A Position on Program Synthesis for Processor Development

Zachary D. Sisco
University of California, Santa Barbara
USA

Timothy Sherwood
University of California, Santa Barbara
USA

Jonathan Balkind
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:

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 '22, March 1, 2022, Virtual, Earth
© 2022 Copyright held by the owner/author(s).
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 post-conditions to guide program synthesis. We define pre- and post-conditions 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 with the value in data memory at address add. With conditional_assignment:

```
with conditional_assignment:
    with write_mem:
        dmem[addr] = acc
    write_acc <<= (op == ADD) ^ (op == LOAD)
    with read_mem:
        read_data <<= imem[pc]
```

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


