Control Logic Synthesis: Drawing the Rest of the OWL

Zachary D. Sisco
University of California, Santa Barbara
USA
zsisco@ucsb.edu

Andrew David Alex
University of California, Santa Barbara
USA
aalex@ucsb.edu

Zechen Ma
University of California, Santa Barbara
USA
zechenma@ucsb.edu

Yeganeh Aghamohammadi
University of California, Santa Barbara
USA
yeganeh@ucsb.edu

Boming Kong
University of California, Santa Barbara
USA
boming_kong@ucsb.edu

Benjamin Darnell
University of Illinois Urbana-Champaign
USA
bzd2@illinois.edu

Timothy Sherwood
University of California, Santa Barbara
USA
sherwood@cs.ucsb.edu

Ben Hardekopf
University of California, Santa Barbara
USA
benh@cs.ucsb.edu

Jonathan Balkind
University of California, Santa Barbara
USA
jbalkind@ucsb.edu

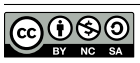
Abstract

System-on-chip (SoC) design requires complex reasoning about the interactions between an architectural specification, the microarchitectural datapath (e.g., functional units), and the control logic (which coordinates the datapath) to facilitate the critical computing tasks on which we all depend. Hardware specialization is now the expectation rather than the exception, meaning we need new hardware design tools to bring ideas to reality with both agility and correctness.

We introduce a new technique, "control logic synthesis", which automatically generates control logic given a datapath description and an architectural specification. This enables an entirely new hardware design process where the designer only needs to write a datapath sketch, leaving the control logic as "holes." Then, guided by an architectural specification, we adapt program synthesis techniques to automatically generate a correct hardware implementation of the control logic, filling the holes and completing the design.

We evaluate control logic synthesis over two classes of control (state machines and instruction decoders) and different architectures (embedded-class RISC-V cores and hardware accelerators for cryptography). We demonstrate how agile-oriented SoC developers can iterate over designs without writing control logic by hand yet still retain formal assurances with only minimal microarchitectural information.

ACM Reference Format:
Zachary D. Sisco, Andrew David Alex, Zechen Ma, Yeganeh Aghamohammadi, Boming Kong, Benjamin Darnell, Timothy Sherwood, Ben Hardekopf, and Jonathan Balkind. 2024. Control Logic Synthesis: Drawing the Rest of the OWL . In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3622781.3674170



ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0391-1/24/04.
https://doi.org/10.1145/3622781.3674170

1 Introduction

Embedded SoCs are the foundation of some of our most critical infrastructure, controlling everything from remote surgical equipment [46], to telecommunications satellites [13], to access to other larger compute and storage resources [30]. In such domains, any correctness issue could be catastrophic. As a result, studies show roughly half of development time is spent on verification [16]. To reduce cost and meet the growing demand for specialised hardware, we must find opportunities for correct-by-construction automation of design. Our new technique, control logic synthesis, meets this goal by freeing design engineers from writing control logic.

In this paper, we describe a method for automatically generating correct-by-construction control logic (or OWL, "operational wires and logic") for embedded-class root-of-trust SoCs. Our technique generates control logic with respect to a formal instruction set architecture (ISA) or instruction-level abstraction (ILA) specification, with only a minimal microarchitectural model, leaving the hardware designer free to iterate over the datapath and specification. Despite the leakiness of the control-datapath abstraction, we find that the datapath captures the designer's *intent* and narrows the innumerable microarchitectural possibilities down to a more manageable set tailored to the most important behaviors.

We work to compose the problem in a way that is tractable for modern program synthesis tools (synthesizing from the entire design and specification fails even for small hardware designs) and to handle the disconnect between the architectural specification and the microarchitecture (pipelining

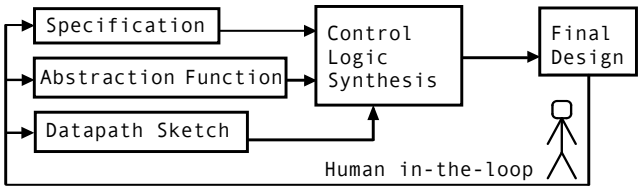


Figure 1. An overview of our technique: starting from an HDL sketch of the datapath combined with a formal architectural specification to generate correct-by-construction HDL code that completes the control logic.

being one example challenge). We focus on the kinds of bespoke embedded SoC designs which current solver-aided techniques can handle and where correctness is crucial.

1.1 The Control-Datapath Divide

Traditionally, embedded SoC design requires human reasoning about all of the behaviors a specific ISA/ILA might embody, down through the microarchitecture, including optimizations such as pipelining, caching, etc., to a complete digital design. Holding such a complex set of relationships in one's head all at once is incredibly difficult. When adopting an existing, open design, today's hardware designer must learn all of this information to extend the architecture and optimise the microarchitecture for their domain-specific goals. To make our reasoning simpler, it is common to divide a design roughly between a *datapath* (the composition of functional units that operate on data and stateful elements) and the *control logic* (the signals that coordinate and route data through appropriate functional units at appropriate times). Designers typically concentrate first on instructions' computational action as they independently traverse these datapaths, leaving their exact orchestration of control for later.

Of course, the control-datapath divide is imperfect as the interactions between them and their relation to the ISA semantics can be subtle and difficult to reason about, particularly for a new or unfamiliar designer. In practice, data flows between the control and datapaths bidirectionally, thanks to matters like data-dependent control flow. Even worse, as the designer iteratively changes either the architecture (e.g., adding custom instructions) or the datapath (e.g., functional units or microarchitectural optimizations) they must reconsider all of these interdependencies which can easily cause pervasive and non-local changes to the control logic.

These problems are further exacerbated by the fact that testing is the most common means for assessing an SoC design's correctness (particularly in small, agile teams). While formal verification techniques are adopted in industry [10, 17, 20, 26, 28, 37, 43, 45, 53], they usually involve manually creating a separate, detailed microarchitectural model that must be updated in lockstep with the design. In contrast, our correct-by-construction approach requires only a lightweight microarchitectural model to handle optimizations such as pipelining in the form of a programmatic mapping from architectural state to microarchitectural components.

1.2 Technique Overview

Figure 1 gives a high-level overview of our technique: the hardware developer provides (1) the datapath in a Hardware Description Language (HDL); (2) the architectural specification that the hardware implements, taken from existing formalizations such as ILA [23–25, 48, 58, 59] or Sail [2]; and (3) the lightweight model connecting the datapath components to the specification, in the form of an abstraction function. Our method takes those inputs and uses program synthesis techniques adapted from the Programming Languages community to automatically create the necessary control logic, thus completing the hardware design (datapath + control logic) and ensuring correctness against the specification. Control logic synthesis enables hardware developers to freely modify and iterate in design of both the ISA/ILA and the datapath without getting caught up in the abstruse details of control. Further, it assures that the final implementation (not just a model of the design) is correct.

We focus our efforts on the design space exemplified by OpenTitan [30] (an open source silicon Root-of-Trust): embedded-class, small, but sophisticated designs for applications requiring bespoke, highly trusted cores and accelerators (e.g., for cryptography). We first target the core RISC-V ISA plus cryptography extensions and investigate both pipelined and non-pipelined microarchitectures. Further, to demonstrate our technique's generality, we generate control logic for a bespoke constant-time cryptography core and also for a cryptographic accelerator targeting the Advanced Encryption Standard (AES). Our major contributions are:

- We introduce a novel HDL Intermediate Representation (IR) named Oyster designed to be amenable to HDL-level program synthesis techniques (Section 3.1).
- We present an HDL program synthesis toolchain that takes a datapath and a specification for ISA/ILA semantics and automatically generates HDL code that implements the control logic (Section 3).
- We evaluate our toolchain on an embedded-class root-of-trust SoC design, encompassing a RISC-V core, constant-time cryptography core, and AES hardware accelerator, automatically extracting semantics from architectural specifications written in ILA [25], and generating correct-by-construction control logic code in the Python-based HDL PyRTL [12].

2 Background

Here we briefly review the concept of the control-datapath divide in hardware design and make clear specifically how we split control and datapath for the class of designs we consider. We broadly define the datapath as "the functional units

that define system operations"; and control logic as "the signals that coordinate and route data through the appropriate functional units at the appropriate times." While in practice the line between control and data can be blurry, to focus the scope of our control logic synthesis technique we describe two control structures commonly found in hardware designs: (1) instruction decoders, and (2) finite state machines (FSMs). For this purpose we present small but illustrative examples of hardware designs with each type of structure and show how we split each design into control logic and datapath.

2.1 Instruction-Level Abstraction

In this work, we use ILA for architectural specifications. We provide an overview of ILA here to aid in understanding our example use cases and direct the reader to the ILA paper [25] for a complete description. ILA provides a mechanism to functionally specify the hardware-software interface for both processors and accelerators. As the name implies, the core unit of computation is modeled as an "instruction." Instruction models capture the software-visible state updates made per unit of computation. Each instruction is specified with functions describing how it is fetched, decoded, and how it updates state. ILA authors specify each instruction's fetch, decode, and update functionality with the help of the ILA C++ library. In the case of a processor, the instruction model is the familiar concept of an ISA instruction specification. ILA abstracts this further by allowing the specification to rely on a wide-range of state-variables and inputs that are not present in general-purpose ISA specifications, but are widely used in MMIO-based accelerators. For example, one may want to trigger an instruction only when certain criteria in its state and input values are met. ILA also allows breaking down complex instruction into a hierarchy of smaller state updates, which further enables reasoning about and specifying complex device interfaces.

2.2 Instruction Decoder Example

A common control structure is instruction decoder-style control logic. This type of control receives an instruction or opcode as input and, based on decode logic from the specification, sets control signals to route data through the design to correctly execute the given operation. Consider the following ILA specification for an ALU machine:

```
ilang::Ila CreateAluIla() {
   auto ila = ilang::Ila("alu_ila");
   // args here are name and bitwidth
   auto op = ila.NewBvInput("op", 2);
   auto dest = ila.NewBvInput("dest", 2);
   auto src1 = ila.NewBvInput("src1", 2);
   auto src2 = ila.NewBvInput("src2", 2);
   // name, addr width, data width
   auto regs = ila.NewMemState("regs", 2, 8);

   auto rs1_val = ilang::Load(regs, src1);
   auto rs2_val = ilang::Load(regs, src2);
```

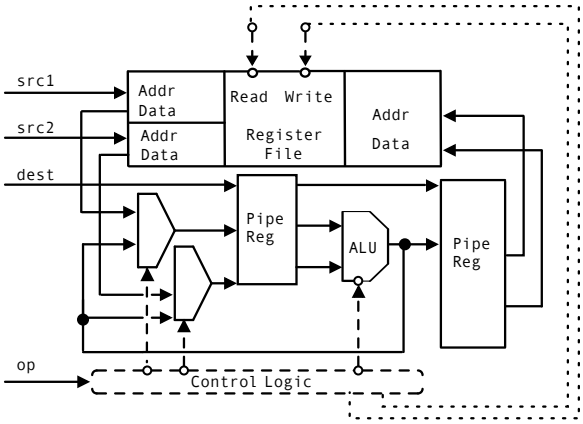


Figure 2. The datapath diagram for a three-stage implementation of the ALU machine. The decoded instruction is input to the control unit, which determines how to set control signals in the datapath.

```
   auto ADD = ila.NewInstr("ADD");
   {
      ADD.SetDecode(op == BvConst(1, 2));
      auto res = rs1_val + rs2_val;
      ADD.SetUpdate(regs, ilang::Store(regs, dest, res));
   }
   // similar for other ALU operations ...
   return ila; }
```

The ALU takes four inputs (`op`, `src1`, `src2`, and `dest`), which are previously decoded from some instruction. The architectural state is made of four registers stored in `regs`. `SetDecode` is an ILA method that specifies the conditional logic to determine whether an instruction is enabled to execute. The decode logic for an `ADD` operation in the ALU specification states that the `op` input must be equal to `01`. Similarly, `SetUpdate` describes the actual state update logic for the instruction. `SetUpdate` operates on one state element at a time; its first argument is the given state element, and the second argument is an expression describing how to update the state. For the `ADD` operation, the update procedure updates register file `regs` using the built-in ILA function `ilang::Store` which stores a new value in memory state.

Suppose the hardware designer wants to implement the ALU machine as a three-stage pipeline; then Figure 2 illustrates the design diagram, clearly labeling which part of the design corresponds to the control logic (with the bulk of the design being the datapath). The hardware designer has inserted two pipeline registers in the datapath, one after reading the `src1` and `src2` registers from the register file and one for storing the result of the ALU operation. The dashed boxes and arrows indicate where the designer would place the control logic to guide the data through the datapath and to select certain paths and functionality depending on the `op` input. Given the datapath portion of this diagram and a specification for the desired behavior, our technique

would automatically infer the correct control logic to fulfill the intended system behavior.

2.3 Finite State Machine Example

Another common class of control logic we consider are FSMs. Consider a simple accumulator machine with the following specification, expressed in ILA:

```
ilang::Ila CreateAccIla() {
    auto ila = ilang::Ila("acc_ila");

    // args are name and bitwidth
    auto reset = ila.NewBvInput("reset", 1);
    auto go = ila.NewBvInput("go", 1);
    auto stop = ila.NewBvInput("stop", 1);
    auto val = ila.NewBvInput("val", 2);
    auto acc = ila.NewBvState("acc", 8);
    auto state = ila.NewBvState("state", 2);

    auto reset_instr = ila.NewInstr("reset_instr");
    reset_instr.SetDecode(state == STOP && reset == 1);
    reset_instr.SetUpdate(acc, 0);
    reset_instr.SetUpdate(state, RESET)

    auto go_instr = ila.NewInstr("go_instr");
    go_instr.SetDecode((state == RESET && go == 1)
                       || (state == GO && stop == 0));
    go_instr.SetUpdate(acc, acc + val);
    reset_instr.SetUpdate(state, GO)

    auto stop_instr = ila.NewInstr("stop_instr");
    stop_instr.SetDecode(state == GO && stop == 1);
    stop_instr.SetUpdate(acc, acc);
    return ila; }
```

The specification describes a design with state `acc`, and three instructions that update the state based on the input signals (`reset`, `go`, and `stop`).

Suppose the hardware designer intends to implement an FSM (illustrated in Figure 3) that matches the accumulator specification. The FSM has three states for the accumulator updates associated with the `reset`, `go`, and `stop` inputs; it defines transitions between the three states with predicates derived from those input signals.

In this case, the datapath is simply the set of FSM states and the control logic is the transitions between them. Given the accumulator updates required for each state as well as the specification, our technique would automatically infer the necessary state encodings, transition conditions, and FSM transitions to fulfill the intended system behavior.

One implementation of the datapath, expressed in pseudocode, looks as follows:

```
state := ??
with state:
  ?? → acc := 0
  ?? → acc := acc + val
  ?? → acc := acc
out := acc
```

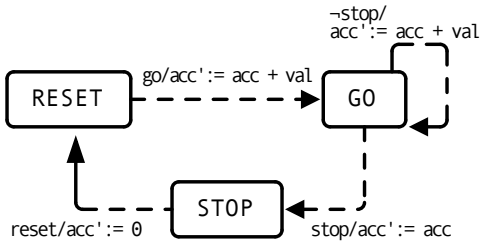


Figure 3. An FSM for the accumulator machine. Each state corresponds to how the machine updates the accumulator register. The input signals, `reset`, `go`, and `stop` predicate the state transitions.

The ?? in the code represents control points in the datapath. The `with` statement is syntactic sugar for conditional assignment predicated on the `state` argument; it describes the conditional updates in the datapath to the accumulator (here the designer implements `acc` as a register).

The key point is that our control logic synthesis technique only requires a datapath sketch (the solid-line components of Figures 2 and 3) and a specification. Control logic synthesis fills in the rest of the design—i.e., all of the dotted-lines in Figures 2 and 3 and their associated logic.

3 Control Logic Synthesis Technique

In this section we describe the high-level process for automatically generating correct-by-construction control logic. We show through our case studies in Section 4 how to specialize it to common architectures and hardware designs.

Figure 4 presents the overall work-flow of our toolchain. The inputs are (1) an architectural specification using ILA [25]; (2) an HDL sketch of the datapath[1]; and (3) a lightweight abstraction function mapping state in the datapath sketch to the architectural specification level. Our tool automatically extracts correctness conditions from the ILA specification plus the abstraction function; translates the datapath sketch into an intermediate representation called Oyster; and finally, via Rosette [50, 51], compiles the Oyster program and correctness conditions together into a symbolic form to generate the control logic. A human can then iterate on the design by modifying the specification and/or the datapath sketch (and updating the abstraction function accordingly) to get new designs.

The control logic synthesis process comprises three main components, detailed in the following subsections:

1. An intermediate representation (IR) tailored for program synthesis (Section 3.1) that captures essential datapath constructs as well as holes for control logic.
2. An abstraction function between the microarchitecture of the datapath sketch and architectural specification

[1]Specified using the PyRTL HDL [12], though other languages such as SystemVerilog could be supported.

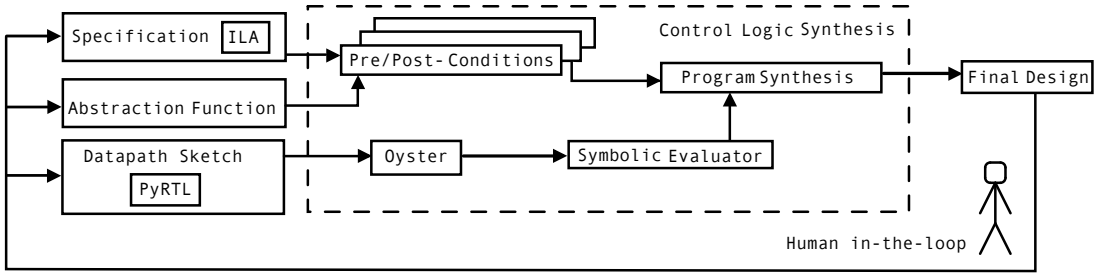


Figure 4. A diagram of the overall workflow.

$$
\begin{aligned}
\text{Design} &::= \textbf{decl}(decl^{+})\ stmt^{+} \\
decl \in \text{Declaration} &::= \textbf{input}\ name\ width \mid \textbf{output}\ name\ width \\
&\mid \textbf{register}\ name\ width \\
&\mid \textbf{memory}\ name\ width\ width \\
&\mid \textbf{hole}\ name\ width \\
stmt \in \text{Statement} &::= var := expr \mid \textbf{write}\ mem\ addr\ data\ enable \\
expr \in \text{Expression} &::= var \mid const \mid \neg\ expr \mid expr\ binop\ expr \\
&\mid \textbf{if}\ expr\ \textbf{then}\ expr\ \textbf{else}\ expr \\
&\mid \textbf{extract}\ expr\ high\ low \mid \textbf{read}\ mem\ addr \\
const \in \text{Constant} &::= width\ '\ value \\
binop \in \text{BinaryOps} &::= \wedge \mid \vee \mid \oplus \mid + \mid = \\
mem, name, var &\in \text{Identifier} \\
high, low, value, width &\in \text{Integer} \\
addr, data, cond, enable &\in \text{Expression}
\end{aligned}
$$

Figure 5. The grammar for Oyster. An "**extract**" expression extracts the bits from the bitvector value in *expr* between the bit-index positions *low* and *high*.

(Section 3.2), serving as a lightweight microarchitectural model which connects architectural state in the datapath to state in the specification.

3. And finally, a program synthesis technique that fills the holes in the datapath sketch using the pre- and postconditions from the formal architectural specification as constraints, generating correct-by-construction control logic (Section 3.3).

3.1 Oyster Intermediate Representation

Our representation must be amenable to automated reasoning and also easily constructed from conventional HDLs. We present an IR named Oyster that is high-level enough to easily translate to/from HDLs such as PyRTL and Verilog yet is also designed to accommodate program synthesis. Oyster embodies a subset of features from conventional HDLs in order to reduce the complexity of automated reasoning while still being complete enough to express non-trivial designs (as shown in our case studies). Oyster programs can be translated to SMT constraints expressed in the theories of bitvectors and uninterpreted functions which allows us to leverage standard program synthesis techniques and tools.

Figure 5 describes the grammar for Oyster. An Oyster program has two components: (1) a set of declarations for inputs, outputs, and stateful elements, and (2) a series of statements that describe the design, how data flows to its output ports, and how to update stateful elements. Oyster represents all variables as bitvectors, with the exception of memories. We model memories as a pair containing an uninterpreted function for reads and an association list to track writes. For space reasons we do not include here all of the operators supported by Oyster expressions, which include many common bitvector operations.

The **hole** construct in Figure 5 allows the hardware designer to specify where the control logic should be filled in for the datapath sketch. Our case studies (Section 4) detail how to use holes in datapath sketches for control logic generation in different design scenarios.

An Oyster interpreter is essentially a cycle-accurate simulator for synchronous hardware designs. Thus, we assume that all Oyster designs are synchronous with a single implicit clock—all writes to registers and memory take effect in the next cycle. We implement the Oyster interpreter in Rosette, a Racket-based framework for solver-aided programming. A key feature of Rosette is that by writing a concrete interpreter for a language, Rosette automatically lifts that interpreter to work with symbolic values, thus generating a symbolic interpreter "for free". This symbolic interpreter then leverages SMT solvers to solve satisfiability questions such as those that we will use to automatically generate control logic.

3.2 Abstraction Function

We use the abstraction function to map datapath components in Oyster code to architecture-level state in the specification. Because of the semantic gap between the architectural specification and the datapath implementation, it is not obvious to formal reasoning tools (such as a program synthesizer) what the connection is between, for example, architectural registers in the specification and a register file in the implementation. An abstraction function maps effectful behavior at the specification level (for example, reading and writing to state) into the semantics of the datapath components for a

particular microarchitecture. This section describes abstraction functions at a high level, and our case studies in Section 4 give more detailed examples.

For an abstraction function, the developer needs to specify for each architectural state element in the specification:

1. The corresponding name of the datapath component;
2. The type of the datapath component: one of either `input`, `output`, `register`, or `memory`;
3. A list of state effects indicating reads or writes from/to the datapath component, annotated with timing (that is, for each read/write, when the effect occurs in the datapath).

We specify abstraction functions (denoted α) for control logic synthesis with the following grammar:

$$
\begin{aligned}
\alpha &::= (\texttt{SpecID: \{name: DatapathID, type: } \textit{type}\texttt{, [}\textit{effect}^{+}\texttt{]\}})^{+} \\
&\quad\ \texttt{with cycles: TimeStep, } \textit{assume}^{*} \\
\textit{type} &::= \texttt{input} \mid \texttt{output} \mid \texttt{register} \mid \texttt{memory} \\
\textit{effect} &::= \texttt{read: TimeStep} \mid \texttt{write: TimeStep} \\
\textit{assume} &::= \texttt{[DatapathID: TimeStep]}^{+}
\end{aligned}
$$

For the three-stage ALU example in Section 2.2, the developer would provide the following abstraction function:

```
op:   {name: 'op',      type: input,  [read: 1]}
src1: {name: 'src1',    type: input,  [read: 1]}
src2: {name: 'src2',    type: input,  [read: 1]}
dest: {name: 'dest',    type: input,  [read: 1]}
regs: {name: 'regfile', type: memory, [read: 1, write: 3]}
with cycles: 3
```

TimeStep $i > 0$ is the state of the datapath after updating all registers and memories with the results of the $(i-1)^{\text{th}}$ step of evaluation (because Oyster evaluates designs synchronously). `op`, `src1`, `src2`, and `dest` are all inputs in the datapath and read at time 1. `regfile` is a memory that maps to the set of architectural registers (`regs`); the datapath reads it at time 1 and writes to it at time 3. The developer also specifies how many cycles to symbolically evaluate the sketch; in this case it is equal to the depth of the pipeline.

The `with` clause optionally accepts a list of signals in the datapath sketch which the symbolic evaluator assumes to be true. Datapath developers provide assumptions in situations where datapath hazards interfere with architectural instruction behavior. For example, a control hazard may flush the pipeline, "killing" the currently executing instruction. In this scenario, the program synthesizer cannot find a satisfying solution for the control logic because it can always find a case where the executing instruction is invalid. Our constant-time cryptography core requires this kind of assumption in its abstraction function (Section 4.2).

It is possible there is no one-to-one mapping between datapath components and architectural state. For instance, an ISA specification may not distinguish between instruction memory and data memory, modeling both together, whereas a datapath targeting that ISA may choose to implement the instruction and data memories as separate memory blocks. In that case, the developer adds multiple entries to the abstraction function, e.g., for the architectural memory example:

```
mem: {name: 'i_mem', type: memory, [read: 1]}
mem: {name: 'd_mem', type: memory, [read: 2, write: 3]}
```

In a multi-cycle design, the implementation may affect architectural state over time. Capturing these timing effects is crucial for designs with pipelining. The pipelined ALU scenario exemplifies the gap between the architectural specification and the datapath implementation. Abstraction functions bridge this gap to give our program synthesis technique enough semantic information about the relation between state in the architecture and datapath sketch to find satisfying solutions for the control logic.

3.3 Program Synthesis for Control Logic

In Figure 4, the process inside the dotted box illustrates the overall flow for the program synthesis step. Given a datapath sketch in an HDL, our technique first compiles the sketch into Oyster and then uses Rosette to translate the Oyster program into an SMT formula via symbolic evaluation using the theories of bitvectors and uninterpreted functions. For multi-cycle designs, the symbolic evaluator runs for the number of steps specified by the user. Then, for an architectural specification, our tool automatically extracts the pre- and postconditions and provides them as constraints to the program synthesizer. We formulate the program synthesis problem as follows:

$$
\begin{aligned}
&\exists e_0, \ \ldots, \ e_n, \forall s_0. \qquad (1)\\
&\quad \mathsf{Interpret}^k(s_0, \ \mathsf{Sketch}[h_0 := e_0, \ \ldots, \ h_n := e_n]) = (s_i)_{i=1}^{k}, \\
&\quad \bigwedge_j \ \mathsf{Pre}_j[s_{\text{spec}} := \alpha(s_0)] \longrightarrow \mathsf{Post}_j[s_{\text{spec}} := \alpha(s_1, \ \ldots, \ s_k)].
\end{aligned}
$$

For all holes $h_0, \ldots, h_n$ in the datapath sketch, the program synthesizer searches for Oyster expressions $e_0, \ldots, e_n$, filling the holes in Sketch with an implementation for the missing control logic. Equation (1) quantifies over the initial state s_0 because the synthesized expressions, $e_0, \ldots, e_n$, must hold for *any* initial state for every instruction in the specification.

$\mathsf{Interpret}^k$ evaluates the Oyster sketch given an initial state s_0 and returns a sequence of environments, $s_1, \ldots, s_k$, capturing the state of the design after each step. Then, for each instruction j, the formula asserts that the precondition implies the postcondition. The precondition Pre_j takes the initial state s_0 after passing through abstraction function α, and the postcondition Post_j takes the computed states $s_1, \ldots, s_k$ transformed according to α.

The abstraction function α acts as a substitution procedure for the pre- and postconditions between state in the specification and state in the datapath. To understand how

α fits into Equation (1), we separate the substitution procedure into two parts, for the precondition and postcondition, respectively.

$\text{Pre}_j[s_{\text{spec}} := \alpha(s_0)]$, where s_{spec} is a state element from the ISA specification; read as, "for the precondition for instruction j, substitute each occurrence of s_{spec} with $\alpha(s_0)$."

$\text{Post}_j[s_{\text{spec}} := \alpha(s_1, \ldots, s_k)]$, with $\alpha(s_1, \ldots, s_k) = s_t$, where t is a TimeStep and $0 < t \leq k$. The substitution procedure checks whether the state element is part of a read or write using t as specified in α. Further, for each *assume* in α, the procedure adds a conjunction that the given datapath signal in s_t is true, where t is the associated TimeStep.

In practice, for large j (the number of instructions in the specification), solving times dramatically increase, as our evaluation shows in Section 5. To overcome this scalability issue, we introduce an optimization for control logic synthesis that can be applied under an assumption about the design.

3.3.1 Optimization for Control Logic Synthesis.

To overcome the scalability limitation of the described program synthesis technique, we scale control logic synthesis by generating the control logic independently *per instruction* and then join the results together into a final overall form according to the preconditions in the specification. We introduce a property we call **instruction independence**, which must hold on the datapath sketch in order to apply this optimization. (In Section 3.3.2, we present an argument for the correctness of this optimization for the class of machines we target.)

Instruction Independence for Control Logic consists of two conditions that must hold on the given datapath sketch in order to solve for control logic independently:

1. **Mutually exclusive preconditions**: The preconditions, or antecedents, for the control logic for each instruction are disjoint.
2. **No feedback in control logic**: Signals output from the control logic cannot feed back into the control logic except for valid wires identified in α.

For the first condition, the decoder and FSM-style control we consider in our case studies necessarily satisfy this condition, as instructions are uniquely decoded. In this way, if the control logic for two instructions share the same preconditions then the control logic is identical.

For the second condition, we require no feedback so that it is possible for the control logic to be solved independently. The exception for valid signals identified by the abstraction function allows the optimization to handle designs that have determined dependencies between instructions. For example, the constant-time cryptography core (Section 4.2) exhibits control hazards when branches resolve and force a flush of the currently fetched instruction. The valid signal that determines the control hazard derives from the control signal controlling a branch. If there is a flush, the signal is false indicating a valid instruction is not executing, thus there is no control logic to dispatch.

```
function ⊔(holes, results):
  control := []
  for hole in holes:
    hole-defn := LogicGen(results[hole])
    control := control + Assign(hole, hole-defn)
  return control

function LogicGen(val→ops):
  val, opcodes := head(val→ops)
  cond := ⋁ opcodes
  return IfThenElse(cond, LogicGen(tail(val→ops)), val)
```

Figure 6. An algorithm for combining individual control logic synthesis results together into a complete implementation under the instruction independence assumption.

The intuition for why this speeds up control logic synthesis is that specifications with large number of instructions produce correspondingly large conjunctions of constraints—following Equation (1)—that SMT solvers struggle to solve. By making the independence assumption about instruction behavior, we break up the conjunction. Then, our control logic synthesis tool sends the individual synthesis queries to the SMT solver which are considerably smaller.

Given an instruction in the specification, the tool extracts the instruction's preconditions (for example, the instruction opcode as calculated by the fetch/decode logic and possibly other specified conditions, such as checking that the destination register is not the *zero* register). Next, the tool extracts the specified state change as a postcondition. The result is a formula that expresses the logical statement, "assuming a specific opcode (and any other relevant preconditions), what values for the existentially quantified variables result in the asserted state change being true?" A satisfying solution from the SMT solver is a concrete bitvector assigning a value to each control signal.

Control logic synthesis repeats this process for each instruction in the specification, resulting in a mapping of control signals to concrete bitvector values. The last step is to translate this mapping into complete Oyster expressions that incorporate the constraints from all instruction semantics, producing satisfying control logic to generate each control signal based on the opcodes and other relevant state.

We call this procedure the *control union*, which we abbreviate as $\sqcup$ and define in the algorithm in Figure 6. The procedure takes as input a list of `holes` in the datapath sketch and synthesis `results` from per-instruction control logic synthesis. The `results` variable maps for each hole the concrete bitvector value solved during control logic synthesis to an instruction (or list of instructions, if multiple instructions map to the same control signal value).

For example, consider the following map of synthesis results from a small RISC-style design with three instructions: ADD, LOAD, and JUMP; and three holes for control signals: write-register, read-memory, and jump.

```
results = {
  "write-register": {0b1: [ADD, LOAD], 0b0: [JUMP]},
  "read-memory": {0b1: [LOAD], 0b0: [ADD, JUMP]},
  "jump": {0b1: [JUMP], 0b0: [ADD, LOAD]}}
```

After running the ⊔ procedure as described in Figure 6 over the results map, we obtain the following Oyster code implementing the control logic:

```
pre-add := op = ADD
pre-load := op = LOAD
pre-jump := op = JUMP
write-register := if (pre-add ∨ pre-load) then 1
                  else if pre-jump then 0
read-memory := if pre-load then 1
               else if (pre-add ∨ pre-jump) then 0
jump := if pre-jump then 1
        else if (pre-add ∨ pre-load) then 0
```

For readability and reuse, the variables pre-add, pre-load, and pre-jump define the preconditions for each instruction in Oyster code based on the specification (derived automatically). While this example is smaller than the control logic in our case studies, the ⊔ procedure is flexible enough to handle signals of larger bitwidths and generate nested multiplexers (through nested **if-then-else** expressions).

3.3.2 Correctness Argument of Union Operation. Here we argue that joining individual generated control logic per-instruction under the ⊔ procedure produces a correct implementation of control logic with respect to the architectural specification. We present our argument starting from the "ideal" problem formulation presented in Equation (1). By solving the control logic for each instruction individually, we rearrange the formula to:

$$\exists c_0^j, \ \ldots, \ c_n^j, \forall s_0 \ . \tag{2}$$
$$\mathsf{Interpret}^k(s_0, \mathsf{Sketch}[h_0 := c_0^j, \ \ldots, \ h_n := c_n^j]) = (s_i)_{i=1}^k,$$
$$\mathsf{Pre}_j[s_{\text{spec}} := \alpha(s_0)] \longrightarrow \mathsf{Post}_j[s_{\text{spec}} := \alpha(s_1, \ \ldots, \ s_k)],$$

where $c_0^j, \ldots, c_n^j$ are Oyster constants.

The new formula says that for each instruction j in the specification, there exists Oyster constants $c_0^j, \ \ldots, \ c_n^j$ that satisfy the holes in the datapath sketch for that instruction. Applying the instruction independence assumption rearranges Equation (1) according to the two conditions (from Section 3.3.1). Because we assume mutually exclusive preconditions, we break the big conjunction of Pre_j and Post_j into a single implication for each instruction j. Assuming no feedback in the control logic, we separate the generated control into disjoint, per-instruction pieces such that $\bigsqcup_j c_0^j, \ \ldots, \ c_n^j \equiv e_0, \ \ldots, \ e_n$. That is, the individual synthesis results after the control union is *a* correct implementation of the control logic and *semantically equivalent* to the Oyster expressions, $e_0, \ \ldots, \ e_n$, generated from Equation (1). As the full formula is a conjunction of all predicates Pre_j and Post_j for each instruction j, we break each expression e_i filled for hole h_i into per-instruction pieces such that $\bigsqcup c_i^j \equiv e_i$.

Note that this correctness argument does not necessarily hold for designs that do not make this assumption or are outside of the class of machines we consider in this work. In Section 5.3, we discuss the limitations of the instruction-independence assumption and highlight future work to support more kinds of microarchitectures.

4 Case Studies

Here we cover three case studies: (1) an embedded-class RISC-V core, (2) a bespoke RISC-V core with a custom instruction set for constant-time cryptography, and (3) a cryptographic accelerator targeting AES. For each, we show how we specialize the core flow of our technique from Section 3.

4.1 Embedded-Class RISC-V Core

In this case study, we demonstrate how our control logic synthesis technique automatically generates the implementation of the instruction decoder-style control logic for different iterations of an embedded-class RISC-V core. We use an existing ILA specification for the RISC-V ISA [22]. The case study iterates on the design over two dimensions—modifying the architectural specification by adding ISA extensions, and modifying the datapath sketch by adding a pipeline.

We begin with the RISC-V 32-bit integer base instruction set (RV32I). This set totals 37 instructions, excluding the ecall and ebreak instructions because the target cores do not implement exceptions or interrupts. Then we add to the base ISA two extensions geared towards cryptography: Zbkb and Zbkc. The Zbkb extension is a set of 12 bit-manipulation instructions which are common in cryptographic applications: rotate (rol, ror, rori), logical-with-negate (andn, orn, xnor), byte reversal (rev8, rev.b), shuffle (zip, unzip), and word packing (pack, packh). Zbkc is an extension that adds two carryless multiply instructions: clmul and clmulh.

4.1.1 Single-Cycle Datapath. We start with a single-cycle datapath sketch, implementing the main components of the processor for executing each instruction class. To write the sketch, the developer identifies control points in the datapath and leaves these as holes, following the instruction-decoder pattern for control logic (described in Section 2.2). The following shows a portion of the datapath sketch in PyRTL, underlining the control signal variables for emphasis:

```
instruction = fetch(i_mem, pc)
opcode, funct3, funct7, imm = decode(instruction)
alu_imm   <<= ??(opcode, funct3, funct7)
alu_op    <<= ??(opcode, funct3, funct7)
reg_write <<= ??(opcode, funct3, funct7)
```

```
read_mem  <<= ??(opcode, funct3, funct7)
# ...
jump      <<= ??(opcode, funct3, funct7)

alu_in2 <<= mux(alu_imm, rs2_val, imm)
alu_out <<= alu(alu_op, rs1_val, alu_in2)

# Register file update
with conditional_assignment:
  with reg_write:
    with read_mem:
      rf[rd] |= d_mem[alu_out]
    with jump:
      rf[rd] |= pc + 4
    with otherwise:
      rf[rd] |= alu_out

# PC update
pc.next <<= mux(jump, pc + 4, target)
```

For each signal, the developer leaves its implementation as a hole (??) and passes as input the parts of the decoded instruction (opcode, `funct3`, and `funct7`).

Abstraction Function. The microarchitecture of the single-cycle core closely matches the architectural specification. There is no special timing and state effect information to consider; all reads and writes happen at time step 1:

```
pc:  {name: 'pc',    type: register, [read: 1, write: 1]}
GPR: {name: 'rf',    type: memory,   [read: 1, write: 1]}
mem: {name: 'd_mem', type: memory,   [read: 1, write: 1]}
mem: {name: 'i_mem', type: memory,   [read: 1]}
with cycles: 1
```

In the ILA specification for RISC-V, `GPR` stands for "general-purpose registers" and is modeled as a vector of registers. In the datapath sketch, `GPR` maps to a memory `rf` which is the register file. The datapath sketch also separates instruction and data memory as `i_mem` and `d_mem`, respectively.

Program Synthesis. As our results show in Section 5, the program synthesis tool is unable to generate control logic for the entire core ISA specification at once. To overcome this limitation, we take advantage of the RISC-V ISA instruction independence (i.e., the control logic for each instruction does not depend on any other instructions) and apply the optimization described in Section 3.3.1, generating control logic for each instruction independently and combining them together according to the algorithm in Figure 6.

Figure 7 shows an example of the generated control logic in PyRTL for a load word instruction (`LW`) from the RISC-V core. The `with` statements in PyRTL specify conditional assignments for wire variables in the design (with the conditional assignment operator denoted by `|=`). The code in Figure 7 executes control logic for a `LW` instruction because the conditional `with` expressions match on the corresponding opcode and 3-bit function code from the decoded instruction. For a load instruction, control logic synthesis determines

```
with op == LOAD:
    with funct3 == 0x2:
        mem_read |= 1
        mask_mode |= 2
        alu_op |= ADD
        alu_imm |= 1
        reg_write |= 1
        mem_write |= 0
        mem_sign_ext |= 0
        jump |= 0
        # Other control signals continue...
```

Figure 7. PyRTL code of the generated control logic for a load word instruction (`LW`) in the RV32I core. `LOAD` and `ADD` are mnemonics for numeric values and used here for readability. The `with` construct in PyRTL is syntactic sugar for nested multiplexers which we present here for readability.

that the following must occur in the datapath to satisfy the ISA instruction semantics for `LW`:

- Signal a memory read (`mem_read |= 1`) with the mask for a word-sized load (`mask_mode |= 2`).
- Perform an ALU operation with the operation signaled by `alu_op |= ADD`, and direct the immediate value from the decoded instruction into one of the ALU's inputs (`alu_imm |= 1`). These control signals coordinate the calculation of the address to be read from memory.
- Signal a write to the register file (`reg_write |= 1`).
- Set other control signals to *false* so that other state elements are not modified in a way that is inconsistent with the ISA instruction semantics (e.g., `mem_write`, and `jump` are all set to `0`).

4.1.2 Two-Stage Pipeline Datapath.

Next, we extend the design to an embedded-class core similar to Ibex [29]. We keep the ISA specification (including extensions) exactly the same as the single-cycle core, and only change the datapath sketch, adding two pipeline stages. The first pipeline stage is instruction fetch, decode and execute. The second pipeline stage is memory and write back.

Abstraction Function. Because we introduce pipelining into the datapath, we need to strengthen the abstraction function by adding timing information related to the microarchitecture. Specifically, we indicate for each corresponding architectural state element in the datapath which cycle (i.e., pipeline stage) that state is read or modified. Due to pipelining, without this timing information the generated pre- and postconditions will not have semantically valid values and the program synthesizer will fail to find a satisfying implementation for the control logic.

```
pc:  {name: 'pc',    type: register, [read: 1, write: 2]}
GPR: {name: 'rf',    type: memory,   [read: 1, write: 2]}
mem: {name: 'd_mem', type: memory,   [read: 2, write: 2]}
```

```
mem: {name: 'i_mem', type: memory,   [read: 1]}
with cycles: 2
```

The main changes to the abstraction function from the single-cycle core are the read and write time steps (underlined). In the two-stage pipeline, reads and writes to the register file occur in parallel (stage 1 and stage 2). All data memory operations occur in stage 2. By indicating a read at time step 1 (i.e., stage 1 of the pipeline), any writes that occurred in parallel in stage two will be available from the perspective of the symbolic evaluator.

Program Synthesis. With the new abstraction function, program synthesis follows the same as the single-cycle core, except the symbolic evaluator runs the sketch for 2 cycles.

4.2 Constant-Time Cryptography Core

As an additional case study, we modify the RISC-V design described above to create a bespoke core for constant-time cryptography. The motivation is that conditional branch instructions introduce variable instruction latency, which reveal timing side channels. We modify the RISC-V ISA specification to remove conditional branch instructions and all other instructions not necessary to execute SHA-256. We then extend it with a custom instruction for conditional move (`CMOV`). In cryptographic deployments, this bespoke instruction set ensures that the number of cycles executed on the core remains independent of the input length, making it resilient to timing side channel attacks.

Starting from the two-stage RISC-V core, we modify the datapath to add a third pipeline stage, remove all conditional branching logic, and extend the decode unit and ALU to support the new `CMOV` instruction. The three stages are: (1) instruction fetch, (2) instruction decode and execute, and (3) memory and write back.

Abstraction Function. The abstraction function for the three-stage pipeline is a modification of the two-stage abstraction function, following the read and write timing of the new datapath.

```
pc:  {name: 'pc',    type: register, [read: 1, write: 2]}
GPR: {name: 'rf',    type: memory,   [read: 2, write: 3]}
mem: {name: 'd_mem', type: memory,   [read: 3, write: 3]}
mem: {name: 'i_mem', type: memory,   [read: 1]}
with cycles: 3, [instruction_valid: 1]
```

The main change is the `instruction_valid` signal assumption in the datapath. The assumption states that this wire should be true at time step 1. This assumption resolves the case when there is a control hazard in the pipeline. An unconditional branch instruction such as `JAL` will resolve in stage 2, and force a flush of the fetched instruction in stage 1. Assuming `instruction_valid` is true will prevent the solver from trying to synthesize control for an instruction that is going to be flushed.

Program Synthesis. The program synthesis step requires no change from the previous case studies; symbolic evaluation runs for 3 cycles.

4.3 AES Hardware Accelerator

In this case study, we demonstrate how our control logic synthesis technique automatically generates the implementation of the FSM-style control logic for an AES-128 hardware accelerator. We take an existing ILA specification for AES-128 encryption [22], and compile it to constraints for our control logic synthesis tool as described in Section 5.1. While the AES specification does not have typical "instructions" as a general-purpose ISA does, it splits the main computation units for AES encryption into three distinct states: "first", "intermediate", and "final". The ILA models each state as a separate ILA instruction, which the device can exist in for one or more "rounds." As an example, the following code is part of the ILA specification for the intermediate round AES computation (where the functions `CipherUpdate_MidRound` and `KeyUpdate_MidRound` compute the update for their respective state elements):

```
auto instr = model.NewInstr("IntermediateRound");
instr.SetDecode((round > 0) & (round < 9));
instr.SetUpdate(round, round + 1);
instr.SetUpdate(ciphertext,
  CipherUpdate_MidRound(ciphertext, round, round_key));
instr.SetUpdate(round_key,
  KeyUpdate_MidRound(round_key, round));
```

The two key components are `SetDecode` and `SetUpdate`. The `SetDecode` function specifies the preconditions for the device existing in that state. The `SetUpdate` function specifies the postconditions, that is, the associated updates for state elements `ciphertext`, `round_key`, and `round`.

For the datapath sketch we implement a multi-cycle datapath for the AES accelerator following an FSM-style control structure. The datapath computes one round of encryption at a time, keeping track of the rounds between cycles. We leave holes for computing the state transition logic as well as holes for the states themselves (in the `with` expressions).

```
state <<= ??
with conditional_assignment:
  with state == ??:
    # Computation for first round ...
  with state == ??:
    # Computation for intermediate rounds ...
  with state == ??:
    # Computation for final round ...
```

The datapath describes *how* the hardware computes with and modifies the architecture-level state such as `round_key` and `ciphertext`, but it does not describe what the states are or how the states transition between each other.

Abstraction Function. The abstraction function bridges the gap between the AES specification and the datapath sketch

by explicitly mapping the inputs and registers in the datapath sketch to the architectural elements in the specification. This design is not pipelined so we do not capture any timing-related information in the datapath.

```
key_in:     {name: 'key_in',     type: input, [read: 1]}
plaintext:  {name: 'plaintext',  type: input, [read: 1]}
round:      {name: 'round', type: regster,
             [read: 1,write: 1]}
round_key:  {name: 'round_key',  type: regster,
             [read: 1, write: 1]}
ciphertext: {name: 'ciphertext', type: regster,
             [read: 1, write: 1]}
with cycles: 1
```

Program Synthesis. The result of control logic synthesis for AES fills in state condition and state transition logic for the FSM, and generates the state encodings.

```
state <<= mux(round == 0,
  mux((round > 1) & (round <= 9), 0b10, 0b01), 0b00)
with conditional_assignment:
  with state == 0b00:
    # Computation for first round ...
  with state == 0b01:
    # Computation for intermediate rounds ...
  with state == 0b10:
    # Computation for final round ...
```

We note that we did not make any changes to the core control logic synthesis technique to support the AES hardware accelerator. The developer follows the same procedure, providing a datapath sketch and ILA specification. This case study demonstrates the generality of our technique and shows promise for applying control logic synthesis to the development of hardware accelerators in other domains such as image processing, AI, and machine learning, as well as other aspects of SoC design such as protocol implementations (for example, cache coherence protocols) [31].

5 Evaluation

In this section, we present the results of control logic synthesis over all designs from our case studies. We ran all experiments on a workstation running Ubuntu 20.04 GNU/Linux (kernel version 5.15) with an Intel Xeon Gold 6226R 3.9 GHz processor and 96 GB RAM.

5.1 Implementation

Our implementation spans several languages for each major component in the tool flow. Overall, the Racket code implementing the Oyster interpreter and program synthesis procedures are just over 1,000 source lines of code (SLOC). Translating PyRTL to Oyster is about 150 SLOC of Python. Our implementation also includes adding support for holes in the PyRTL language. With the exception of the bespoke cryptography core, we use unmodified, off-the-shelf ILA specifications for all of the case studies.

$$
\begin{aligned}
\text{DecodeExpr} &::= \textbf{SetDecode}(\mathit{expr}) \\
\text{UpdateExpr} &::= \textbf{SetUpdate}(\mathit{state\_var}, \mathit{expr}) \\
\text{expr} &::= \mathit{sym} \mid \mathit{expr}\ \mathit{binop}\ \mathit{expr} \mid !\mathit{expr} \\
&\quad \mid \textbf{Extract}(\mathit{expr}, \mathit{int}, \mathit{int}) \\
&\quad \mid \textbf{Load}(\mathit{expr}, \mathit{expr}) \mid \textbf{Load}(\mathit{expr}) \\
&\quad \mid \textbf{Concat}(\mathit{expr}, \mathit{expr}) \\
&\quad \mid \textbf{Ite}(\mathit{bool\_expr}, \mathit{expr}, \mathit{expr}) \\
&\quad \mid \textbf{ZExt}(\mathit{expr}, \mathit{int}) \\
\text{binop} &::= + \mid == \mid \& \mid \ldots \\
\text{sym} &::= \mathit{int} \mid \mathit{state\_var} \mid \mathit{input\_var} \\
T[[\mathit{DecodeExpr}]] &::= (\textbf{assume}\ T[[\mathit{expr}]]) \\
T[[\mathit{UpdateExpr}]] &::= (\textbf{assert}\ (\textbf{bveq}\ T[[\mathit{expr}]] \\
&\qquad (\textbf{post}\ (\alpha\ \mathit{state\_var})))) \\
T[[\mathit{expr}\ \mathit{binop}\ \mathit{expr}]] &::= (T[[\mathit{binop}]]\ T[[\mathit{expr}]]\ T[[\mathit{expr}]]) \\
T[[!\mathit{expr}]] &::= (\textbf{bvnot}\ T[[\mathit{expr}]]) \\
T[[\textbf{Extract}(\mathit{expr}, \mathit{int}, \mathit{int})]] &::= (\textbf{extract}\ T[[\mathit{expr}]]\ \mathit{int}\ \mathit{int}) \\
T[[\textbf{Load}(\mathit{expr}, \mathit{expr})]] &::= (\textbf{read-mem}\ (\textbf{pre}\ (\alpha\ T[[\mathit{expr}]])) \\
&\qquad (\textbf{bv}\ T[[\mathit{expr}]]\ \mathit{addr\_width})) \\
T[[\textbf{Load}(\mathit{expr})]] &::= (\textbf{pre}\ (\alpha\ T[[\mathit{expr}]])) \\
T[[\textbf{Concat}(\mathit{expr}, \mathit{expr})]] &::= (\textbf{concat}\ T[[\mathit{expr}]]\ T[[\mathit{expr}]]) \\
T[[\textbf{Ite}(\mathit{expr}, \mathit{expr}, \mathit{expr})]] &::= (\textbf{if}\ T[[\mathit{expr}]]\ T[[\mathit{expr}]]\ T[[\mathit{expr}]]) \\
T[[\textbf{ZExt}(\mathit{expr}, \mathit{int})]] &::= (\textbf{zero-extend}\ T[[\mathit{expr}]] \\
&\qquad (\textbf{bitvector}\ \mathit{int})) \\
T[[+]] ::= \textbf{bvadd} \quad T[[==]] &::= \textbf{bveq} \\
T[[\&]] ::= \textbf{bvand} \quad \ldots &
\end{aligned}
$$

Figure 8. The grammar for ILA decode and update expressions with their Rosette transformation rules. $T[[\,]]$ defines the translation function. **pre** is the initial state environment. **post** is the sequence of environments produced after symbolic evaluation (dependent on the number of steps). α is the abstraction function.

The ILA to Rosette compiler is 550 SLOC of C++. Figure 8 presents a grammar that defines the compilation process. Bold names in the grammar correspond to ILA intrinsic functions that model common bit manipulation and comparison operations whereas bold names in the translation function correspond to Rosette functions. The `DecodeExpr` and `UpdateExpr` are the top-level rules that are translated into `assume` and `assert` statements in Rosette, respectively. An ILA-modeled instruction is valid if the `expr` argument is true. A modeled instruction may also update one or more state variables with a call to `SetUpdate` and passing the variable as well as the new value. Translation proceeds by syntactically rewriting the rest of the expression tree.

ILA specifications for FSM-based designs model one state for each instruction. The conditions for decoding the state are the architectural preconditions for the device existing in or entering into that state and the state update is the change expected to be made after that state finishes execution. Because the modeling for FSM-based designs is analogous to

Design	Variant	Sketch Size	Control Logic Synthesis Time (s)
AES Accelerator	-	250	253.8
AES Accelerator †	-	250	315.9
Single-Cycle Core	RV32I	358	6.6
	RV32I + Zbkb	531	10.2
	RV32I + Zbkc	668	12.8
	RV32I †	358	Timeout
Two-Stage Core	RV32I	393	96.3
	RV32I + Zbkb	566	75.4
	RV32I + Zbkc	703	131.7
Crypto Core	CMOV ISA	426	6.7

Table 1. Control logic synthesis results over all case studies: the AES hardware accelerator, two variants of an embedded-class RISC-V core, and the constant-time cryptography core. The "Sketch Size" column gives the size of the datapath sketch in lines of Oyster code. Control logic synthesis times are given in seconds. †: Indicates the experiment synthesizes control logic without the instruction-independence optimization. All other experiments use the per-instruction control logic synthesis strategy with the union operator (as described in Section 3.3.1).

traditional CPU instructions, our compiler is able to generate constraints without extra information about the type of control it is generating.

As discussed in Section 4.3, we demonstrate our technique on FSM-based control for an AES accelerator. A unique detail of AES, which separates its ILA from a standard processor's, is that it relies on various lookup tables for computation. These are modeled in ILA as MemConst objects representing read-only memory. Instead of modeling these with uninterpreted functions as with other state elements, the ILA-to-Rosette compiler generates Racket-level immutable vectors.

5.2 Results

Table 1 presents our experimental results. In most cases, control logic synthesis takes minutes. We include one experiment where we attempt control logic synthesis over the entire RV32I RISC-V ISA at once, to show the effectiveness of the instruction-independence optimization. We set a 3 hour timeout, which this experiment exceeded, while the experiment on the same design with instruction-independence optimization took only 6.6 seconds. We also compare times for the AES accelerator with and without our per-instruction optimization. While AES does not time out without the optimization, the per-instruction version finishes faster.

Table 2 compares the size of the processor configurations with generated control logic to a hand-written reference. For space, we only show the comparison for the single-cycle core, as the other designs follow the same pattern. The size of the generated HDL code for the control logic primarily depends on the number of instructions in the ISA and the number of control signals. Overall, its size is larger than the handwritten implementation. However, after hardware synthesis, the processors with generated control logic use about 10% more gates than the reference. We also ran the generated control logic through a logic optimizing pass in Yosys [54] which results in about 3% more gates total.

For the constant-time cryptography core, we compile a SHA-256 program to our bespoke ISA without conditional branches and using the new CMOV instruction. We simulate this on the core with test cases varying input string length from 4 to 32. The simulation results yield the same number of CPU cycles independent of input length, showing our bespoke core is constant-time. Further, we compared these simulations of the cryptography core with automatically generated control logic against a hand-written reference. The results show both cores spend the same number of cycles to produce the same result.

5.3 Limitations and Future Work

Given the time and effort required to implement and verify a processor in an iterative agile design process, it is notable that we generate correct control logic in minutes. Here we discuss some limitations and directions for future work.

One limitation comes from the size and complexity of constraints sent to the SMT solver for program synthesis. For large designs evaluated over multiple time steps, solving times increase dramatically. Exploding solving times is a known problem in program synthesis and research has studied how to diagnose and fix performance issues related to symbolic evaluation [7, 41], more recently targeted for hardware designs [44]. Given the relatively little attention to HDLs in program synthesis there is space for these tools to better accommodate HDLs.

There are many interesting microarchitectural features to explore with our control logic synthesis technique; we group these features into two categories:

1. Based on the limitations brought by the instruction-independence assumption, there are microarchitectural features our technique currently cannot handle, like out-of-order execution. We leave this to future work on how to lift, generalize, and scale our technique without the assumption.

2. There are designs with features that are worth exploring and which are *not* blocked by the instruction-independence assumption. By adding more invariants through the abstraction function, our technique can encode more microarchitectural dependencies such as branch predictors, stalls and exceptions, and resilience to other side-channels.

At present, our tool only generates *correct* control logic. The HDL code generated for our RISC-V processor—and the synthesized circuit—is larger than a handwritten reference. There is room in our technique to generate HDL code that is correct *and* also optimal with respect to some objective function (size of HDL code, area of circuit, power, etc.). Similarly, the generated code is not optimized for readability as our technique essentially produces a netlist. Recent techniques such as hardware decompilation [47] lift low-level circuits up to high-level HDL code, producing a more readable artifact.

Design	Variant	HDL Control Logic (Reference)	HDL Control Logic (Generated)	Netlist Size (Reference)	Netlist Size (Generated)	Netlist Size (Optimized)
Single-Cycle Core	RV32I	177	627	41K	46K	42K
	RV32I + Zbkb	214	797	60K	66K	62K
	RV32I + Zbkc	192	643	68K	73K	70K

Table 2. Size of designs with generated control logic compared to a hand-written reference implementation. HDL Control Logic records source lines of code in PyRTL of the generated control logic versus the reference. Netlist Size measures the number of gates in the circuit synthesized from the completed designs using the PyRTL compiler. Netlist Size (Optimized) records the number of gates in the design after running the generated control logic through a logic optimizer (using Yosys [54]).

Improving feedback for developer experience is further future work. For instance, if the datapath sketch is incorrect with respect to the ILA, the tool will fail to find a satisfying solution for the control logic. Future work can extend the tool to indicate which part of the datapath is incorrect.

6 Related work

6.1 Symbolic Evaluation for Hardware Design

Existing work like SketchiLog [5, 6] generates Verilog code given a sketch and a reference implementation, but is limited to combinational circuits. VeriSketch [1] is another sketch-based Verilog code generation tool that leverages CEGIS and information flow tracking to synthesize combinational and sequential circuits that adhere to information flow security properties. Our work instead uses program synthesis goals guided by specifications independent of the HDL code. Other work symbolically evaluates processors for verification or other analyses [9, 52] like tailoring a processor to a specific application by reducing area and power through eliminating unused gates via symbolic analysis [11].

Knox is a framework that uses Rosette to symbolically evaluate circuits in order to formally verify hardware security modules [4] building off of previous work that translates Verilog designs into a shallow embedding in Rosette [3]. Similarly, Pensieve uses Rosette for modeling microarchitectures to find speculative execution vulnerabilities [55].

6.2 Hardware Languages and Design Tools

PDL (*Pipeline Description Language*) [57] is an HDL that raises the abstraction level for implementing pipelined processors by letting developers write "one instruction at a time" semantics for their design and outputs a Bluespec System Verilog (BSV) pipeline [42]. PDL intersects with our work as it tackles the problem of designing and reasoning about pipelined processors from a language perspective. While PDL relies on the BSV compiler for generating control logic, our generated control logic is proven correct with respect to a formal ISA specification.

Xtensa is an extensible processor design tool [19] which enables developers to "drop-in" components into a processor pipeline and automates connecting the components together. Part of Xtensa is the TIE language, which allows specifying semantics of single-cycle and multi-cycle register-to-register instructions [49]. Our technique instead allows for arbitrary HDL code from a developer, not only drop-in components. Additionally, our tool formally verifies the generated control logic against an existing ISA specification.

6.3 Formal Verification

Our work intersects with research using automated theorem provers in verification of microarchitecture models for processors similar to those considered in this paper (in-order execution with shallow pipelines) [10], models with deeper pipelines [53], and more complex microarchitectures [20, 26, 45]. Much of this work relies on an abstraction function which "flushes" the implementation whereas in other work, compositional, or refinement-based, proof techniques obviate the need for flushing [26, 37, 39, 59]. Our work differs by starting at the HDL code level rather than a microarchitecture-level model, and builds on prior microarchitecture verification by automatically *generating* correct-by-construction control logic for an incomplete hardware implementation.

Broadly, work in formal hardware verification and model checking [17, 28, 43], intersects with ours as well. Well-established tools such as ABC [8] use SAT solving for logic simulation, synthesis, and verification tasks [60]. Advances in SMT solvers found their way into model checkers for hardware such as EBMC [15, 27, 40]. Pono [36]—successor of CoSA [38]—and AVR [18] are model checkers that work over transition systems and often run multiple model checking algorithms in parallel. Further, past work builds formal verification into existing HDL toolchains [14, 56].

The *Check* suite use an interactive theorem prover (Coq) to prove a microarchitecture's handwritten MCM is correct with respect to a suite of litmus tests [32–35] or extract a model from RTL code for MCM verification [21].

7 Acknowledgments

We thank our shepherd, Andrea Lattuada, and the anonymous reviewers for their suggestions to improve the formalisms in this paper. We also thank Bo-Yuan Huang for help with ILA.

A Artifact Appendix

A.1 Abstract

The provided artifact includes the full implementation of our control logic synthesis toolchain, the full source and specifications for the evaluated benchmarks, scripts to recreate the results reported in Table 1, and scripts to set up a Docker environment with all of the required software dependencies.

A.2 Artifact check-list (meta-information)

- **Run-time environment:**
  - Python 3.11
  - PyRTL at bfe7141
  - Racket 8.7
  - Rosette 4.1
  - Boolector 3.2.3
  - CVC4 1.8
- **Experiments:** Control logic synthesis using a program synthesis toolchain over case studies covering an embedded-class SoC; compilation from hardware description languages and formal specifications into solver-aided languages.
- **How much disk space required?:** 2 GB.
- **How much time is needed to prepare workflow?:** 1 hour.
- **How much time is needed to complete experiments?:** 1–2 hours.
- **Publicly available?:** Yes, https://github.com/UCSBarchlab/owl
- **Code licenses:** BSD 3-Clause
- **Archived:** https://doi.org/10.5281/zenodo.11506063

A.3 How to access

We recommend cloning the git repository for the latest code: https://github.com/UCSBarchlab/owl.

A.4 Installation

Users can follow the README file in the artifact to install the required software dependencies and build the toolchain. Optionally, users can build and run the provided Docker environment.

A.5 Evaluation and expected results

The artifact provides documentation and scripts to run the different parts of the toolchain for compiling PyRTL to the Oyster IR and compiling ILA specifications to set of pre- and postconditions used by the solver. Further, the artifact provides scripts to recreate the results reported in Table 1.

A.6 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

[1] Armaiti Ardeshiricham, Yoshiki Takashima, Sicun Gao, and Ryan Kastner. Verisketch: Synthesizing secure hardware designs with timing-sensitive information flow properties. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1623–1638, New York, NY, USA, 2019. Association for Computing Machinery.

[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. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 71.

[3] Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Notary: A device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, page 97–113, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying hardware security modules with Information-Preserving refinement. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 503–519, Carlsbad, CA, July 2022. USENIX Association.

[5] A. Becker, D. Novo, and P. Ienne. SKETCHILOG: Sketching combinational circuits. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, 2014.

[6] Andrew Becker, David Novo, and Paolo Ienne. Automated circuit elaboration from incomplete architectural descriptions. In *2013 Asilomar Conference on Signals, Systems and Computers*, pages 391–395. IEEE, 2013.

[7] James Bornholt and Emina Torlak. Finding code that explodes under symbolic evaluation. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[8] Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[9] Niklas Bruns, Vladimir Herdt, and Rolf Drechsler. Processor verification using symbolic execution: A RISC-V case-study. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2023.

[10] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *Proceedings of the 6th International Conference on Computer Aided Verification*, CAV '94, page 68–80, Berlin, Heidelberg, 1994. Springer-Verlag.

[11] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. Bespoke processors for applications with ultra-low area and power constraints. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 41–54, New York, NY, USA, 2017. Association for Computing Machinery.

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

[13] Ang Cui. The next frontier in cyberwar: Embedded devices. *GCN*, 2022.

[14] Andrew Dobis, Tjark Petersen, Hans Jakob Damsgaard, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. Chiselverify: An open-source hardware verification library for chisel and scala. In *2021 IEEE Nordic Circuits and Systems Conference (NorCAS)*, pages 1–7, 2021.

[15] Vijay D'Silva, Mitra Purandare, and Daniel Kroening. Approximation refinement for interpolation-based model checking. In *Verification,*

Model Checking, and Abstract Interpretation (VMCAI), volume 4905 of *Lecture Notes in Computer Science*, pages 68–82. Springer, 2008.

[16] Harry Foster. Wilson research group functional verification study. https://blogs.sw.siemens.com/verificationhorizons/2020/10/27/prologue-the-2020-wilson-research-group-functional-verification-study/, 2020.

[17] Dapeng Gao and Tom Melham. End-to-end formal verification of a risc-v processor extended with capability pointers. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 24–33, 2021.

[18] Aman Goel and Karem Sakallah. Avr: Abstractly verifying reachability. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–422, Cham, 2020. Springer International Publishing.

[19] R.E. Gonzalez. Xtensa: a configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[20] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 521–537, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[21] Yao Hsiao, Dominic P. Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. Synthesizing formal models of hardware from RTL for efficient verification of memory model implementations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 679–694, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Bo-Yuan Huang. Imdb-archive. https://github.com/PrincetonUniversity/IMDb-Archive, 2018.

[23] Bo-Yuan Huang, Sayak Ray, Aarti Gupta, Jason M. Fung, and Sharad Malik. Formal Security Verification of Concurrent Firmware in SoCs using Instruction-Level Abstraction for Hardware. In *Proc. Design Automation Conference*, page 91, 2018.

[24] Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik. ILAng: A Modeling and Verification Platform for SoCs using Instruction-Level Abstractions. In *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–357, 2019.

[25] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. Instruction-level abstraction (ila): A uniform specification for system-on-chip (soc) verification. *ACM Trans. Des. Autom. Electron. Syst.*, 24(1), dec 2018.

[26] Ranjit Jhala and Kenneth L. McMillan. Microarchitecture verification by compositional model checking. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, page 396–410, Berlin, Heidelberg, 2001. Springer-Verlag.

[27] Daniel Kroening. Computing over-approximations with bounded model checking. In *Proceedings of the Third International Workshop on Bounded Model Checking (BMC 2005)*, volume 144, pages 79–92, January 2006.

[28] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. Verified compilation on a verified processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1041–1053, New York, NY, USA, 2019. Association for Computing Machinery.

[29] lowRISC. Ibex: An embedded 32 bit RISC-V CPU core. https://ibex-core.readthedocs.io/en/latest/, 2018.

[30] lowRISC. Opentitan. https://docs.opentitan.org/, 2022.

[31] Huaixi Lu, Yue Xing, Aarti Gupta, and Sharad Malik. Soc protocol implementation verification using instruction-level abstraction specifications. *ACM Trans. Des. Autom. Electron. Syst.*, 28(6), oct 2023.

[32] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Verifying correct microarchitectural enforcement of memory consistency models. *IEEE Micro*, 35(3):72–82, 2015.

[33] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. COATCheck: Verifying memory ordering at the hardware-OS interface. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 233–247, New York, NY, USA, 2016. Association for Computing Machinery.

[34] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. RTLcheck: Verifying the memory consistency of RTL designs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 463–476, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. CCICheck: Using μhb graphs to verify the coherence-consistency interface. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 26–37, 2015.

[36] Makai Mann, Ahmed Irfan, Florian Lonsing, Yahan Yang, Hongce Zhang, Kristopher Brown, Aarti Gupta, and Clark Barrett. Pono: A flexible and extensible smt-based model checker. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 461–474, Cham, 2021. Springer International Publishing.

[37] Panagiotis Manolios and Sudarshan K. Srinivasan. A refinement-based compositional reasoning framework for pipelined machine verification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(4):353–364, 2008.

[38] Cristian Mattarei, Makai Mann, Clark Barrett, Ross G. Daly, Dillon Huff, and Pat Hanrahan. CoSA: Integrated verification for agile hardware design. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–5, 2018.

[39] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1–3):279–309, may 2000.

[40] Rajdeep Mukherjee, Daniel Kroening, and Tom Melham. Hardware verification using software analyzers. In *IEEE Computer Society Annual Symposium on VLSI*, pages 7–12. IEEE, 2015.

[41] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 225–242, New York, NY, USA, 2019. Association for Computing Machinery.

[42] R. 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, 2004.

[43] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with isa-formal. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 42–58, Cham, 2016. Springer International Publishing.

[44] Kaki Ryan and Cynthia Sturton. Sylvia: Countering the path explosion problem in the symbolic execution of hardware designs. In *2023 Formal Methods in Computer Aided Design (FMCAD)*, pages 110–121, 2023.

[45] Jun Sawada and Warren A Hunt. Verification of FM9801: An out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability. *Formal Methods in System Design*, 20(2):187–222, 2002.

[46] Azad Shademan, Ryan S. Decker, Justin D. Opfermann, Simon Leonard, Axel Krieger, and Peter C. W. Kim. Supervised autonomous robotic soft tissue surgery. *Science Translational Medicine*, 8(337):337ra64–337ra64, 2016.

[47] Zachary D. Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. Loop rerolling for hardware decompilation. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[48] Pramod Subramanyan, Yakir Vizel, Sayak Ray, and Sharad Malik. Template-based Synthesis of Instruction-Level Abstractions for SoC

Verification. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design*, pages 160–167, 2017.

[49] Cadence Design Systems. TIE language—the fast path to high-performance embedded SoC processing. Technical report, Cadence Design Systems, Inc., San Jose, CA, USA, 2016.

[50] 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 and Software*, Onward! 2013, page 135–152, New York, NY, USA, 2013. Association for Computing Machinery.

[51] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. *SIGPLAN Not.*, 49(6):530–541, June 2014.

[52] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. Solver-aided constant-time hardware verification. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 429–444, New York, NY, USA, 2021. Association for Computing Machinery.

[53] Miroslav N. Velev and Randal E. Bryant. Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, page 112–117, New York, NY, USA, 2000. Association for Computing Machinery.

[54] Claire Wolf. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/, 2022.

[55] Yuheng Yang, Thomas Bourgeat, Stella Lau, and Mengjia Yan. Pensieve: Microarchitectural modeling for security evaluation. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[56] YosysHQ. Symbiyosys. https://github.com/YosysHQ/sby, 2022.

[57] Drew Zagieboylo, Charles Sherk, Edward Suh, and Andrew Myers. PDL a high-level hardware design language for pipelined processors. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2022, New York, NY, USA, 6 2022. Association for Computing Machinery.

[58] Hongce Zhang, Caroline Trippel, Yatin A Manerkar, Aarti Gupta, Margaret Martonosi, and Sharad Malik. ILA-MCM: Integrating Memory Consistency Models with Instruction-Level Abstractions for Heterogeneous System-on-Chip Verification. In *Proc. Conf. Formal Methods in Computer-Aided Design*, pages 1–10, 2018.

[59] Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, and Sharad Malik. Synthesizing environment invariants for modular hardware verification. In Dirk Beyer and Damien Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 202–225, Cham, 2020. Springer International Publishing.

[60] Jin S Zhang, Subarna Sinha, Alan Mishchenko, Robert K Brayton, and Malgorzata Chrzanowska-Jeske. Simulation and satisfiability in logic synthesis. *Computing*, 7:14, 2005.