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[5] 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[6], should address those shortcomings and significantly ease the use of Rust for hardware design.

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 [7], 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 prevent many common mistakes and bugs
- Package management and a growing open-source ecosystem allow for collaboration and composability
- Built-in test capabilities for rigorous and automated testing.
- Generics and const generics make parametric design safe and accessible.

The RHDL framework and its 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, as well as to provide a natural path for existing C and Verilog developers to migrate to a safer language. 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. Note that I use the term "hardware description" here to mean FPGA firmware (or possibly ASIC designs), as Rust is already established as a language for embedded systems programming (which is also often referred to as "Firmware").

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

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 [8], and Chisel is embedded in Scala [9]. The immediate implication is that:

- All RustHDL designs must pass the strenuous correctness checks of the Rust compiler `rustc`.
- An entire class of potential bugs are eliminated, such as type mismatches, use-before-initialization, unassigned outputs, etc.
- Tools such as `clippy` and `rust-analyzer` can immediately be used to check, lint and analyze code with no additional investment.
- The Rust test framework can be used to test the designs directly.

Note that unlike MyHDL, RustHDL does not use a generator pattern and infer the required hardware. Instead, the AST itself is transformed into the circuit description. So, for example, a MUX is inferred from an 'if' statement, not at run time, and not via construction. This is also in contrast to a combinator style of hardware description, for example in [4], where the language used is Rust, but hardware description is functional.

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 wires: SPIWiresMaster,
    local_signal: Signal<Local, Bit>,
    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 requires explicit allocation and types for all local variables. The member `cs_off` (along with others omitted) is a constant constructed at runtime that encodes 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) include both

input and output signals, and can be used to "connect" complex components with a single line.

Note that in this instance, `state: DFF<SPIState>` is equivalent to a module instantiation. The `DFF` is a flip-flop, and `SPIState` is a C-style enum that represents the state of of the controller. By including it as a member of the struct, we request an instance of it be created in the generated design.

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 address: Signal<In, Bits<13>>,
    pub write_data: Signal<In, Bits<D>>,
    pub read_data: Signal<Out, Bits<D>>,
}
```

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. The `join` statement is used inside of an `update` function as the following demonstration:

```
#[derive(LogicBlock)]
struct I2CControllerTest {
    clock: Signal<In, Clock>,
    controller: I2CController,
    target_1: I2CTestTarget,
    target_2: I2CTestTarget,
    test_bus: I2CTestBus<3>,
}

impl Logic for I2CControllerTest {
  #[hdl_gen]
  fn update(&mut self) {
      I2CBusDriver::join(&mut self.controller.i2c,
        &mut self.test_bus.endpoints[0]);
      I2CBusDriver::join(&mut self.target_1.i2c,
        &mut self.test_bus.endpoints[1]);
      I2CBusDriver::join(&mut self.target_2.i2c,
        &mut self.test_bus.endpoints[2]);
  }
}
```

In this example, the controller, and 2 DUTs are connected to a bus. Since all of the logic is simply connecting the interfaces together, it consists mainly of `join` statements.

3 MENTAL MODEL

RustHDL attempts to build on HDLs like Lucid[10] 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`.

```
// Design is parametric over N - the size of the counter
impl<const N: usize> Logic for Counter<N> {
  #[hdl_gen]
  fn update(&mut self) {
    self.counter.d.next = self.counter.q.val();       // Latch prevention
    if self.enable.val() {                            // MUX
      self.counter.d.next = self.counter.q.val() + 1; // Adder
    }
  }
}
```

Rust lacks write-only semantics, so the framework checks for read-before-write on the `x.next` endpoint. This mental model is coupled with analysis passes that look (with the aid of Yosys[11] in RustHDL) for latch inferences due to missing assignments and other such issues.

The mental model of RustHDL is not ideal (and is replaced in RHDL). A signal's `.next` endpoint can be written to as many times as desired inside of an `update` function. Only the last value it takes will matter when the function completes. In essence, the last successful write to a signal "wins", where success may be conditional (in this case, for example, the value of 'self.counter.d.next' depends on the value of 'self.enable.val()'). Similarly, local variables can be both written to and read, as long as a write preceeds and subsequent reads or writes.

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. Note that because RustHDL supports combinatorial connections across modules that the simulator iterates until it reaches a fixed point. The iterations will terminate with an error if some upper limit is reached.

Speed is a critical factor in simulation. RustHDL is a reasonably fast simulator, and 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. A direct comparison with Verilator proved difficult as Verilator rejected the Verilog generated by RustHDL. A fair comparison is a target for RHDL, which supports multi-threaded simulation, and should be even faster than RustHDL.

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, combined with 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, pin-outs, 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 [12]. 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 non-trivial 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 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 are needed for more than just Verilog.

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 an auxiliary 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 returns, match and if expressions (as opposed to statements), and other Rust-isms that are not common in HDLs, but are common in Rust.

7 CONCLUSIONS

I believe Rust is a promising basis for a hardware description language. It offers many powerful tools that can be utilized to build composable, reusable and correct hardware designs. The RustHDL framework was a first step in this direction, and the in-development RHDL framework promises to address many of the shortcomings of the first attempt. I look forward to presenting more details on RHDL as it evolves.

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] Sungsoo Han, Minseong Jang, and Jeehoon Kang, "ShakeFlow: Functional Hardware Description with Latency-Insensitive Interface Combinators", ASPLOS 2023.
[5] "RustHDL - Write FPGA Firmware using Rust!", https://rust-hdl.org/ (Accessed Feb 1, 2024).
[6] "RHDL - Rust Hardware Description Language", https://github.com/samitbasu/rhdl (Accessed Feb 1, 2024).
[7] "Stack Overflow Developer Survey 2023", https://insights.stackoverflow.com/survey/2023 (Accessed Feb 1, 2024).
[8] "MyHDL - From Python to Silicon!", https://www.myhdl.org/ (Accessed Feb 1, 2024).
[9] "Chisel - Software-defined hardware", https://www.chisel-lang.org/ (Accessed Feb 1, 2024).
[10] J. Rajewski, "Lucid - FPGA Tutorials", https://alchitry.com/lucid/ (Accessed Feb 18, 2024).
[11] C. Wolf, "Yosys Open SYnthesis Suite", https://yosyshq.net/yosys/ (Accessed Feb 18, 2024).
[12] "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).