Generate Compilers from Hardware Models!

Gus Henry Smith
University of Washington
Seattle, USA
gussmith@cs.washington.edu

Ben Kushigian
University of Washington
Seattle, USA
benku@cs.washington.edu

Vishal Canumalla
University of Washington
Seattle, USA
vishalc@cs.washington.edu

Andrew Cheung
University of Washington
Seattle, USA
acheung8@cs.washington.edu

René Just
University of Washington
Seattle, USA
rjust@cs.washington.edu

Zachary Tatlock
University of Washington
Seattle, USA
ztatlock@cs.washington.edu

Abstract

Compiler backends[1] should be automatically generated from hardware design language (HDL) models of the hardware they target. Generating compiler components directly from HDL can provide stronger correctness guarantees, ease development effort, and encourage hardware exploration. Past work has already championed this idea; here we argue that advances in program synthesis make the approach more feasible. We present a concrete example by demonstrating how FPGA technology mappers can be automatically generated from SystemVerilog models of an FPGA's primitives using program synthesis.

ACM Reference Format:
Gus Henry Smith, Ben Kushigian, Vishal Canumalla, Andrew Cheung, René Just, and Zachary Tatlock. 2023. Generate Compilers from Hardware Models!. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

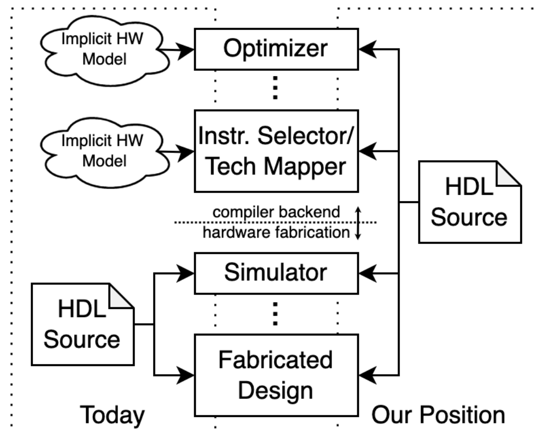

Figure 1. How the components of a software/hardware toolchain for a piece of hardware are built today: current state of the world (left) vs. according to our position (right).

1 Our Position

The semantics of HDLs are very rich. From the advanced type systems of new languages such as Aetherling [9] or Filament [13], to the high-level, algorithmic expressiveness of High Level Synthesis, hardware design languages convey much useful information about the hardware they describe. Even stalwart SystemVerilog and VHDL accurately capture a precise description of how a hardware design functions, including the ability to specify low-level details like latency.

Despite its richness, the HDL description of a hardware design is currently only used by the lowest layers of the software/hardware toolchain. Figure 1 (left) visualizes this. The HDL model of a design *is* used to build simulators and compile the final fabricated design, but the same model *is not* used when building higher-level toolchain components such as code optimizers or instruction selectors. Instead, these parts of the toolchain often contain handwritten (and sometimes implicit) models of the target hardware, e.g., a model of the hardware's memory hierarchy built into the optimizer.

It is our position that **compiler backends should be automatically generated from the HDL model of the target hardware.** Automatically generating compiler backends **(1) provides stronger correctness guarantees** as compiler components no longer rely on handwritten, implicit, potentially buggy models of hardware. Instead, compiler backends would rely on the same HDL source from which the hardware is fabricated, guaranteeing that the compiler's hardware model matches the fabricated hardware. For similar reasons, automatically generating compiler backends **(2)**

[1]We broadly define a *compiler backend* as any program that modifies, optimizes, or lowers high-level, hardware-independent code into low-level, hardware-specific code. This broad definition includes software compilers like gcc and libraries like CUDA, but also hardware compilers like FPGA synthesis or High-Level Synthesis (HLS) tools.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

reduces compiler development effort by removing the need to build a duplicate hardware model into the compiler backend. Lastly, we believe this approach **(3) encourages hardware exploration.** By providing more confidence in the correctness of the toolchain and reducing the burden of building a compiler for a new piece of hardware, hardware designers will be emboldened to experiment with new designs.

Furthermore, it is our position that **recent advances in programming languages make automated compiler construction feasible.** The idea of automatically generating compiler backends is not new: previous work includes synthesizing instruction selectors [3, 6–8] and code generators [4, 11], among other work. However, much of this work is a decade old, if not more, and does not benefit from advances in languages and type systems for hardware [9, 12, 13], equational reasoning via equality saturation [17, 19], program synthesis [15, 18], and machine learning for program generation [1, 2].

To ground our position, we present a concrete example, in which we use SystemVerilog models of FPGA primitives to automatically build technology mappers using modern program synthesis techniques.

2 Generating Technology Mappers

Technology mapping is an FPGA compiler backend step in which a high-level hardware design is lowered to use hardware *primitives* (small functional blocks) available on the target FPGA. Currently, technology mappers are often implemented as hand-written pattern matchers, which look for patterns in high-level HDL code and rewrite them to instances of FPGA primitives.[2] Some automation does exist; the VTR project [14] seeks to automatically provide compiler backends for hardware given just an architecture description using tools like ABC [5] and ODIN-II [10].

Existing technology mapping approaches—hand-written pattern matchers and automated tools—have a number of weaknesses. They fail to provide strong correctness guarantees: hand-written patterns can be incorrect. They require significant developer effort: when an automated tool cannot support an FPGA primitive, developers must support the primitive by hand. Finally, current tools limit exploration: each new FPGA primitive represents a potentially high cost to support.

We have prototyped a tool which generates technology mappers automatically from the HDL models of the target FPGA. Our tool automatically extracts bitvector semantics from the SystemVerilog models of FPGA primitives provided by each FPGA vendor. We then apply *program synthesis,* a technique which utilizes SMT solvers to generate programs. We use the bitvector semantics extracted from each primitive's SystemVerilog model to check—with the help of the solver—whether the primitive can be configured to implement the input high-level hardware design. Furthermore, we build an intermediate representation which allows for the construction of platform-independent *templates,* which capture patterns common across FPGA architectures.

Table 1. FPGA primitives imported automatically (and thus available for technology mapping) from vendor-provided SystemVerilog models, with source lines of code of the original SystemVerilog models.

FPGA	Primitive	SystemVerilog
Xilinx Ultrascale+	LUT6	88
	CARRY8	23
	DSP48E2	1426
Lattice ECP5	LUT2	5
	LUT4	7
	CCU2C	60
	ALU24B	672
	MULT18X18D	985
SOFA [16]	frac_lut4	69
Intel Cyclone	altmult_accum	1460

Our prototype approach to technology mapping provides strong correctness guarantees, reduces development effort, and can support hardware exploration. Our approach's strong correctness guarantees come not only from our use of SMT solvers, but also from the fact that we use the primitive semantics extracted directly from SystemVerilog, rather than relying on handwritten, and potentially incorrect, semantics. We quantify our approach's reduction of development effort by listing the primitives automatically imported (and thus supported) by our tool in table 1. Finally, our approach can encourage the exploration of new FPGA primitives, by quickly generating technology mappers for hardware prototypes during the development process.

3 Conclusion and Future Directions

We have argued that compiler backends should be automatically generated from the HDL models of the hardware they target. Furthermore, we provided a concrete demonstration of this idea via a prototype tool which generates FPGA technology mappers given the SystemVerilog models of an FPGA's hardware primitives.

We call on others in the field to revive this idea with us via the application of modern techniques, such as machine learning or equational reasoning via equality saturation.

[2]For one example of these patterns in the open-source FPGA compilation tool Yosys [20], see https://github.com/YosysHQ/yosys/blob/cee3cb31b98e3b67af3165969c8cfc0616c37e19/techlibs/xilinx/xcu_dsp_map.v

References

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[3] Florian Brandner, Dietmar Ebner, and Andreas Krall. Compiler generation from structural architecture descriptions. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 13–22, 2007.

[4] Florian Brandner, Viktor Pavlu, and Andreas Krall. Automatic generation of compiler backends. *Software: Practice and Experience*, 43(2): 207–240, 2013.

[5] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*, pages 24–40. Springer, 2010.

[6] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. Synthesizing an instruction selection rule library from semantic specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 300–313, 2018.

[7] Ross Daly, Caleb Donovick, Jackson Melchert, Rajsekhar Setaluri, Nestan Tsiskaridze Bullock, Priyanka Raina, Clark Barrett, and Pat Hanrahan. Synthesizing instruction selection rewrite rules from rtl using smt. In *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2022*, page 139, 2022.

[8] Joao Dias and Norman Ramsey. Automatically generating instruction selectors using declarative machine descriptions. *ACM Sigplan Notices*, 45(1):403–416, 2010.

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

[10] Peter Jamieson, Kenneth B Kent, Farnaz Gharibian, and Lesley Shannon. Odin ii-an open-source verilog hdl synthesis tool for cad research. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 149–156. IEEE, 2010.

[11] Rainer Leupers and Peter Marwedel. Retargetable generation of code selectors from hdl processor models. In *Proceedings European Design and Test Conference. ED & TC 97*, pages 140–144. IEEE, 1997.

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

[13] Rachit Nigam, Pedro Henrique Azevedo De Amorim, and Adrian Sampson. Modular hardware design with timeline types. *arXiv preprint arXiv:2304.10646*, 2023.

[14] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B Kent, Peter Jamieson, and Jason Anderson. The vtr project: architecture and cad for fpgas from verilog to routing. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 77–86, 2012.

[15] Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[16] Xifan Tang, Ganesh Gore, Grant Brown, and Pierre-Emmanuel Gaillardon. Taping out an fpga in 24 hours with openfpga: The sofa project. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 400–400. IEEE, 2021.

[17] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 264–276, 2009.

[18] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152, 2013.

[19] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5 (POPL):1–29, 2021.

[20] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, page 97, 2013.