Extended Abstract: Effective simulation and debugging for a high-level hardware language using software compilers

Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, Arvind, Adam Chlipala
MIT CSAIL

1 Motivation

Hardware designs are expressed in a spectrum of languages ranging from low-level RTL (Verilog [15] or VHDL [8]) to sequential software languages with annotations for high-level hardware synthesis (Vivado HLS [17], Handel-C [6], Clash [12], etc.). Different points on this scale entail different trade-offs. Verilog offers limited programming abstractions and composability and thus is tedious to write and debug, but it provides developers fine-grained control over the resulting circuits. HLS systems, i.e., the hardware design systems that start from software languages to generate Verilog, offer rich abstractions and excellent debugging and simulation facilities but poor control over generated circuits. This is not surprising: the sequential computation model of software languages is deeply at odds with the hardware computation models, which try to run all parts of a circuit in parallel all the time.

Rule-based languages, such as Bluespec [9], Kôika [2], and Kami [3], offer an interesting middle ground, with both predictable performance and high-level, usable semantics. Rule-based designs describe the manipulation of (hardware) state elements using state-transforming atomic *rules*, which (appear to) execute sequentially. An RTL compiler for such a rule-based system introduces concurrency by translating rules into individual circuits that run in parallel preserving *one-rule-at-a-time* semantics.

While significant effort has been dedicated to synthesizing high-quality hardware from Bluespec designs, comparatively little effort has been expended on simulation, debugging, and testing of rule-based designs: these tasks are typically performed at the generated-Verilog level. This is non-ideal for two reasons:

1. generated circuits suffer from poor readability due to compiler optimizations breaking high-level abstractions, worsening the debugging experience, and
2. generated circuits are optimized for the hardware execution model where concurrency is free, rather than the mostly sequential model of CPUs—this is inefficient for simulation.

Our work is motivated by the idea that architecture hobbyists and researchers should be able to create hardware designs easily: we want to enable architects to write in high-level hardware-description languages without sacrificing any of (1) the ability to generate efficient, synthesizable circuits with predictable performance characteristics, (2) direct visbility of high-level abstractions when debugging, and (3) simulation performance (in fact, we can leverage high-level semantic properties to simulate *faster*).

2 Limitations of the State of the Art

Architects currently have three main options for simulation: high-level non-architectural simulation, hardware simulation using an FPGA, and cycle-accurate simulation on a CPU. *High-level, non-architectural simulation* [1, 13] is fast. However, the models do not cycle-accurately represent synthesizable designs and have an unpredictable cost model. *Simulating on an FPGA* is both fast and cycle-accurate. However, the synthesis, placement, and routing flow to compile a design to run on an FPGA is slow, and so iterating on a design is time-intensive. Furthermore, debugging on an FPGA is challenging: even printf, the simplest tool used in debugging, is unavailable.

Hence, we focus on *cycle-accurate simulation* on a CPU. Verilog is a common language for writing RTL and describing circuits cycle-accurately. However, Verilog (analogously to assembly) is intrinsicially hard to debug. Furthermore, Verilog is often auto-generated, which makes it even more inscrutable: translations are not designed for readability, and high-level abstractions are lost in translation. Programmers debugging Verilog mainly use printf and assertion debugging, or they examine low-level waveforms. This is primitive in comparison to software debugging, which offers powerful tools like gdb [14], reverse debugging [4], and code profiling.

Tooling for RHDLs is also limited. Bluesim [10] (a Bluespec-level cycle-based simulator) generates models for simulation after multiple passes of compilation, which leads to hard-to-read models. Furthermore, its performance is poor, and so it is common to first compile to RTL and then simulate the resulting circuit. This approach again suffers from lack of readability and high-level abstraction: in particular, there is limited support for exploring executions below the cycle granularity.

In summary, with the state of the art, models generated for simulation are not simultaneously designed for cycle-accurate semantics, readability/debuggability, and simulation performance.

3 Key Insights

Circuits compiled from high-level HDLs are optimized for the concurrent execution model of hardware, rather than the sequential model of CPUs. The hardware model has a tradeoff between area and critical path: often you get faster circuits with a shorter critical path by doing potentially redundant work. Thus, a significant amount of code in Verilog is "dead" at any given point. Modern simulators have been unable to exploit this, and many signals are evaluated unnecessarily.

As an example, rule-based languages have an "early-exit" semantics, with the compiler dynamically tracking whether a rule "fails" at any point in its execution (e.g. by conflicting with another rule resulting in breakage of the one-rule-at-a-time semantics). A circuit does not have this luxury: after a failure, the rest of the circuitry is still there, and it computes all intermediate values of the rules. By specializing our compilers for simulation, we can leverage this high-level information to "early-exit" to avoid simulating dead code.

One key insight is the following: simulating a high-level design and synthesizing hardware from a circuit description are fundamentally different activities that deserve fundamentally different compilers making fundamentally different assumptions and compilation choices. Thus, we should separate the simulation and synthesis pipelines to allow compiler specialization. Furthermore, by compiling to C++, we can leverage existing compilers and toolchains (e.g. gdb, gcov [7], gprof [5], rr [11]) to get fast and debuggable software models for free (as compared to compiling directly to machine code, which fails to take advantage of existing compiler optimizations and would be unusable for debugging).

The result is an improved hardware-development workflow: architects write in an RHDL with convenient abstractions, compile to a readable, cycle-accurate C++ model, debug and profile with standard software tools, then synthesize to RTL.

4 Main Artifacts

We present a methodology for better simulation of RHDLs, leading to a better hardware design experience, and an open-source compiler to C++ for the Kôika RHDL (*Cuttlesim*).

We evaluate the performance of Cuttlesim by benchmarking embedded processor designs and DSP building blocks; see Table 1. We compare primarily against Verilator, an open-source, state-of-the-art Verilog simulator[1]. To evaluate the hardware design and debugging process, we conduct a series of case studies (functional-correctness-debugging of a cache coherence protocol, functional validation of a design using randomized testing, performance-debugging of an embedded processor core, and design exploration adding a branch predictor to an existing processor).

5 Key Results and Contributions

Performance results. As shown in Figure 1, Cuttlesim consistently out-performed Verilator by a factor ranging from 2x to 3x on small but realistic designs, including flavors of an embedded-class RISC-V core and DSP designs (FFT, finite impulse filter). Figure 2 shows that simulation with Cuttlesim out-performed simulation with circuits generated by both Kôika and a commercial Bluespec compiler from equivalent designs. As Kôika has not yet been used to design very large systems, our benchmarks are all small to medium-sized. While we do not believe that the results would be different, more experience would be needed to generalize the claims to very large designs.

Debugging and design-exploration methodology. The case studies (Section 4.2) demonstrate novel and interesting ways to apply software tools to hardware designs: code-coverage tools operating at the source-line level provide a wealth of architectural data without adding a single hardware counter (getting, for example, branch misprediction counts for free); reverse debuggers and hardware watchpoints allow quick bug-finding without consulting waveforms; and *sub-cycle*, step-by-step debugging of *readable* models lets one explore an unfamiliar design interactively, at a level matching the original Kôika semantics. Ultimately, leveraging existing software toolchains enables a whole range of novel and interesting hardware debugging and design exploration methodologies.

Contributions.

1. We show how using completely separate toolchains for software simulation and hardware synthesis leads to faster simulation and improved debugging experience.
2. We describe techniques to build fast software models of rule-based designs, using lightweight transactions.
3. We show that rule-based designs are amenable to heavy optimization through static analysis that exploits the simplicity of the input language.
4. We give concrete evidence of the value of this approach using simulation performance benchmarks and debugging and design-exploration case studies.

6 Citation for Most Influential Paper Award

This paper presented the last piece of a new hardware-development toolchain based on completely separating simulation and synthesis pipelines, enabling the design of circuits with predictable performance, a debugging experience on par with the one software developers have come to expect, and simulation abilities matching the fastest-available commercial and free simulators.

It was instrumental in allowing architecture hobbyists and researchers to create hardware designs and explore architectural ideas as easily as programmers write software.

[1]The authors of Verilator write that "*Verilator has typically similar or better performance versus the closed-source Verilog simulators (Carbon Design Systems Carbonator, Modelsim, Cadence Incisive/NC-Verilog, Synopsys VCS, VTOC, and Pragmatic CVer/CVC).*" [16]

References

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[2] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 243–257, New York, NY, USA, 2020. Association for Computing Machinery.

[3] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017.

[4] Jakob Engblom. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pages 1–6. IEEE, 2012.

[5] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982.

[6] Mentor Graphics. Handle-C. https://www.mentor.com/products/fpga/handel-c/.

[7] GNUGPL License. Gcov: Gnu coverage tool.

[8] Zainalabedin Navabi. *VHDL: Analysis and modeling of digital systems*. McGraw-Hill, Inc., 1997.

[9] Rishiyur Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, pages 69–70. IEEE, 2004.

[10] Rishiyur S Nikhil. What is bluespec? *ACM SIGDA Newsletter*, 39(1):1–1, 2009.

[11] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability: Extended technical report. *arXiv preprint arXiv:1705.05937*, 2017.

[12] QBayLogic. Clash: A modern, functional, hardware description language. https://clash-lang.org/.

[13] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 475–486, New York, NY, USA, 2013. Association for Computing Machinery.

[14] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with GDB. *Free Software Foundation*, 675, 1988.

[15] Donald E. Thomas and Philip Moorby. *The Verilog hardware description language (3. ed.)*. Kluwer, 1996.

[16] Veripool. Verilator. https://www.veripool.org/wiki/verilator.

[17] Xilinx. Vivado hls. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.