# Marlin: A Hardware Testing and Simulation Frontend in Rust

Ethan Uppal
Cornell University
USA

## ABSTRACT

Marlin is a library that bridges SystemVerilog and modern hardware description languages (HDLs) into Rust [3]. Hardware simulators like Verilator [8] and simulator frontends like cocotb [5] require tinkering with build systems. In contrast, Marlin enables simulation and testing of hardware designs while invisibly integrating with Rust. It supports modern HDLs as first-class instead of through intermediate compilation. Marlin is publicly available at https://github.com/ethanuppal/marlin.

## 1 INTRODUCTION

There are many tools for hardware simulation and testing. Verilator, for instance, compiles SystemVerilog into multi-threaded C++ libraries. Testing can be done by directly working with simulators or through frontends like cocotb, which abstracts over many simulators and exposes an asynchronous Python API. Verilog also has native constructs for writing testbenches that simulators can execute.

However, these methods have drawbacks. SystemVerilog is not an optimal programming environment for complicated sequential testing. Working with libraries generated by Verilator involves a disconnected process of regenerating header files on any interface change and linking with generated libraries. Cocotb does offer a unified interface, but using it requires Makefiles or custom Python runners, which complicates the testing setup. Furthermore, because it loads designs at runtime, it cannot offer editor completion or static checking for port fields.

Modern HDLs further complicate using these tools because they have a seperate translation step to, *e.g.*, SystemVerilog. Spade [7], for instance, uses various “hacks”: the Verilator integration involves confusing preprocessor macros, and the cocotb integration uses ordinary comments to convey information to the test driver. Also, Spade has algebraic data types, so in both integrations, ports are written and read as strings parsed as Spade code:

```
SpadeExt(dut).i.a = "&Some(Escaped::Yes(20))"
```

Listing 1

In Veryl [6], SystemVerilog and Python testbenches are written inline alongside Veryl code.

This paper introduces Marlin, a Rust simulation frontend with the design philosophy that things should “just work.” The main contributions of Marlin are:

- Editor integration with HDL code using Rust’s language server
- Native support for modern HDLs
- Direct linking with simulators like Verilator instead of cocotb’s usage of the slower [1] verification procedural interface (VPI)
- The ability to write in Rust, a systems language with a powerful type system consistently voted most-loved programming language [2]

Marlin transparently integrates with the entire Rust ecosystem, giving access to a wide array of packages, excellent tooling, and its testing framework. For instance, after writing a game in RTL, you can use ordinary Rust to render to a window and call into the RTL for graphics and logic.

## 2 DESIGN

Marlin comes with prebuilt integration for SystemVerilog and Spade (with experimental Veryl support), and offers:

- A declarative API for writing tests in plain Rust, treating the hardware models as ordinary `struct`s with fields and member functions
- A safe procedural API for dynamically constructing bindings to SystemVerilog and interacting with opaque hardware models at runtime
- A library for any HDL that compiles to SystemVerilog to get all of the above

Suppose we have written some RTL:

```
module Wire(
    input[31:0] a,
    output[31:0] b
);
    assign b = a;
endmodule
```

Listing 2

To use this design in Rust, we declare it as a `struct`:

```
#[verilog(src = "wire.sv", name = "Wire")]
struct Wire;
```

Listing 3

Then, we can create a `Wire` using the Verilator runtime:

```
let runtime = VerilatorRuntime::new(...);
let mut wire = runtime.create_model_simple::<Wire>()?;
```

Listing 4

Now, wire can be used like a normal Rust value:

```
wire.a = 5;
wire.eval();
assert_eq!(wire.b, 5);
```

Listing 5

## 3 IMPLEMENTATION

To simultaneously support static checking with designs from any HDL without requiring additional build steps beyond the standard way of running Rust code, RTL is compiled for simulation during program execution (currently with Verilator) and dynamically linked into the program. The runtime maintains an artifacts directory and incrementally builds and caches intermediate files so that the amortized runtime over repeated executions is effectively the same as if the RTL was recompiled upfront on any change.

The #[verilog] annotation shown in listing 3 is a metaprogramming construct (in this case, a *procedural macro*) that parses the contents of the specified Verilog file and generates a complete type definition along with instructions on how to perform combinational evaluation of the design. Similar annotations exist for Spade and Veryl, and it is relatively trivial to add a corresponding annotation for any other modern HDL.

## 4 CURRENT & FUTURE WORK

There are many small ways Marlin can be improved, for instance, supporting FST tracing or the option to use a build script for static linking (despite dynamic linking enabling many of Marlin's contributions).

Currently Verilator is the only supported runtime, but any tool that compiles RTL to a C library can be added as an alternate runtime, such as the Yosys project CXXRTL [4]. CXXRTL additionally supports querying the signals of internal modules, which could allow Marlin to bridge the full design hierarchy into Rust.

Modern HDLs like Spade and Veryl have type systems more expressive than SystemVerilog's — for example, Spade supports algebraic data types — and with well-defined semantics for how complicated values translate into bit vectors. These HDLs can implement procedural macros to generate the Rust equivalents of their custom types and use those types in port fields. There is currently a proof-of-concept implementation of this functionality for Spade[1]. Consider the following Spade design:

```
entity option_out(
  valid: bool, x: uint<6>, result: inv &Option<uint<6>>
) {
    set result = &if valid {Some(x)} else {None};
}
```

Listing 6

The syntax may be familiar to Rust programmers. inv & is used to declare an output port, Option is a sum type that represents the presence or absence of a value, and if is an expression.

We can then use this design from Rust:

```
spade_types!();

#[spade_marlin(top = "spade_marlin::option_out")]
struct OptionOut;
```

Listing 7

Spade has a similar module system to Rust, so the procedural macro spade_types!() generates types nested in Rust modules that correspond to Spade ones:

```
pub mod spade_types {
    pub mod std {
        pub mod option {
            enum Option<T: SpadeType> {
                None {},
                Some { value: T },
            }
        }
    }
}
```

Listing 8

(Types other than Option were omitted for brevity.) Then, OptionOut is generated with a result field of type std::option::Option<SpadeUint<6>> instead of u8. Thus, we can interact with the design using these types instead of raw bit vectors:

```
dut.i.valid = false;
dut.eval();
assert_eq!(main.o.result, None);
```

Listing 9

(In the particular cases of bool and Option, an optimization is made to translate Spade bool/Options to Rust's native bool/Options.)

Many RTL designs are parametrized. Currently, Marlin can only interface with non-parametrized designs. However, because it uses dynamic linking, it is possible to bridge parameters SystemVerilog in constant integer parameters in Rust, or even type parameters in Spade to ordinary type parameters in Rust.

Cocotb supports an asynchronous programming model, using Python's async and await keywords to abstract over waiting for clock cycles. Rust also allows for custom asynchronous runtimes and has its own async/await keywords, so Marlin could be extended to support a similar programming model.

Marlin has been used in two university courses: one at Luleå University of Technology[2] and one at Technical University of Denmark[3].

[1]https://gitlab.com/spade-lang/spade/-/tree/marlin-in-spade/spade-marlin

[2]https://github.com/perlindgren/vips

[3]https://docs.spade-lang.org/agile/intro.html

## REFERENCES

[1] 2024. Connecting to Verilated Models - Verification Procedural Interface (VPI). Retrieved January 30, 2026 from https://web.archive.org/web/20251228071240/https://verilator.org/guide/latest/connecting.html#verification-procedural-interface-vpi

[2] 2025. Stack Overflow Developer Survey 2025. Retrieved January 31, 2026 from https://survey.stackoverflow.co/2025/

[3] Rust. Retrieved January 30, 2026 from https://rust-lang.org/

[4] Catherine 'whitequark'. CXXRTL. Retrieved from https://github.com/YosysHQ/yosys/tree/main/backends/cxxrtl

[5] cocotb contributors. cocotb. Retrieved from https://www.cocotb.org/

[6] Naoya Hatta, Taichi Ishitani, and Ryota Shioya. 2024. Veryl: A New Hardware Description Language as an Altarnative to SystemVerilog. Retrieved from https://arxiv.org/abs/2411.12983

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

[8] Wilson Snyder, Paul Wasson, Duane Galbi, Geza Lore, and et al. Verilator. Retrieved from https://github.com/verilator/verilator