declarative, and a debugging simulator.

The overarching theme of this vision is that tightly-integrated interactive development for hardware will give hardware designers greater flexibility to quickly experiment with and iterate on ideas, akin to their software brethren.

2 REPLs

REPLs do exactly what they sound like; they allow the user to type away at a command-line (or over a socket/pipe from your favorite editor) and submit code to evaluate. When evaluation completes, the REPL prints the result, if there is any. This short turnaround development cycle encourages and epitomizes the iterative development cycle; the "save-compile-run" cycle turns into a single press of the Enter key, allowing ideas to be tested in a self-contained environment, without needing to interact with a build system whatsoever. Interactive notebooks take this cycle to the next level, allowing complex visualization to be done with just a keypress.

Language REPLs often leverage the language's compiler, but sacrifice code optimization for interaction speed, and usually interprets either a bytecode or the AST/IR directly. REPLs are common in many modern languages today, including: Lisps/Schemes, Python, Scala, Swift, Haskell, and JavaScript. Python's REPL allows machine learning engineers to experiment with complex GPU code rapidly.

2.1 Deeply-Integrated Software REPLs

That being said, not all REPLs are built the same. Some are merely thin wrappers over the language's interpreter itself, e.g. Python [13]. On the flip side, Lisp REPLs showcase the capabilities of a deeply-integrated REPL-based interactive development environment. When Common Lisp encounters an unhandled runtime error (REPL **or** pre-compiled), an interactive debugging prompt stops the program with all the information that led up to the error available. This debugger can inspect and modify *all* aspects of the program, allowing the user to change values, evaluate arbitrary expressions,

LATTE '26, Pittsburgh, PA, USA

re-execute the failing step, *redefine functions and variables*, and even rewind the callstack and start again elsewhere!

Some of the deeper powers of Lisp REPLs are hidden away by default. Knowledgeable text editors can launch these REPLs with additional flags to open more information flows to gain deeper insight about the program(s) at hand, effectively making them full-fledged IDEs. These additional flows give Lisp REPLs similar project-level navigation as LSP (Language Server Protocol) implementations [9], including introspective online documentation, go-to definitions, go-to uses, and even auto-completion suggestions, all *without* requiring a separate program, configuration, or setup.

3 Pervasive Deeply-Integrated REPLs for Hardware

We are designing Chil's language from scratch rather than building on top of already-existing hardware eDSLs because we were interested in seeing how a hardware language designed around and for REPL interactivity changes the language's designs and goals. Further, we want REPL interactions to be pervasive throughout the Chil ecosystem; from the language itself, to the language's build system, to the simulators that test the design, all the way to the debuggers used on the simulators. There is nothing technical preventing an already-existing hardware eDSL in software languages with REPLs from providing the kind of hardware REPL we envision, although the difficulty depends on the eDSL's implementation. Chisel makes this more difficult because it is both a DSL library *and* a Scala compiler plugin. Whereas Amaranth (or any hardware eDSL written in Python that provides its DSL constructs as a "regular library") would lend itself to such an extension, particularly with Python's strong runtime reflection features.

Providing a language with deeply-integrated interactivity would be powerful, but that is only the beginning. REPLs integrate the programming language and its tools into a single environment; Lisp REPLs allow one to not only write code, but also debug it, compile it, and systematically deploy it all without leaving the REPL. Importantly, they an *explore* a problem and potential solutions before taking one as "the solution".

A significant amount of designer energy goes into optimizing non-functional aspects of a design; optimizing frequency, pipelining a design, reusing hardware subcomponents, etc. all of which require exploring solutions. Such optimizations require careful understanding of the implementation at hand. We envision the language's REPL allowing designers to quickly experiment with different module implementations while the simulator's hot-reload and debugging (§3.2) capabilities would provide immediate feedback. An interesting feedback possibility opens up when the REPL is the main driver; designers can interactively query design characteristics, such as timing or resource estimation, while iterating on their design!

3.1 Build System

When connecting to the build system, we **do not** mean that the REPL is just "able to invoke `make`". We mean that the build system (and by extension the compiler) are visible to the REPL and able to be modified interactively as well. Not only could the designer start a build, but the build system could also offer feedback, or the designer could add a custom phase to compilation without permanently committing to it.

In our vision, the build system and compiler become a database-backed daemon allowing designers to query the project and compiler instead of writing separate tools. This is not a new idea; Montana's CodeStore [8, 17] provided a database that enabled developers to directly interact with the C++ compiler and its internal artifacts. Montana leveraged this to offer code formatting suggestions based on the code's AST. More recently, rust-analyzer has this same feature and uses Salsa [14] for their database and queries. The database-structured approach would allow for developers to write declarative queries to perform analyses on the code the *compiler* sees, without needing to use intermediate files and external tools. This is immediately useful for driving automated "on-save" operations, such as formatting, linting, or running tests only for the tree of *modules* that have changed.

3.2 Simulations

In our vision, defining a hardware module is analogous to defining a software function. The natural thing to do after defining a function is to test it, particularly in an interactive REPL. This is often where the illusion of "hardware is software" falls apart, because unless the hardware module is purely combinational, the notion of time immediately bleeds through. To help alleviate this, we envision a REPL-friendly simulation-focused DSL that can composably build complex test harnesses, allowing designers to drive simulations, manually tick the simulator, reset simulations, etc.

This space of reducing simulation turnaround time has seen many approaches. Cascade [15] is one such approach that reduced the turnaround time to starting simulations of designs that are put on FPGAs. Cascade performs JIT (Just-In-Time) synthesis of designs, running them through traditional software simulation while synthesizing in the background, eventually seamlessly transferring the simulation to the FPGA. Their goal was to reduce the amount of time required for the designer to see the effect of their HDL on an FPGA. We currently do not have plans to work with FPGAs, nor accelerating design simulation with FPGAs, but their state snapshotting approach could prove useful.

3.3 Debugging and Time-Travel

We want to enable users to interactively run their simulations; but this is not enough for us. We believe users really want a way to interactively *debug* their interactive simulation. Calyx [10] recognized this exact situation and built Cider [6] to meet Calyx's needs. One of our end-goals for interactive debugging is time-travel debugging, allowing users to freely jump both forward and backward through the simulation.

Tying simulators and debuggers together through the REPL produces a tight feedback loop. Minor errors can be corrected quickly without performing a full recompile and run cycle, while more insidious faults can be interactively investigated to find their root cause. Once the error is identified, a REPL-focused workflow would allow for hot-reloading of hardware modules, hopefully *while* the simulation is running!

We show a mockup of our vision of the debugging simulator and its ability to perform operations in a REPL with the simulator

in Listing 1. The example starts by running a simulation until an assertion is violated, at which point the simulation is stopped so that users can debug the simulation and/or the design. The user can then ask for the history of events seen by the simulator, *time-travel* to a different event (analogous to a clock tick), and ask for a trace of what happened during that cycle. With this information, the designer can edit the design's source, fixing the bug, reload the design into the simulator, and then restart the simulation.

```
* (run-simulation example-dut.chil)
...
Assertion violation in example-dut.chil:54.
Amount of data to handle > internal buffer length!
Entering debug REPL.

(debug) [1] * (history)
Clock 80:
  180: (recv listen #0<promise>)
  181: (recv msg #0<fulfilled-promise> fulfill
             (assign amt-remaining -1))
  182: (recv msg #0<fulfilled-promise> fulfill
             (assign amt-data -1 :hidden 't))
  183: (recv msg #0<on-listener> fulfill
             (assert (<= amt-data buffer-entries)))

(debug) [1] * (event-debug 180)
NOTE: Entering a sub-REPL.

(debug) [2] * (trace-history)
Clock 74:
  174: (next-state writing-done? 'compute-loop-variables)
  175: NOTE: Now in compute-loop-variables state
  175: (assign amt-remaining (- buffer-size num-elements))
  175: (defconstant buffer-entries 8)
  163: (assign num-elements 9)
  175: (assign amt-remaining (- 8 9))
  176: (next-state 't 'fetch1)

;; Edit design in text editor & save
(debug) [2] * (reload-design example.chil)
(debug) [2] * (restart-simulation)
...
Simulation completed successfully in 145 cycles.
```

Listing 1: A fabricated example interaction with Chil's debugging REPL. This highlights the ability to interact with the simulator, reverse time, reload the design, and restart the simulation.

Hot-reloading a design after a change is a key desired feature of Chil's simulator. The amount of time required to hot-reload the simulator is an important factor, because waiting too long will break the designer's flow. LiveSim [16] is a simulator designed with this latency in mind and hot-reloads modules *quickly*. In particular, LiveSim was designed to target and achieved a 2 second turnaround time between saving the file and the simulation being updated to reflect the change, even for very large designs. Chil's simulator should take notes from LiveSim's to keep reload times as short as possible.

4 Current Efforts

The Chil[1] project is currently in *very* early development and exposes itself to Lisp as a collection of libraries. Chil currently has a very rudimentary hardware eDSL that resembles traditional HDLs and can generate Verilog module declarations that pass Verilator's linter. The language leverages the tools already available in Common Lisp, so writing Chil modules can immediately access the REPL's rich interactive powers, including completion candidates and auto-completion, type errors that throw to a debug prompt, and even altering select components of the module (name, IO, parameters) without fully redefining the module.

While working on the language, we also started investigating time-travel debugging. We used rr [11] (which uses GDB internally) with multiple simulators and several designs. rr allows software programs to be easily recorded and later replayed/reversed with a single command, while programs only run slightly slower (10s of percent). In all of our tests, we found that rr trace sizes were not affected by simulation level (functional vs. gate-level) nor design complexity. Table 1 compares the simulated designs, the simulation level and simulation generator used, and the software run by the simulations (test cases). Interestingly, a standardized instruction unit test on Rocket-Chip [2] with Verilator produced an rr trace that is not significantly larger than the simplest `AND`-gate with CVC, despite running for tens of thousands of cycles longer. This continues to hold true when scaling to Out-of-Order cores with completely custom `crt` "firmware" that run larger-scale integration tests that use multiple-privilege C-style exception handling.

Table 1: rr Trace Sizes remain reasonable across multiple usage dimensions, including the design's complexity, the simulator used and its simulation level (functional vs. gate-level), and the program run by the simulated hardware.

Design	Simulator	Test Case	Trace Size
2-input `AND`-gate	Verilator	All Cases	34 kB
2-input `AND`-gate	CVC	All Cases	21 kB
Rocket-Chip	Verilator	Insn. Unit Test	1.6 MB
M68k	CVC	UART Transfers	35 kB
BOOM	Verilator	Custom `crt` test	3 MB

Using rr, we were able to interactively debug Verilator simulations both forwards *and backwards*! The debugging experience was less than enjoyable, since the simulators are not easily debugged in general. The main difficulty is connecting the symbol and package hierarchy from Verilog to the generated software simulator, which must be done entirely manually. Users must read both the simulator's input HDL and the simulator's generated software and manually "connect" these two symbols. Even then, the user must be aware of where in the software's generated module hierarchy the symbol lies. Prior work found heuristic solutions that alleviate the problem of matching simulator symbols to RTL names which gave acceptable results [18], but was limited to subtrees of the module hierarchy; it could not identify a single symbol across a whole design. We believe ergonomic time-travel debugging can be achieved by co-designing the language, compiler, simulator, and debugger to cooperate and share information.

Our next steps with the language are to flesh out how we represent and write module bodies and connect the necessary machinery to make project-level cross-referencing work. Once this baseline eDSL is completed, we intend to layer additional DSLs atop. We have already begun considering a DSL specifically for representing state machines for example.

[1] https://github.com/chil-hw

References

[1] Amaranth Project. 2025. *Amaranth HDL*. Amaranth Project. https://github.com/amaranth-lang/amaranth

[2] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Benjamin Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical UCB/EECS-2016-17. University of California - Berkeley, Berkeley, California. 11 pages. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[3] C.P.R. Baaij. 2015. *Digital Circuit in CλaSH: Functional Specifications and Type-Directed Synthesis*. Ph. D. Dissertation. University of Twente, Netherlands. doi:10.3990/1.9789036538039 eemcs-eprint-23939.

[4] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell. In *Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD '10)*. IEEE Computer Society, USA, 714–721. doi:10.1109/DSD.2010.21

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. doi:10.1145/2228360.2228584

[6] Griffin Berlstein, Rachit Nigam, Christophe Gyurgyik, and Adrian Sampson. 2023. Stepwise Debugging for Hardware Accelerators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 778–790. doi:10.1145/3575693.3575717

[7] Google. 2025. *XLS: Accelerated Hardware Synthesis*. Google. https://github.com/google/xls

[8] Michael Karasick. 1998. The architecture of Montana: an open and extensible programming environment with an incremental C++ compiler. *SIGSOFT Softw. Eng. Notes* 23, 6 (Nov. 1998), 131–142. doi:10.1145/291252.288284

[9] Microsoft. 2025. *Language Server Protocol*. Microsoft. https://microsoft.github.io/language-server-protocol/

[10] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 804–817. doi:10.1145/3445814.3446712

[11] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. *Engineering Record And Replay For Deployability: Extended Technical Report*. arXiv:1705.05937 [cs.PL] doi:10.48550/arXiv.1705.05937

[12] Project Jupyter. 2025. *Jupyter Notebooks*. Project Jupyter. https://jupyter.org/

[13] Pablo Galindo Salgado, Łukasz Langa, Niklaou Lysandros, and Emily Morehouse-Valcarcel. 2024. *PEP 762 – REPL-acing the default REPL*. Python Enhancement Proposals. https://peps.python.org/pep-0762/#motivation

[14] Salsa Team. 2026. *R: A Language and Environment for Statistical Computing*. https://salsa-rs.netlify.app/

[15] Eric Schkufza, Michael Wei, and Christopher J. Rossbach. 2019. Just-In-Time Compilation for Verilog: A New Technique for Improving the FPGA Programming Experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 271–286. doi:10.1145/3297858.3304010

[16] Haven Skinner, Rafael Trapani Possignolo, Sheng-Hong Wang, and Jose Renau. 2020. LiveSim: A Fast Hot Reload Simulator for HDLs. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 126–135. doi:10.1109/ISPASS48437.2020.00028

[17] D. Soroker, M. Karasick, J. Barton, and D. Streeter. 1997. Extension mechanisms in Montana. In *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering (ICCSSE '97)*. IEEE Computer Society, USA, 119–128. doi:10.1109/ICCSSE.1997.599883

[18] Keyi Zhang, Mark Horowitz, Zain Asgar, and Clark Barrett. 2022. *Source-level debugging for hardware generator frameworks*. Ph. D. Dissertation. Stanford University, Stanford, California. https://purl.stanford.edu/wb864cp0807

Received 30 January 2026; Accepted 18 February 2026