Rapid Open Hardware Development Framework

Max Korbel
Intel Corporation
Santa Clara, USA
max.korbel@intel.com

***Abstract*—The Rapid Open Hardware Development (ROHD) framework is an open-source framework for describing and verifying hardware in the Dart programming language. ROHD enables engineers to build and traverse a graph of connectivity between module objects using unrestricted software. ROHD improves development iteration time by orders of magnitude and enables higher quality and more succinct designs, better abstraction through composition, and more powerful testbenches. It leverages modern software industry innovations like build, debug, profiling, and dependency management provided through the Dart ecosystem. ROHD includes a fast, event-based, four-value simulator with cosimulation support. Hardware models can be converted into logically equivalent, structurally similar, human readable, EDA tool compatible SystemVerilog with signal and port names maintained. ROHD can instantiate and interact with SystemVerilog modules, enabling gradual adoption of ROHD into derivative projects. The architecture of the framework is extensible to generate different kinds of output beyond just SystemVerilog. The ROHD Verification Framework, built upon ROHD, enables development of complex testbenches. ROHD aims to revolutionize hardware development and become a new industry standard.**

***Index Terms*—hardware, design, verification, framework, simulator**

I. Introduction

Many hardware developers today are frustrated by the industry-standard ecosystem consisting of SystemVerilog, UVM, and proprietary vendor tools. While hardware development has made slow progress over recent decades, the software industry has made outstanding progress in language design, static code analysis, integrated development environments (IDEs), design patterns, dependency management, application programming interfaces (APIs), performance, debugging, profiling, open-source collaboration, and much more. This paper introduces the Rapid Open Hardware Development (ROHD) framework, which aims to become the new industry-standard for front-end hardware design and verification. ROHD is an open-source framework written in Dart for describing and verifying hardware enabling developers to build, traverse, and simulate hardware models and intuitively interact with them. In addition to unlocking powerful new capabilities for design generation, verification stands to benefit massively from a fully featured modern programming language which can natively interact with hardware. ROHD comes with a built-in hardware simulator (with cosimulation support) and leverages a variety of software industry innovations. ROHD is offered as free and open-source software at https://github.com/intel/rohd and takes as one of its main goals to bring the benefits of open-source collaboration to hardware development [1].

A. Background

SystemVerilog is too limited for modern hardware description needs, as evidenced partly by frequent reliance on ad hoc scripts, editor macros, and code generators for signal insertion, interface connection, IP integration, etc. Developing a large system often requires a complex stack of libraries, packages, options, configurations, etc., which can discourage engineers from following good development practices to avoid interacting with delicate build systems. Many hardware testbenches, comprising of behavioral models, stimulus, checkers, coverage, etc. are written in SystemVerilog, which is seriously behind popular modern programming languages across a variety of metrics.

Several alternative methods for hardware design and verification have been proposed. Chisel is a hardware generator framework written in Scala, but it treats verification as second class and generated hardware is netlist-like [2]. The cocotb project enables Python testbenches to interact with SystemVerilog simulators, but it does not consider hardware generation and still relies on other tools for simulation [3]. High-Level Synthesis (HLS) allows designers to compile high level algorithms with specialized tools into hardware implementations, but it complexifies some lower-level tasks and often requires re-validation on the generated output code [4]. These and numerous other solutions don't sufficiently address language problems facing hardware engineers. ROHD can drastically improve productivity of engineers, increase quality of designs and test benches, and make hardware development as enjoyable as it should be.

B. The Dart Language

Dart is a modern, open-source, multi-platform, general purpose programming language [5]. The language is fast, includes a flexible type system, type inference, sound null safety, and garbage collection. The Dart analyzer performs static analysis and integrates well with a variety of popular IDEs. Dart also includes a great documentation generator and powerful profiling and debug tools. Dart has built-in support for asynchronous programming without multi-threading using Futures, Streams, and async/await, which are intuitive for modeling and interacting with hardware. Dart is easy to learn and is one of the fastest growing and most popular programming languages [6]. Dart has a thriving package ecosystem and an excellent package manager.

II. The Rapid Open Hardware Development (ROHD) Framework

ROHD is a generator framework built in the Dart language. Hardware models in ROHD are built by instantiating modules (instances of classes which extend Module) and connecting them with logical signals (each of type Logic). The only entities in ROHD that represent hardware are instantiated hardware objects, and any software approach can be taken to construct a design, sidestepping any concerns over synthesizability as exist in SystemVerilog. ROHD does not utilize any software reflection: the names of variables and the structure of generator code has no impact on the models' behavior or generated outputs, marking a meaningful advantage over some other frameworks in terms of flexibility. Overloaded built-in language operators conveniently instantiate most building block modules automatically. "Primitive" building block modules specify two critical pieces of information: the functional behavior of the logic in a simulation and the equivalent representation in a generated output format such as SystemVerilog. Users may define new primitive modules or create more abstract modules by composing other modules together. A module's contents are determined by tracing for logic which exists between the designated input and output ports of that module. The flexibility of ROHD, coupled with the excellent dependency management from Dart, enable very low barriers to reusability of designs, verification collateral, and infrastructure of any size.

ROHD is intended to fit in with the existing hardware development ecosystem to enable a gradual transition. ROHD designs and testbenches can interact with SystemVerilog designs via cosimulation and compose mixed designs of SystemVerilog and ROHD. Designs can be simulated and validated using the native ROHD simulator. Designs, once converted from ROHD to SystemVerilog, will include SystemVerilog submodule instantiations (including parameterization) alongside the rest of the hierarchy, and can be integrated into other SystemVerilog designs. Downstream tools like emulation, synthesis, formal verification, SystemVerilog simulators, etc. can consume generated SystemVerilog. Figure 1 shows some of these ways that ROHD can interact with legacy designs, environments, and flows.

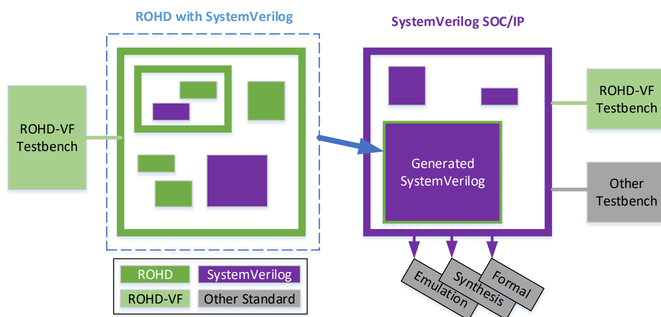

Fig. 1. Some of the ways in which ROHD designs, ROHD testbenches, SystemVerilog designs, legacy testbenches, the ROHD simulator, and industry standard tools can interact with each other.

A. Composition and Abstraction

Logic signals can be arbitrarily wide, be connected to other Logics or Modules, and have 4-value (0, 1, X, and Z) values of type LogicValue at a given simulation time. LogicValue is also a convenient type for testbench development because it has overloaded operators and extra functionality that are often missing in primitive types in non-hardware-oriented programming languages.

Much connectivity in ROHD happens through permanent assignments. The concept of Conditional objects in ROHD enables procedural execution of logical assignments to occur conditionally, similar to the contents of "always" blocks in SystemVerilog. Combinational and Sequential blocks consume collections of Conditional objects, making it easy to dynamically generate arbitrary logic.

ROHD comes with some built-in composite abstractions such as the pipelining abstraction which makes refactoring easy and optionally includes ready/valid protocol. The Pipeline object in ROHD allows specification of Conditional operations to be performed in a given pipeline stage, eliminating manual management of registers, resets, intermediate signals, and even things like ready/valid protocols. Stages can refer to signals in other stages via both absolute index or relative position, which makes adding or removing stages dramatically easier. Other abstractions are easy to create, reuse, and compose into even more powerful abstractions, without removing designers from RTL modelling where it makes sense.

A well-commented, interesting example of a small ROHD design for a logarithmic-height tree of arbitrary two-input/one-output functionality is available in the ROHD repository's example area at https://github.com/intel/rohd/blob/main/example/tree.dart.

B. The ROHD Simulator

ROHD comes with a built-in, fast, event-based, four-value simulator with support for X and Z propagation and VCD waveform dumping. The simulator maintains a chronologically sorted list of actions to perform at specified times and executes them until none remain. Logical simulation occurs via driving signal toggles (called "glitches"), where each signal propagates transitions to downstream listeners, such as other signals or modules (see Figure 2). When any module with custom logical behavior receives a glitch, it performs its defined functional behavior based on the new set of input values and then passes update glitches onto its outputs. Purely combinational logic propagates glitches immediately and automatically without any involvement from a central simulator.

The simulator is phased so that a single final "changed" event is emitted once all signals have settled. The changed event is intended for listeners like the testbench and is the root event for other events like posedge and negedge. The phases also enable proper sequential logic sampling behavior. Any signal can be used as a clock for sequential logic.

The ROHD simulator also supports cosimulation with SystemVerilog simulators via VPI. The simulator executes in

the same process as the rest of the framework and user-written code which improves efficiency, visibility, profiling, and debuggability to simulations.

C. Generated Outputs

ROHD uses objects called Synthesizers to interpret hardware models and construct the desired output format from them. Some Synthesizers require all primitive modules to provide conversion functionality, while others may infer a default behavior based on surrounding context and hierarchy. ROHD includes a native SystemVerilog Synthesizer written in Dart which can convert ROHD modules into logically equivalent, structurally similar, human readable SystemVerilog (see Figure 2). SystemVerilog generation is vital for modern workflows, as practically all tools for back-end synthesis, emulation, formal verification, etc. read SystemVerilog. A survey of ROHD-generated code has been successfully taken through industry-standard static flows.

Many other generator frameworks perform optimization on hardware and produce a netlist-like representation which is unpredictable and difficult to read, debug, or ECO. By contrast, ROHD predictably maps each module and signal one-to-one to generated SystemVerilog without optimization, maintaining signal and port names and hierarchy.

Other Synthesizers are included and/or under development, and users can also create their own. ROHD has a Synthesizer for CIRCT, which is an intermediate representation and library of tools based on MLIR for generating hardware outputs including SystemVerilog [7]. Intermediate representations like CIRCT can provide a path to higher levels of abstraction enabled by compiler technology without migrating to a new front-end, so ROHD can reap benefits of future innovations. A UPF Synthesizer, coupled with added functionality for power-aware simulation, enables a hardware model to simulate with power intent and generate a complete set of associated UPF files. The extensible design of ROHD Synthesizers ensures the framework can keep up with future requirements.

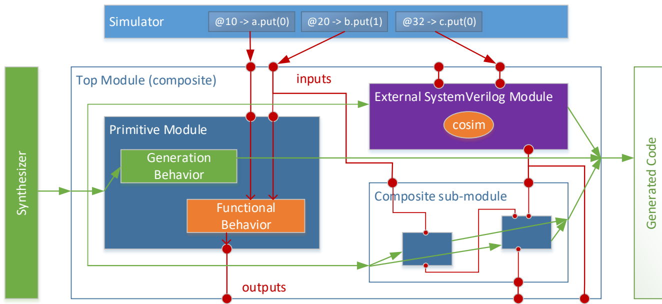

Fig. 2. ROHD modeling and interactions with hardware. Hierarchy is determined by connectivity between inputs and outputs of modules. The simulator assigns signal values at specified times which propagate through the hierarchy, executing any primitive functional behavior. The Synthesizer hierarchically executes generation behavior to create an output. SystemVerilog modules are cosimulated and instantiated by module name in generated SystemVerilog.

III. The ROHD Verification Framework

Verification collateral with ROHD designs is written in Dart with unbridled access to the full power of the language, including its excellent base libraries and huge collection of open-source packages. Signals and modules can be referenced intuitively as objects. Dart's asynchronous language features are more powerful alternatives to SystemVerilog's tasks, forks, and testbench analysis ports. Validation code executes as native software, rather than through a black-box tool, drastically improving performance and debuggability. Unit testing small portions of hardware is dramatically easier with ROHD than traditional methods. Dart's unit testing framework is on par with the best, and the simple build system makes instantiating hardware outside of its typical location very low-effort.

UVM, an industry-standard framework for SystemVerilog testbenches, encourages a popular methodology of component organization [8]. The implementation, however, consists of countless macros, requires excessive boilerplate code, and is heavily opinionated on design patterns. The ROHD Verification Framework (ROHD-VF) is a separate open-source framework (hosted at https://github.com/intel/rohd-vf) built upon ROHD which supports similar testbench organization to UVM, but without the macros, boilerplate, or strong opinions [9]. It includes base classes such as Driver, Monitor, Sequencer, Agent, Sequence, etc. like UVM does, as well as helpful utilities for logging, tracker generation, randomization, constraint solving, and more. Testbench code written in Dart is not convertible to SystemVerilog because it would inherently limit what portions of the language can be used to the subset which can also be represented in SystemVerilog.

IV. Results & Discussion

ROHD promises to improve overall developer efficiency, which is inherently a difficult thing to measure. This section discusses some successful usages of ROHD and provides some of the measurable comparisons, which are not always exactly apples-to-apples. Table I shows some metrics related to the reimplementation with ROHD of a relatively small, configurable, parameterized IP used in upcoming silicon products. The ROHD design only took a day to write for an unfamiliar designer relying on a high-level architectural specification, empowered by drastically reduced iteration times and the ability to intuitively create design-specific levels of abstraction. Build and simulation experienced massive performance boosts, even when run on a significantly less powerful computer.

While the "lines of code" metric does not necessarily directly correlate with efficiency, maintainability, or quality, it is a simple metric to measure which can offer some insight into overall effort and complexity. Table II shows the differences in number of lines of code for real designs in their original implementation compared to reimplementations with ROHD. The hardware interfaces referenced in the table are parameterized interfaces used in real productized IPs. Original interface definitions are written in a Tcl-based format that can be used to connect IPs sharing the same interface. Implementations

TABLE I

COMPARISON OF METRICS FOR IP "A" – A SMALL, CONFIGURABLE IP CONNECTING TO OTHER IPS OVER A STANDARDIZED INTERFACE.

Metric	Original	New with ROHD	Improvement
Development Time	Approximately 1 quarter for initial development, written in SystemVerilog	Less than 2 days for initial full functionality and a testbench with BFMs, with 1 engineer	97.8% reduction in development time
Lines of Code	About 1500 lines of SV for design only	About 450 lines of ROHD for design only	70% reduction in lines
Build time	About 300 seconds for design and testbench, run on a server machine	About 1.65 seconds for design and testbench, including generation, run on a laptop	182x speedup
Simulation Cycles per Second	About 170 CPS on EDA vendor simulator for test with >50k cycles and random stimulus	About 1400 CPS on ROHD simulator with similar input stimulus over >50k cycles	8.2x speedup

TABLE II

LINES OF CODE COMPARISONS BETWEEN ORIGINAL IMPLEMENTATIONS AND ROHD REIMPLEMENTATIONS FOR SOME REAL DESIGNS.

Hardware	Original Lines of Code	Lines of Code with ROHD	Improvement
AXI Interface	1161 lines of Tcl	229 lines of ROHD	80% reduction
Hardware Interface "C"	588 lines of Tcl	283 lines of ROHD	50% reduction
IP "D" HIP instantiation and connectivity	941 lines of SV, with generate statements and macros	114 lines of ROHD	88% reduction

with ROHD's Interface class generate synthesizable hardware interfaces in a more succinct way.

IP "D" is a highly configurable and parameterized IP which has a soft IP (SIP) top-level and a need to instantiate and generate connectivity for a configuration-selected hard IP (HIP) as a sub-module. The original implementation in SystemVerilog relies on clever usage of parameters, static functions, generate statements, and macros, and it is notoriously difficult to debug. Reimplementing this generation logic in ROHD is dramatically simpler and shorter.

ROHD can also be used as a powerful tool for generating hardware from machine-readable specifications. For example, a simple IP under development using ROHD has generated an entire hardware sub-block and interface by parsing a YAML file which identifies functionality and ports.

ROHD can also enable easy software modeling at various levels of abstraction. For example, a large and complex IP's team is building a microarchitectural model of their IP using ROHD. It is being used for performance analysis to guide microarchitectural decisions but is designed in a way to enable reuse of components for unit-level verification. The model is cycle approximate where convenient and cycle accurate at critical hardware interfaces.

V. CONCLUSION

This paper has introduced the Rapid Open Hardware Development (ROHD) framework as a powerful new method for developing hardware. ROHD drastically improves engineering productivity and enables hardware developers to build higher quality products faster. The framework is free and open-source and encourages participation in the open-source community. Numerous engineers have contributed to the ROHD ecosystem through direct bug reports and pull requests as well as new package development. Adoption and usage have been growing quickly since it became available.

ROHD aims to solve a wide set of hardware industry problems and become the new standard for hardware development. It addresses a broad set of challenges for both design and verification in a comprehensive way that alternatives do not. ROHD is extensible by design for other and new purposes. It unlocks areas of innovation which were previously hidden behind proprietary EDA vendor tools. ROHD is execution-ready and does not mandate an all-or-nothing approach: ROHD plays nicely with legacy code and existing infrastructure to enable gradual transitions to this more modern solution. Hardware developers will be amazed at the improvements in productivity and development enjoyment with ROHD.

ACKNOWLEDGMENT

ROHD would not have been possible without the guidance, mentorship, and support of Sandeep Chaparala. Josh Kimmel, Eric Norige, Alan Saw, Vivek Rajan, and Desmond Kirkpatrick have given invaluable advice, suggestions, and contributions. Teams and individuals which were early adopters for design, verification, and modelling are much appreciated.

REFERENCES

[1] M. Korbel, "Rapid open hardware development (ROHD) framework," https://github.com/intel/rohd, 2022.

[2] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, 2012, pp. 1212–1221.

[3] C. Higgs, "Cocotb," in *ORCONF 2015*, Geneva, Switzerland, 2015.

[4] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[5] Google. (2022, May) Dart overview. [Online]. Available: https://dart.dev/overview

[6] S. O'Grady. (2022, Mar.) The redmonk programming language rankings: January 2022. [Online]. Available: https://redmonk.com/sogrady/2022/03/28/language-rankings-1-22/

[7] C. Lattner and A. Lenharth, "Lattner, chris; lenharth, andrew," in *LLVM Developer Meeting*, 2021.

[8] Accellera Systems Initiative. (2015, Oct.) Universal verification methodology (uvm) 1.2 user's guide. [Online]. Available: https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf

[9] M. Korbel, "ROHD verification framework," https://github.com/intel/rohd-vf, 2022.

Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.