A Position on Program Synthesis for Processor Development

Zachary D. Sisco
University of California, Santa Barbara
USA

Jonathan Balkind
University of California, Santa Barbara
USA

Timothy Sherwood
University of California, Santa Barbara
USA

Ben Hardekopf
University of California, Santa Barbara
USA

ABSTRACT

Program synthesis is a programming-languages technique not often seen in hardware design. However, hardware designs, and processors in particular, contain characteristics that match well with problems solved using program synthesis. We can use specifications as oracles to guide program synthesis and generate correct-by-construction HDL code. The hierarchical structure of hardware lends itself to "sketching", or partial implementations, where components can be solved individually. CEGIS-based synthesis techniques, which use SMT solvers, are a natural match for modeling netlists and RTL designs using the theory of bitvectors. We are exploring different directions in applying and adapting program synthesis for processor development. Presented as preliminary work in this position paper, we use program synthesis techniques to generate HDL code that implements the control logic for a sketch of a processor's datapath. There are a number of challenges to address such as scaling program synthesis tools to handle real-world hardware designs, and adapting tools to reason about "hardware semantics". Overcoming these challenges we argue program synthesis should be particularly beneficial to processor and hardware accelerator development, speeding up development time to keep pace with changes in specifications and microarchitecture-level optimizations.

1 OVERVIEW

Advances in program synthesis have been used to great success in software settings including code repair [6, 11, 13, 15], data wrangling (e.g., Excel FlashFill) [7], compiler superoptimization [8, 12, 14], graphics [9, 10], and more. These settings often center around domain-specific languages and tools that benefit from program synthesis. However, the domain of hardware design, driven by hardware description languages (HDLs), has received little attention from program synthesis (see [1, 3] for sketch-based Verilog code generation). We argue in this position paper that hardware designs, and processors in particular, have characteristics that match well with the kinds of problems solved using program synthesis. Further, we present preliminary work that exploits these characteristics by synthesizing HDL code that implements the control logic for a partial implementation of a processor. We present three characteristics:

LATTE '22, March 1, 2022, Virtual, Earth

Ubiquity of Specifications. Program synthesis requires some sort of specification to encode user intent and guide the search problem. Specifications range from formal (rigorous, mathematical or logical descriptions of behavior) to informal (sets of input-output examples). In software settings, where formal specifications are few, example-driven synthesis is common, but suffers from imprecision and overfitting. Hardware designs, however, are often already specified. Further, constituent components of the system also generally have behavioral specifications to describe how they fit into the entire system. We can leverage these existing specifications in program synthesis tools as oracles to generate HDL code that is correct-by-construction.

Hierarchical structure. Hardware naturally lends itself to hierarchical, structured components. As mentioned in the previous point, existing specifications also define the interfaces between these components. "Program sketches" (or partial implementations) are a common technique in program synthesis that allow pieces of the code to remain unknown or unspecified, to be later filled in by synthesis. Given the natural composability of hardware designs, program sketches in HDL code can help program synthesis scale.

Development Lifetime. Hardware designs are often modified and updated over their development lifetimes. For example, for SoC designs, changes to ISA, or microarchitecture-level optimizations can have non-local effects that permeate through the entire design and require exhaustive refactoring and verification. Program synthesis can alleviate these pains through automatically generating HDL code according to the updated specification with the guarantee that the changes are correct.

Given the relatively small adoption of program synthesis in the hardware design space, there are a number of challenges to adapt program synthesis to processor development.

Scalability. While this is already a well-known problem in the software world [4], program synthesis for HDL code has unique issues to contend with for scalability. The common artifact of HDL code is a netlist. For real-world hardware designs, netlists contain large orders of wires and gates. Modeling each and every component is intractable. Where the software world has seen success in program synthesis through domain-specific applications, similarly we find that specific classes of problems in hardware design necessitate specific solutions from program synthesis. Such solutions may call for intermediate representations or abstracted models of the hardware in order to scale.

Hardware semantics. Synthesizing HDL code requires reasoning about different models of computation compared to software. We

$$\frac{imem[pc] = \text{“LOAD addr”} \quad acc' = dmem[\text{addr}]}{(pc, acc, imem, dmem) \rightarrow (pc+1, acc', imem, dmem)} \textsc{Load}$$

$$\frac{imem[pc] = \text{“ADD addr”} \quad acc' = acc + dmem[\text{addr}]}{(pc, acc, imem, dmem) \rightarrow (pc+1, acc', imem, dmem)} \textsc{Add}$$

$$\frac{imem[pc] = \text{“STORE addr”} \quad dmem' = dmem[\text{addr} \mapsto acc]}{(pc, acc, imem, dmem) \rightarrow (pc+1, acc, imem, dmem')} \textsc{Store}$$

$$\frac{imem[pc] = \text{“BRZ addr”} \quad acc = 0 \quad pc' = \text{addr}}{(pc, acc, imem, dmem) \rightarrow (pc', acc, imem, dmem)} \textsc{BranchZero-T}$$

$$\frac{imem[pc] = \text{“BRZ addr”} \quad acc \neq 0}{(pc, acc, imem, dmem) \rightarrow (pc+1, acc, imem, dmem)} \textsc{BranchZero-F}$$

Figure 1: Operational semantics for the instructions for an accumulator-style ISA.

```
1  # fetch
2  inst <<= imem[pc]
3
4  # decode
5  op <<= inst[0:2]
6  addr <<= inst[2:].zero_extended(32)
7
8  # control logic
9  add <<= ??(op)
10 branch <<= ??(op)
11 write_acc <<= ??(op)
12 read_mem <<= ??(op)
13 write_mem <<= ??(op)
14
15 with conditional_assignment:
16   with read_mem:
17     read_data |= dmem[addr]
18
19 with conditional_assignment:
20   with write_acc:
21     with add:
22       acc.next |= read_data + acc
23     with otherwise:
24       acc.next |= read_data
25
26 with conditional_assignment:
27   with write_mem:
28     dmem[addr] |= acc
29
30 with conditional_assignment:
31   with (acc == 0) & branch:
32     pc.next |= addr
33   with otherwise:
34     pc.next |= pc + 1
```

Figure 2: Sketch of the datapath for the accumulator-style ISA as a single-cycle processor, written in the Python-based HDL PyRTL [5].

need to adapt program synthesis tools to reason about hardware semantics (models of hardware), deal with high degrees of parallelism, statefulness, and timing.

2 PRELIMINARY WORK

Consider a scenario where we have a specification for an ISA, and an implementation of a processor's datapath. With the datapath in place, the remaining unknown in the implementation is the control logic. We show that with these two pieces (an ISA specification and a datapath sketch), we can leverage program synthesis techniques to automatically generate the control logic for this processor.

For conciseness, we adapt a minimal, accumulator-style ISA from [16]. It has four instructions: `LOAD addr`, `ADD addr`, `STORE addr`, and `BRZ addr`. The state includes an accumulator register (*acc*), program counter (*pc*), instruction memory (*imem*), and data memory (*dmem*). We present the operational semantics for the instructions in Figure 1.

We can extract ISA instruction semantics from a formal specification written in a language like Sail [2], which lets programmers define ISA instructions functionally. We use these definitions to extract the goals needed for program synthesis. The semantics of the instructions are agnostic to the actual implementation of the processor. This kind of specification is higher level than a microarchitectural specification and more detached from the RTL, but as we will show is sufficient for program synthesis to generate HDL code for the processor's control logic

The second piece is a partial implementation, or sketch, of the processor for this ISA. Let's assume that the developer implemented the datapath for a single-cycle version of the processor with the HDL code shown in Figure 2. This implementation is a *sketch* because we introduce "holes" (denoted by ??) for the definitions of the control signals (lines 9–13). These holes will be filled in by program synthesis.

The goal now is, for each instruction in the ISA, to determine how the control signals should be set in order to correctly execute the instruction, then generate the HDL code that implements the control logic. To accomplish this we symbolically evaluate the processor sketch to find values for the control signals that hold under the constraints given by each ISA instruction. For instance, to execute an `ADD` instruction, the control signals `read_mem`, `write_acc` and `add` must be asserted.

Our prototype lifts the HDL code sketch to a solver-aided IR. This IR symbolically evaluates the hardware design into constraints in the theory of bitvectors. For program synthesis we use Rosette [17], a framework for solver-aided programming. With the instruction semantics from Figure 1 we can generate preconditions and postconditions to guide program synthesis. We define pre- and postconditions only over the ISA-level state so that synthesis goals are agnostic to the microarchitecture and RTL implementation details. For example, the precondition for `ADD` asserts that the current instruction is an `ADD` opcode. The postcondition asserts that the accumulator register updates to be the sum of the current value in *acc* with the value in data memory at address `addr`.

Running the symbolic evaluation process for all four instructions we generate the control logic. First, our immediate result is a table of control signal values according to opcode. From this table, we can generate the following HDL code that implements the control logic:

```
add <<= (op == ADD)
branch <<= (op == BRZ)
write_acc <<= (op == ADD) ^ (op == LOAD)
read_mem <<= (op == ADD) ^ (op == LOAD)
write_mem <<= (op == STORE)
```

Our prototype supports continual development. Using our technique we synthesized multiple implementations of the accumulator processor's control logic for three different microarchitectures (one single-cycle, two multi-cycle)—all using the same high-level ISA specification.

We are extending our preliminary work to synthesize the complete control logic for a RISC-V processor given a Sail specification for the ISA. It currently supports a subset of the RV32I ISA for a single-cycle datapath. To showcase our work on practical designs we are extending our prototype to handle pipelining and more advanced microarchitecture features found in modern processors and accelerators.

REFERENCES

[1] Armaiti Ardeshiricham, Yoshiki Takashima, Sicun Gao, and Ryan Kastner. 2019. VeriSketch: Synthesizing Secure Hardware Designs with Timing-Sensitive Information Flow Properties. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1623–1638. https://doi.org/10.1145/3319535.3354246

[2] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3290384 Proc. ACM Program. Lang. 3, POPL, Article 71.

[3] A. Becker, D. Novo, and P. Ienne. 2014. SKETCHILOG: Sketching combinational circuits. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–4. https://doi.org/10.7873/DATE.2014.165

[4] James Bornholt and Emina Torlak. 2018. Finding Code That Explodes under Symbolic Evaluation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 149 (Oct. 2018), 26 pages. https://doi.org/10.1145/3276519

[5] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–7. https://doi.org/10.23919/FPL.2017.8056860

[6] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantiative Objectives. In *27th International Conference on Computer Aided Verification (CAV 2016)* (27th international conference on computer aided verification (cav 2016) ed.). https://www.microsoft.com/en-us/research/publication/qlose-program-repair-with-quantiative-objectives/

[7] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[8] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 62–73. https://doi.org/10.1145/1993498.1993506

[9] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. 2011. Synthesizing Geometry Constructions. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 50–61. https://doi.org/10.1145/1993498.1993505

[10] Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. Association for Computing Machinery, New York, NY, USA, 379–390. https://doi.org/10.1145/2984511.2984575

[11] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. 2005. Program Repair as a Game. In *Proceedings of the 17th International Conference on Computer Aided Verification* (Edinburgh, Scotland, UK) *(CAV'05)*. Springer-Verlag, Berlin, Heidelberg, 226–238. https://doi.org/10.1007/11513988_23

[12] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A Goal-Directed Superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 304–314. https://doi.org/10.1145/512529.512566

[13] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 772–781.

[14] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 297–310. https://doi.org/10.1145/2872362.2872387

[15] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2491956.2462195

[16] Mark Smotherman. 2019. *A Brief History of Microprogramming*. https://people.cs.clemson.edu/~mark/uprog.html

[17] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. *SIGPLAN Not.* 49, 6 (June 2014), 530–541. https://doi.org/10.1145/2666356.2594340