What are the Semantics of Hardware?

Gilbert Bernstein
UC Berkeley

Ross Daly
Stanford University

Jonathan Ragan-Kelley
MIT

Pat Hanrahan
Stanford University

ABSTRACT

Recently, numerous hardware description, hardware generator languages, and intermediate representations have been developed to facilitate creation and programming of new accelerators. These languages require a reference interpreter to describe their semantics. These semantics differ according to the level of abstraction at which a circuit is described. We propose an endeavor to formalize and relate each level of semantics using abstract interpretation.

1 INTRODUCTION

There has been a proliferation of hardware design languages [3, 5, 10, 16, 22], domain-specific languages for hardware [2, 11, 12, 14, 17, 18], high level synthesis languages (HLS) [4, 6], and intermediate languages [9, 13, 20, 23]. These languages appeal to a variety of semantics, at varying levels of clarity and formality. For the developers of such languages, questions of semantics pragmatically resolve into the questions "how do I implement a reference interpreter?" "how do I implement a performant simulator?" and "why do I believe different interpreters/simulators for my language are consistent?"

There are at least three major points of view on implementing interpreters/simulators. If the language resembles sequential code (e.g. HLS), then the host software-language semantics are often appealed to. In functional HDLs, some dataflow interpretation[15] is commonly used. Verilog itself uses a form of event-based simulation that resembles dataflow[1]. Lastly, many high-performance simulators flatten module hierarchy, eliminate loops by breaking the circuit at registers, and compile a single transition function for updating the state at each synchronous clock.

Each of these approaches run into various issues. The sequential code view breaks down in the presence of "loopy" hardware that isn't simply a feed-forward pipeline. Verilog's semantics suffer from inconsistencies across simulators, and a complex specification of event-processing order within moments of time. Flattening enables optimization of the circuit simulation, but destroys modularity in the form of separate compilation. Lastly, the dataflow point of view (which is the most similar to our proposal) does not handle combinational loops and other lower-level aspects of hardware simulation.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
LATTE '21, April 15, 2021, Virtual, Earth
© 2021 Copyright held by the owner/author(s).

More fundamentally, we are stuck with the problem that there is no single correct *level of abstraction* at which to simulate hardware. For instance, how should time be handled? There are at least high-level asynchronous[15, 18], clock-synchronous, and low-level asynchronous (e.g. delays and latch-timings) models. Values also get more exotic than simply 0 or 1 logical voltage levels. Verilog has four values: 0, 1, X, and Z. VHDL allows a 9-valued logic.

Our idea is to use the theoretical framework of Abstract Interpretation [7] to define semantics, and therefore reference interpreters, for HDLs at *multiple* levels of abstraction. At the heart of abstract interpretation are two big ideas. The first reflects its grounding in denotational semantics: recursion and loopy dataflow behavior is defined via fix-point solutions to recursive systems of equations. However, arbitrary systems of equations do not always have solutions. Therefore the second idea is the construction of abstract domains using lattices, in which some unique solution can unconditionally be computed in finite time.

Abstract Interpretation, Model Checking, Theorem Proving, etc. are often applied to hardware as *formal methods* (e.g. [21]) to prove claims about what can or can't happen in a simulation. Our idea here differs in that we are using abstract interpretation to define what simulation *means* — what the behavior of a specified circuit is, in the first place. Pragmatically, this means writing reference interpreters and simulators for HDLs, rather than creating and checking auxiliary proof collateral for some particular circuit.

2 ABSTRACT INTERPRETATION

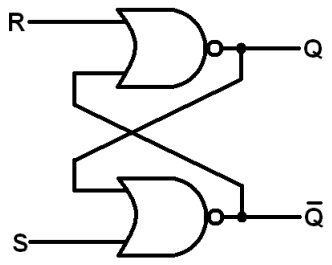

Figure 1: SR-Latch built from two NOR gates

Consider the SR-latch in Figure 1 built using two NOR gates. Even though this circuit contains combinational loops, we can simulate it as follows. Let $\{0, 1, \top, \bot\}$ be our set of abstract lattice values ($\top$ is read "top", $\bot$ is read "bottom"). Each of these values x corresponds to a subset of all possible concrete values, which we write as $\gamma(x)$. For this lattice, $\gamma(0) = \{0\}, \gamma(1) = \{1\}, \gamma(\top) = \{0, 1\}, \gamma(\bot) = \{\}$. In the inverse direction, we have abstraction $\alpha(y)$ as a map from sets of concrete values to abstract values. Thus, the behavior of

x	y	NOR(x,y)
1	.	0
0	0	1
0	$\top$	$\top$
$\top$	$\top$	$\top$
$\bot$	.	$\bot$
$\bot$	$\bot$	$\bot$

Table 1: Behavior of NOR-gate on abstract values (symmetries omitted, and . represents anything except $\bot$)

any specific gate can be *lifted* to working on abstract values by (i) sending all inputs back through γ, (ii) applying the gate to all possible combinations of inputs, (iii) re-abstracting the resulting set of possible outputs with α. (See table 1 for an example.)

To simulate, we initialize the values on all wires to $\top$, except for input wires R and S, where the values are known. Now, suppose we set R to 1 and S to 0. Then, we can evaluate the NOR gate with R as an input. NOR of 1 and $\top$ yields 0. We then use the lattice *meet* operation (written $\wedge$) to merge 0 into the previous R-NOR-gate output wire, which had value $\top$. The behavior of meet reflects *intersection* of the corresponding concrete sets of values, so $0 \wedge \top = 0$. (For our whole lattice, $x \wedge \top = x$, $x \wedge x = x$, $x \wedge \bot = \bot$, and $0 \wedge 1 = \bot$) Continuing this simulation *until convergence* yields the output 0 from the latch. If we instead began our simulation with S = R = 0, then we converge to output $\top$.

As another example, consider a loop of three NOT-gates, as one might use to build a clock. If we set any given wire to 0 or 1 and then simulate, the simulation will converge to all wires having value $\bot$. In general, our lattice construction ensures that any simulation will converge to a unique *fix-point* solution in finite time, regardless of the order in which gates are evaluated. Thus, any simulation in this mold inherently supports parallelization and distribution.

The $\top$ value discussed above helps us clarify what values like X (in Verilog) ought to mean. However, note that the concept here is clearer than "unknown": $\top$ means a wire *might* be 0 or 1, but not necessarily. $\bot$ is also "unknown" but with the meaning that no solutions are possible. Z (from Verilog) doesn't correspond to either of these abstract values.

A physical triple inverter loop will generate an oscillatory signal, but at this atemporal *level of abstraction*, the concept of oscillation is nonsense. Likewise, setting all inputs to the SR-latch to 0 yields $\top$ as output, rather than the "previous" value, because our values (and simulation) is atemporal.

2.1 Time

In order to account for time in our simulations, we can replace our simple domain of values (i.e. $\{0, 1, \top, \bot\}$) with *signals-over-time* in at least two ways. Given a fixed global clock (i.e. a synchronous model), we can think of signals as infinite (or bounded) sequences of values. Otherwise, we can think of signals as piece-wise constant functions of time, encoded as sequences of (`time`, `val`)-pairs, where `time` is always increasing. Each pair encodes a change in the function at `time` to `value`.

Both of these models are themselves lattices, where their values are drawn from a lattice. We can compute the $\wedge$ of synchronous sequences by taking the element-wise $\wedge$. For asynchronous sequences, we can merge the two time-sorted sequences and $\wedge$ at each point in time, using the last change in the other signal. We can also establish an abstraction relationship between these two lattices themselves. Given a clock-rate, a synchronous signal can be converted to an asynchronous signal by annotating it with regular time-stamps. In the other direction, any asynchronous signal can be converted to a synchronous signal by "joining" ($\vee$, which is dual to $\wedge$) all values within each regular time-step; this "join" is the abstract analog of the union of sets of concrete values.

2.2 Meaning of Z

In Verilog, the purpose of the "Z" value is to allow the simulation of circuits whose behavior relies on *high-impedance*. This occurs where wires lack an intrinsic dataflow direction (e.g. a bus wire driven by multiple tri-state buffers). The simplest such case is a transistor, where there is no inherent directionality between the source and the drain. By extending the framework of abstract interpretation to allow for components like transistors to define *relations* rather than functions (like the NOR gate earlier), we can also simulate multi-directional behavior on wires. In doing so, we find that "Z" is not properly thought of as a value in the first place.

3 FURTHER DIRECTIONS

If we replace our underlying $\{0, 1\}$ value-domain with a continuous model of voltages (in the interval $[0, 1.8]$) using polyhedral abstract values [8], then we can analyze analog circuit behavior. We can also abstract voltages via tolerance-defined bins, which we expect will give us leverage on disentangling the meaning of VHDL's 9-valued logic. Ultimately, this procession towards lower levels of abstraction reaches physics, where behavior is defined as the solution of systems of differential equations—also amenable to abstraction.

New "just-in-time" [19] or distributed simulation methods propose to *separately compile* parts of circuits that must then "synchronize" or converge via communication. The fix-point formulation of semantics clarifies the conditions under which such simulations converge to a unique solution.

Applying formal methods below the clock-synchronous level of abstraction is possible using these semantics. For instance, we believe it should be possible to show that a given component bridging clock domains does not exhibit meta-stability errors.

Most importantly, our existing prototype is less than 500 lines of code. Drastically reducing the complexity of developing robust and fully-featured HDLs has important ramifications for pedagogy and future research.

REFERENCES

[1] 2017. IEEE Draft Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE P1800/D3, April 2017 (Revision of IEEE Std 1800-2012)* (2017), 1–1316.

[2] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. C? ash: Structural descriptions of synchronous hardware using haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. IEEE, 714–721.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*. IEEE, 1212–1221.

[4] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of*

the 19th ACM/SIGDA international symposium on Field programmable gate arrays. 33–36.

[5] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–7.

[6] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[7] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[8] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Tucson, Arizona) *(POPL '78)*. Association for Computing Machinery, New York, NY, USA, 84–96. https://doi.org/10.1145/512760.512770

[9] Ross Daly, Lenny Truong, and Pat Hanrahan. 2018. Invoking and linking generators from multiple hardware languages using coreir. In *Proceedings of the 1st Workshop on Open-Source EDA Technology*.

[10] Jan Decaluwe. 2004. MyHDL: a Python-Based Hardware Description Language. *Linux journal* 127 (2004), 84–87.

[11] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 408–422.

[12] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. 2016. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–11.

[13] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 209–216.

[14] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 296–311.

[15] Edward A Lee and Thomas M Parks. 1995. Dataflow process networks. *Proc. IEEE* 83, 5 (1995), 773–801.

[16] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A unified framework for vertically integrated computer architecture research. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 280–292.

[17] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 393–407.

[18] Rishiyur Nikhil. 2004. 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.* IEEE, 69–70.

[19] Eric Schkufza, Michael Wei, and Christopher J. Rossbach. 2019. Just-In-Time Compilation for Verilog: A New Technique for Improving the FPGA Programming Experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 271–286. https://doi.org/10.1145/3297858.3304010

[20] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. Llhd: A multi-level intermediate representation for hardware description languages. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 258–271.

[21] Carl-Johan H. Seger and Randal E. Bryant. 1995. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Form. Methods Syst. Des.* 6, 2 (March 1995), 147–189. https://doi.org/10.1007/BF01383966

[22] Lenny Truong, Raj Setaluri, and Pat Hanrahan. 2016. Magma. https://github.com/phanrahan/magma.

[23] Clifford Wolf. 2016. Yosys open synthesis suite.