RHDL: Rust as a Hardware Description Language

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

ABSTRACT

In [1], I proposed Rust as an excellent language for hardware description[1]. The initial attempt described in that paper was published as RustHDL [2]. However, feedback from developers learning RustHDL revealed several weaknesses in the design. As a result, I have developed RHDL [3], which is a complete rewrite of RustHDL that attempts to address these shortcomings and significantly expand the capabilities of the tool.

1 INTRODUCTION

In my prior paper, I described the various reasons that Rust makes an excellent choice for a hardware description language. This includes strong typing, functional programming features, package management and tooling, generics and a strong open ecosystem [4], [5]. I also introduced RustHDL, which transforms a carefully selected subset of Rust into synthesizable Verilog. Within a set of implicit rules, gateware can be built using Rust code. There are, however, several shortcomings that became apparent as more engineers began to use RustHDL. A subset of the features requested:

- Algebraic Data Types (enums with data).
- Type inference and local variable support.
- Early returns, match and if expressions.
- Timing estimation and analysis.

Adding these features has required the development of a co-compiler that runs alongside `rustc` to analyze the Rust source code and generate a series of HDL-compatible representations that are successively lowered to the hardware. Both compilers work together to ensure that the language invariants are met at all stages, and that undefined behavior is prevented. The new framework, called RHDL is meant to be a zero-cost abstraction, meaning that the framework itself does not limit the performance of the design.

2 NEW FEATURES

RHDL (like its predecessor RustHDL) is embedded in the Rust programming language, much as MyHDL is embedded in Python[6] and Chisel is embedded in Scala [7]. As a result, all RHDL code must be valid Rust and must meet all the requirements of the Rust compiler. This design means an entire class of errors are eliminated, as the language enforces correct usage of types, prevents use-before-initialization, unassigned outputs, etc. Furthermore, by using Rust itself, as opposed to something Rust-like or a DSL[2], all the tools that work with Rust can be used with RHDL unmodified. Of course, the challenge with this approach is that only a very small set of Rust code can be directly mapped to Verilog. Mapping a large subset of Rust to hardware requires significant analysis of the underlying code to enable the relevant transformations. A brief summary of some of the more significant changes follows.

2.1 Algebraic Data Types

The main request from RustHDL users was support for Algebraic Data Types. These are essentially tagged unions, where the tag is created and tracked by the compiler, *and it is guaranteed via the type system that a value is a valid combination of tag and data.* This is an incredibly powerful feature in Rust that RustHDL was unable to support due to lack of any direct equivalent in Verilog. RHDL provides first class support for ADTs. For example, the following can be used in synthesizable designs, where the language invariants guarantee that the payload is valid if and only if the appropriate tag is set in the discriminant of the enum:

```
enum MyEnum {
  A, // No payload
  B(b4, b6), // 4-bit and 6-bit tuple payload
  C{ x: b4, y: b6, z: [b3; 3] }, // Struct payload
}
```

All of these *variants* are stored in a union that will be sized large enough to store the largest variant together with the discriminant. In this case, the type is 21 bits wide. See Figure 1 for an auto-generated layout diagram of the type.

Using ADTs in Rust requires pattern matching and destructuring. These are lowered into `case` expressions automatically:

```
let w = MyEnum::B(1, 2);
let v = match w {
  MyEnum::A => bits(1),
  MyEnum::B(a,..) => a,
  MyEnum::C{ x,..} => x,
}
```

⟶

```
// Assume r0 = w, r6 = v
r1 <- r0# // r1 <- discriminant(w)
r2 <- r0#1 // r2 <- payload(w, 1)
r3 <- r2.0 // r3 <- r2.a
r4 <- r0#2  // r4 <- payload(w, 2)
r5 <- r4.x // r5 <- r4.x
r6 <- case r1 {
   0 => 1, 1 => r3, 2 => r5
}
```

Here, the right hand side shows the intermediate form generated by the compiler. It is a register-based, static single assignment (SSA) series of opcodes that are later lowered into RTL and then into Verilog. At this stage, each of the register values is strongly typed.

2.2 Type Inference and Local Variables

Rust relies heavily on type inference to make the language less verbose. RHDL uses a type inference pass to deduce and annotate the types of all variables in the original code. This type pass *must* agree with the types inferred by `rustc` to avoid miscompilation. Passes are included to ensure that all variables are typed by RHDL and that the typing is consisting with `rustc`:

```
function(a: b4, b: b4) -> b4 {
  let x = 1;
  let y = (x, x + 1);
  y.0 + y.1
}
```

⟶

```
let a: b4;
let b: b4;
let x: b4;
let y: (b4, b4);
let _return: b4;
```

[1] FPGA Gateware or ASIC design
[2] Both Spade [8] and XLS [9] use a Rust-inspired syntax for hardware description.

LATTE '25, March 30, 2025, Rotterdam, Netherlands

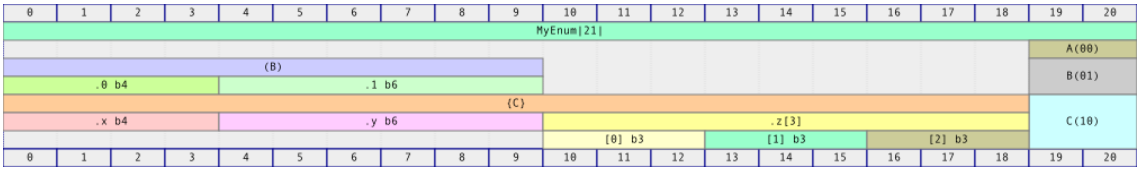

Figure 1: Autogenerated layout of MyEnum.

Type inference and type checking are also used to enforce correctness of the design. For example, in RHDL, a `Signal` type indicates the clock domain it belongs to by use of a marker type parameter (arbitrarily chosen to be a color). So, for example, `Signal<b4, Red>` indicates a nibble that changes on edges in the `Red` clock domain, and `Signal<b4, Blue>` signifies a nibble that changes in the `Blue` clock domain. The color marker type distinguishes signals from different clock domains as being different types, so that `rustc` will prevent inadvertent substitution. Here is an example that fails to compile due to an adder that crosses clock domainsm with the error shown in Figure 2:

```
function(a: Signal<b4, Red>, b: Signal<b4, Blue>)
   -> (Signal<b4, Red>, Signal<b4, Blue>) {
  let a = a.val(); // Extract the value of a
  let b = b.val(); // Extract the value of b
  let a = a + b;   // <--- Illegal!
  (signal(a), signal(b))
}
```

Clock domain crossings require special constructs provided in the RHDL core library.

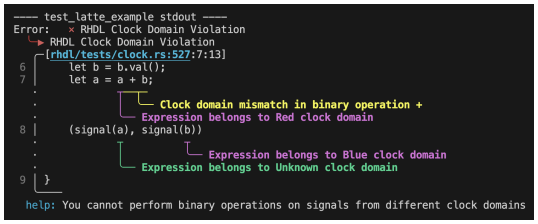

Figure 2: Clock domain error message.

2.3 Expression Transformations

Many Rust constructs are not directly mappable to Verilog. For example, in Rust, all `if` constructs are expressions, and can be used in any context where a value is expected. Blocks also have values (in addition to side effects). These are transformed into a series of mux expressions with renaming of local variables.

```
let mut z = 0;
let x = if y {
  z += 1;
  z
} else {
  z
};
```

→

```
// Assume r0 = z (at the beginning),
// l0 = 1, r1 = y
// r3 = x, r4 = z (at the end)
r2 <- r0 + l0; // z_if_y = z + 1
r3 <- r1 ? r2 : r0; // x = y ? z_if_y : z
r4 <- r1 ? r2 : r0; // z = y ? z_if_y : z
```

In this case, type inference will also be required as the types of all of the variables are implicit. Similar transformations are applied for `match` expressions, early returns and other flow control constructs.

2.4 Timing Estimation

A significant problem that arises in high level HDLs is the difficulty in fixing timing closure issues[3] with the generated design. This results from the very loose coupling between the design as expressed in the HDL and the resulting low level representation that feeds the synthesis tools. RHDL includes a simple critical timing path estimator that can reference back to the source code, see Figure 3. While basic, the intent is to build upon this capability to eventually include a closed loop from 3rd party tools back to the original source code. The user can supply their own timing estimator, or use the built-in heuristic, which counts the number of non-trivial operations on every path (after optimization).

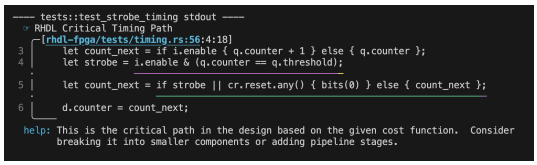

Figure 3: Timing path estimation in RHDL

3 INTERNALS

RustHDL [1] is a transpiler, which generates Verilog syntax that is matched to the allowed subset of Rust that RustHDL supports, stripping type information out of the AST, and providing small shims to convert, e.g., `match` statements into Verilog `case` statements. RHDL, on the other hand, includes a compiler that treats Verilog like a machine-code target, lowering in steps that include:

- RHDL Hardware Intermediate Form (RHIF), which is a strongly typed, static single assignment (SSA), register-based virtual machine instruction set.
- RTL, which is an untyped SSA register-based virtual machine instruction set.
- Flow graph, which is a netlist representation (which may not be directed if the design has loops).

Invoking the compiler is as trivial as adding an annotation to the Rust source code and adding a dependency on the appropriate packages.

4 CONCLUSION

RHDL is a significant step forward from RustHDL. It enables the use of a more complete subset of Rust, and supports many more of the language features that make Rust a powerful language for software development. These features should enable RHDL to support a wider range of hardware designs with code that is easier to write, correct, and efficient.

[3] Or any issues identified in downstream processing, such as floor planning, etc.

REFERENCES

[1] S. Basu, "Rust as a Hardware Description Language", LATTE 2024, San Diego, CA.

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

[3] "RHDL - Rust Hardware Description Language", https://github.com/samitbasu/rhdl (Accessed Jan 25, 2025).

[4] "Stack Overflow Developer Survey 2023", https://insights.stackoverflow.com/survey/2023 (Accessed Feb 1, 2024).

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

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

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

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

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