Rust as a Hardware Description Language

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

ABSTRACT

Rust [1] makes an excellent language for hardware description. A number of new HDLs are Rust-inspired in syntax [2],[3], but RustHDL[4] is actually a framework for turning ordinary Rust code into firmware. The first attempt, while successful had several shortcomings that are discussed in this paper. The new framework, RHDL[5], should address those shortcomings and significantly increase the power and usability of Rust as a hardware description language.

1 INTRODUCTION

While the field of HDLs may be crowded, I propose the use of the Rust Programming Language (RPL) as a hardware description language. Beyond its status as most loved of the programming languages [6], Rust has steadily been gaining traction as a serious language for systems programming, embedded software, and other mission critical applications. The particular features of Rust that are relevant to hardware description include:

- Static typing and sane syntax
- Functional programming features
- A powerful macro system
- Package management and a growing open-source ecosystem
- Built-in test capabilities
- Significant tooling and infrastructure support
- Generics and const generics

The RHDL framework, and it's predecessor RustHDL took advantage of all of these features to create an environment for hardware description that is powerful, easy to use, extensible and open. The end goal is to enable a wider class of engineers to develop high quality hardware by reusing their skills as Rust developers in the hardware domain. This paper briefly describes the RustHDL approach to developing FPGA firmware using the RPL, and then identifies the observed shortcomings and how they may be addressed in the upcoming RHDL framework. Finally, I touch upon some of the unsolved problems in using Rust for hardware description.

2 SYNTAX

RustHDL is not a new language. Instead it is a set of libraries and macros along with a subset of the Rust programming language that can be used to generate firmware. The key principle of RustHDL is:

> RustHDL designs are valid Rust programs that can be compiled and run on a host computer using the included event-based simulator.

In this sense, RustHDL is embedded in the Rust language much as MyHDL is embedded in Python [7], and Chisel is embedded in Scala [8]. The immediate implication is that:

- All RustHDL designs must pass the strenuous checks of the Rust compiler `rustc`.
- A number of critical checks come for free, including use before assignment, unused variables, unassigned values, etc.
- Tools such as `clippy` and `rust-analyzer` can immediately be used to check, lint and analyze code with no additional effort.
- The Rust test framework can be used to test the designs directly.

The syntax should be fairly familiar to anyone comfortable with Rust. The following is an example of a simple SPI master in RustHDL, generic over the transaction size `N`, edited for brevity:

```
#[derive(LogicBlock)]
pub struct SPIMaster<const N: usize> {
    pub clock: Signal<In, Clock>,
    pub data_outbound: Signal<In, Bits<N>>,
    pub data_inbound: Signal<Out, Bits<N>>,
    pub start_send: Signal<In, Bit>,
    pub wires: SPIWiresMaster,
    state: DFF<SPIState>,
    cs_off: Constant<Bit>,
}
```

The `pub` keyword is used to denote the visibility of the signals. Signals marked with a direction, and type. Internal components such as flip flops and strobes are all included in the top level struct, which is initialized using normal Rust code. The `Local` signal represents a local variable used in the update function, but not otherwise exposed. As RustHDL had no type inference, it required explicit allocation and types for all local variables. The `cs_off, mosi_off, cpha, cpol` are all constants constructed at compile time that encode the SPI mode of the bus. Finally, the `SPIWiresMaster` is a struct that describes the interface to the actual SPI bus. Interfaces (unlike structs in SystemVerilog, for example) included both input and output signals, and could be used to "connect" complex components with a single line.

As an example, an interface to an SDRAM chip with a `D-bit` data bus and a 13 bit address bus is defined as:

```
#[derive(LogicInterface, Clone, Debug, Default)]
#[join = "SDRAMDriver"]
pub struct SDRAMDevice<const D: usize> {
    pub clk: Signal<In, Clock>,
    pub we_not: Signal<In, Bit>,
    pub cas_not: Signal<In, Bit>,
    pub ras_not: Signal<In, Bit>,
    pub address: Signal<In, Bits<13>>,
    pub write_data: Signal<In, Bits<D>>,
    pub read_data: Signal<Out, Bits<D>>,
```


LATTE '24, March 26, 2023, Vancouver, BC, Canada

```
    pub write_enable: Signal<In, Bit>,
}
```

and can be connected to the corresponding signals in another IP block with a single `join` statement. This significantly reduces the amount of error-prone wiring that must be done by code or graphically.

An update function calculates the next value of the signals (external and internal) based ont the current state stored in the DFF `state`, which itself is a C-style enum. Rust ensures that the state machine match/case is exhaustive. Note that re-assigning to a the `.next` endpoint of a signal feels entirely natural and translates into the priority based assignments in Verilog.

3 MENTAL MODEL

RustHDL attempts to build on HDLs like Lucid to provide a more understandable mental model for how hardware works. In an imperfect implementation, RustHDL defines a `Signal` struct that has a read only endpoint `x.val()` for signal `x`, and a write endpoint `x.next`. The write endpoint is meant to be write-only, but those semantics do not exist in Rust. This way the idea of non-blocking assignments is removed, and the engineer need only think about a conditional model - i.e., given the current value in the set of signals, what next value do I want them to take? This mental model is coupled with analysis passes that look (with the aid of yosys in RustHDL) for latch inferences due to missing assignments and other such issues.

4 SIMULATION

Testing of designs in RustHDL does not require the use of third party tools or tooling. Tests utilize a built-in event-based simulation engine that can simulate any RustHDL design. Black box IP cores can be simulated by providing Rust equivalents of the hardware descriptions. The simplest example is something like a block RAM, which can be trivially instantiated in Verilog, but requires a behavioral model in RustHDL. In RustHDL that behavioral model is written in Rust, and can be substituted into the simulation environment. Other black box IP cores can be equivalently simulated in Rust.

Speed is a critical factor in simulation. While RustHDL is not as fast as Verilator, for example, and cannot currently multi-thread a single design, the Rust test framework is inherently parallel, and can run multiple tests in parallel. Using system calls/shell-outs, the entire synthesis and bitstream generation process can be handled inside the Rust ecosystem.

5 REUSE

Hardware designs in RustHDL are simply `structs`, and are composed of other hardware components via composition. This allows for easy reuse of components, the construction of complex designs out of simpler, smaller components, and sane rules of scoping and encapsulation. Furthermore, each of the subcomponents can be tested in isolation, and then tested after composition in the larger design.

Rust is a very composable language, and `crates.io` provides a natural mechanism for sharing and reusing components. As an example, in RustHDL, handling of hardware specific details (such as synthesis tools and constraints files for specific FPGAs and boards) is handled through a *board support package*. This is simple a library that provides the defaults and other mapping information needed to generate a bitstream for a given piece of hardware. As an open-ended and potentially unbounded problem, the BSP can be published as a `crate` (package) in the Rust ecosystem by contributors [9]. This decentralizes control over one of the more challenging parts of maintaining support for a bewildering array of devices.

6 SHORTCOMINGS AND THE FUTURE

RustHDL has been used for commercial firmware development and is deployed. It has also seen some level of interest and adoption from the open source community. Feedback from early users lead to the following list of shortcomings:

- The subset of Rust supported by RustHDL (which is the subset of the language that can be directly translated into Verilog) is too small to write "natural" Rust code.
- RustHDL does not support Algebraic Data Types (data-carrying enums).
- Local variables and type inference (which go together) are critical to writing clean and idiomatic Rust code.
- Composition of functions/behavior is not possible.
- Writing test-benches required an understanding of the simulator mechanics.
- Backends were requested for a variety of HDLs.

Solving all of these problems essentially necessitated a rewrite of RustHDL. The new framework, called RHDL (Rust Hardware Description Language) is currently under development. The primary technical difference to RustHDL is the introduction of a full secondary compiler into the processing. This compiler works in tandem with `rustc` to convert an AST of the code into a RTL-like HDL, and then transform and optimize that representation into a form that can be synthesized. The compiler is key to support of things like early return, match and if expressions (as opposed to statements), and other Rust-isms that are not common in HDLs, but are common in Rust. The compiler also provides ADT support with control over the layout of the data, and easy composition of data types into structs of arbitrary complexity.

REFERENCES

[1] "Rust - A language empowering everyone to build reliable and efficient software", https://rust-lang.org (Accessed Feb 1, 2024).
[2] F. Skarman and O. Gustafsson, "Spade: An Expression-Based HDL With Pipelines", Open Source Design Automation Conference, 2023.
[3] "XLS: Accelerated HW Synthesis", https://google.github.io/xls/ (Accessed Feb 1, 2024).
[4] "RustHDL - Write FPGA Firmware using Rust!", https://rust-hdl.org/ (Accessed Feb 1, 2024).
[5] "RHDL - Rust Hardware Description Language", https://github.com/samitbasu/rhdl (Accessed Feb 1, 2024).
[6] "Stack Overflow Developer Survey 2023", https://insights.stackoverflow.com/survey/2023 (Accessed Feb 1, 2024).
[7] "MyHDL - From Python to Silicon!", https://www.myhdl.org/ (Accessed Feb 1, 2024).
[8] "Chisel - Software-defined hardware", https://www.chisel-lang.org/ (Accessed Feb 1, 2024).
[9] "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).