Veryl: A Modern Hardware Description Language for Open-Source Computer Architecture

Naoya Hatta
PEZY Computing K.K.
hatta@pezy.co.jp

Taichi Ishitani
PEZY Computing K.K.
ishitani@pezy.co.jp

Nathan Bleier
University of Michigan
nbleier@umich.edu

***Abstract*—Hardware Description Languages (HDLs) form the foundation of digital hardware design, yet many popular HDLs predate modern software engineering practices and lack the tooling ecosystem that has revolutionized software development. We present Veryl, a modern HDL designed as a SystemVerilog alternative that brings advanced safety features, improved developer ergonomics, and robust tooling to hardware development while maintaining seamless interoperability with existing SystemVerilog components. Veryl addresses key challenges in open-source hardware design through simplified syntax, clock domain safety analysis, generics for code reuse, real-time diagnostics, and comprehensive tooling. By lowering barriers to entry and enhancing developer productivity, Veryl contributes to a more accessible and collaborative open-source hardware design ecosystem.**

I. Introduction

Hardware Description Languages (HDLs) remain the primary method for digital hardware design, yet leading languages like SystemVerilog contain legacy features from decades past and lack the modern developer tools common in software engineering. These limitations create significant barriers to entry and reduce productivity, particularly impacting open-source hardware communities where collaboration, code reuse, and accessibility are paramount. Common challenges include:

- Error-prone constructs and limited safety guarantees
- Poor tooling support compared to modern languages
- Limited facilities for code reuse and component sharing
- Difficult interoperability between projects and toolchains
- Critical issues like clock domain crossings often detectable only at simulation time

These challenges have constrained the growth of open-source hardware ecosystems compared to their software counterparts. The Veryl Hardware Description Language addresses these limitations while maintaining interoperability with existing SystemVerilog components and toolchains.

II. Design and Implementation

Veryl is designed with three key principles: syntactic simplicity, SystemVerilog interoperability, and productivity. The language maintains familiarity for SystemVerilog experts while introducing modern features for safer, more efficient hardware design.

A. Language Features

Veryl introduces several key innovations:

Optimized Syntax: Veryl simplifies common hardware design patterns while removing ambiguities and unsynthesizable constructs. Veryl simplifies common hardware design patterns while removing amibguities and unsynthesizable constructs. For example, if a module contains only one clock and one reset, the Veryl compiler automatically infers their usage without requiring explicit declaration. At the same time, it allows explicit handling of multiple clocks when needed. This approach simplifies the description of common patterns while ensuring expressiveness for rare cases.

Strong Type System: Veryl's type system includes a `clock` type, and requires explicit annotation of clock domains and unsafe clock domain crossing (CDC) blocks, enabling automatic detection of unintended CDC issues during compilation rather than simulation or post-synthesis:

```
module ModuleA (
    i_clk_a: input  'a clock,
    i_dat_a: input  'a logic,
    i_clk_b: input  'b clock,
    o_dat_b: output 'b logic,
) {
    unsafe (cdc) {
        assign o_dat_b = i_dat_a; } }
```

Generics System with Prototypes: Veryl implements a powerful generics system that goes beyond traditional parameter overrides, supporting parametrization of function signatures; and module, data structure, interface, and package definitions. Veryl also supports module and interface prototypes, which enable using modules and interfaces themselves as generic parameters, similar to Rust traits, or Haskell classes:

```
prototype ModInterface::<DWIDTH: const,
        AWIDTH: const> (
    i_clk  : input  clock          ,
    i_reset: input  reset          ,
    i_valid: input  logic          ,
    i_data : input  logic<DWIDTH>,
    i_addr : input  logic<AWIDTH>,
    o_ready: output logic          ,
);

module MemoryController::<T: module
```

```
ModInterface::<8, 16>> (
    i_clk  : input clock,
    i_reset: input reset,
) {
    // Instantiate the generic module
    inst memory_interface: T (
        i_clk  ,
        i_reset,
        // ...
    );
}
```

Real-time Diagnostics: Issues such as undefined, unused, or unassigned variables are detected and reported during editing rather than at compile or simulation time.

Visibility Control: Modules can be marked with the `pub` keyword to indicate they are part of a module's public API, allowing better encapsulation of implementation details.

B. Modern Tooling Ecosystem

Veryl is built with a comprehensive, modern tooling ecosystem that brings software development best practices to hardware design:

Streamlined Installation: Unlike traditional EDA tools with complex installation processes, Veryl can be installed with a single command using either Cargo (Rust's package manager) or the dedicated `verylup` version manager. The `verylup` tool enables users to easily switch between different Veryl versions, install specific versions for different projects, and update to the latest release—similar to `rustup`, `nvm`, or `pyenv` in software development. This frictionless installation process significantly lowers the barrier to entry, particularly for new users and in educational settings where complex tool setup often consumes valuable learning time.

Language Server Protocol (LSP): Veryl implements the Language Server Protocol, providing real-time diagnostics, go-to-definition, find-all-references, hover information, and auto-completion within all major editors including VSCode, Vim, Emacs, and JetBrains IDEs. This critical feature enables developers to identify errors instantly rather than waiting for compilation or simulation failures.

Automatic Formatting: The built-in formatter ensures consistent code style across projects and teams, eliminating style debates and reducing code review friction. The formatter can be invoked manually, integrated into editors for on-save formatting, and enforced in continuous integration pipelines.

Package Management: Built-in dependency management enables easy incorporation of libraries by specifying repository paths and versions in a project configuration file, similar to npm, cargo, or pip in software ecosystems.

Testing Infrastructure: Test code written in SystemVerilog or cocotb can be embedded directly in Veryl code and executed through the `veryl test` command, enabling test-driven development workflows that are standard in software but challenging with traditional HDLs.

Documentation Generation: Support for Markdown, WaveDrom waveform diagrams, and Mermaid diagrams within documentation comments enables comprehensive, visually rich documentation that evolves alongside the code.

III. SystemVerilog Interoperability

A key feature of Veryl is its seamless interoperability with existing SystemVerilog codebases. Unlike many language transpilers that produce obfuscated or machine-optimized output, the Veryl compiler generates exceptionally clean, human-readable SystemVerilog code that follows established coding conventions. This high-quality output is a deliberate design choice that provides several critical benefits:

- Transparency for debugging and verification processes
- Easy code reviews of the generated SystemVerilog
- Compatibility with existing SystemVerilog toolchains and workflows
- Ability to manually modify the generated code if needed
- Lower barrier to adoption through clear relationship between Veryl code and its SystemVerilog equivalent, including a Veryl-SystemVerilog map file

SystemVerilog modules, interfaces, and packages can be referenced directly using the `$sv` namespace, allowing seamless bidirectional interoperability, e.g.: `let a: logic = $sv::PackageA::ParamA` and `inst b: $sv::ModuleB(.*);`

This high-quality code generation, combined with the `$sv` namespace approach, enables gradual adoption within existing projects without requiring an all-or-nothing migration strategy.

IV. Evaluation and Feedback

Initial user feedback indicates significant productivity improvements, with users reporting faster development cycles and fewer debugging sessions. The language's built-in safety features have proven particularly valuable for identifying cross-clock domain issues and type inconsistencies early in the design process, problems that traditionally manifest only during simulation or on physical hardware.

The abstraction of clock and reset polarity/synchronicity allows the same Veryl code to target both ASIC flows (typically using negative asynchronous reset) and FPGA flows (typically using positive synchronous reset) through build-time configuration rather than code changes.

V. Conclusion and Future Work

By lowering barriers to entry, improving code quality, and enhancing developer productivity, Veryl contributes to the broader movement toward accessible and collaborative open-source hardware design. Our future work focuses on:

- Expanding language server capabilities for more sophisticated code analysis
- Integrating with additional simulators and synthesis tools
- Building a comprehensive standard library for common hardware components
- Implementing advanced static analysis for timing and resource optimization

Through these efforts, we aim to make hardware design more accessible to a broader community and accelerate innovation in open-source computer architecture.