RustHDL - Rust as a Hardware Description Language

Samit Basu
Fremont, California USA basu.samit@gmail.com

***Abstract*—RustHDL is an open source framework in which the Rust programming language is repurposed for describing hardware. It aims to bring the benefits of the Rust programming language to the task of firmware design, and focuses on enabling the use of features such as strong typing, ease of design reuse, and rigorous safety and linting in the implementation of firmware. RustHDL has a fairly extensive library of IP cores for common hardware tasks, and has been fielded in commercial systems. This paper describes the core concepts of RustHDL, and the lessons learned from the initial releases.**

***Index Terms*—Hardware description languages, Rust Programming Language, Field Programmable Gate Arrays, Design automation**

I. Introduction

There are a number of new hardware description languages (HDL) [1]- [5] that are being introduced that attempt to remedy some of the shortcomings of the traditional Verilog or VHDL based workflow. These HDLs tend to focus on newer features borrowed from the rapidly advancing fields of software programming language design and compiler development. Typically, these new HDLs come with custom toolchains, new syntaxes and grammars, as well as new ways of thinking about how hardware designs should best be expressed in text.

There is, however, an additional challenge that cannot be understated. Hardware development is typically quite difficult for those with traditional software engineering backgrounds. The procedural, imperative mode of software development does not lend itself naturally to the design of hardware systems. As such, it is important to consider how "natural" an HDL might feel to the developer using it. As an example, MyHDL [3] is a Python-based HDL that describes circuit function in terms of generators, which are functions that return values over time. The engineer is thus not exposed to the concepts of blocking and non-blocking assignments, priority assignments, etc. Instead, the engineer writes fairly normal looking Python code, and the framework takes care of the translation to Verilog.

The use of a programming language as the basis for an HDL is not new. However, modern programming languages offer significant advantages over their predecessors. The Rust programming language (RPL) includes a number of powerful features in a mainstream language such as:

- Strong static typing
- Generic and const-generic programming
- Functional programming features
- A powerful package manager and ecosystem
- Built in test automation and documentation features
- High performance and multithreading
- A powerful macro system

These features lead the author to attempt to use the RPL as the basis for a hardware description language. The goal was to leverage the power of the RPL to make the development of hardware designs easier and safer, and to bring the benefits of RPL to the task of hardware design. The result of this effort is the open source RustHDL framework that allows for the development of firmware for Field Programmable Gate Arrays (FPGAs) using the Rust programming language. The framework has been used to develop commercial grade firmware, and has been fielded in commercial products. A large number of sample designs are available on `crates.io`, and the framework has seen some moderate level of adoption by the open source community.

The key components of RustHDL are:

- The use of the Rust type system to describe complex data without a synthesis overhead, and independently of the underlying toolchain's support for types.
- The use of simple structs and composition to describe hierarchies of design with encapsulation and the hiding of internal details.
- The use of Rust itself as the programming language. No new grammar is required and all of the infrastructure for supporting the Rust programming language, including training, compilers, linters, editors, etc. "just work" with RustHDL. As a side effect, significant issues such as code sharing, documentation, and automated testing of hardware designs are all handled by the Rust ecosystem.
- The ability to integrate external IP cores and legacy designs (e.g., memory controllers, serializer/deserializers, PLLs, etc.) which cannot be described from first principles.
- High performance simulation of hardware designs is built into the framework, so that testbenches and verification can be performed without the use of additional tools.

The organization of this paper is as follows. Section II describes related works. This is a particularly fruitful time for innovation in this space, and only a sample of Rust-related projects are discussed. Section III describes the basics of how Rust is used as an HDL using the RustHDL framework. This includes a discussion of how RustHDL leverages existing infrastructure for the Rust programming language to ease collaboration, testing, and sharing of hardware designs. Section IV discusses the limitations observed from the initial

release of RustHDL, and the motivation for a rewrite. Finally, Section V presents conclusions.

II. Background

There are many modern approaches to HDL in development (see [1] and the references therein for example). This section will focus exclusively on HDLs that feature the Rust programming language as a basis. From a high level programming languages can influence HDLs in one of two ways:

- Through grammar similarity. A number of HDLs use syntaxes that are inspired or similar to programming languages in order to make the transition from software to hardware design easier. For example, Chisel [2] uses a Scala-like syntax, and Spade [1] uses a Rust-like syntax. The underlying implementation language is irrelevant (it happens to be Rust in the case of Spade). What is important is that the background experience and knowledge of the developer can be brought to bear when understanding hardware descriptions if they use syntax and patterns from a broadly used programming language.
- As a host environment for the design. In this case, a subset of the programming language (very similar in nature to a "synthesizable subset") is carved out of the broader programming language through some means, and then used to generate hardware designs within the context of the overall programming language. An example here is MyHDL [3], which uses Python as the host language, and designs are expressed in a subset of Python that is synthesizable into Verilog.

In the first category, the most prominent examples are XLS [4] and Spade [1] and Veryl [5]. In all three of these cases, the language grammar and type system are inspired by Rust, but the actual tooling is separate from the Rust programming language and ecosystem. In the case of [1], [5], the relevant compilers are also written in Rust. But XLS, which is quite close to Rust syntax, is written in C++.

While there are several examples of HDLs that use Rust as the inspiration language, RustHDL falls into the second category, where it appears to be unique. RustHDL is unique in that *hardware designs are expressed as valid Rust programs*. This means that before a design can be synthesized, it must first pass the checks of the Rust compiler, and be compiled into some form of a working, valid software program. It also means that much of the infrastructure of the Rust programming language can be reused by hardware designers. Things like package management, documentation, test management and IDE integration all come for free.

III. RustHDL Core Principles

As this is not meant to be a tutorial, a very brief summary of the concepts used in RustHDL is presented in order to orient the reader. The core concepts of RustHDL are:

- RustHDL is a subset of Rust. As such all hardware designs must be valid Rust programs. This includes all aspects of Rust validity checking including borrow checking, type checking, initialization before use, etc.
- Circuit elements are described architecturally as structs, and composition is used to combine circuit elements into larger designs. Internal details of circuit elements can be exposed using the `pub` keyword, just as in a normal Rust struct.
- An `update` function is used to describe the behavior of the circuit, and is translated into Verilog using the macro extension capabilities of RPL.
- RustHDL includes a reasonably high performance simulator with extensive tracing capabilities. This allows for the simulation of complex designs with a high degree of confidence, as well as the use of Rust to write testbenches.
- Extensibility and collaboration are provided through the use of the Rust package manager `cargo`, and the ability to publish and share designs as crates on `crates.io`.

While no one aspect of the above may be novel, in combination they present a novel and powerful way to describe hardware designs. In the following subsections, each of these elements will be briefly touched upon and explained. More thorough documentation is available on the RustHDL website [7].

A. Circuit Elements

Circuit elements in RustHDL are simply structs composed of other circuit elements. The idea is to provide effortless reuse and composition of circuit components. Every circuit in RustHDL is composed of three parts:

- A struct that describes the architecture of the circuit.
- An `update` function that describes signal propagation in the circuit internals.
- A constructor function that initializes the circuit.

As an example, the following from the RustHDL tutorial demonstrates a simple strobed blinker. First, the circuit element is defined as a struct, annotated with `LogicBlock` to invoke the macro that generates the necessary boilerplate required by the framework.

```
use rust_hdl_core::prelude::*;
use crate::{dff::DFF, dff_setup};

/// A [Strobe] generates a periodic pulse train,
/// with a single clock-cycle wide pulse
/// at the prescribed frequency.  The argument
/// [N] of the generic [Strobe<N>] is used
/// to size the counter that stores the internal
//  delay value.
#[derive(Clone, Debug, LogicBlock)]
pub struct Strobe<const N: usize> {
    /// Set this to true to enable the pulse train.
    pub enable: Signal<In, Bit>,
    /// This is the strobe signal
    /// it will fire for 1 clock cycle such that
    /// the strobe frequency is generated.
    pub strobe: Signal<Out, Bit>,
    /// The clock that drives the [Strobe].
    /// All signals are synchronous to this clock.
    pub clock: Signal<In, Clock>,
    threshold: Constant<Bits<N>>,
    counter: DFF<Bits<N>>,
}
```

Signals have both a direction and a type. The type of the signal can be any Rust type that implements the `Synth` trait,

and can include custom user types and structs. The `derive` macro is used to generate the necessary boilerplate to make the `Strobe` struct implement the `Block` and `Logic` traits. The details of these implementations are unimportant, and the user can simply use the `Strobe` struct as if it were a normal Rust struct. The D-type flip flop (DFF), which is critical to synchronous designs, is simply another circuit element that is included in the internal structure of the `Strobe` struct. The RustHDL DFF is parameterized (or generic) over the type of data it holds, so in this case, it is holding an N-bit wide value. The value of `N` is provided when the strobe is created.

Note also that the `pub` keyword is used to expose parts of the circuit that are considered to be part of its public-facing interface. The counter and threshold are internal implementation details and can be changed without altering how this circuit is used in higher level designs.

The second part of the circuit encapsulates its behavior. This is done by implementing the `Logic` trait, and requires only a single function, called `update`. The `update` function describes the behavior of the circuit via the propagation of signals through the internal structure of the circuit. It is simply a Rust function that must obey a set of rules to ensure that the circuit can be synthesized. Feedback on the compliance of the code to the rule set is provided at compile time, and is immediate. Here is the `update` function for the `Strobe` circuit:

```
impl<const N: usize> Logic for Strobe<N> {
  #[hdl_gen]
  fn update(&mut self) {
    // Connect the counter clock to my clock
    // Also ensures that no latches are inferred
    // due to unassigned signals.
    dff_setup!(self, clock, counter);
    if self.enable.val() {
      self.counter.d.next = self.counter.q.val() + 1;
    }
    self.strobe.next = self.enable.val() &
      (self.counter.q.val() == self.threshold.val());
    if self.strobe.val() {
      self.counter.d.next = 1.into();
    }
  }
}
```

The `#[hdl_gen]` attribute is attached to the `update` function to provide a way to convert the function into Verilog. The key thing to note, however, is that *without this attribute, update is still a valid Rust function*. This means that the `update` function can be tested, debugged, and run as a normal Rust function. The `#[hdl_gen]` adds additional constraints to the code to ensure that it is synthesizable, but the Rust compiler still does the work of ensuring that the program input is valid. This is in contrast to other Domain Specific Languages implemented in Rust where the language only uses Rust's token structure, but the actual grammar and semantics are different. In those cases, removing the macro attribute results in a non-compilable program.

The `update` function follows a few rules that are laid out in the documentation. In short:

- Signals have two endpoints. A `.val()` endpoint that represents their current value, and a `.next` endpoint that represents the value that the circuit is driving them to.
- Several (but not all) Rust flow control primitives are allowed, including `match`, and `if` *statements* and a very limited `for` loop.
- Local variables are allowed, but they must be declared in the architecture of the circuit, like any other element. These signals have a type of `Local`, and are declared as `my_sig: Signal<Local, T>`. The types of any local variables must be expressed as part of the struct declaration. There is no type inference.

The last part of the circuit is the construction and initialization. Here again, there is nothing special in constructing a struct that represents a RustHDL circuit. The constructor function is just a normal Rust function that populates the contents of the struct. Because it can do anything that a normal Rust function can do, arbitrary checks and computations (which are done prior to hardware synthesis) can be accomplished in the constructor. For example, the constructor for `Strobe` includes some checks that the desired strobe frequency will not overflow the size of the counter chosen for the design at the specified clock frequency. This is implemented as a set of assertions in the constructor and will fail at run time. Making these checks compile-time is not currently possible on stable Rust, but may be possible in the future [10].

B. Simulation and Testing

One powerful aspect of RustHDL is the ease with which designs can be tested and simulated *without* resorting to external tools. Figure 1 illustrates some of the potential flow paths for a RustHDL design. RustHDL includes a high performance event-based simulator that can be used to simulate the design, and generate traces of the circuit signals. It can also be used to test the circuits by placing them into software test fixtures. These tests can lend a high degree of confidence in the correctness of the design before it is synthesized.

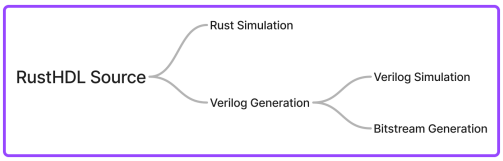

Fig. 1. Flow of a RustHDL design

As a result, RustHDL allows the engineer to write those testbenches using the full power of Rust. The following example tests an SDRAM-backed FIFO implemented in RustHDL, that includes a simulated SDRAM, and reads and writes random data to the FIFO concurrently with checks for the integrity of the results.

```
#[test]
fn test_hls_sdram_fifo_works() {
  let mut uut = HLSSDRAMFIFOTest::default();
  let mut sim = Simulation::new();
  // v-- generate random data using Rust
  let data = (0..256)
      .map(|_| rand::thread_rng().gen::<u16>())
      .collect::<Vec<_>>();
  let data2 = data.clone();
  // v-- generate the clock
  sim.add_clock(4000, |x: &mut Box<_>| {
      x.clock.next = !x.clock.val()
  });
  // v-- this testbench feeds data to the fifo
  sim.add_testbench(move |mut sim: Sim<_>| {
      let mut x = sim.init()?;
      wait_clock_cycles!(sim, clock, x, 20);
      hls_fifo_write_lazy!(sim, clock, x,
        fifo.bus_write, &data);
      sim.done(x)
  });
  // v-- this testbench drains data from the fifo
  //    and panics if it doesn't match the input
  sim.add_testbench(move |mut sim: Sim<_>| {
      let mut x = sim.init()?;
      wait_clock_cycles!(sim, clock, x, 20);
      hls_fifo_read_lazy!(sim, clock, x,
        fifo.bus_read, &data2);
      sim.done(x)
  });
  // v-- both testbenches are run concurrently
  //    by the RustHDL event-based simulator
  sim.run_to_file(Box::new(uut), 200_000_000,
      &vcd_path!("hls_sdram_fifo.vcd")).unwrap();
}
```

The resulting trace file as shown in Figure 2 is quite complicated. But the key is that the testbench includes assertions to ensure correct behavior of the circuit, and will fail/panic if at any time the FIFO behaves incorrectly. No manual visual inspection of the trace file is required. Indeed, it is far faster to run the simulation with no trace output at all, and simply have the testbenches encode the validity checks as assertions. The simulation can then be run to verify the integrity of the design with a simple `cargo test` command as part of a regular integration cycle. Visual inspection of the trace output can be used when a test fails or when debugging the circuit design.

In the RustHDL test suite, a subset of the tests include a full synthesis, and are run on actual FPGA boards attached to the host computer. But these can be disabled if the necessary hardware and supporting software is missing.

C. Synthesis, Extensibility, and Coherence

Obviously, to get a RustHDL design onto an FPGA, it must be synthesized. It is here that another advantage of the Rust ecosystem becomes apparent. RustHDL includes the concept of a *board support package* (a term borrowed from the embedded world), which is a software crate that provides the necessary infrastructure to convert a RustHDL hardware design into a bitstream targeted at a specific device. The BSP provides a function such as:

```
pub fn generate_bitstream<U: Block>
    (mut uut: U, prefix: &str) {}
```

which takes a generic struct that implements a circuit, and converts it into a bitstream, using whatever tooling is appropriate for the target device.

A key advantage of RustHDL is that the software module that provides the BSP for a specific physical devices is completely decoupled from the core library, and can be written, published, and maintained by third parties (see [8] for an example of this in the wild). The core library can also be extended with new circuit elements (dubbed "widgets") that can be published on `crates.io` and used by other engineers. As such one of the significant challenges in hardware design - that of sharing reusable components - is addressed by the Rust ecosystem.

The first class support that Rust provides for software package management also allows the core library to be extended by others in the ecosystem. Simply adding a package to the `Cargo.toml` can enable new features such as easier wrapping of Verilog cores into RustHDL [9]. Support for different use cases, such as ASICs, and different FPGA families can be handled through external crates, avoiding the problems of a single point of coordination, testing and publishing.

When synthesizing designs, it is sometimes the case that the synthesis tools provide some feedback on the design. Examples include timing closure challenges, and other such issues. The engineer must be able to map these suggestions back into the original source HDL, and maintain coherence between the two. Coherence is further complicated when the source HDL is richer than the one fed to the toolchain.

As noted in [1], there is a significant challenge associated with using a standard programming language to express hardware designs - in particular, the range of acceptable expressions must be limited to the common set from the destination language/HDL and the source language. Without a full compiler, the RustHDL grammar is limited to handle only those expressions that can be directly translated into Verilog. Concepts like early returns, and match expressions, which do not exist in Verilog, cannot be used in RustHDL.

While the next version of RustHDL (RHDL) attempts to overcome these limitations, the benefit to such a close association between the two languages is that the generated Verilog is quite readable, and maintaining coherence between the generated code and the input source is fairly straightforward. To demonstrate, consider the following snippet of RustHDL:

```
#[derive(LogicBlock)]
pub struct AlchitryCuPulser {
    pulser: Pulser,
    clock: Signal<In, Clock>,
    leds: Signal<Out, Bits<8>>,
}

impl Logic for AlchitryCuPulser {
    #[hdl_gen]
    fn update(&mut self) {
        self.pulser.enable.next = true;
        clock!(self, clock, pulser);
        self.leds.next = 0x00.into();
        if self.pulser.pulse.val() {
            self.leds.next = 0xAA.into();
        }
```

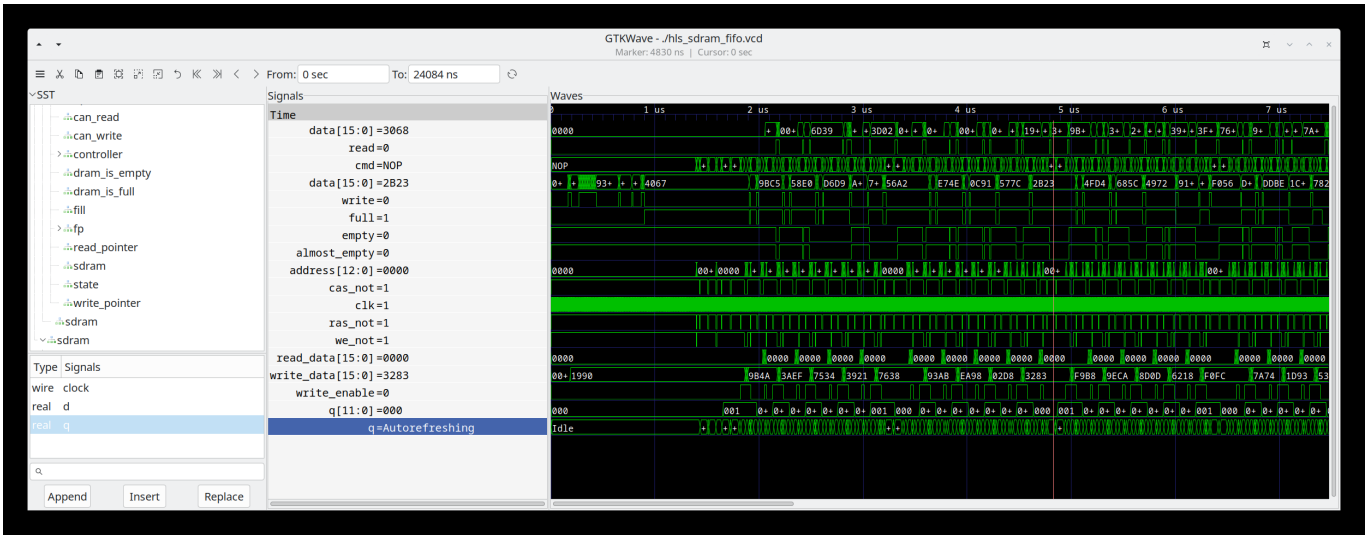

Fig. 2. Sample simulation trace for the HLSSDRAMFIFOTest testbench

```
    }
}
```

This simple blinker is translated into the following top level Verilog module:

```
module top(clock,leds);

  // Module arguments
  input wire  clock;
  output reg  [7:0] leds;

  // Stub signals
  reg  pulser$clock;
  reg  pulser$enable;
  wire  pulser$pulse;

  // Sub module instances
  top$pulser pulser(
      .clock(pulser$clock),
      .enable(pulser$enable),
      .pulse(pulser$pulse)
  );

  // Update code
  always @(*) begin
      pulser$enable = 1'b1;
      pulser$clock = clock;
      leds = 32'h0;
      if (pulser$pulse) begin
          leds = 32'haa;
      end
  end
endmodule // top
```

The relationship between the two is fairly evident, and one could theoretically maintain the Verilog code without reference to the original RustHDL sources.

This close equivalence is important when trying to understand and interpret the results of the toolchain while processing the design. A report from the place and route process that refers back to a struct in RustHLD is much easier to act upon than one that refers only to generic identifiers and low level primitive instances. This coherence issue will remain critical for all new HDL environments. Because they ultimately rely upon existing toolchains, it will be necessary to interpret the results of those toolchains in terms of the original HDL.

IV. Lessons Learned and the Path to RHDL

RustHDL has been used to develop commercial grade firmware and is deployed in commercial products. It has also seen some level of adoption by the open source community, with third party crates published on `crates.io` that add board support packages to RustHDL for different FPGAs, or augment the capabilities of the framework. However, based on feedback from users getting started with the framework, the following items became clear:

- Engineers coming to a Rust-based HDL environment are likely to be familiar with the Rust programming language (RPL), and have grown accustomed to patterns and language features that are not supported by RustHDL. For example, the use of `match` *expressions*, and the use of complex types with their need to employ pattern matching to destructure them are all idiomatic in Rust, but not supported by RustHDL. This lack of support stems from the close connection to Verilog of the generated code.
- Many idiomatic Rust patterns require some kind of support for Algebraic (Sum) Data Types (ADTs). While RustHDL supports simple C-style enums, that is not sufficient to express many of the patterns common in Rust, including the `Result` and `Option` patterns.
- The need to declare all local variables and their types up front is cumbersome. Being able to freely rebind names and assign them is also critical to several standard Rust patterns (like shadowing). This is also not possible in RustHDL.
- Function composition is not possible. This makes the development of modular *behavior* difficult.
- Testing is hard to grasp. Testbenches are written in Rust, but require an understanding of the underlying event-

based simulator that is used. This makes testbenches difficult to write, and makes it difficult to use those testbenches in other environments (e.g., Verilog).
- Backends for multiple HDLs are needed including FIRTL, VHDL and others. Some of those backends support rich type information to be carried on signals, and so would benefit from propagating that typed information through to the HDL instead of simply generating bit vectors.

After taking all of these considerations into account, a new framework, named RHDL was planned that addressed these issues. The core differences between RustHDL and RHDL are:

- Full support for Rust data types, including ADTs, tuples, etc.
- Full support of Rust syntax inside functions (with the exception of pointers), which includes match and if expressions, early returns, etc.
- Full support for function composition.
- Support for local variables/rebindings, as well as pattern matching in `match` expressions.
- The use of a function iterator-style description of testbenches in which testbenches are written in straight Rust, and consumed by the simulator (so that the details of how the circuit is simulated can be ignored).
- Support for thorough analysis of the design, including identification of potential timing issues, as well as potential metastability detection, and clock domain crossings.
- Support for multiple backend languages is possible, even for those that want type information.

These items required a significantly broader scope and far more effort on the compiler side of the framework. In particular, the first four items required the development of a full compiler that works alongside `rustc` to process the source. This embedded compiler performs the following passes, each of which is necessary to achieve the full support of Rust:

- Type inference to add support for local rebindings and pattern matching.
- Lowering of Rust expressions to an intermediate representation (which is a kind of RTL-SSA assembly code). This step is where early returns, if expressions and pattern matching are lowered to ROM tables and muxes.
- Optimization passes to remove unneeded registers, and unreachable code.
- Validation passes that ensure there are no latches generated, and that all types are preserved through the lowering.
- Analysis passes that look for potential timing issues, and metastability.
- Linking of translation units into a single design. This is where support for function composition and black boxes is provided.
- Generation of Verilog (or whatever target HDL is desired - Verilog only at this time).

Apart from the analysis passes, the remainder of the compiler is implemented and working. The analysis passes are still in development. As mentioned in Section III, one significant challenge is maintaining coherence between the analysis results reported by the synthesis tools and the RHDL source code. This is a significant challenge and still an area of active research and experimentation.

V. Conclusions

Rust is an excellent language to serve as the basis for hardware design. RustHDL demonstrates how the compiler, tooling, infrastructure and community of the Rust programming language can be leveraged to build a robust and collaborative environment for hardware design. The initial release and use of RustHDL, along with the feedback provided by the open source community lead to a roadmap for the successor of RustHDL (RHDL), which employs additional software technology to bring the full expressiveness of the underlying language to bear. These improvements promise to make the development of hardware designs safer, more accessible, easier to test and verify, and simpler to share and reuse.

References

[1] F. Skarman and O. Gustafsson, "Spade: An Expression-Based HDL With Pipelines", Open Source Design Automation Conference, 2023.

[2] "Chisel - Software-defined hardware", https://www.chisel-lang.org/ (Accessed Feb 1, 2024).

[3] "MyHDL - From Python to Silicon!", https://www.myhdl.org/ (Accessed Feb 1, 2024).

[4] "XLS: Accelerated HW Synthesis", https://google.github.io/xls/ (Accessed Feb 1, 2024).

[5] "Veryl - A modern hardware description language", https://crates.io/crates/veryl (Accessed Feb 1, 2024).

[6] "Rust - A language empowering everyone to build reliable and efficient software", https://rust-lang.org (Accessed Feb 1, 2024).

[7] "RustHDL - Write FPGA Firmware using Rust!", https://rust-hdl.org/ (Accessed Feb 1, 2024).

[8] "rust-hdl-bsp-step-mxo2-lpc - rust-hdl board support package for STEP-MXO2-LPC", https://crates.io/crates/rust-hdl-bsp-step-mxo2-lpc (Accessed Feb 4, 2024).

[9] "wrap_verilog_in_rust_hdl_macro - A proc-macro to wrap Verilog in a rust-hdl module", https://crates.io/crates/wrap_verilog_in_rust_hdl_macro (Accessed Feb 4, 2024).

[10] "Tracking Issue for complex generic constants: features(generic_const_exprs)", https://github.com/rust-lang/issues/76560 (Accessed Feb 5, 2024).