UNIVERSITY of CALIFORNIA
Santa Barbara

Automated Reasoning for Agile and Robust Chip Design

A Dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Zachary David Sisco

Committee in Charge:

Professor Ben Hardekopf, Co-Chair
Professor Jonathan Balkind, Co-Chair
Professor Timothy Sherwood
Professor Zachary Tatlock, University of Washington

June 2025

The Dissertation of Zachary David Sisco is approved.

Professor Timothy Sherwood

Professor Zachary Tatlock, University of Washington

Professor Jonathan Balkind, Committee Co-Chair

Professor Ben Hardekopf, Committee Co-Chair

June 2025

Automated Reasoning for Agile and Robust Chip Design

Dedication

To my parents and brothers.

To Linda and Eric for the Thanksgivings away from home.

To Jeremy for the good jams.

To Drew, we'll make that game someday.

To Pingyuan for your love and support.

Acknowledgements

I cannot enumerate all of the people—mentors, mentees, colleagues, and more—who helped me in carrying out the research in this dissertation. I could not have done it alone. There are a few I would like to specially thank:

Ben Hardekopf For guiding me through this research and helping me hone my writing and teaching skills.

Jonathan Balkind For jumping in with us on this wild idea called "hardware decompilation," and staying on! Over the years you have been an invaluable mentor in research and navigating my academic career, nudging me towards many opportunities to engage with folks in the PL–Hardware community.

Tim Sherwood For your enthusiasm and openness to applying esoteric ideas to computer architecture, and for giving me an opportunity to teach about those ideas.

Adam Bryant For introducing me to computer science research, and giving me the chance to work with some smart folks at Galois.

Harlan Kringen For countless hours of shop talk and for being a good friend. Your deep wells of PL theory knowledge are inexhaustible and yet somehow all trace back to the 1970s. You have shared so much excellent music which has become the soundtrack to my PhD.

Andrew Alex For jumping onto the OWL project and braving the jungle of parentheses that is its Racket codebase.

Thanawat Techaumnuaiwit For trying (and nearly succeeding) to make me a better developer by showing me better tools (though I'm still not using Nix).

Gus Smith For inviting me to give a talk at the UW PLSE lab that eventually kicked off a collaboration over our shared research areas.

The UCSB PL Lab To the senior members: Mehmet Emre, Michael Christensen, and Lawton Nichols for welcoming me into the lab. And to the current members: Harlan Kringen, Jingtao Xia, Hitomi Nakayama, Peter Boyland, and Sarah Canto. I've enjoyed sharing the lab with you all.

The UCSB Arch Lab For helping me understand what a computer is and how we build them. And also for organizing fun events outside of the lab. A special thank you to the folks in the lab (current and former) who I have collaborated with: Zechen Ma, Boming Kong, Yeganeh Aghamohammadi, Pranjali Jain, and Sijie Kong.

Curriculum Vitæ

Zachary David Sisco

Education

2025	Ph.D. in Computer Science, University of California, Santa Barbara.
2018	M.S. in Computer Science, Wright State University.
2014	B.S. in Mathematics, Ohio University.

Publications

Z. D. Sisco, A. D. Alex, Z. Ma, Y. Aghamohammadi, B. Kong, B. Darnell, T. Sherwood, B. Hardekopf, and J. Balkind. Control Logic Synthesis: Drawing the Rest of the OWL. In *Architectural Support for Programming Languages and Operating Systems, Volume 4* (*ASPLOS*). 2024.

G. H. Smith, **Z. D. Sisco**, T. Techaumnuaiwit, J. Xia, V. Canumalla, A. Cheung, Z. Tatlock, C. Nandi, and J. Balkind. There and Back Again: A Netlist's Tale With Much Egraphin'. *Workshop on Languages, Tools, and Techniques for Accelerator Design* (*LATTE*). 2024.

Z. D. Sisco, J. Balkind, T. Sherwood, and B. Hardekopf. Loop Rerolling For Hardware Decompilation. In *Programming Language Design and Implementation* (*PLDI*). 2023.

H. Kringen, **Z. D. Sisco**, J. Balkind, T. Sherwood, and B. Hardekopf. Semi-Automated Translation of a Formal ISA Specification to Hardware. *Programming Languages for Architecture* (*PLARCH*). 2023.

Z. D. Sisco, J. Balkind, T. Sherwood, and B. Hardekopf. A Position on Program Synthesis for Processor Development. *Workshop on Languages, Tools, and Techniques for Accelerator Design* (*LATTE*). 2022.

Z. D. Sisco, A. R. Bryant. A Semantics-Based Approach to Concept Assignment in Assembly Code. *International Conference on Cyber Warfare and Security* (*ICCWS*). 2017.

Z. D. Sisco, P. P. Dudenhofer, A. R. Bryant. Modeling Information Flow for an Autonomous Agent to Support Reverse Engineering Work. *Journal of Defense Modeling and Simulation*. 2017.

Awards

2024	PhD Student of the Year, Computer Science Department, UCSB.
2024	Neal Fenzi – Resonant Founder Fellowship, Resonant, Inc.
2022	2nd Place Award, ACM SIGPLAN PLDI Student Research Competition.

Experience

2024–2025	Computing Fellow, College of Creative Studies, UCSB.
Summer 2024	Research Mentor, Research Mentorship Program, UCSB.
Summer 2023	Intern, Galois, Inc.
2020–2023	Research Assistant, UCSB.
2016–2018	Research Assistant, Wright State University.
2013–2016	Programmer Analyst, Motorists Insurance Group.

Teaching – Instructor of Record

Spring 2025	UCSB CMPTG 130E: Exploring the Hardware–Software Interface.
Winter 2025	UCSB CMPTG 1L: Programming Laboratory.
Summer 2024	UCSB CS 9: Intermediate Python Programming.
Fall 2022	UCSB CS 32: Object-Oriented Design & Implementation.
Summer 2022	UCSB CS 24: Problem Solving with Computers II.
Summer 2021	UCSB CS 16: Problem Solving with Computers I.
Winter 2021	UCSB CS 16: Problem Solving with Computers I.
Summer 2020	UCSB CS 138: Automata & Formal Languages.

Teaching – Lead Teaching Assistant

Fall 2023	UCSB CS 501: Techniques of Computer Science Teaching.
Fall 2022	UCSB CS 501: Techniques of Computer Science Teaching.

Teaching – Teaching Assistant

Fall 2024	UCSB CMPTG 1L: Programming Laboratory.
Winter 2023	UCSB CS 154: Computer Architecture.
Spring 2020	UCSB CS 138: Automata & Formal Languages.
Winter 2020	UCSB CS 56: Advanced Applications Programming.
Fall 2019	UCSB CS 56: Advanced Applications Programming.
Fall 2017	WSU CS 7830: Machine Learning.
Fall 2017	WSU CS 4350: Operating System Internals and Design.

Abstract

Automated Reasoning for Agile and Robust Chip Design

by

Zachary David Sisco

Modern chip design embodies enormous complexity, from general-purpose processors to specialized hardware accelerators. With the trend towards specialization, chip designers need techniques that let them quickly iterate over a design while fitting into familiar programming languages and tools. However, designing a chip with speed and robustness remains a challenge. Chip design requires reasoning between different layers of abstraction, however these tools do not provide mechanisms to connect specifications with implementations to ensure correctness. Programming languages for chip design rely on technology-specific components, but lack helpful abstractions needed to support common deployment platforms, making it difficult to adapt and compose designs. And further, the design ecosystem is fragmented between systems and tool chains without the ability to interoperate.

This thesis presents my research on improving chip design tools with automated reasoning techniques. I use program synthesis techniques to bridge the gap between an architectural specification and a low-level hardware implementation, developing a new technique called *control logic synthesis*. I establish a new field called *hardware decompilation*, which is about lifting common hardware artifacts to high-level source code, enabling design transpilation and automating the effort of re-targeting designs to different technologies. And finally, to address challenges with technology constraints, I developed a memory design language that uses equational reasoning techniques to automatically target multiple memory technologies from a single interface. Through

the application of these automated reasoning techniques, I opened two wholly new areas in the chip design space enabling novel design processes that were not possible before, improving developer agility and design verifiability.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The cost of designing new, specialized chips for emerging applications is too high, with industry studies showing roughly half of development time is spent on verification [1]. To handle the growing complexity of specialized hardware as well as enabling newcomers to the space with smaller teams and resources, we need new approaches to address the current state-of-the-art in hardware development. However, modern hardware description languages (HDLs) and electronic design automation tools have not kept pace with the demands brought by the "golden age" of computer architecture [2]. In this thesis, I address three challenges that hamper developer agility and verifiability during the chip design process:

Multilevel Reasoning The current system-on-chip (SoC) development process suffers from decoupling between all of the layers of abstraction. Enormous effort goes into verification between the specifications, models, and implementation—the difficulty being the semantic gap between these layers. Further, verification during this process is inherently *post hoc*: designers write microarchitectural models, *then* verify; developers write RTL code, *then* verify. This disjointed process leads to long design iterations, hindering agility and increasing verification burden.

Interoperability The growth of new HDLs and open-source hardware design tools adds new features that improve the process of designing hardware over legacy languages and proprietary tools. However, these new languages and tools often lack the ability the interoperate, fragmenting the communities that use them and making migration difficult, as manual porting is impractical. Limited interoperability means that new HDLs cannot leverage the vast body of existing designs written in legacy languages, and existing designs cannot take advantage of the new features provided by these modern tools.

Technology Constraints Languages and tools for chip design rely on technology-specific components, but lack helpful abstractions needed to support common deployment platforms, making it difficult to adapt and compose designs. Targeting platforms, such as ASIC and FPGA, requires technology-specific code and so each new technology being targeted requires updating an existing part of the code with multiple independent descriptions of the same component.

The recognition of these three challenges facing chip design brings me to my thesis statement:

1.1 Thesis Statement

The integration of automated-reasoning techniques, such as constraint solvers and equational reasoning, into programming languages used for hardware design improves developer agility and verifiability.

In this thesis, I will show how these techniques enable new hardware design processes via automated verification and program synthesis, overcoming design challenges related to multilevel reasoning, interoperability, and technology constraints.

1.2 Organization of this Document

The following describes the organization of this thesis:

Chapter 2 To address the first challenge, *multilevel reasoning*, I adapt program synthesis techniques to languages used for chip design, bridging the gap between an architectural formal specification and a low-level hardware implementation, automating the programming of tedious and error-prone control logic. This technique, called **Control Logic Synthesis**, allows designers to quickly iterate over chips designs, focusing on the critical design questions without getting bogged down in low-level details of control. It is based on work published in ASPLOS 2024 [3], and LATTE 2022 [4]. This work was done in collaboration with Andrew David Alex, Zechen Ma, Yeganeh Aghamohammadi, Boming Kong, Benjamin Darnell, Timothy Sherwood, Ben Hardekopf, and Jonathan Balkind. Specifically, Andrew David Alex contributed the compiler to generate Rosette code from ILA specifications, the AES accelerator benchmark, as well as writing. Zechen Ma, Yeganeh Aghamohammadi, Boming Kong, and Benjamin Darnell contributed in benchmark development and case study evaluation.

Chapter 3 To address the *interoperability* challenge, I established a new field called **Hardware Decompilation**, which is about lifting gate-level netlists to high-level code in hardware description languages. In this work, I apply advanced program synthesis and compiler techniques to the hardware domain. This work enables transpiling designs from one language to another, automates the effort of re-targeting designs to different technologies, and speeds up analyses over gate-level netlists. It is based on work published in PLDI 2023 [5], and supported by the National Science Foundation under Grants No. 2006542, 1763699, 1717779.

Chapter 4 To address the *technology constraints* challenge, I developed a **Memory Design Language** which includes a rich memory abstraction that automatically targets many technologies from a single interface. From this memory abstraction, I show how we can build a compiler for memories as well as a decompiler for memories from netlists—both based on the same formalized rule set. In this work I use powerful equational reasoning techniques normally used in software compilers enabling automated memory technology targeting, making designs more robust and reusable. The memory decompiler is a continued exploration of hardware decompilation established in Chapter 3. It is based on work that has been submitted to a conference and is currently under revision. I am the first author on this work, and my co-authors are Sijie Kong, Daniel Ruelas-Petrisko, Jingtao Xia, Julian Springer, Varun Rao, Spencer Wang, Gus Henry Smith, Ben Hardekopf, and Jonathan Balkind. In particular, Sijie Kong made significant contributions to the implementation for the memory decompiler as well as the dynamic programming algorithm for memory mapping with technology constraints. Daniel Ruelas-Petrisko helped with the BlackParrot case study and its evaluation, as well as XCI ingestion. Jingtao Xia, Julian Springer, and Varun Rao contributed parts of the memory compilation and decompilation implementation. Spencer Wang contributed to the evaluation, validating memory mapping results. Gus Henry Smith contributed writing about equality saturation as well as development of the abstract memory formalization.

Chapter 2

Control Logic Synthesis

2.1 Introduction

Embedded SoCs are the foundation of some of our most critical infrastructure, controlling everything from remote surgical equipment [6], to telecommunications satellites [7], to access to other larger compute and storage resources [8]. In such domains, any correctness issue could be catastrophic. 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 chapter, we describe a method for automatically generating correct by construction control 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 possibili-

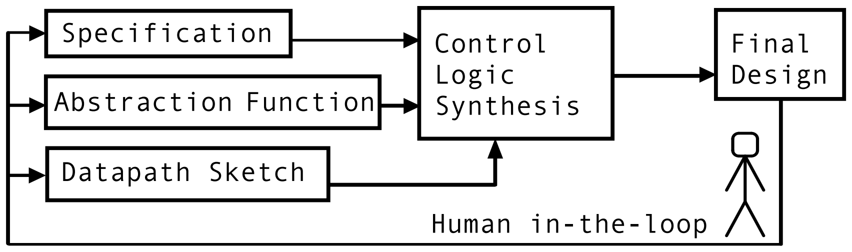


Figure 2.1: An overview of our technique: starting from an HDL sketch of the datapath combined with a formal architectural specification to generate correct-by-construction HDL code that completes the control logic.

ties 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 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.

2.1.1 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 [9, 10, 11, 12, 13, 14, 15, 16, 17], 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.

2.1.2 Technique Overview

Figure 2.1 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 [18, 19, 20, 21, 22, 23] or Sail [24]; 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 [8] (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 2.3.1).
- We present an HDL program synthesis toolchain[1] that takes a datapath and a specification for ISA/ILA semantics and automatically generates HDL code that implements the control logic (Section 2.3).
- 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

[1]Available as a free and open-source artifact: `https://github.com/UCSBarchlab/owl`

accelerator, automatically extracting semantics from architectural specifications written in ILA [19], and generating correct-by-construction control logic code in the Python-based HDL PyRTL [25].

2.2 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.

2.2.1 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 [19] 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.

2.2.2 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);
   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 2.2 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 datap-

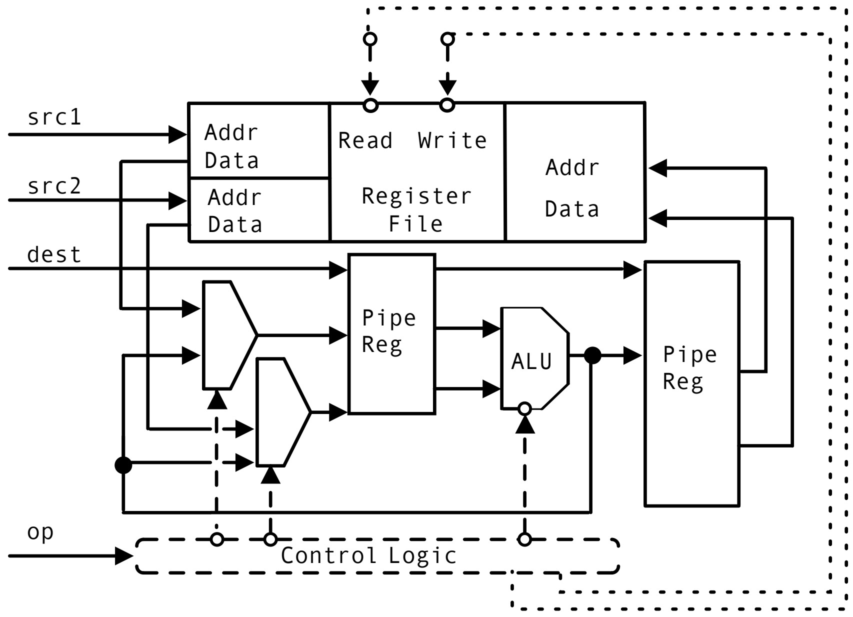


Figure 2.2: The datapath diagram for a three-stage implementation of the ALU machine. The decoded instruction is input to the control unit, which determines how to set control signals in the datapath.

ath 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.

2.2.3 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 2.3) 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

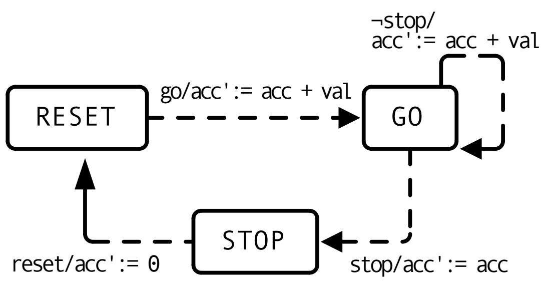


Figure 2.3: An FSM for the accumulator machine. Each state corresponds to how the machine updates the accumulator register. The input signals, `reset`, `go`, and `stop` predicate the state transitions. Control logic synthesis generates the parts of the design indicated in the dotted lines, including the transition logic.

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
```

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 2.2 and 2.3) and a specification. Control

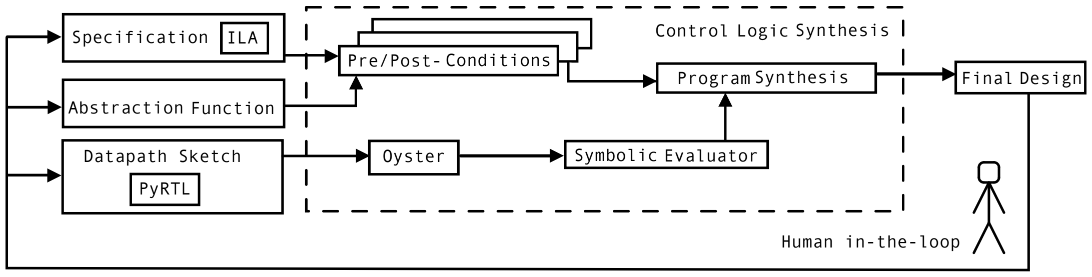


Figure 2.4: A diagram of the overall control logic synthesis workflow.

logic synthesis fills in the rest of the design—i.e., all of the dotted-lines in Figures 2.2 and 2.3 and their associated logic.

2.3 Control Logic Synthesis Technique

In this section we describe the high-level process to automatically generate correct-by-construction control logic. We show through our case studies in Section 2.4 how to specialize it to common architectures and hardware designs.

Figure 2.4 presents the overall work-flow of our toolchain. The inputs are (1) an architectural specification using ILA [19]; (2) an HDL sketch of the datapath[2]; 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 [26, 27], 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.

[2]Specified using the PyRTL HDL [25], though other languages such as SystemVerilog could be supported.

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

1. An intermediate representation (IR) tailored for program synthesis (Section 2.3.1) 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 (Section 2.3.2), 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 2.3.3).

2.3.1 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 2.5 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

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

Figure 2.5: The grammar for Oyster. An "**extract**" expression extracts the bits from the bitvector value in $expr$ between the bit-index positions low and $high$.

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 2.5 allows the hardware designer to specify where the control logic should be filled in for the datapath sketch. Our case studies (Section 2.4) 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.

2.3.2 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 2.4 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:

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

For the three-stage ALU example in Section 2.2.2, 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 2.4.2).

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.

2.3.3 Program Synthesis for Control Logic

In Figure 2.4, 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:

$$\begin{aligned} &\exists e_0,\ \ldots,\ e_n, \forall s_0. \\ &\quad \mathsf{Interpret}^k(s_0,\ \mathsf{Sketch}[h_0 := e_0,\ \ldots,\ h_n := e_n]) = (s_i)_{i=1}^k, \\ &\quad \bigwedge_j \mathsf{Pre}_j[s_{\mathsf{spec}} := \alpha(s_0)] \longrightarrow \mathsf{Post}_j[s_{\mathsf{spec}} := \alpha(s_1,\ \ldots,\ s_k)]. \end{aligned} \tag{2.1}$$

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 (2.1) 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 post-conditions between state in the specification and state in the datapath. To understand how α fits into Equation (2.1), we separate the substitution procedure into two parts, for the precondition and postcondition, respectively.

$\mathsf{Pre}_j[s_{\mathsf{spec}} := \alpha(s_0)]$, 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 $\alpha(s_0)$."

$\mathsf{Post}_j[s_{\mathsf{spec}} := \alpha(s_1, \ldots, s_k)]$, 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 2.5. To overcome this scalability issue, we introduce an optimization for control logic synthesis that can be applied under an assumption about the design.

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 2.3.3, 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 2.4.2) 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.

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 (2.1) — 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 instance, 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, e.g., in RISC-V). 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 2.6. 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

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

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 2.6 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 $\sqcup$ procedure is flexible enough to handle signals of larger bitwidths and generate nested multiplexers (through nested **if-then-else** expressions).

Correctness Argument of Union Operation

Here we argue that joining individual generated control logic per-instruction under the $\sqcup$ 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 (2.1). By solving the control logic for each instruction individually, we rearrange the formula to:

$$\begin{aligned}&\exists c_0^j,\ \ldots,\ c_n^j, \forall s_0\ . \\ &\quad \mathsf{Interpret}^k(s_0,\ \mathsf{Sketch}[h_0 := c_0^j,\ \ldots,\ h_n := c_n^j]) = (s_i)_{i=1}^k, \\ &\quad \mathsf{Pre}_j[s_{\mathsf{spec}} := \alpha(s_0)] \longrightarrow \mathsf{Post}_j[s_{\mathsf{spec}} := \alpha(s_1,\ \ldots,\ s_k)],\end{aligned} \tag{2.2}$$

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 (2.1) according to the two conditions (from Section 2.3.3). 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 (2.1). 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 2.5.3, we discuss the limitations of the instruction-independence assumption and highlight future work to support more kinds of microarchitectures.

2.4 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 2.3.

Through these case studies, we emulate an "agile" design flow. The case studies start with a base datapath sketch and off-the-shelf ILA specification, where we demonstrate the control logic synthesis technique. Then, we modify either the specification or the datapath, or both, and show how control logic synthesis can again be invoked to automatically re-generate the control logic given the design changes.

2.4.1 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 [28]. 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`.

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 2.2.2). 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 2.5, 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 2.3.3, generating control logic for each instruction independently and combining them together according to the algorithm in Figure 2.6.

Figure 2.7 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 2.7 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 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`).

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.

```
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 2.7: 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.

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.

2.4.2 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.

2.4.3 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 [28], and compile it to constraints for our control logic synthesis tool as described in Section 2.5.1. 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 (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) [30].

2.5 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.

2.5.1 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.

The ILA to Rosette compiler is 550 SLOC of C++. Figure 2.8 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.

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

Figure 2.8: 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 `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 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 2.4.3, 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.

2.5.2 Results

Table 2.1 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

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 2.1: 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 2.3.3).

our per-instruction optimization. While AES does not time out without the optimization, the per-instruction version finishes faster.

Table 2.2 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 [31] 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.

2.5.3 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

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 2.2: 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 [31]).

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 [32, 33], more recently targeted for hardware designs [34]. 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 via

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.).

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.

2.6 Related work

2.6.1 Symbolic Evaluation for Hardware Design

Existing work like SketchiLog [35, 36] generates Verilog code given a sketch and a reference implementation, but is limited to combinational circuits. VeriSketch [37] 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 [38, 39] like tailoring a processor to a specific application by reducing area and power through eliminating unused gates via symbolic analysis [40].

Knox is a framework that uses Rosette to symbolically evaluate circuits in order to formally verify hardware security modules [41] building off of previous work that translates Verilog designs into a shallow embedding in Rosette [42]. Similarly, Pensieve uses Rosette for modeling microarchitectures to find speculative execution vulnerabilities [43].

2.6.2 Hardware Languages and Design Tools

PDL (*Pipeline Description Language*) [44] 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 [45]. 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 [46] 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 [47]. 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.

2.6.3 Formal Verification

Our work intersects with research using automated theorem provers in verification of microarchitecture models for processors similar to those considered in this chapter (in-order execution with shallow pipelines) [9], models with deeper pipelines [10], and more complex microarchitectures [11, 12, 13]. 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 [48, 12, 14, 23]. 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 [15, 16, 17], intersects with ours as well. Well-established tools such as ABC [49] use SAT solving for logic simulation, synthesis, and verification tasks [50]. Advances in SMT solvers found their way into model checkers for hardware such as EBMC [51, 52, 53]. Pono [54]—successor of CoSA [55]—and AVR [56] 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 [57, 58].

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 [59, 60, 61, 62] or extract a model from RTL code for MCM verification [63].

2.7 Conclusion

Control logic synthesis enables a new design approach which embeds formal methods *in tandem* with the development process. This technique provides a correct-by-construction approach which requires only a lightweight microarchitectural model and handles the semantic gap between architectural state and microarchitectural components — addressing the challenge of *multilevel reasoning* outlined in Chapter 1. While this initial effort carries some limitations in terms of the complexity of microarchitectural optimizations control logic synthesis can support, it is nonetheless an important step forward in supporting verifiable, agile SoC development. The end goal of this line of work is a specification-forward development process, where properties such as correctness and design optimization metrics are first-class constraints. Importantly, this research provides these guarantees hand-in-hand with automated code generation, using state-of-the-art program synthesis techniques to assist with error-prone aspects of the development process. The benefit is many engineer-hours saved from complex verification tasks.

Chapter 3

Loop Rerolling for Hardware Decompilation

3.1 Introduction

Hardware description languages (HDLs) are a key tool in the hardware development process. HDLs provide high-level programmatic abstractions for designing, simulating, verifying, and synthesizing hardware. Synthesizing HDL code generates a layout of wires and logical gates represented as a graph called a *netlist*. After synthesis, the resulting netlist loses many of the high-level details from the HDL code such as loops, functions, and modules. The netlist is also considerably larger than the HDL code that generates it.

This chapter introduces a new research problem: *hardware decompilation*, that is, transforming a netlist into a semantically identical HDL program at a higher level of abstraction. The idea is analogous to software decompilation, wherein an executable binary is lifted back to source code in a high-level programming language, but targets netlists (rather than executables) and HDLs (rather than general-purpose programming languages).

3.1.1 Motivating Hardware Decompilation

If we had a solution to the hardware decompilation problem, there are a number of applications that would benefit hardware designers. This chapter only begins to explore hardware decompilation and its applications, but our work shows that these ideas have potential. Here is a non-exhaustive list of such applications:

Transpilation Between HDLs A netlist serves as a common target for all HDLs (e.g., SystemVerilog, Chisel [64], PyRTL [25], etc). When decompiling a netlist, the target HDL does not need to be the same as the original HDL from which the netlist was generated. Thus, hardware decompilation enables an automated translation between two different HDLs: take the original HDL code, synthesize it into a netlist, then decompile that netlist into the second HDL.

Speeding Up Simulation Time Simulation is a huge part of the hardware design process, both for exploration and validation of designs. Designs are repeatedly simulated, edited, and simulated again. However, simulating netlists can be significantly slower than simulating HDL source code [65]; thus, being able to recover HDL code from a netlist can help improve simulation time.

Artifact Compaction A netlist can be significantly larger than the HDL code that generated it [66]. One way to compress a netlist design (above and beyond using standard compression algorithms) would be to recover the much smaller, but equivalent, HDL-level code.

The work presented in this chapter shows the feasibility of these three applications:

transpilation (we convert designs between SystemVerilog and PyRTL HDL code); simulation time (we demonstrate that simulation of recovered HDL code is faster than the corresponding netlist, with mean speedup of 6x); and artifact compaction (we demonstrate that the recovered HDL takes significantly less space than the corresponding netlist representation, with mean compaction of 39% across our benchmark suite). Further, a hardware decompiler opens the way to other potential applications:

Understanding and Analysis In industry hardware development, it is common to use third-party component libraries (also known as Intellectual Property or IP catalogs). These components are provided only as netlists, without the higher-level HDL source code. Creating designs using these components makes human analysis and automated static analysis difficult; being able to recover high-level HDL would greatly benefit such efforts. In this way, a hardware decompiler enables security and verification analyses designed for higher-level HDL code but over a decompiled netlist (this is one of the main motivators for software decompilation as well).

Propagating Netlist Edits Back to HDL A common occurrence in hardware design is to synthesize HDL code to a netlist and then discover that the resulting design needs to be tweaked for various reasons (e.g., physical layout or timing closure). Given a change made directly to the netlist, it can be extremely difficult to reason about what specific parts of the original HDL code would need to be modified, and in what specific way, in order to ensure that the updated HDL would then generate the desired new netlist. Using hardware decompilation, that process can be entirely automated.

In a departure from the software decompilation analogy, hardware decompilation has unique value during the *design* process, not just for reverse engineering or post-design analysis. For instance, hardware synthesis and backend tools often mangle the

```
module ripple_carry_adder #(parameter N)
 (input [N-1:0] a, input [N-1:0] b, output logic
    [N-1:0] sum);
 logic cin, cout;
 always_comb begin
   cin = 1'b0;
   for (int i=0; i < N; i++) begin
     sum[i] = a[i] ^ b[i] ^ cin;
   cout = a[i] & b[i] | a[i] & cin | b[i] & cin;
   cin = cout;
   end
 end
endmodule
```

```
module accumulator (input [3:0] x, input clk, output
     reg [3:0] acc);
  logic [3:0] sum;
  ripple_carry_adder #(.N(4)) adder(acc, x, sum);
  always @(posedge clk) begin
    acc <= sum;
  end
endmodule
```

Figure 3.1: An accumulator circuit instantiated with a 4-bit ripple-carry adder written in SystemVerilog.

semantics of the HDL code, producing a netlist that is not semantically equivalent to the original design. Hardware designers then need to run simulations and logic equivalence checks over the netlist for functionality and correctness verification. Thus, hardware decompilation is a valuable tool during this design phase for speeding up simulation time by lifting the netlist to a more compact and higher-level representation.

3.1.2 Hardware Loop Rerolling

There are a number of sub-problems that need to be solved to completely translate all aspects of a netlist back to idiomatic HDL, namely identifying and recovering a range of abstractions such as: loops; procedures; modules; protocol interfaces; and clean divisions between control and data-path logic. We do not solve all of these sub-problems in this work; instead we identify one of them as a stepping stone towards solving the others. Our specific focus in this chapter is on recognizing repeated logic in netlists and decompiling them into loops in higher-level HDL code. We call this process *hardware loop rerolling*, as it mirrors an analogous operation seen in software compilers at the source and binary level [67, 68, 69, 70, 71, 72]. However, loop rerolling for a hardware decompiler occurs in the context of hardware design, where the execution model differs significantly from software. We discuss the differences between hardware loop rerolling and software loop rerolling in Section 3.7. For brevity, henceforth in the chapter we use "loop rerolling" to refer specifically to hardware loop rerolling.

In the original HDL code that synthesized the netlist, repeated logic is syntactically expressed as loops (or recursion in the case of functional HDLs [73, 74, 75, 76]). For example, Figure 3.1 presents code for an n-bit ripple-carry adder written in SystemVerilog. To generalize this function to arbitrary n-bit designs, the code uses a `for`-loop parameterized over the length, or bitwidth, of the wire vectors. The body of the `for`-loop generates a repeated pattern of add and carry operations which compute the sum of each bit and push the carry-out bit forward to the next iteration.

During hardware synthesis, loops in the HDL code are completely unrolled in the resulting netlist. The `accumulator` module in Figure 3.1 parameterizes the ripple-carry adder over wire vectors of width 4. When this accumulator circuit is synthesized, it results in the netlist found in Figure 3.2. Upon closer inspection, we find that the `for`-loop in Figure 3.1 is *unrolled* in the resulting netlist in Figure 3.2. There are four repetitions of the same set of `xor`, `and`, and `or` operations with each of the four bits of the input `x` and register `acc`. Each add and carry bit computation feeds into the proceeding one, starting from the zeroth bit to the third bit, until they are concatenated together and connected to the output register. Our goal, then, is to transform the netlist in Figure 3.2 into HDL code similar (though not necessarily identical) to that in Figure 3.1.

We break the loop rerolling problem into two major subproblems, and for each subproblem we adapt a different existing programming languages technique to create a

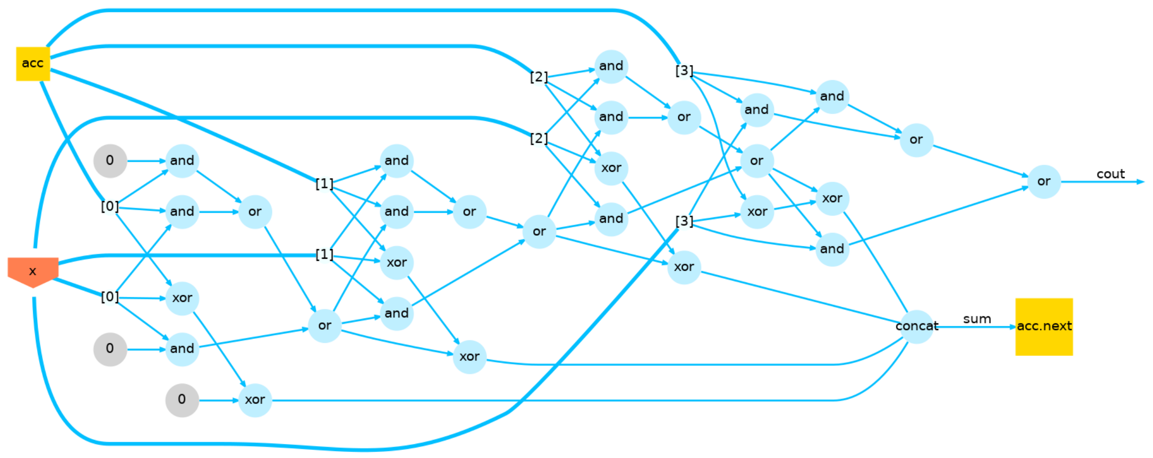


Figure 3.2: Graph representation of a netlist for a 4-bit accumulator synthesized from the HDL code in Figure 3.1. We use acc.next to denote that the register acc receives the value and is updated in the *next* cycle.

solution. The first subproblem is *loop identification*, i.e., analyzing the netlist to detect potential candidates for loops. We leverage techniques from software clone detection, applying them to netlists instead of source code text or abstract syntax trees [77, 78]. Once we have identified a candidate, the second subproblem is the actual *loop rerolling* itself, which requires reasoning about pre-, post-, and intra-loop dependencies that need to exist in the generated code. We leverage techniques from solver-based program synthesis [79] (not to be confused with hardware synthesis) in order to generate semantically equivalent looping HDL code.

3.1.3 Contributions

The contributions of this chapter are:

- I introduce the new research problem of hardware decompilation, focusing specifically in this chapter on recovering loops in hardware designs.

- I describe a technique to identify candidate slices of a netlist that are suitable for lifting up as an HDL loop, based on software clone detection techniques (Section 3.3).
- I describe a technique to take a netlist slice and synthesize semantically equivalent looping HDL code, based on program synthesis techniques (Sections 3.4 and 3.5).
- I implement our techniques[1] and evaluate their effectiveness on a benchmark suite of SystemVerilog and PyRTL hardware designs (Section 3.6). The evaluation examines the potential of hardware decompilation for transpilation, fast simulation, and artifact compaction.

3.2 The Maki Intermediate Language

We present a bespoke intermediate language called Maki that is the common connecting format for each phase of our decompilation technique: the netlist is transformed into Maki code, each phase of decompilation operates on Maki code, and the final result is transliterated into HDL code. The purpose of Maki is to provide abstractions that sit between the low-level world of a netlist and the high-level world of a full-featured HDL. As such, it should be able to completely specify a netlist but also contain abstractions such as loops and arrays. This multi-level representation allows us to represent both low-level and high-level code in the same program representation.

Figure 3.3 describes the grammar for Maki. A program in Maki starts with three components: the input wire vectors (in), the output wire vectors (out), registers (reg), and a series of statements that describe the netlist ($stmt^+$). The input and output wire vector and register declarations are lists of pairs, with the first element being a variable identifier and the second element being the bitwidth of that wire vector (i.e., the length

[1]Available as a free and open-source artifact (`https://doi.org/10.5281/zenodo.7686503`) and online repository (`https://git.sr.ht/~zachs/hardware-loop-rerolling`).

$$
\begin{array}{rcl}
\textsf{Netlist} & ::= & \mathit{in}\ \mathit{out}\ \mathit{stmt}^{+} \\
\mathit{in} \in \textsf{Input} & ::= & (\textbf{input}\ (\mathit{name}\ \mathit{width}\textbf{,})^{*}) \\
\mathit{out} \in \textsf{Output} & ::= & (\textbf{output}\ (\mathit{name}\ \mathit{width}\textbf{,})^{*}) \\
\mathit{reg} \in \textsf{Registers} & ::= & (\textbf{registers}\ (\mathit{name}\ \mathit{width}\textbf{,})^{*}) \\
\mathit{stmt} \in \textsf{Statement} & ::= & \mathit{var} := (\mathit{wexp} \mid \mathit{aexp}) \mid \textbf{for-range}\ \mathit{index}, \mathit{range}\{\mathit{stmt}^{+}\} \\
\mathit{wire} \in \textsf{WireVector} & ::= & \mathit{name} \mid \textbf{const}\ \mathit{value}\ \mathit{width} \\
\mathit{wexp} \in \textsf{WireExpression} & ::= & \mathit{wire} \mid \texttt{AND}\ \mathit{wexp}\ \mathit{wexp} \mid \texttt{OR}\ \mathit{wexp}\ \mathit{wexp} \mid \texttt{NOT}\ \mathit{wexp} \mid \texttt{XOR}\ \mathit{wexp}\ \mathit{wexp} \\
 & & \mid \textbf{mux}\ \mathit{wexp}\ \mathit{wexp}\ \mathit{wexp} \mid \textbf{concat}\ (\mathit{wexp}^{+}) \mid \textbf{select}\ \mathit{wexp}\ (\mathit{aexp} \mid \mathit{nexp}) \\
\mathit{aexp} \in \textsf{ArrayExpression} & ::= & \textbf{array-create}\ \mathit{length} \mid \textbf{array-store}\ \mathit{nexp}\ \mathit{var} \mid \textbf{array-ref}\ \mathit{var}\ \mathit{nexp} \\
\mathit{nexp} \in \textsf{NumExpression} & ::= & \mathit{var} \mid \mathit{num} \mid (+ \mid - \mid * \mid \div \mid \%)\ \mathit{nexp}\ \mathit{nexp} \\
\mathit{index}, \mathit{name}, \mathit{var} \in \textsf{Identifier} & & \\
\mathit{length}, \mathit{num}, \mathit{range}, \mathit{value}, \mathit{width} \in \textsf{Integer} & &
\end{array}
$$

Figure 3.3: The grammar for Maki.

of the bit-vector that the wire can carry). Note that we use the term *wire vector* to distinguish a bit-vector typed variable in a Maki program, as opposed to a single-bit *wire* in a netlist. Additionally, for expository purposes, the version of Maki presented here only describes a subset of HDL features. The full Maki language and our implementation supports sequential features including memory, and we include both sequential and combinational hardware designs in our evaluation.

A variable in Maki is one of three types: wire vectors (bit-vector of a set length), integers, or arrays of wire vectors. Maki has two kinds of statements: variable definitions (:=), and loops (**for-range**). A variable definition $\mathit{var} := (\mathit{wexp} \mid \mathit{aexp})$ creates and binds a new wire vector or array expression to a variable identifier, deriving its value from the right-hand side expression. A loop defines an integer loop-counter variable index initialized to zero, a loop bound range, and a sequence of statements for the loop body.

The right-hand side of a variable definition can be either a wire expression $wexp$ or an array expression $aexp$. A wire expression denotes the operations common for describing digital circuits (logical, arithmetic, bit select, wire concatenation). An array expression denotes array declarations, as well as reading from and writing to arrays. Array variables do not directly represent a specific hardware component, but are used to store accumulated results of wire expressions (e.g., storing each result of a one-bit add operation in a loop).

Figure 3.4 presents a selection of big-step structural operational semantics for Maki. There are four main types of rules in the presentation of the semantics: (1) rules for NumExpressions (NumExpOp), (2) rules for WireExpressions (WireExpOp, Mux0, Mux1), (3) rules for ArrayExpressions (ArrayRef, ArrayStore), and (4) rules for Statements. For space, we omit rules for NumExpression, WireExpression, and ArrayExpression. The more interesting rules are around Registers. For sequential elements, such as registers, there is a special store σ_{Reg} which holds updates to sequential elements until the end of one execution step. The rule StepCycle describes how Maki evaluates one step and handles register elements. Statements $s_1 \ldots s_n$ evaluate all combinational expressions, then $r_1 \ldots r_m$ update the register store σ_{Reg}. After evaluating all statements, the updates in σ_{Reg} are merged into the primary store σ so that the registers have the updated values ready for the next step.

Note that Maki programs describe one cycle of hardware execution (mapping from the state present at the beginning of the cycle to the state present at the end of the cycle) and are guaranteed to terminate. Specifically, all ForRange loops are finite and their range is known a priori (from the number of repetitions found in the netlist).

$$\frac{\langle x_1, \sigma\rangle \Downarrow_n n_1 \quad \langle x_2, \sigma\rangle \Downarrow_n n_2}{\langle x_1 \oplus_n x_2, \sigma\rangle \Downarrow_n (n_1 \oplus_n n_2)}\ \textsc{NumExpOp} \qquad \frac{\langle w_1, \sigma\rangle \Downarrow_w v_1 \quad \langle w_2, \sigma\rangle \Downarrow_w v_2}{\langle w_1 \oplus_w w_2, \sigma\rangle \Downarrow_w (v_1 \oplus_w v_2)}\ \textsc{WireExpBinop}$$

$$\frac{\langle w_c, \sigma\rangle \Downarrow_w 0 \quad \langle w_0, \sigma\rangle \Downarrow_w v_0}{\langle \textbf{mux}\ w_c\ w_0\ w_1, \sigma\rangle \Downarrow_w v_0}\ \textsc{Mux0} \qquad \frac{\langle w_c, \sigma\rangle \Downarrow_w 1 \quad \langle w_1, \sigma\rangle \Downarrow_w v_1}{\langle \textbf{mux}\ w_c\ w_0\ w_1, \sigma\rangle \Downarrow_w v_1}\ \textsc{Mux1}$$

$$\frac{\langle e, \sigma\rangle \Downarrow v}{\langle x := e, \sigma\rangle \Downarrow \sigma[x \mapsto v]}\ \textsc{Define} \qquad \frac{r \in \mathsf{Registers} \quad \langle e, \sigma\rangle \Downarrow_w v}{\langle r := e, \sigma, \sigma_{Reg}\rangle \Downarrow_r \sigma_{Reg}[r \mapsto v]}\ \textsc{RegUpdate}$$

$$\frac{\langle x, \sigma\rangle \Downarrow_a a \quad \langle n, \sigma\rangle \Downarrow_n n'}{\langle \textbf{array-ref}\ x\ n, \sigma\rangle \Downarrow_a a[n']}\ \textsc{ArrayRef} \qquad \frac{\langle x_1, \sigma\rangle \Downarrow_a a \quad \langle e, \sigma\rangle \Downarrow_n i \quad \langle x_2, \sigma\rangle \Downarrow_w v}{\langle x_1 := \textbf{array-store}\ e\ x_2, \sigma\rangle \Downarrow \sigma[a[i \mapsto v]]}\ \textsc{ArrayStore}$$

$$\frac{\langle s_1, \sigma\rangle \Downarrow \sigma_1 \quad \langle s_2, \sigma_1\rangle \Downarrow \sigma_2}{\langle s_1\ s_2, \sigma\rangle \Downarrow \sigma_2}\ \textsc{StmtSeq} \qquad \frac{\begin{array}{c}\langle i := 0, \sigma\rangle \Downarrow \sigma' \quad \langle s_1 \ldots s_n, \sigma'\rangle \Downarrow \sigma_1 \\ \langle i := i+1, \sigma_1\rangle \Downarrow \sigma'_1\ \ldots\ \langle s_1 \ldots s_n, \sigma'_{r-1}\rangle \Downarrow \sigma_r\end{array}}{\langle \textbf{for-range}\ i\ r\ s_1 \ldots s_n, \sigma\rangle \Downarrow \sigma_r}\ \textsc{ForRange}$$

$$\frac{\begin{array}{c}\langle s_1, \sigma\rangle \Downarrow \sigma_1\ \ldots\ \langle s_n, \sigma_{n-1}\rangle \Downarrow \sigma_n\ \ldots \\ \langle r_1, \sigma_n, \sigma_{Reg}\rangle \Downarrow_r \sigma^1_{Reg}\ \ldots\ \langle r_m, \sigma_n, \sigma^{m-1}_{Reg}\rangle \Downarrow_r \sigma^m_{Reg}\ \ldots\end{array}}{\langle s1 \ldots s_n\ r_1 \ldots r_m, \sigma, \sigma_{Reg}\rangle \Downarrow \sigma_n[\forall_{x \in \sigma^m_{Reg}} x \mapsto \sigma^m_{Reg}(x)]}\ \textsc{StepCycle}$$

Figure 3.4: A selection of big-step structural operational semantics for Maki. The relations $\Downarrow_n$, $\Downarrow_w$, $\Downarrow_a$, and $\Downarrow_r$ are expressly for evaluating NumExpression, WireExpression, ArrayExpression, and register updates, respectively. The environments is σ and σ_{Reg} map variable identifiers to values and registers.

Translating a Netlist to Maki

We translate a netlist to Maki by performing a topological sort over the netlist, starting from the input wires. This sort linearizes the netlist graph in a way that gives the same ordering for the same netlist (i.e., it is deterministic), and, in practice, groups related operations together. Each gate (node in the graph) is translated to a Maki wire vector variable definition by making the left-hand side the outgoing edge (the wire vector being defined), while the right-hand side becomes a wire expression. We translate the wire expression according to the wire operation and its arguments (the incoming edges). The input and output wires are the initial wire vector variables. All other edges in the graph become intermediate wire vector variables that are defined exactly once. The result of the translation is a Maki program in SSA form. Figure 3.5 shows Maki code for the 4-bit accumulator netlist from Figure 3.2 after linearization. Note that the initial translation of the netlist to Maki only contains the low-level features of Maki (i.e., only wire expressions defining wire vector variables). The high-level features, like loops and arrays, will come from decompilation, at which point the design may not be in SSA form.

Linearizing a netlist is efficient and works well in practice; a topological sort tends to order related wires and operations together. However, it is possible a linearization may disguise some repeated logic if the netlist is linearized differently for different repetitions. Another possible approach without this limitation would be to operate on the netlist graph directly. However, detecting repeated logic would amount to detecting repeated subgraph isomorphisms, which is an NP-complete problem and expensive in practice. We choose to linearize the code in order to leverage efficient techniques from software clone detection.

3.3 Loop Identification

Here we describe our technique for identifying slices of repeated logic in a netlist that may reasonably correspond to a loop in the higher-level HDL. Translating the netlist to Maki linearizes the graph of wires and gates into a straight-line program in SSA form. Our loop identification task is to find continuous segments of repeated statements in the program.

```
(input x 4)
(register acc 4)
t3 := const 0 1
t8 := select acc (0)
t9 := select x (0)
t10 := XOR t8 t9
t11 := XOR t10 t3
t12 := AND t8 t9
t13 := AND t8 t3
t14 := OR t12 t13
t15 := AND t9 t3
t16 := OR t14 t15
t17 := select acc (1)
t18 := select x (1)
t19 := XOR t17 t18
t20 := XOR t19 t16
t21 := AND t17 t18
t22 := AND t17 t16
t23 := OR t21 t22
t24 := AND t18 t16
t25 := OR t23 t24
t26 := select acc (2)
t27 := select x (2)
t28 := XOR t26 t27
t29 := XOR t28 t25
t30 := AND t26 t27
t31 := AND t26 t25
t32 := OR t30 t31
t33 := AND t27 t25
t34 := OR t32 t33
t35 := select acc (3)
t36 := select x (3)
t37 := XOR t35 t36
t38 := XOR t37 t34
t39 := AND t35 t36
t40 := AND t35 t34
t41 := OR t39 t40
t42 := AND t36 t34
t43 := OR t41 t42
t44 := concat (t38 t29 t20 t11)
acc := t44
```

Figure 3.5: Maki representation of the 4-bit accumulator netlist from Figure 3.2 after linearization.

We take inspiration from software clone detection [77, 78]. In our case, a "clone" A is a segment of Maki code that is identical to some other Maki code segment B, modulo variable identifiers (i.e., identifiers are not considered). We specifically look for *tandem repeats* [80], that is, a sequence of consecutive clones without anything in-between the repeated code segments. The entire loop identification process consists of three phases:(1) tokenize the Maki program; (2) scan the resulting token stream for tandem repeats; (3) heuristically filter out tandem repeats that are undesirable candidates for loop rerolling. The end result is a set of slices of the Maki program, each slice being a candidate for loop rerolling.

3.3.1 Tokenization

We transform the Maki program into a sequence of *tokens*, similarly to lexical analysis in parsing but applying certain abstractions to ignore irrelevant differences such as variable identifiers (because hardware synthesis would have unrolled a loop and given each iteration its own wires, thus including identifiers would mean that no clones can exist). Since statements in Maki are only two types (variable definitions and loops), and there are no loops in the initial translation of the netlist to Maki, tokenizing a Maki program considers only one case: variable definitions.

If the right-hand side of a variable definition is a wire expression, then we create a token for the wire operation. Array expressions are ignored (because we start from a netlist there will be no array expressions initially). If the wire expression is assignment, as in `b := a`, we record the bitwidth in the token as well as if it is an output wire vector (see the last token in Figure 3.6 for an example). As an example, Figure 3.6 shows the accumulator netlist (from Figure 3.5) as a token stream.

3.3.2 Finding Tandem Repeats

A repeated sequence of tokens with no interruptions indicates a candidate for loop rerolling. Note that because Maki linearizes the netlist graph, not all loops in the original HDL code may result in identical code segments for each loop iteration (if different iterations are linearized differently); we show in our evaluation that in practice using the linearized form works well. We use a standard sequence alignment technique using suffix trees to detect longest common prefixes [81]; this technique returns the location, length, and number of repeats contained in each tandem repeat present in the token stream, which when mapped back to the Maki code yields a potential loop rerolling candidate.[2] When we apply this process to the token stream in Figure 3.6, it shows that the boxed tokens in that figure represent the first repetition of a tandem repeat of length four.

Note that a tandem repeat may *not* represent a valid loop in HDL code, i.e., this loop identification process is an over-approximation of the token stream. This approxima-

[2]We filter out candidates of only two repetitions because these rarely correspond to useful loops.

$\boxed{\langle\textbf{select}\rangle\ \langle\textbf{select}\rangle\ \langle\mathsf{XOR}\rangle\ \langle\mathsf{XOR}\rangle\ \langle\mathsf{AND}\rangle\ \langle\mathsf{AND}\rangle\ \langle\mathsf{OR}\rangle\ \langle\mathsf{AND}\rangle\ \langle\mathsf{OR}\rangle}$

$\langle\textbf{select}\rangle\ \langle\textbf{select}\rangle\ \langle\mathsf{XOR}\rangle\ \langle\mathsf{XOR}\rangle\ \langle\mathsf{AND}\rangle\ \langle\mathsf{AND}\rangle\ \langle\mathsf{OR}\rangle\ \langle\mathsf{AND}\rangle\ \langle\mathsf{OR}\rangle$

$\langle\textbf{select}\rangle\ \langle\textbf{select}\rangle\ \langle\mathsf{XOR}\rangle\ \langle\mathsf{XOR}\rangle\ \langle\mathsf{AND}\rangle\ \langle\mathsf{AND}\rangle\ \langle\mathsf{OR}\rangle\ \langle\mathsf{AND}\rangle\ \langle\mathsf{OR}\rangle$

$\langle\textbf{select}\rangle\ \langle\textbf{select}\rangle\ \langle\mathsf{XOR}\rangle\ \langle\mathsf{XOR}\rangle\ \langle\mathsf{AND}\rangle\ \langle\mathsf{AND}\rangle\ \langle\mathsf{OR}\rangle\ \langle\mathsf{AND}\rangle\ \langle\mathsf{OR}\rangle$

$\langle\textbf{concat}\rangle\ \langle\textbf{register}, 4\rangle$

Figure 3.6: A tokenized version of the 4-bit accumulator netlist from Figure 3.5.

tion is because our tokenization necessarily abstracts the Maki code and ignores things like variable identifiers; a tandem repeat may, once we look at the actual wire definitions it contains, not correspond to an iterative repetition of logic that is characteristic of a loop.

On the other hand, it may also be the case that a tandem repeat can be successfully rerolled but does not correspond to a loop in the original HDL code. The reason for this discrepancy is due to how hardware synthesis lowers HDL code to a netlist. During hardware synthesis, higher-level operations over wire vectors expand to lower-level operations over single-bit wires. For instance, consider an `AND` operation over two 4-bit input wire vectors `a` and `b`. In an HDL like SystemVerilog or PyRTL, this can be written simply as `c = a & b`, but in the resulting netlist this operation gets expanded into 4 repeated `AND` operations (for each bit in the wire vector) with the result concatenated into an output wire. While this repeat is not a loop in the original HDL code, it is nonetheless a sequence of repeated logic we can detect and reroll into a valid loop.

3.4 Sketch Generation for Loop Rerolling

Given a loop candidate, i.e., a tandem repeat in the Maki code as identified using the technique described in Section 3.3, we want to rewrite the candidate into Maki code that uses higher-level loop and array abstractions. One might think that we could simply take a single element of the tandem repeat (corresponding to a single iteration of the desired loop) and wrap it inside a loop expression. The reality is more difficult: the newly-created loop must maintain the correct pre-, post-, and intra-loop wire dependencies to guarantee semantic equivalence to the original Maki code, and must also infer non-trivial bit-selecting arithmetic (not present in the original low-level code) in order to allow a single loop body to compute different iterations of the loop correctly.

One potential strategy would be to use heuristic code transformations that attempt to preserve the necessary wire dependencies and semantic equivalence to the original code. However, our experience is that the required heuristics are very design-dependent and differ widely across different netlists, and that inferring the necessary arbitrary bit-selecting arithmetic using static analysis is non-trivial and often fails.

Instead, we leverage sketch-based program synthesis [82]. This takes a sketch of the desired code (i.e., Maki code containing *holes* for synthesis to fill in) and an oracle for determining correctness (in this case, the original Maki program), then applies an SMT solver to produce a new Maki program based on the provided sketch, with the holes filled in, that is guaranteed to be semantically equivalent to the original Maki program. In the remainder of this section we discuss how to automatically create a suitable sketch given a specific tandem repeat. In the next section we discuss how to use that sketch for Maki-specific program synthesis.

We add two new constructs to Maki's syntax to represent holes, `?w` and `?n`, defined as:

$$?\mathtt{w} ::= \mathit{wire} \mid \textbf{array-ref}\ \mathit{var}\ \mathit{nexp}$$

$$?\mathtt{n} ::= \mathit{nexp}$$

?w represents a WireVector hole and can take the place of any Maki expression that resolves to a wire vector-typed value (including reading from arrays of wire vectors). ?n represents a NumExpression hole and is used for filling in the indexing arguments to array references, array stores, and wire vector bit-selects.

We begin sketch generation by picking an arbitrary element of the tandem repeat to serve as the loop body and wrap it within a **for-range** expression to create an initial sketch. Sketch generation is split into two main passes based on (1) *reaching definitions* and (2) *liveness* properties of variables in the Maki program. These passes are made efficient by the fact that the netlist translated into Maki is already in SSA form. Figure 3.7a shows the initial sketch for our accumulator example.

3.4.1 Reaching Definitions Pass

The first pass for sketch generation uses reaching definitions information of wire variables in a Maki program. This pass focuses on variable *uses*, and will only insert holes into the right-hand side of variable definitions. We compute a *use-def* chain to capture the reaching definitions of a Maki program. This data structure is commonly used in compilers for data-flow analysis. The use-def chain of a program maps each use of a variable to definitions which reach that use. Because our initial Maki netlist is in SSA form, an element in a use-def chain maps to precisely one definition.

The use-def chain tells us where in the program we have broken data dependencies — in the form of unreachable or missing definitions — from inserting the new loop sketch. With this, we perform a pass over the Maki program using Algorithm 1. The

```
(input x 4)
(register acc 4)
t3 := const 0 1
for-range i, 4 {
    t8 := select acc (0)
    t9 := select x (0)
    t10 := XOR t8 t9
    t11 := XOR t10 t3
    t12 := AND t8 t9
    t13 := AND t8 t3
    t14 := OR t12 t13
    t15 := AND t9 t3
    t16 := OR t14 t15 }
t44 := concat (t38 t29 t20 t11)
acc := t44
```

(a)

```
(input x 4)
(register acc 4)
t3 := const 0 1
for-range i, 4 {
    t8 := select ?w (?n)
    t9 := select ?w (?n)
    t10 := XOR t8 t9
    t11 := XOR t10 ?w
    t12 := AND t8 t9
    t13 := AND t8 ?w
    t14 := OR t12 t13
    t15 := AND t9 ?w
    t16 := OR t14 t15 }
t44 := concat (?w ?w ?w ?w)
acc := t44
```

(b)

Figure 3.7: Intermediate reroll sketches of an accumulator design in Maki. (a) Initial reroll sketch of the accumulator. Note that the sketch has not been made generic yet. That is, before inserting any holes we start by just copying the first iteration's statements into a new loop body. (b) Reroll sketch of the accumulator after inserting holes from the reaching definitions pass Algorithm 1.

map returned from the procedure informs which parts of which statements to update with the given hole. The first loop (lines 3–10) scans the right-hand side of variable definition statements in the loop body. First, the pass scans the right-hand side of each variable definition statement in the loop body. If the use of a variable has no reaching definition in the loop body, it replaces that variable use with a `?w` hole. This pass also scans the right-hand side of variable definitions for numeric and constant arguments—replacing any numeric bit-select argument with a `?n` hole (since any NumExpression-typed value comes only from a wire select operation). It also replaces any **const**-typed expression with a `?w` hole.

In lines 7–8 the algorithm replaces any numeric bit-select argument with a `?n` hole (since any NumExpression-typed value will only come from a wire select operation). In lines 9–10, the algorithm replaces any **const**-typed expression with a `?w` hole. The second loop (lines 11–14) scans the right-hand side of variable definition statements after the loop. Next, the pass scans the right-hand side of variable definition statements *after* the loop. The algorithm replaces any variable uses which have no reaching definition with a `?w` hole. After running Algorithm 1 over the code in Figure 3.7a we get the intermediate sketch in Figure 3.7b.

3.4.2 Liveness Pass

The second pass for sketch generation uses liveness information of wire variables in a Maki program. This pass focuses on program variable *definitions*, and will introduce new variable definitions into the sketch. This time, we compute a *def-use* chain to capture the liveness data in a Maki program. The def-use chain of a program maps each variable definition to the uses which reach that definition. Note that a use-def chain and def-use chain for the same program are not symmetric. There is information in

Algorithm 1 Sketch generation pass based on reaching definitions. $Statements$ is a list of Maki statements indexed from 0 to n. UD holds the use-def chain for the Maki code in $Statements$. Returns $holes$, a map of statement indices to a set of pairs of variable identifiers with their respective holes.

```
 1: procedure ReachingDefsPass
 2:     $holes \leftarrow \emptyset$
 3:     for all $i \in \{LoopStart, \ldots, LoopEnd\}$ do
 4:         for all $use \in Statements[i].rhs$ do
 5:             if $UD[use] \notin \{LoopStart, \ldots, i\}$ then
 6:                 $holes[i] \leftarrow holes[i] \cup (use, ?\mathtt{w})$
 7:             if Type($UD[use]$) $=$ NumExpression then
 8:                 $holes[i] \leftarrow holes[i] \cup (use, ?\mathtt{n})$
 9:             if Type($UD[use]$) $=$ const then
10:                 $holes[i] \leftarrow holes[i] \cup (use, ?\mathtt{w})$
11:     for all $i \in \{LoopEnd + 1, \ldots, n\}$ do
12:         for all $use \in Statements[i].rhs$ do
13:             if $UD[use] \notin \{0, \ldots, LoopStart - 1\} \cup \{LoopEnd + 1, \ldots, n\}$ then
14:                 $holes[i] \leftarrow holes[i] \cup (use, ?\mathtt{w})$
15:     return $holes$
```

one that is not captured in the other.

The information in the def-use chain tells us where in the program we have broken data dependencies—in the form of dead variables—from inserting the new loop sketch. With the def-use chain, we perform a second pass over the Maki program using liveness information Algorithm 2. The map returned from the procedure tells us which definitions to add to the sketch. There are two cases for modifying the sketch to fix liveness:

1. Lines 5–6: The variable is dead. Any uses of the variable were removed when the remaining statements in the original unrolled code were discarded. This case indicates some data is feeding forward into subsequent loop iterations. From the accumulator example, this is the carry-in bit (variable `t3`). The solution is to provide a definition *before* the loop. The uses have already been replaced with `?w` holes from the reaching definitions pass (Algorithm 1).

2. Lines 7–10: The variable is defined within the loop and is live *after* the loop. This case indicates that the loop is accumulating some results each iteration. The solution is to declare an array *before* the loop, and update the array each iteration with the just defined variable. From the accumulator example, the intermediate sums are stored in an array to be used after the loop.

After applying the liveness pass (Algorithm 2) over the code in Figure 3.7b we get the final sketch in Figure 3.8a. Generating a sketch that preserves data dependencies before, after, and within the new loop is general enough to work for netlists that contain sequences of repeated logic. The resulting sketch over-approximates the number of unknowns but it ensures that no data dependencies are broken after inserting the loop and transforming the Maki program.

Algorithm 2 Sketch generation pass based on liveness information. DU holds the def-use chain for the Maki code in $Statements$. Returns $NewDefs$, a map from statement indices to a set of pairs of variable identifiers with their respective definitions.

```
1:  procedure LivenessPass
2:      $NewDefs \leftarrow \emptyset$
3:      for all $i \in \{LoopStart, \ldots, LoopEnd\}$ do
4:          for all $def \in Statements[i].lhs$ do
5:              if $DU[def] \not\subseteq \{LoopStart, \ldots, i\} \cup \{LoopEnd + 1, \ldots, n\}$ then
6:                  $NewDefs[LoopStart - 1] \leftarrow NewDefs[LoopStart - 1] \cup (def, ?w)$
7:              if $DU[def] \subseteq \{LoopEnd + 1, \ldots, n\}$ then
8:                  $x \leftarrow$ FreshVariable
9:                  $NewDefs[LoopStart - 1] \leftarrow NewDefs[LoopStart - 1] \cup (x,$ array-create $LoopReps)$
10:                 $NewDefs[i] \leftarrow NewDefs[i] \cup (x,$ array-store ?n $def)$
11:     return $NewDefs$
```

3.4.3 Properties of Sketch Generation

If a tandem repeat from loop identification is a valid, rerollable loop then, with one caveat, our sketch generation process introduces sufficient holes to reroll it. The caveat is that that we do not consider alternative schemes for bundling wires together in the netlist other than the one present in the original Maki code. Doing so could potentially allow more identified loops to be rerolled than our current technique, and is an interesting future direction.

We argue that, modulo the wire bundling scheme, this property is true because all points of dependencies between variables inside and outside the loop are addressed through ?w and ?n holes. That is, initially creating the loop sketch (as in Figure 3.7a) breaks some data dependencies in the Maki program. However, our pre-, post-, and intra-loop strategies for patching the dependencies cover all relevant points (aided by

```
(input x 4)
(register acc 4)
t3 := const 0 1
t4 := array-create 4
t16 := ?w
for-range i, 4 {
    t8 := select ?w (?n)
    t9 := select ?w (?n)
    t10 := XOR t8 t9
    t11 := XOR t10 ?w
    t4 := array-store ?n t11
    t12 := AND t8 t9
    t13 := AND t8 ?w
    t14 := OR t12 t13
    t15 := AND t9 ?w
    t16 := OR t14 t15 }
t44 := concat (?w ?w ?w ?w)
acc := t44
```

(a)

```
(input x 4)
(register acc 4)
t3 := const 0 1
t4 := array-create 4
t16 := t3
for-range i, 4 {
    t8 := select acc (i)
    t9 := select x (i)
    t10 := XOR t8 t9
    t11 := XOR t10 t16
    t4 := array-store i t11
    t12 := AND t8 t9
    t13 := AND t8 t16
    t14 := OR t12 t13
    t15 := AND t9 t16
    t16 := OR t14 t15 }
t44 := concat ((array-ref t4 3) (array-ref t4 2)
     (array-ref t4 1) (array-ref t4 0))
acc := t44
```

(b)

Figure 3.8: (a) Final reroll sketch of the accumulator design after inserting holes from the liveness pass Algorithm 2. (b) The rerolled designed for the accumulator sketch in (a) after program synthesis.

the reaching definitions and liveness analysis of the variables in the original SSA form of the program), and the holes are flexible enough that if there exists a way to reroll the candidate into a valid loop then the resulting sketch provides at least one way.

3.5 Program Synthesis for Loop Rerolling

With our generated sketch of the hardware design with loops, we want to find a solution that fills in the holes and produces a design equivalent to the original netlist.

We use an established program synthesis technique called *counterexample-guided inductive synthesis* (CEGIS) [79]. CEGIS completes a program sketch by searching for a candidate solution (i.e., a way to fill the holes) and then verifying it against the reference specification. Here, the reference specification is the unmodified netlist. CEGIS searches for a solution by translating the candidate to constraint formulas and feeding them into an SMT solver. While verifying a candidate solution, if the SMT solver finds a counterexample that falsifies the solution, CEGIS generates a new candidate solution taking into account previously found counterexamples. This solve-verify loop continues until a solution is synthesized that satisfies the specification, or it determines that a solution does not exist.

3.5.1 Solver-Aided Maki

We wrote a symbolic interpreter that "runs" programs written in Maki. The symbolic interpreter keeps track of the program's state—that is, the variable definitions—where all input wires to the netlist are symbolic bitvectors. This process compiles a netlist specification into a set of symbolic constraints. These constraints enable solver-based verification and synthesis. These solver-aided functions come from Rosette, a Racket framework for CEGIS [27].

Rosette is a language-driven framework for building program synthesizers. By defining a language and an interpreter for that language, Rosette can lift the evaluation of programs in that language to work with symbolic values. This "symbolic evaluation" is the process that converts a program into a set of constraint formulas that an SMT solver can understand.

3.5.2 The Loop Rerolling Pipeline

Continuing the accumulator example, loop rerolling via program synthesis works as follows. Given a netlist translated to Maki, and a sketch of that netlist with loops generated by the methodology in Section 3.4, do the following:

1. First, symbolically interpret the netlist.
2. Using a CEGIS solver, produce a candidate solution for the sketch.
3. Symbolically interpret the candidate solution and verify it against the reference netlist. Check the equivalence of each of the symbolically defined output wires.
4. If the solver finds a counterexample, add it to the current set of constraints, then generate a new candidate. Repeat until it determines a solution does not exist.
5. Otherwise, if the candidate satisfies the reference netlist specification, substitute the holes in the sketch according to the expressions found in the solution. The resulting program is the semantically-equivalent synthesized Maki code with loops.

Figure 3.8b presents the synthesized code with a rerolled loop for the 4-bit accumulator design. Note that `t16` is initialized to a constant before the loop (as a result of the pre-loop dependency check), and is updated at the end of each iteration inside the loop. This variable holds the carry-in and carry-out values for the adder. The `?n` holes inside the loop body fill in with loop variable `i`. `?w` holes inside the loop body fill in with previously defined wire variables. The `?w` holes after the loop in the **concat** operation resolve to **array-ref** operations that retrieve the stored sum of each of the four bits of the input wires. These **array-ref** holes correspond to the post-loop dependencies introduced in sketch generation.

```
module accumulator (input  clk, input [3:0] x,
     input reg [3:0] acc);
 logic t0; logic [3:0] t1;
 always_comb begin
   t0 = 1'b0;
   for (int i=0; i < 4; i++) begin
     t1[i] = (acc[i] ^ x[i]) ^ t0;
     t0 = ((acc[i] & x[i]) | (acc[i] & t0)) |
          (x[i] & t0);
   end
 end
 always_ff @(posedge clk) begin
   acc <= {t1[3], t1[2], t1[1], t1[0]};
 end
endmodule
```

(a)

```
from pyrtl import *
acc = Register(bitwidth=4)
x = Input(bitwidth=4)
t0 = Const(0, bitwidth=1)
t1 = [None]*4
for i in range(4):
  t1[i] = (acc[i] ^ x[i]) ^ t0
  t0 = ((acc[i] & x[i]) | (acc[i] & t0)) | (x[i] & t0)
acc.next <<= concat(t1[3], t1[2], t1[1], t1[0])
```

(b)

Figure 3.9: Decompiled SystemVerilog (a) and PyRTL (b) code for the 4-bit accumulator.

3.5.3 Output to HDL Code

After loop rerolling we can easily translate Maki code to an HDL. For instance, our tool translates Maki to SystemVerilog and PyRTL [25]. Figures 3.9a and 3.9b present the rerolled and decompiled 4-bit accumulator in SystemVerilog and PyRTL, respectively. In Section 3.1, we presented the original SystemVerilog code for the accumulator in Figure 3.1. Note the similarity between the rerolled code in Figure 3.9a and the original code[3].

Also note that the original SystemVerilog code splits the design into two separate modules (or two functions, for PyRTL). Here, the decompiler emits the design as one flattened module. Procedural abstraction is one branch of future work, whereby a candidate fragment of the netlist is wrapped into a module or function body, and all occurrences of the fragment are replaced with a module instantiation/function call.

3.6 Evaluation

Here we evaluate our implementation of the loop identification and rerolling techniques. Our implementation has two parts that span two languages: we implement loop identification in Python 3.9 and loop rerolling (sketch generation plus program synthesis) in Racket 8.7 using the Rosette CEGIS framework [27]. To evaluate our loop identification and rerolling we use two sets of hardware design benchmarks written in two different HDLs: PyRTL (Table 3.1) and SystemVerilog (Figure 3.10). The PyRTL benchmarks are a mix of combinational and sequential designs for basic components such as adders, shifters, and caches.

The SystemVerilog benchmarks are taken from the BaseJump Standard Template

[3]For readability, we rewrite expressions in the rerolled loops from the "three-address code" form into nested wire expressions.

Table 3.1: PyRTL benchmark information. “Loops” are the number of loops present in the original source code. The “small”, “medium”, and “large” columns denote a particular parameterization of the design and the number of wires and gates in the resulting netlist. A benchmark noted as ’recursive’ means that the loops are implemented via recursive function calls.

		Small		**Medium**		**Large**	
Module	**Loops**	**Wires**	**Gates**	**Wires**	**Gates**	**Wires**	**Gates**
Barrel shifter	1	44	42	196	194	839	836
Cache (directed)	1	199	158	391	310	775	614
Cache (n-way set associative)	1	165	125	326	249	645	495
Demultiplexer	1	88	83	274	269	507	502
Priority encoder (recursive)	1	79	57	146	107	277	205
Pseudo-random number generator	1	43	40	139	136	526	522
Ripple-carry adder (iterative)	1	79	74	295	290	583	578
Ripple-carry adder (recursive)	1	97	92	385	380	769	764
Shifter	1	160	140	320	284	640	572

Library [83] (BaseJump STL) found in the BSG Micro Designs repository [84]. BaseJump STL is a standard template library of components commonly found across many different hardware designs. We chose designs from BaseJump STL without regard to their original SystemVerilog code containing loops in the interest of finding loop rerolling opportunities where there were none originally.

Our translator from netlist to Maki operates on PyRTL-format netlists. For BaseJump STL we converted each module into BLIF format [85] via Yosys [31] and imported it into PyRTL to have the netlist in the correct format. Due to this multi-step conversion process we only evaluate a subset of the BaseJump STL modules. For instance, PyRTL does not support asynchronous designs and so those modules were not included.

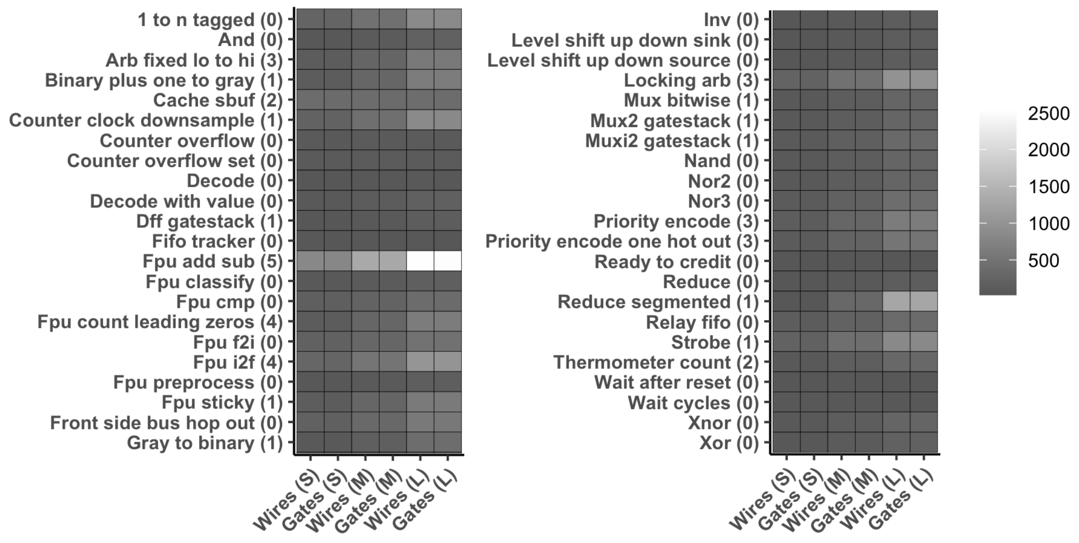


Figure 3.10: Heat map of benchmark sizes for the BaseJump modules, shows count of wires and gates for each parameterization (S/M/L) of the module. The number of loops present in the SystemVerilog source code is shown in parentheses next to the module name.

We parameterize each benchmark over three configurations we denote as "small", "medium", and "large"; for many designs this meant setting the width parameter to 16, 32, and 64, respectively. Note that the size categories are benchmark-specific, not meant to be compared across benchmarks. We test our implementation on the netlists produced from these modules for each size configuration.

We rely on PyRTL's compiler for netlist linearization; when building a netlist PyRTL performs a topological sort over wires in the graph and assigns them a deterministic order where related operations are close together in practice. With this sort, our tool lifts each netlist to Maki, tokenizes the code, and performs loop identification. The Maki program and loop information then feed into the loop rerolling phase. If the sketch is satisfiable, the loop rerolling tool outputs the decompiled HDL code. Note that our decompiler supports PyRTL and SystemVerilog as an output language regardless of which HDL generated the input netlist. Both phases of loop identification and rerolling were run on a machine with a 6-core Intel Xeon E5-2420, 32 GB RAM, running Ubuntu 18.04.5.

3.6.1 Loop Identification and Rerolling Results

Table 3.2 presents the loop rerolling results over the nine PyRTL benchmarks. Since a netlist represents a particular parameterization of a hardware design, we ran loop identification and rerolling on three configurations (“small”, “medium”, and “large”) for each benchmark. Each PyRTL benchmark contained one loop in its source code, and our decompiler identified and rerolled that loop in each case. The decompiler often found more loops than were contained in the original HDL code; for example, on the large version of the priority encoder the decompiler identified 6 potential loops and rerolled 4 of them. Loop identification over-approximates the number of loops that can be rerolled. Thus, some potential loops are false positives and the decompiler does not always reroll as many loops as it finds.

PyRTL can also represent repeated logic using recursive functions. We include two recursive benchmarks (priority encoder and the recursive ripple-carry adder) in our evaluation to show that at the netlist level a recursive PyRTL function still gets unrolled into a set of repeated gates and wires, and our loop rerolling tool can still identify and reroll those operations into an equivalent loop (though converted into an iterative loop rather than recursion).

Figure 3.11 presents the loop rerolling results over the BaseJump STL benchmarks. These benchmarks are generally larger than the PyRTL benchmarks and that is reflected in the number of loops rerolled. The large version of “Fpu count leading zeros”

Table 3.2: PyRTL benchmark loop identification and rerolling results. The "small", "medium", and "large" rows for each benchmark denote a particular parameterization of the design. Both the loop identification and loop rerolling phases have a timeout of 1 hour.

Module	Size	Loops found / expected	Loops rerolled	Loop detection time (s)	Loop rerolling time (s)
Barrel shifter	Small	1 / 1	1	0.1	6.8
	Med.	1 / 1	1	0.6	11.4
	Large	1 / 1	1	6.3	27.9
Cache, directed	Small	1 / 1	1	0.4	8.8
	Med.	9 / 1	1	1.8	53.1
	Large	1 / 1	1	7.1	198.3
Cache, n-way set associative	Small	1 / 1	1	0.3	13.8
	Med.	3 / 1	2	0.8	38.9
	Large	4 / 1	2	4.2	146.3
Demultiplexer	Small	1 / 1	1	0.2	6.3
	Med.	3 / 1	2	1.1	16.3
	Large	2 / 1	1	4.1	26.7
Priority encoder (recursive)	Small	3 / 1	2	0.1	9.1
	Med.	5 / 1	3	0.3	16.2
	Large	6 / 1	4	0.8	31.1
Pseudo-random number generator	Small	1 / 1	1	0.1	8.4
	Med.	1 / 1	1	0.2	14.2
	Large	1 / 1	1	2.5	78.9
Ripple-carry adder (iterative)	Small	1 / 1	1	0.1	5.9
	Med.	1 / 1	1	0.6	8.1
	Large	1 / 1	1	3.9	12.1
Ripple-carry adder (recursive)	Small	2 / 1	1	0.1	6.9
	Med.	2 / 1	1	2.0	24.1
	Large	2 / 1	1	15.6	102.5
Shifter	Small	9 / 1	1	0.3	21.1
	Med.	1 / 1	1	1.0	21.5
	Large	33 / 1	1	5.9	441.8

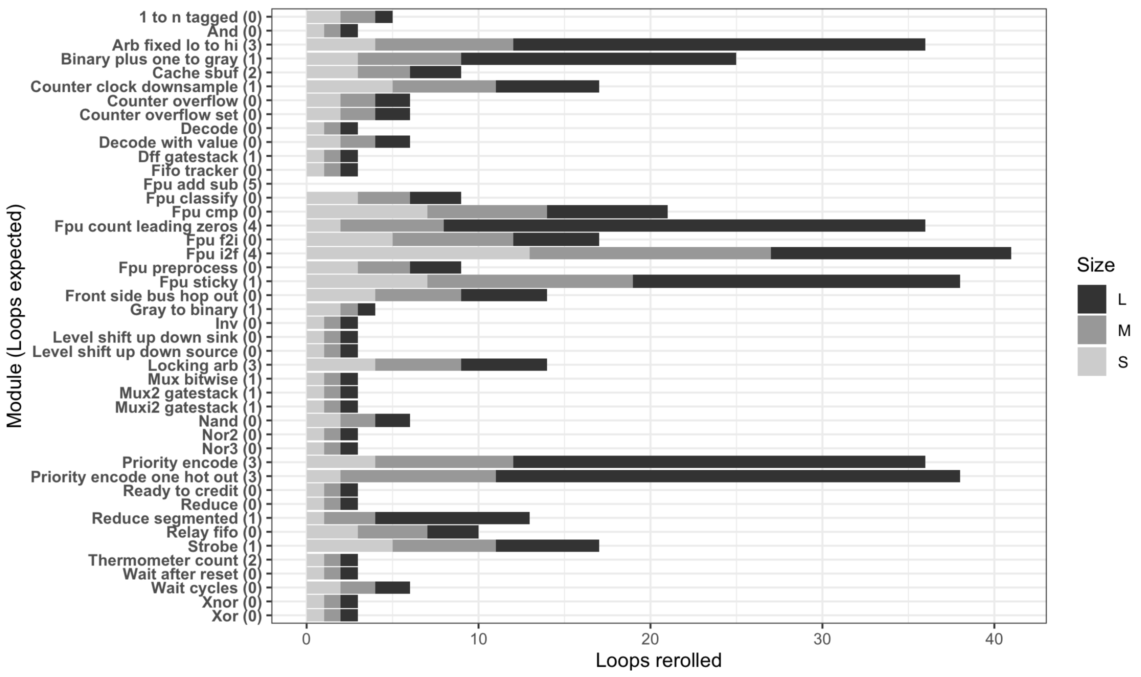


Figure 3.11: Number of loops rerolled across all BaseJump benchmarks (across all parameterizations). The number of expected loops present in the SystemVerilog source code is shown in parentheses next to the module name.

rerolled 28 of the 31 loops identified, the most rerolled in our evaluation. Also note that the BaseJump STL modules have instances where the original SystemVerilog code has no loops, but after analyzing the netlist our tool finds opportunities to reroll loops. Overall, with the exception of "Fpu add subtract", our tool identified and rerolled at least one loop in every module (often more).

Discussion

Considering the original HDL code in our benchmarks contained few loops, it is noteworthy that our tool often found and rerolled many more loops. There are two explanations for this phenomenon: (1) The HDL code may explicitly repeat the logic instead of parameterizing it over a loop. The BaseJump STL modules with zero expected loops often do this in the original SystemVerilog code. Nevertheless, this point is helpful to the decompiler user for understanding what the repeated operation is and how it is parameterized. (2) As noted in Section 3.3.2, when hardware synthesis lowers HDL code to a netlist, common higher-level operations in the original HDL code are expanded into chunks of lower-level repeated logic, increasing the loop identification count. The lowering process introduces non-obvious looping behavior that was not present in the higher-level design. Nonetheless, our tool identifies these repeated wire operations and attempts to reroll them. If successful, this kind of loop uncovers a repeated operation that is likely part of some larger structure that was lowered during synthesis. For these kinds of loops we argue this is beneficial for the decompiler user as the rerolled loop identifies some repeated logic (out of a large graph of similar nodes) and generates a concise representation in high-level code.

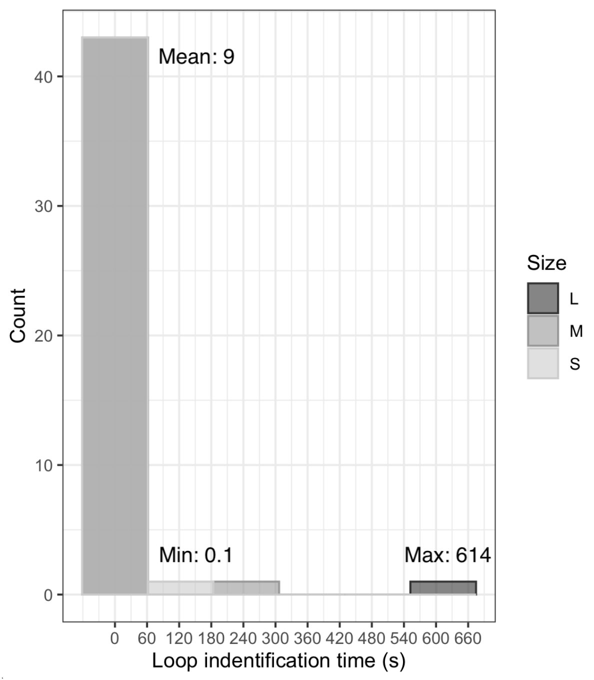
There are instances where we reroll loops that do not cover all of the iterations of the original loop. Either the first or last iterations may be missing. This mismatch is due to the limitations of our tandem repeat analysis during loop identification where token sequences must be *exact* matches. In some cases, the first or last iteration of a repeat may differ in a way that results in a different token sequence from the other repeats.

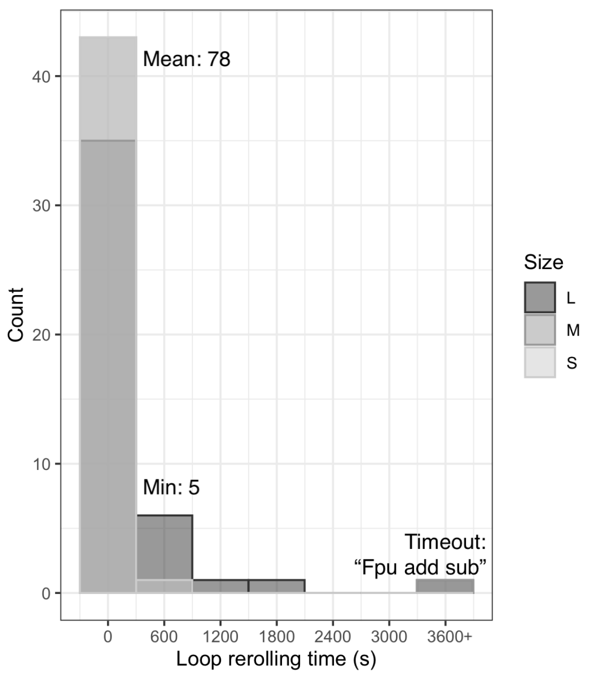
Based on our evaluation we find that nested loops in the original HDL will translate to larger non-nested loops in the decompiled HDL. For instance, the "Priority encode one hot out" benchmark actually has a nested loop inside one of its submodules. However, after conversion to Verilog and then to BLIF, the nested loops are essentially unrolled and the resulting netlist loses any notion of it. In this case, the best our technique recovers is a single loop with a larger body.

For netlist linearization, our evaluation empirically shows that a topological sort works well in practice. Alternative approaches to linearizing a netlist are interesting to consider (and might be part of future work). While we rely on the PyRTL compiler for netlist linearization, the utility of the topological sort is not limited to PyRTL-produced netlists as we also evaluate netlists generated from SystemVerilog code. These netlists have also undergone optimization passes in Yosys before being output to BLIF, showing that loop identification is also effective in the presence of optimizations.

Loop identification and rerolling times are shown in Figures 3.12a and 3.12b, respectively. One limitation of our loop rerolling tool comes from the constraints it sends to the SMT solver. For large netlists, solving times dramatically increase from a few seconds to minutes to over an hour. As Figure 3.12b shows, the "Fpu add sub" module timed out (mainly due to the netlist's size). 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 [32]. To scale to larger netlists, our tool needs to overcome this bottleneck at the program synthesis stage.

While the benchmarks in our evaluation are small, their module-level behavior is representative of the kinds of components used in real-world hardware designs—this





(a) (b)

Figure 3.12: (a) Histogram of loop identification performance across all BaseJump benchmarks. (b) Histogram of loop rerolling performance across all BaseJump benchmarks. The two points in the “3600+” bin indicate a timeout for the “Fpu add subtract” module sizes “medium” and “large”.

is one motivation for choosing the BaseJump STL. To scale module-level hardware loop rerolling to larger designs, there are techniques that can infer module boundaries in large netlists [86]. These techniques are orthogonal and complementary to our work in hardware decompilation in that they can be used to decompose a netlist into modules which then can be decompiled by our technique.

3.6.2 Transpilation Between HDLs

Since our hardware decompilation tool operates over a common IR, Maki, as well as outputs SystemVerilog and PyRTL, it enables automated translation between HDLs. For example, we can take a design that starts in SystemVerilog, synthesize it to a netlist, and decompile it into PyRTL code. The PyRTL code for the rerolled accumulator in Figure 3.9b is one example of transpilation from SystemVerilog. All of the PyRTL benchmarks in our evaluation can be synthesized and decompiled into SystemVerilog. The same is true for decompiling the SystemVerilog benchmarks into PyRTL.

As Maki is a small language, we mapped all Maki constructs to equivalent constructs in SystemVerilog and PyRTL. One point is that Maki is closer in design to PyRTL than SystemVerilog. Since the clock is implicit in Maki and PyRTL, we add a clock to the exported SystemVerilog code and wrap any sequential logic into a `always_ff @(posedge clk)` block where `clk` is the added clock. All combinational logic is wrapped in a `always_comb` block.

3.6.3 Speeding Up Simulation Time

In this section we compare simulation times for the BaseJump STL modules that successfully rerolled loops against their original netlists. We run the simulations using Verilator, an open-source SystemVerilog simulation tool [87]. For each module, we supply pseudorandom inputs to the netlist and decompiled HDL code.

Figure 3.13 presents the speedups for simulating each module. For a majority of the modules, hardware loop rerolling speeds up simulation time, with “Fpu cmp” seeing the largest speedup at 30x. Overall, the mean speedup is 6x. However, smaller modules such as “And” and “Xor” did not gain any speedup. The reason for the slower times ties back to some of the limitations of our approach. In particular, netlist linearization can

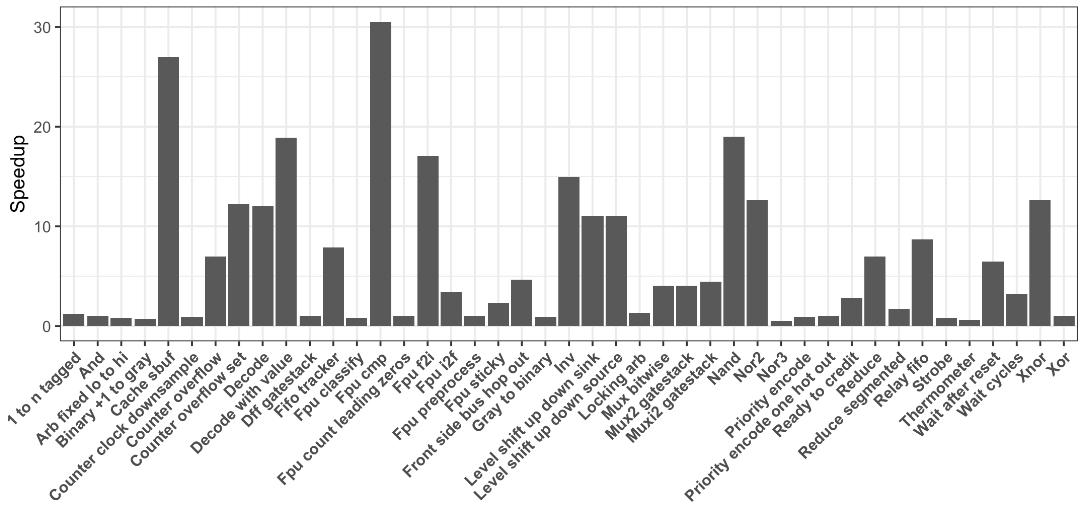


Figure 3.13: Speedups in Verilator simulation times across all "large" versions of the BaseJump STL modules that successfully rerolled loops compared to the original netlist.

affect the performance of rerolled loops in Verilator simulation. For instance, if a netlist linearization shifts the bit-select for a wire vector by 2 from the original monotonic ordering, our loop reroller will still reroll it into a loop. However, instead of referencing the bit-select with loop index `i` the synthesizer generates a more complicated arithmetic expression (e.g., `(i + 2) % n`, for a wire vector with bitwidth `n`). While still a correct loop with respect to the original netlist, the loop eludes easily applicable optimizations in Verilator. We can overcome these slowdowns in some cases by rewriting single-operation loops into one-line bitwise operations where feasible (e.g., `c = a & b`).

3.6.4 Artifact Compaction

To evaluate the benefits of loop rerolling for decompilation, we also measured artifact compaction. Since the goal of decompilation is to produce HDL code, we compare the rerolled code translated to PyRTL with the original Maki code translated to PyRTL without loop rerolling. We record the size of the artifact in bytes after the it is compressed with `gzip` on the highest compression level.

Overall, as parameter sizes grow, so does the degree of compaction between the rerolled code and the netlist. For some designs this is a significant difference, seeing up to 90% artifact compaction (the "large" versions of the PyRTL barrel shifter and iterative ripple-carry adder) and 39% compaction on average across the entire benchmark suite.

Loop rerolling alone makes a sizable impact here. However, a few outliers, typically smaller netlists, do not benefit as much from loop rerolling. For instance, as one of the smallest netlists, the BaseJump "Fifo tracker" module actually grew in size after rerolling. Due to the small size of the original netlist, loop rerolling has a proportionally small effect.

3.7 Related Work

3.7.1 Netlist Reverse Engineering

Research in this area presents techniques to recover module functionality, control logic, and data flow from a netlist graph. Most netlist reverse engineering work focuses on security scenarios, such as finding Trojans in digital circuits. There are two primary approaches for analyzing netlists: structural and functional. A structural analysis considers the shape, or topology, of the circuit to identify subcircuits [88] and recover control logic [89, 90, 91].

Functional analyses recover data flow and match subcircuits to templates of commonly used components. These analyses leverage QBF/SAT solvers to identify library components and word-level data paths [92, 93, 94, 95]. Other work identifies high-level blocks through graph embeddings and connectivity information [96]. Subramanyan et al. combine structural and functional analyses for reverse engineering circuits—first identifying submodule boundaries using a structural analysis, then mapping potential modules to a component template library via functional analysis [86].

These techniques in netlist reverse engineering work by producing a more structured netlist graph or finite-state machine annotated with higher-level constructs—as opposed to generating HDL code. The analogy to software binary reverse engineering is akin to recovering a control-flow graph and annotating it without decompiling to C code.

Further, work in netlist reverse engineering focuses on extracting structural information of the circuit, but not necessarily recovering the HDL code that synthesized the netlist. Some recent work makes the step to recovering register-transfer level (RTL) code [97, 98], but does not recover higher-level programming abstractions as our work does. We differentiate hardware decompilation from previous work that only recovers RTL code. Hardware decompilation lifts low-level details in the netlist to higher-level programmatic abstractions found in HDL code (such as loops, procedures, modules, etc).

3.7.2 Program Synthesis

Recent research has also used program synthesis techniques to automatically generate HDL code. Sketchilog generates Verilog code given a sketch, but is limited to combinational circuits [36]. VeriSketch is another sketch-based HDL code generation tool that uses CEGIS and information flow tracking to synthesize combinational and sequential circuits that adhere to a set of security properties [37]. Both of these tools focus on the design aspects of hardware, whereas our work comes from the opposite direction with decompilation.

Although in a different domain, another area of research conceptually related to our work uses rewrite-driven equality saturation to find loops in 3D geometric models [99]. The motivation is similar in that decompiled low-level triangle meshes used in 3D printing are large and unstructured. Instead of syntax-guided program synthesis, Nandi et al. use rewrite rules via an equality saturation engine to reroll loops in a DSL for Constructive Solid Geometry. Using an equality saturation engine such as egg [100], a rewrite-driven approach may improve synthesis times in our loop rerolling tool.

3.7.3 Software Loop Rerolling

Research in software loop rerolling focuses on rerolling for code size reduction, targeting resource-constrained environments [67, 68, 69, 70]. Modern compilers also have loop rerolling strategies. LLVM implements a heuristic-based loop rerolling pass which operates over LLVM IR and rerolls partially unrolled iterations of single-block loops. No previous work in loop rerolling uses program synthesis techniques to reroll loops.

Recent work looks at loop rerolling at the source-code [71] and binary level [72]. RoLAG rerolls loops by aligning blocks of straight-line code in SSA form [71]. Aligned SSA graphs correspond to isomorphic code and are then rolled into a single loop. RollBin rerolls loops at the binary level using a custom data-dependency analysis to handle shuffled instructions and loop-carry dependencies [72]. RollBin identifies loops and infers their unrolling factor by observing memory accesses.

Our work differs from software loop rerolling because the semantics and execution model of HDLs and netlists are different from that of software and binary executables. Importantly, the *semantics* of a loop in an HDL is different from loops in software. Unlike the existing work, our work specifically uses program synthesis to generate rerolled higher-level code—using symbolic evaluation to guarantee that the rerolled loop code is semantically equivalent to the original netlist.

3.8 Conclusion

In this chapter we defined and explored a new problem—hardware decompilation. This problem is the task of lifting a low-level netlist back to structured, high-level HDL code. It is a large problem, so in this chapter we tackle the first step for decompiling high-level HDL code with loops. Inspired by techniques in software clone detection, we find candidate loops in netlists using a token-based analysis and sequence matching algorithms. With loop information, we generate a sketch of the code with rerolled loops and send it to a program synthesis tool that can reason about hardware designs. We evaluate hardware loop rerolling on a set of SystemVerilog and PyRTL hardware design benchmarks, noting the number of loops successfully identified and rerolled, and its impact on transpilation between HDLs, faster simulation times over netlists, and artifact compaction.

This chapter lays the groundwork for future research in hardware decompilation. The hardware-oriented program synthesis tool we developed opens the door to an entire class of problems that can be solved through this technique. In the future we envision developing more circuit-based analyses to recover other high-level programming features and extending the program synthesis tool to decompile those back into HDL code.

Chapter 4

A Memory Design Language for Automated Memory Mapping

4.1 Introduction

Hardware description languages (HDLs) such as Verilog drive SoC development. When writing behavioral HDL code, engineers need to target different technologies to support different deployment platforms (for simulation, ASIC, FPGA, etc.). However, industry-standard HDLs are not equipped to cleanly target these different platforms from a single, generic implementation. As a result, a standard approach is for engineers to duplicate parts of the code into separate blocks for each technology, implementing the same functional behavior but targeting the specialized semantics of each specific technology. High-level source code is often littered with `ifdef` blocks, akin to software targeting specific features of different ISAs (x86, ARM, etc.), but exacerbated by the diversity and divergence of the many ASIC and FPGA technologies that engineers need to target (see Figure 4.1).

This practice is particularly problematic for memories, where the structure and se-

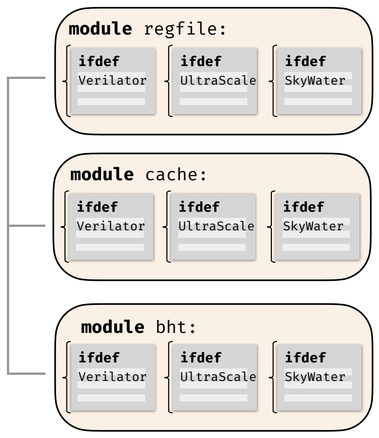


Figure 4.1: An illustration of manual memory technology mapping using `ifdef` blocks in each memory module. Each memory module requires `ifdef` blocks for each targetable technology which each block matching the semantics of that memory.

mantics of different memory technologies vary widely with respect to deployment platform. There have been attempts to address this problem — the state-of-the-art, called *memory inference*, is a syntactic template-based matching approach where the designer writes HDL code following syntactic patterns dictated in technology-specific vendor manuals, which allows automated mapping for that vendor's technology [101]. However, memory inference only works for one specific technology at a time (i.e., the developer can only use it to target one technology, not a set of technologies) and adoption is limited to FPGAs. So while memory inference helps engineers in a restricted way with one aspect of the problem, they still need to develop separate solutions for other technologies, including for different FPGAs which need separate templates. These practices result in a brittle code base, with repeated but subtly different technology-specific blocks of code, increasing the burden on verification, agility, and extensibility.

In this work, we focus on the problem of automated technology mapping for memories. Our insight is incorporating an abstract memory representation into the HDL which is "write once, map anywhere", meaning the memory representation has a rich enough semantics to target all of the relevant technologies behind a single, generic interface. We implement this abstract memory representation as an extension of the Python-based HDL PyRTL [25], named ELEPHANT[1], which performs automated technology mapping for memories across multiple platforms and technologies using a single, common HDL-level memory description.

Of course, there are already a wealth of existing hardware designs written in other HDLs which target specific memory technologies. Instead of porting these existing code bases to ELEPHANT, we present a hardware decompilation-based memory identification technique which lifts memories from a gate-level netlist to ELEPHANT, enabling technology *re-targeting*. A *netlist* is a graph representation of the wires and logical gates describing a digital circuit and serves as a common, language-agnostic representation format for hardware designs. The key insight of our approach enables technology targeting and re-targeting through a unified set of algebraic rewrite rules, which specify both memory *compilation* and *decompilation*. One direction ($\rightsquigarrow$) specifies memory technology mapping, where the opposite direction ($\leftsquigarrow$) specifies memory decompilation.

We show that ELEPHANT effectively targets five backends for three different technology platforms—for simulation, ASIC, and FPGA—from a single interface over a suite of representative designs. Further, we evaluate our decompilation-based memory lifting technique against the state of the art, demonstrating higher accuracy, inferring more semantics of the memories, and enabling automated re-targeting. Our major contributions are:

- An abstract memory representation called ELEPHANT which targets three common deployment platforms, concretely realized as an extension to the PyRTL HDL (Section 4.3).

[1]Our abstraction can be added to any HDL; we choose PyRTL to make a concrete implementation for evaluation.

- A hardware decompilation-based memory lifting technique, implemented as a bespoke equality saturation procedure, capable of identifying memories (specialized for various technologies) in gate-level netlists that the state of the art cannot (Section 4.4).
- We show that our technique enables technology re-targeting, starting from a design mapped to one technology and re-targeting it to another. We demonstrate the end-to-end re-targeting flow on real-world open-source SoC designs (Section 4.5).

We first describe the necessary background with a typical design scenario for an engineer tasked with targeting their HDL code to several technologies, along with its attendant problems (Section 4.2). In the rest of the chapter we show how to avoid this scenario by using a unified abstract memory representation.

4.2 Background

In this section, we provide background on the problem domain through a motivating example. Consider a scenario where a developer is implementing a core for a System-on-a-Chip (SoC). Such a core may require a number of memory blocks for instructions, data, caches, etc. As a small but non-trivial example, we focus on a register file—a memory block with 32 entries of 32-bit values.

In behavioral Verilog, it is straightforward to implement the register file as an array of 32-bit registers (as `regfile` in the code below). The developer expresses reading and writing to the register file using the familiar array notation. In our scenario, the target architecture specifies RISC-style instructions which operate over two registers, so the register file needs two read ports and one write port. This style of behavioral Verilog suffices for simulation in, for example, the open-source simulator Verilator [87].

```
reg [31:0] regfile[31:0];
// Read logic
assign r0_data_o = regfile[r0_addr_r];
assign r1_data_o = regfile[r1_addr_r];
// Write logic
always @(posedge clk)
  if (w_valid_i)
    regfile[w_addr_i] <= w_data_i;
```

However, the developer then learns they need to deploy this core to a particular FPGA. The FPGA only supports a particular kind of memory, a block RAM (BRAM). Verilog does not have built-in support for BRAMs. Instead, the FPGA vendor provides a BRAM module as an intellectual property (IP) library, and through a detailed manual documents how to interface with the module and instantiate it in Verilog code. The developer then pores through the vendor manual and writes what they think is the correct way to instantiate a BRAM for the register file in their design. For context, the UltraScale Architecture Memory Resources manual from AMD/Xilinx is 139 pages in length with 80 pages on BRAMs [102]. If the developer is "lucky" (read: likes to use a GUI), the vendor may provide a configuration wizard to generate these wrappers, but it still takes careful use and consultation with the vendor manual to choose the correct settings and parameters. An example is given below:

```
bram_2r1w_wrapper #(
   .DEPTH         (32),
   .ADDR_WIDTH    (5),
   .DATA_WIDTH    (32)
)   regfile (
   .MEMCLK  (MEMCLK), .RESET_N (RESET_N),
   .CEA     (CEA),    .AA      (AA),
   .AB      (AB),     .RDWENA  (RDWENA),
   .CEB     (CEB),    .RDWENB  (RDWENB),
   .BWA     (BWA),    .DINA    (DINA),
   .DOUTA   (DOUTA),  .BWB     (BWB),
   .DINB    (DINB),   .DOUTB   (DOUTB) );
```

To support both the Verilator simulation and FPGA deployment, the developer then splits register file source code into two parts using `ifdef` blocks, shown as follows:

```
`ifdef VERILATOR
reg [31:0] regfile[31:0];
  ...
`endif

`ifdef UltraScale
bram_2r1w_wrapper #(...)
  ...
`endif
```

But the developer is not finished yet. Next they are told that they also need to support an ASIC deployment. Again, Verilog does not explicitly support the kinds of memory technologies used in ASIC designs, specifically SRAMs. This time, the developer reaches for an open-source solution—the Basejump STL [83], which is an open-source standard component library for common hardware components providing process-specific implementations for memory blocks in supported technologies. After consulting the Basejump STL documentation, the developer adds a third `ifdef` block to their register file implementation, instantiating yet another technology-specific module with its own set of parameters and port mappings:

```
`ifdef VERILATOR
reg [31:0] regfile[31:0];
  ...
`endif
```

```
`ifdef UltraScale
bram_2r1w_wrapper #(...)
  ...
`endif

`ifdef ASIC
bsg_mem_2r1w_sync #(...)
regfile (.w_addr_i(w_addr_i), .w_data_i(w_data_i), ...)
`endif
```

Overall, this implementation is brittle in three dimensions. (**1**) The `ifdef` blocks each support only one particular technology. If the code base needs to support another FPGA device, the developer needs to add *another* `ifdef` to instantiate the BRAM supported for that particular FPGA and decide how to differentiate between the different, sometimes overlapping macros they have introduced. (**2**) The implementation here is specific to *one* memory block in the overall SoC design; there are generally many more memory blocks, each with their own particular semantics and configurations, for which the developer needs to repeat this process over and over. (**3**) The implementation is brittle to design changes. If the architecture changes, e.g., to support more complex instructions, the developer may need to update the register file to support three read ports. This change requires updating all of the `ifdef` blocks for the register file to reflect the change in port mapping. This issue arises with optimizations as well. "Write-to-read forwarding" forwards the value of a write operation to the read port if they share the same address which bypasses an unnecessary read operation. Some memory technologies support this optimization and can enable it through a parameter. Others may not support the optimization and the developer needs to implement the logic manually.

Our solution alleviates the developer from this convoluted situation. In Elephant, the developer only writes a single generic interface for each memory block that can be automatically targeted to all of these technologies. Next, we present Elephant as a language, its rewrite-driven semantics, and show how we can achieve a more robust implementation.

4.3 Elephant for Automated Memory Technology Mapping

In this section, we present the HDL extension we call Elephant which defines a rich memory abstraction for use in memory technology mapping. Because Elephant is an HDL extension it is designed to be added to an existing HDL, and thus we present the language as a *subset* of an HDL, excluding any HDL features that are not directly relevant to memory semantics. In this formal presentation, we give an abstract syntax for Elephant which is agnostic to any particular HDL, but in Section 4.5.1 we describe how we extend PyRTL with the memory abstraction defined here.

4.3.1 Elephant Grammar

Figure 4.2a presents the grammar for Elephant. The syntax only has constructs relevant for the abstract memory representation. The top-level construct in Elephant is a memory. A memory itself has a number of read ports, write ports, and options indicating implementation-specific optimizations and configurations. $\langle update \rangle$ and $\langle expr \rangle$ terms only appear through lowering—that is, "elaborating"—a memory abstraction.

Figure 4.2b presents signatures for key constructs in Elephant. The `ReadPort` construct and `WritePort` construct are tagged unions, with fields indicating each part of the port: an enable signal (`en`), an address value (`addr`), a `data` value (as output for `ReadPort`, and input for `WritePort`), and an optional `mask` for write data. Option types

$$\begin{aligned}
\langle \mathit{memory} \rangle &::= \textbf{Memory(}\langle \mathit{port} \rangle^{+}\textbf{,}\langle \mathit{option} \rangle^{+}\textbf{)} \mid \langle \mathit{update} \rangle \\
\langle \mathit{port} \rangle &::= \textbf{ReadPort(}\langle \mathit{var} \rangle\textbf{,}\langle \mathit{var} \rangle\textbf{,}\langle \mathit{var} \rangle\textbf{)} \mid \textbf{WritePort(}\langle \mathit{var} \rangle\textbf{,}\langle \mathit{var} \rangle\textbf{,}\langle \mathit{var} \rangle\textbf{,}\langle \mathit{var} \rangle\textbf{)} \\
\langle \mathit{option} \rangle &::= \textbf{LatchLastRead} \mid \textbf{WriteReadForward} \mid \textbf{Sync} \\
\langle \mathit{update} \rangle &::= \langle \mathit{update} \rangle\ \textbf{;}\ \langle \mathit{update} \rangle \mid \langle \mathit{var} \rangle\ \textbf{:=}\ \langle \mathit{expr} \rangle \mid \langle \mathit{var} \rangle\textbf{[}\langle \mathit{var} \rangle\textbf{]}\ \textbf{:=}\ \langle \mathit{expr} \rangle \\
&\quad \mid \langle \mathit{expr} \rangle\ \textbf{?}\ \langle \mathit{update} \rangle \\
\langle \mathit{expr} \rangle &::= \textbf{Mux(}\langle \mathit{expr} \rangle\textbf{,}\langle \mathit{expr} \rangle\textbf{)} \mid \textbf{Demux(}\langle \mathit{expr} \rangle\textbf{,}\langle \mathit{expr} \rangle\textbf{)} \mid \textbf{\{}\ \langle \mathit{expr} \rangle^{+}\ \textbf{\}} \\
&\quad \mid \langle \mathit{expr} \rangle \wedge \langle \mathit{expr} \rangle \mid \langle \mathit{expr} \rangle \vee \langle \mathit{expr} \rangle \mid \neg\langle \mathit{expr} \rangle \mid \langle \mathit{var} \rangle\textbf{[}\langle \mathit{var} \rangle\textbf{]}
\end{aligned}$$

(a)

$$\begin{aligned}
&\texttt{In(n), Out(n), Reg(n) ::}\ \mathbb{B}^{n}\texttt{, Option ::}\ \mathbb{B}^{1} \\
\\
&\texttt{ReadPort\ \ :: \{en: In(1), addr: In(}m_r\texttt{), data: Out(}n_r\texttt{)\}} \\
&\texttt{WritePort :: \{en: In(1), addr: In(}m_w\texttt{), data: In(}n_w\texttt{), mask: In(}n_w\texttt{)}^{?}\texttt{\}} \\
&\texttt{Memory\ \ \ \ :: [ReadPort]}\times\texttt{[WritePort]}^{?}\times\texttt{[Option]}\rightarrow\texttt{[}x_i\texttt{: Reg(n) | i = 0..}2^{m}\texttt{] | m = }m_r \\
&\quad\texttt{= }m_w\texttt{, n = }n_r\texttt{ = }n_w \\
\\
&\texttt{Mux\ \ \ ::}\ \mathbb{B}^{n}\times\mathbb{B}^{2^{n}}\rightarrow\mathbb{B}^{1} \\
&\texttt{Demux ::}\ \mathbb{B}^{1}\times\mathbb{B}^{n}\rightarrow\mathbb{B}^{2^{n}}
\end{aligned}$$

(b)

Figure 4.2: (a) The grammar for ELEPHANT. `m[a]` represents addressing, either addressing a row from a memory, or addressing a particular bit from a variable. `e ? x := y` indicates a *guarded* update, which only executes if `e` evaluates to 1. `{ a, b, c }` represents concatenation of the values of expressions `a`, `b`, and `c`. (b) A selection of type signatures for key constructs in ELEPHANT. The base type is a bit-vector, indicated by $\mathbb{B}^n$. `In`, `Out`, and `Reg` are constructors for input and output ports, and registers. `n` and `m` are natural numbers.

are indicated with '?'—for instance, a `Memory` does not require a `WritePort` (it can be read only). Square brackets indicate an array (one or more)—for example, a `Memory` can have multiple `ReadPorts`. The type signature for `Memory` contains two refinements to ensure a well-typed memory: The outer refinement ensures that the address and data fields for the read and write ports are the same size, `m` and `n`, respectively ($m = m_r = m_w,\ n = n_r = n_w$). The refinement on the return value, the array of registers, asserts that the number of registers is proportional to the address size ($[x_i\text{: Reg}(n) \mid i = 0..2^m]$).

Following the example in Section 4.2, we describe that same proposed register file in Elephant:

```
r0: ReadPort :: {en: In(1), addr: In(5), data: Out(32)}
r1: ReadPort :: {en: In(1), addr: In(5), data: Out(32)}
w: WritePort :: {en: In(1), addr: In(5), data: In(32)}
rf := Memory([r0, r1], [w], [Sync])
```

This code expresses the register file as an abstract memory block with two read ports and one write port, and the `Sync` option enabled indicating that reads to the memory are synchronous. From this representation, we will show how Elephant enables automated memory technology mapping, replacing the lines of Verilog and `ifdef` blocks with a single `Memory` instance. The information encoded in the type signatures for the memory, including its read and write ports, will become important for elaboration, described next.

4.3.2 Elaboration in Elephant

Program execution in Elephant is different than a typical language. Rather than evaluate data flowing from the input ports to the output ports, Elephant *generates a*

$$\frac{\begin{array}{c} r_i : \textbf{ReadPort} \quad mem \mathrel{\textbf{:=}} \textbf{Memory}([r_0, ..., r_n], \vec{w}, \vec{o}) \\ i = 0, ..., n \quad mem' \mathrel{\textbf{:=}} \textbf{Memory}([r_0, ..., r_{n-1}], \vec{w}, \vec{o}) \end{array}}{mem = (mem' \mathbin{\textbf{;}} r_n.en \mathbin{\textbf{?}} r_n.data \mathrel{\textbf{:=}} mem'[r_n.addr])}\textsc{ReadPort}$$

$$\frac{\begin{array}{c} mem_i = \textbf{Mux}(r.addr,\ \vec{x}[i]) \\ r : \textbf{ReadPort} \quad \vec{x} : \overrightarrow{\textbf{Reg}}(m) \quad i = 0, ..., m \end{array}}{mem[r.addr] = \{\, mem_0, ..., mem_{m-1} \,\}}\textsc{ReadAddress}$$

$$\frac{\begin{array}{c} w_i : \textbf{WritePort} \quad mem \mathrel{\textbf{:=}} \textbf{Memory}(\vec{r}, [w_0, ..., w_n], \vec{o}) \\ i = 0, ..., n \quad mem' \mathrel{\textbf{:=}} \textbf{Memory}(\vec{r}, [w_0, ..., w_{n-1}], \vec{o}) \end{array}}{mem = (mem' \mathbin{\textbf{;}} w_n.en \mathbin{\textbf{?}} mem'[w_n.addr] \mathrel{\textbf{:=}} w_n.data)}\textsc{WritePort}$$

$$\frac{\begin{array}{c} e \mathrel{\textbf{:=}} \textbf{Demux}(w.en, w.addr) \\ d_j = e[j] \mathbin{\textbf{?}} x_j[i] \mathrel{\textbf{:=}} w.data[i] \\ w : \textbf{WritePort} \quad \vec{x} : \overrightarrow{\textbf{Reg}}(m) \quad i = 0, ..., m \quad j = 0, ..., n \end{array}}{w.en \mathbin{\textbf{?}} mem[w.addr] \mathrel{\textbf{:=}} w.data = d_0 \mathbin{\textbf{;}} ... \mathbin{\textbf{;}} d_{n-1}}\textsc{WriteAddress}$$

$$\frac{s : \mathbb{B}^1 \quad a : \mathbb{B}^1 \quad b : \mathbb{B}^1}{\textbf{Mux}(s,\ \{\, a, b \,\}\,) = (a \ \wedge\ \neg s) \ \vee\ (b \ \wedge\ s)}\textsc{Mux}$$

$$\frac{s : \mathbb{B}^{log_2 n} \quad t : \mathbb{B}^1 \quad a : \mathbb{B}^n \quad b : \mathbb{B}^n}{\textbf{Mux}(\,\{\, s, t \,\}\,,\ \{\, a, b \,\}\,) = \textbf{Mux}(t,\ \{\, \textbf{Mux}(s, a),\ \textbf{Mux}(s, b) \,\}}\textsc{Mux2N}$$

$$\frac{x : \mathbb{B}^1 \quad s : \mathbb{B}^1}{\textbf{Demux}(x, s) = \{\, x \ \wedge\ \neg s,\ x \ \wedge\ s \,\}}\textsc{Demux}$$

$$\frac{x : \mathbb{B}^1 \quad s : \mathbb{B}^{log_2 n} \quad t : \mathbb{B}^1 \quad \{\, d_0,\ d_1 \,\} \mathrel{\textbf{:=}} \textbf{Demux}(x, t)}{\textbf{Demux}(x,\ \{\, s, t \,\}\,) = \{\, \textbf{Demux}(d_0, s),\ \textbf{Demux}(d_1, s) \,\}}\textsc{Demux2N}$$

$$\frac{\begin{array}{c} w : \textbf{WritePort}(mask : \textbf{In}(n)) \quad mem : \textbf{Memory} \quad i = 0, ..., n \\ read = mem[w.addr] \quad m_i = \textbf{Mux}(w.mask[i],\ \{\, read[i], w.data[i] \,\}\,) \end{array}}{w.en \mathbin{\textbf{?}} mem[w.addr] \mathrel{\textbf{:=}} w.data = w.en \mathbin{\textbf{?}} mem[w.addr] \mathrel{\textbf{:=}} \{\, m_0, ..., m_{n-1} \,\}}\textsc{WriteMask}$$

Figure 4.3: The core elaboration rules for Elephant. A rule $a = b$ means a term a rewrites to an equivalent term b. For each rule, the bottom half defines the term rewrite while the top half specifies conditions that must hold for the rewrite to occur.

description of the corresponding logic obtained from refining the abstract memory given a particular configuration of ports and parameters. This pattern is common in HDLs such as Chisel [103] and PyRTL [25] which also use this "elaboration through execution" strategy.

We formalize elaboration as rewrites over terms in Figure 4.3. The goal of elaboration is to start from an abstract memory interface, and through successive rewrites, refine the abstract memory into a logical implementation. Different technology targets dictate how to refine the abstract memory. Thus, the resulting implementation may be expressed at varying levels—from behavioral all the way to gate-level logic. In this way, Elephant is a multi-level representation; it can express both abstract memory blocks and low-level logic in the same language. The elaboration rules make use of this multi-level representation. The core rules assume full elaboration to gate-level logic (as for simulation); Section 4.3.3 will discuss how to target different technologies. Next, we outline the core elaboration rules for Elephant, starting with read logic.

Read Logic

The core rules for read logic are ReadPort and ReadAddress in Figure 4.3. The ReadPort rule pulls an abstract read port out of a memory, and expresses it behaviorally — e.g., `en ? data := mem[addr]`. The ReadAddress rule takes this one step further by expanding a behavioral read (`mem[addr]`) into the underlying logic, generating a multiplexer tree which selects the correct register according to the address.

We continue with the register file example, showing how to apply the rewrites for the example following the general rules. Considering the two read ports for the register file, the goal for rewriting is to produce two $\langle update \rangle$ statements for each of the output ports in `r0` and `r1`. Thus, through two applications of ReadPort, we obtain:

```
rf = r0.en ? r0.data := rf'[r0.addr] ;
      r1.en ? r1.data := rf'[r1.addr]
```

The resulting code expresses one level of refinement from the abstract memory to a statement explicitly defining an output port in terms of addressing the memory as a whole. We can continue to lower the read logic, next rewriting the memory addressing logic for the read port via the ReadAddress rule. Consider the expression `rf[r0.addr]`:

```
Mux : 𝔹⁵ × 𝔹³² → 𝔹¹
x0, ..., x31 : Reg(32)
rf[r0.addr] = { Mux(r0.addr, x0[0], ..., x31[0]),
                 Mux(r0.addr, x0[1], ..., x31[1]), ...,
                 Mux(r0.addr, x0[31], ..., x31[31]) }
```

The semantics “hidden” in the left-hand side expand into the multiplexing logic on the right. Internally, the memory block is a matrix of one-bit registers. The multiplexing logic uses the address to select to corresponding bit of the addressed row. That is, it generates the concatenation of `m` one-bit values which make up the full n-bit row of the selected data in memory.

Rules Mux2n and Mux then continually decompose the multiplexing logic. That is, a $2n$-to-1 multiplexer can be decomposed into two n-to-1 multiplexers which feed into a 2-to-1 multiplexer (via Mux2n). This decomposition continues until it reaches the base case of a 2-to-1 multiplexer (via Mux), expressing the logic directly using Boolean logic gates.

Elaborating these rules over the full `Memory` lowers the memory block to purely logic gates and one-bit registers—a gate-level netlist. Note that the multiplexer trees for the read logic are a generic representation. Some technologies will “specialize” read logic forgoing the generic multiplexer tree. For example, an SRAM will select the read output via charge outputting onto the bitlines, selecting a single entry all at once. Next, we turn to the write logic.

Write Logic

Elaboration rules for write logic proceed similarly to read logic but moving in the "opposite" direction. The core rules for write logic are WritePort and WriteAddress in Figure 4.3. The WritePort rule pulls an abstract write port out of a memory, and expresses it behaviorally—`en ? mem[addr] := data`. The corresponding WriteAddress rule expands a behavioral write into the underlying logic; this time generating demultiplexer logic which enables writing to the correct register according to the address.

Continuing the register file example, `rf`, the write logic for its single write port expands to the following:

```
rf := Memory(_, [w], _)
rf = _ ; w.en ? rf'[w.addr] := w.data
```

Not shown are the previous applications of the ReadPort rule from Section 4.3.2. The pattern `_ ; s` is a shorthand for appending `s` to the list of update statements. Similar to the rules for multiplexers, the Demux2n rule rewrites a 1-to-$2n$ demultiplexer into two 1-to-n demultiplexers preceded by a 1-to-2 demultiplexer. The base case is a 1-to-2 demultiplexer, rewritten to logic gates via the Demux rule.

Data Masks

Write ports optionally accept masks which apply a given input mask to the write data before writing. The WriteMask rule in Figure 4.3 presents an elaboration for bit masks but masks of larger granularity can be supported too. The mask operates on the data input of the write port. The lowering logic generates a series of multiplexers which selects the i^{th} bit of the incoming write data only if the i^{th} bit of the mask is high.

Options

Elaboration rules for read logic, write logic, and masks comprise the core semantics for Elephant. The remaining rules are the options set in the memory interface. The options we present here are representative but not exhaustive; memories in Elephant can support more options by extending the set of elaboration rules.

Here, we explain **Sync**, **WriteReadForward**, and **LatchLastRead**. The **Sync** option models a synchronous memory, meaning the read data is not available until the next cycle. When lowering the design, this option requires adding a register (a_{sync} below) for each read port to store the input read address, as shown in the following elaboration rule:

$$\frac{r : \textbf{ReadPort} \quad r.addr : \mathbb{B}^n \quad a_{sync} : \textbf{Reg}(n) \\ mem : \textbf{Memory}(\textbf{Sync}) \quad s = (r.en \;\textbf{?}\; a_{sync} \;\textbf{:=}\; r.addr)}{r.en \;\textbf{?}\; r.data \;\textbf{:=}\; mem[r.addr] = s \;\textbf{;}\; r.data \;\textbf{:=}\; mem[a_{sync}]}$$

The option **WriteReadForward** is an optimization that forwards a written value to the read port if they are accessing the same address. Because writes are applied to the memory at the next clock edge, a read to that address will have to wait one cycle after the write takes effect. This optimization makes the read value available earlier instead of waiting for the write to complete. We express **WriteReadForward** in the following rule, the main additional logic is the multiplexer that checks if the read address and write address are the same:

$$\frac{r : \textbf{ReadPort} \quad w : \textbf{WritePort} \\ mem : \textbf{Memory}(\textbf{WriteReadForward})}{r.data = \textbf{Mux}(r.addr = w.addr,\ \{\ mem[r.addr], w.data\ \}\ )}$$

The last option **LatchLastRead** saves the enable signal and last read data in separate registers. The output port takes the last read data until another read is enabled. We express the rule as follows, where e_{llr} and d_{llr} are the new registers for the enable and last read data, respectively:

$$\frac{\begin{array}{c} r : \textbf{ReadPort} \quad mem : \textbf{Memory}(\textbf{LatchLastRead}) \\ e_{llr} : Reg(1) \quad d_{llr} : Reg(n) \quad r.data : \mathbb{B}^n \\ s = (e_{llr} \mathbin{\textbf{:=}} r.en \mathbin{\textbf{;}} d_{llr} \mathbin{\textbf{:=}} \textbf{Mux}(e_{llr},\ \{\ mem[r.addr], d_{llr}\ \}\ )) \end{array}}{r.data \mathbin{\textbf{:=}} mem[r.addr] = s \mathbin{\textbf{;}} r.data \mathbin{\textbf{:=}} d_{llr}}$$

These cover the three options that configure an abstract memory. This set can be extended by adding a new option and defining the corresponding rewrites.

4.3.3 Technology Targeting with Constraints

The "backend" of Elephant consists of passes that target particular technologies. Through elaboration, Elephant propagates information from read and write ports into a given abstract memory interface, along with timing-related features (i.e., options). Recall that there are three target platforms: simulation, ASIC, and FPGA, but multiple technologies within each platform. How much elaboration needs to happen depends on a given technology. Some technologies, for example, memories in the Basejump STL, bear a close resemblance to the abstract memory interface in Elephant and can be targeted with minimal elaboration. The general rewrite rules presented in Section 4.3.2 handle simulation at different levels of abstraction from behavioral all the way to gate-level netlist, which level depends on the target platform. Given the comprehensive elaboration rules, many different backends can be targeted.

The Elephant backend generates templated descriptions for a set of supported memory technologies (see Section 4.5.1 for the specific technologies). Further, due to the refinement-style elaboration and algebraic rewrite rules, we can develop backend passes that incorporate technology-specific constraints, enabling memory mapping for more complex scenarios. For example, a developer may need to map a `2r2w` memory to an FPGA that only supports dual-port BRAMs (`2rw`), a non-trivial mapping problem that may take extensive developer effort.

We developed a technique for mapping abstract memories in Elephant according to technology constraints. The input of this algorithm contains two parts: a high-level memory and a set of available memory technologies with their constraints. The technology library describes the specifications of each target memory, including data width, number of ports, different features like (a)synchronous read, latch last read, etc., and a user-defined cost. The output is a module with certain instantiations of the target memories and wrappers over them to meet the high-level memory interface.

Algorithm 3 describes how memory mapping works in two main phases. First, it applies a dynamic programming algorithm to compute the feasible port fitting plan for every memory with i read ports, j write ports, and k read-write ports given the technology library. The second phase is a greedy algorithm that deals with data width and features. It selects memories from the technology library that have enough capacity to hold the data and meet the features. It prefers memories with lower cost and those already have the required features, otherwise it wraps the memories with additional logic, e.g., a forwarding unit will be added when the underlying memory does not support it.

Let's consider an example of mapping an abstract memory with 2 read ports and 2 write ports to a technology library with only one type of BRAM which has 2 read-write ports and costs 1, regardless of their capacity and features. To compute a port fitting plan with minimal cost, the algorithm initializes a 3-dimensional array $f(i, j, k)$, where i, j, and k are the number of read, write, and read-write ports respectively, to $+\infty$. First, it sets $f(0, 0, 2) = 1$ since the target technology has 2 read-write ports. Then it starts from $f(2, 2, 0)$ and recursively searches for the plans. In order to implement $f(2, 2, 0)$, it uses two $f(3, 1, 0)$ by splitting the write ports. For each $f(3, 1, 0)$, it again uses three $f(1, 1, 0)$ by splitting the read ports, where $f(1, 1, 0) = f(0, 0, 2)$. Finally, we have $f(2,2,0) = 6$, which means we need to use 6 `2rw` BRAMs to implement this memory.

Algorithm 3 Dynamic programming algorithm for memory mapping with technology constraints.

```
1: procedure MemoryMap(abstract_mem, tech_lib)
2:     Initialize 3D DP table $dp[r][w][rw]$ with cost $+\infty$
3:     Populate $dp$ entries based on available physical memories in tech_lib
4:     for all valid $(r, w, rw)$ configurations do
5:         for each possible port split of read ports do
6:             Update $dp[r][w][rw]$ using minimum cost of split subproblems
7:         for each possible casting of read or write to readwrite ports do
8:             Update $dp[r][w][rw]$ based on transformed port configuration
9:     for each cast of readwrite to separate read and write ports do
10:        Update final $dp$ entry accordingly
11:    return Minimum cost plan in $dp[nr][nw][nrw]$
```

4.4 Memory Decompilation

In this section we present our memory decompilation technique with the goal of lifting memories in a gate-level netlist up to abstract memories in Elephant. Since netlists are a common artifact for distributing IP, memory decompilation helps designs written in other HDLs take advantage of Elephant's automated memory technology mapping capabilities. The technique we present starts from a gate-level netlist with single-bit registers (also called D-flip flops, or DFFs). We split the technique into four phases, each driven by rewrite rules in Elephant: (1) group registers by common enable signals; (2) lift write logic from gates to demultiplexers to behavioral writes; (3)

lift read logic from gates to multiplexers to behavioral reads; (4) lift behavioral reads and writes into an abstract memory. The key insight is that we decompile memories by using the same elaboration rules described in Section 4.3, only we flip the direction of the rewrite from $a \rightsquigarrow b$ to $a \leftsquigarrow b$.

4.4.1 Register Enable Signals

Because ELEPHANT is a multi-level representation, we can use it to express designs in terms of gate-level logic. This allows us to translate netlists directly into ELEPHANT and then rewrite the ELEPHANT code into higher-level abstractions according to the (reversed) elaboration rules. The first step is to group single-bit registers into wider registers. Sets of multi-bit registers of the same width will serve as candidates for memory decompilation in the later phases. To cut down on the search space of all DFFs in a netlist, we use the heuristic of only considering DFFs with enable pins. This heuristic is not required for our technique to work, it is simply a practical optimization; even if DFFs with enable pins are not specifically used in a netlist, there will normally be some kind of logic which enables reading and writing to registers inside of a memory block as a power saving mechanism. In ELEPHANT, we represent a DFF with an enable pin as `q := e ? d`, where `e` is the enable and `d` is the incoming data. Then, we group registers by common enable signals through a basic algebraic rewrite which distributes enables over register terms in ELEPHANT:

$$\frac{e,\ d_0,\ d_1 : \mathbb{B}^1 \quad q_0,\ q_1 : Reg(1)}{e\ \mathbf{?}\ q_0 \mathrel{\mathbf{:=}} d_0\ \mathbf{;}\ e\ \mathbf{?}\ q_1 \mathrel{\mathbf{:=}} d_1 = e\ \mathbf{?}\ (q_0 \mathrel{\mathbf{:=}} d_0\ \mathbf{;}\ q_1 \mathrel{\mathbf{:=}} d_1)}\ \textsc{EnableDist}$$

Consider a small example for a memory where the address is only 2-bits and the data is 2-bits, so there are four possible entries in the memory block. The wires for the data signal are `d0` and `d1`. The eight 1-bit registers are `x00`, `x01`, `x10`, `x11`, `x20`, `x21`, `x30`, and `x31`, with enable signals `x0_en`, `x1_en`, `x2_en`, and `x3_en`. We use meaningful variable identifiers only for the ease of understanding in the presentation of the example; such naming information is lost in the netlist. The individual registers definitions are below followed by the application of the EnableDist rule:

```
  x0_en ? x00 := d0 ; x0_en ? x01 := d1 ;
  x1_en ? x10 := d0 ; x1_en ? x11 := d1 ;
  x2_en ? x20 := d0 ; x2_en ? x21 := d1 ;
  x3_en ? x30 := d0 ; x3_en ? x31 := d1
⇝
  x0_en ? (x00 := d0 ; x01 := d1) ;
  x1_en ? (x10 := d0 ; x11 := d1) ;
  x2_en ? (x20 := d0 ; x21 := d1) ;
  x3_en ? (x30 := d0 ; x31 := d1)
```

4.4.2 Decompiling Write Logic

After grouping registers through common enables, we show how to apply rewrites to recover memory write logic. This logic follows exactly the rules for write logic from Figure 4.3, except in the opposite direction, from logic gates, to demultiplexers, up to high-level memory addressing logic—specifically, rules Demux, Demux2n, and WriteAddress. Here, through an example, we show how to go from a low-level Elephant program and rewrite it step-by-step into an equivalent program with the write logic represented in high-level Elephant. Following the example from Section 4.4.1, we consider the definitions of the register enable signals below:

```
x0_en := en ∧ (¬a0 ∧ ¬a1) ; x1_en := en ∧ (a0 ∧ ¬a1) ;
x2_en := en ∧ (¬a0 ∧ a1)  ; x3_en := en ∧ (a0 ∧ a1) ;
```

The write enable signal for the memory write port is `en`. The wires for the address signal are `a0` and `a1`. The conjuncts `en ∧ ¬a0` and `en ∧ a0` form a 1-to-2 demultiplexer, which, via the Demux rule, gets rewritten into:

```
{ t0, t1 } := Demux(en, a0) ; [Demux]
x0_en := t0 ∧ ¬a1 ;
x1_en := t1 ∧ ¬a1 ;
x2_en := t0 ∧ a1 ;
x3_en := t1 ∧ a1 ;
```

After this rewrite, two more demultiplexers form using the two outputs from the first demultiplexer (`t0` and `t1`) as inputs:

```
{ t0, t1 } := Demux(en, a0) ;
{ x0_en, x2_en } := Demux(t0, a1) ; [Demux]
{ x1_en, x3_en } := Demux(t1, a1) ; [Demux]
```

Finally, this step matches rule Demux2n for a 1-to-4 demultiplexer based on the enable signal on the address signal: `{ x0_en, x1_en, x2_en, x3_en } := Demux(en, {a0, a1})`. This `Demux` combined with all of the guarded update statements for register `x00`, . . ., `x31` can then be merged into a single memory write statement via WriteAddress: `en ? mem[{a0, a1}] := {d0, d1}`.

4.4.3 Decompiling Read Logic

Similar to the write logic, recovering read logic follows the reversed elaboration rules from Figure 4.3, specifically rules Mux, Mux2n, and ReadAddress.

For each read port, we follow a multiplexer tree stemming from the registers until it stops at a series of n wires, where n is the width of each register. We can follow the same example but for read logic, where `rd0` and `rd1` are output signals for the data value of the read port:

```
  rd0 := (((x00 ∧ ¬a0) ∨ (x10 ∧ a0)) ∧ ¬a1) ∨
         (((x20 ∧ ¬a0) ∨ (x30 ∧ a0)) ∧ a1)
⇝ [Mux]
  rd0 := (Mux(a0, {x00, x10}) ∧ ¬a1) ∨
         (Mux(a0, {x20, x30}) ∧ a1)
⇝ [Mux2N]
  rd0 := Mux({a0, a1}, {x00, x10, x20, x30})
```

Following the same rewrites for `rd1` (not shown) produces a corresponding multiplexer, selecting bit 1 instead of bit 0: `rd1 := Mux({a0, a1}, {x01, x11, x21, x31})`. The concatenation of these two expressions generates a valid memory read via the ReadAddress rule: `{rd0, rd1} := mem[{a0, a1}]`. If there is a write mask, it will be rewritten in this phase, following the opposite direction of the WriteMask rule.

4.4.4 Decompiling to an Abstract Memory

With all gate-level logic rewritten into behavioral reads and writes, the final step pulls this logic into distinct read and write ports in an abstract memory using the reverse direction of the ReadPort and WritePort rules. In this way, we reverse the elaboration rules to deduce instantiations of `ReadPort` and `WritePort`, lifting up to a full `Memory` in Elephant. We demonstrate these steps on the small example, instantiating a single abstract memory interface with a read port and write port:

```
  {rd0, rd1} := mem[{a0, a1}]
  en ? mem[{a0, a1}] := {d0, d1}
⇝ [ReadPort, WritePort]
  rp := ReadPort(1, {a0, a1}, {rd0, rd1})
```

```
wp := WritePort(en, {a0, a1}, {d0, d1})
mem := Memory([rp], [wp])
```

Identifying options such as **LatchLastRead** and **WriteReadForward** also follow the same method—running the elaboration rules from Section 4.3.2 in reverse based on the gate-level logic, with the result collapsing the logic into an option in the memory interface.

4.4.5 Memory Decompilation via Equality Saturation

The previous section presents an ideal scenario to decompile memories from gate-level netlists. In practice, however, the reversed elaboration rules by themselves are insufficient to decompile memories in arbitrary netlists generated from other languages and hardware synthesis tools. The reason is that hardware synthesis and logic optimization obscure the original structure of the high-level memory block. Additional transformations may be necessary to reveal the structure of a memory in the low-level gates of the netlist, although it is not always obvious which sequence of rewrites will do so.

Solving this problem requires searching over a large space of possible transformations. Our memory decompilation technique overcomes this problem by using *equality saturation* [104], a non-destructive term rewriting technique which uses the e-graph [105] data structure to group terms into equivalence classes. Instead of applying a fixed sequence of transformations over a program, equality saturation applies all possible rewrites over the whole program in a convergent process, extracting the "best" term according to a given cost function.

There are three challenges unique to the hardware domain that hinder equality saturation from performing efficiently over hardware designs: (1) the massive scale of netlists; (2) working between multiple levels of abstraction; (3) aggressive optimizations that obscure the structure of low-level logic. To address these challenges, we developed an equality saturation technique specially tailored to overcome these three challenges from the hardware domain: *scoped rewrites* to scale to million-gate designs, *phased rewrites via subsumption* to handle multi-level abstraction recovery, and *anti-unification* to repair logic obscured by aggressive optimizations.

Scoped rewrites Conventional equality saturation globally applies all rewrites across the entire e-graph. For a task such as recovering read port logic, the decompiler will apply the rule for rewriting combinational logic gates into 2-to-1 multiplexers (Mux in Figure 4.3). However, for a netlist with millions of gates, it is possible that many of the rewritten multiplexers have no relation to a memory's read port logic.

The insight with "scoped" rewrites is that some rewrites are only profitable in given contexts, and so equality saturation will only apply those rewrites on certain parts of the e-graph. For the read port logic example, the only relevant multiplexers to recover are those whose inputs are derived from a grouped register. Then, successive rewrites are only applied "outward" from the outputs of the rewritten multiplexers.

Phased rewrites via subsumption Due to the multilevel representation of memories in Elephant, memory decompilation can be split into distinct phases. That is, there are some sets of rewrites that should be saturated before running others. It is not always profitable to run *all* rewrites over the netlist at once, for example, recovering write ports before registers are even grouped together. So, as an optimization, instead of applying all sets of rewrite rules at the same time, the memory decompiler has four main phases with their associated rules (from Figure 4.3):

1. Register grouping: EnableDist.

2. Read port logic: Mux, Mux2n, ReadAddress.

3. Write port logic: Demux, Demux2n, WriteAddress.

4. Memory extraction: ReadPort, WritePort.

Subsumption is one key mechanism for implementing phased rewrites. To *subsume* in the e-graph means to prevent an entry in an e-class from being extracted (as opposed to deleting it entirely). Recall the register grouping rule:

$$\frac{e,\ d_0,\ d_1 : \mathbb{B}^1 \quad q_0,\ q_1 : Reg(1)}{e \ \textbf{?}\ q_0 \ \textbf{:=}\ d_0 \ \textbf{;}\ e \ \textbf{?}\ q_1 \ \textbf{:=}\ d_1 = e \ \textbf{?}\ (q_0 \ \textbf{:=}\ d_0 \ \textbf{;}\ q_1 \ \textbf{:=}\ d_1)}\textsc{EnableDist}$$

Trivial saturation of EnableDist will result in a combinatorial explosion of the number of possible register groups. To avoid this blow up, the equality saturation technique groups as many registers as possible in a single round, then *subsumes* grouped registers (preventing them from being extracted). Subsumption is similarly used in other phases. For example, during multiplexer tree reduction, the decompiler subsumes smaller multiplexers after rewriting them into a single larger multiplexer.

Logic repair via anti-unification Unlike read ports, write ports are not as straightforward to extract for two reasons. (1) Theoretically, a write port can be decomposed into a demultiplexer tree. However, a demultiplexer's outputs are loosely coupled and the internal logic is often twisted with other unrelated logic in aggressively optimized netlists, which cannot be recovered by standard equality saturation. (2) Even in the most ideal case, the demultiplexer tree can be broken with missing branches. For example, RISC-V, ARMv8-A, and MIPS processor register files have a “zero” register, which is never written to, resulting in certain paths of the demultiplexer tree being permanently unused, thus effectively pruned. Such an aggressive logic optimization in the write logic may reduce a subcircuit to a single wire, making it expensive to speculatively transform that wire back into arbitrary logic.

This problem comes down to profitably "undoing" an optimization that reveals some higher-level structure for the sake of decompilation. However, standard equality saturation does not deal particularly well with rewrites that dramatically increase the size of the e-graph. Previous work in other domains develop customizations on top of equality saturation to handle these scenarios. For example, work in decompiling 3D CAD programs developed *inverse transformations* which speculatively add equivalences to the e-graph that expose latent structure [99].

We develop an *anti-unification* technique for netlists that repairs the missing write logic. Anti-unification is computing the most concrete pattern that matches two given terms, and has been used in equality saturation in other domains [106]. The idea is to anti-unify the optimized logic with the "ideal" write logic to recover the logic that was optimized away. We do this by introducing a *reverse* write port into the write logic, constructed from the register enable signals and the write data signal. The reverse write port keeps the equivalence of the original netlist. Then, after another saturation pass, much of the logic is simplified away, leaving the difference between the "ideal" write port and netlist's. For an unoptimized netlist, the reverse write port will be totally eliminated.

4.5 Evaluation

In this section, we evaluate Elephant and our decompilation-based memory lifting technique over a suite of representative hardware design benchmarks. Through our evaluation, we seek to answer the following questions:

RQ 1. Can our abstract memory representation target multiple technologies across sim-

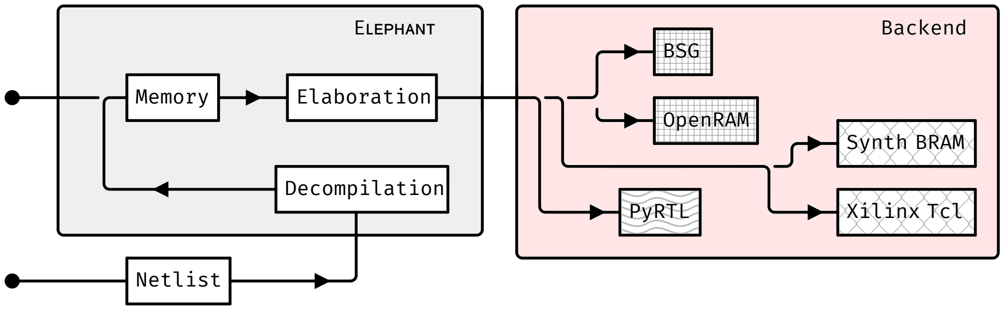


Figure 4.4: An illustration of the ELEPHANT tool flow. A user inputs either a memory design expressed in ELEPHANT, or a netlist. After elaboration, ELEPHANT targets a particular backend. The backends are shaded by technology platform. Boxes with square grids are ASIC backends; boxes with cross hatching are FPGA backends; and boxes with wavy lines are simulation backends.

ulation, ASIC, and FPGA platforms?

RQ 2. Can our decompilation-based memory lifting technique identify memories that state-of-the-art memory inference cannot?

RQ 3. Can our composed techniques enable technology re-targeting on a real-world open-source SoC design, starting from a design mapped to one technology and re-targeting it to another?

4.5.1 Implementation

Our implementation spans two parts: (1) the ELEPHANT language itself, and (2) the memory decompiler. Figure 4.4 illustrates the overall tool flow where a user inputs a memory design expressed in ELEPHANT. For designs coming from other sources, the user inputs a netlist, which gets passed into the decompilation-based memory-inference flow, lifting the registers in the netlist to memory blocks in Elephant. After elaboration, Elephant targets a particular backend. We target five backends for three different technology platforms: Basejump STL (BSG) and OpenRAM for ASIC; Vivado Tcl scripts and synthesizable BRAMs for FPGA; and PyRTL for simulation. Currently, the ASIC and FPGA backends are implemented via templating, whereas the PyRTL simulation flow follows the core elaboration rules for Elephant.

To evaluate Elephant we need to realize it in a concrete HDL, so we implement Elephant as an extension to the Python-based HDL PyRTL. We specify interfaces for `ReadPort`, `WritePort`, and `Memory`, following the signatures given in Figure 4.2b. We implement the rewrite-driven lowering passes following the rules in Section 4.3.2.

To support additional backends, there are two required pieces. The first, and most basic, is a "template" for the specific memory technology; either instantiating the module in HDL code, or generating a script such as for Vivado Tcl or OpenRAM. Then, there needs to be a mapping from the abstract memory interface to the fill in the template—for example, mapping data and address sizes and port inputs and outputs. Second, for mapping scenarios where the memory cannot be directly mapped to the specified technology, constraints meta-data is required. The constraints can be derived readily from vendor manuals. The following is an example of meta-data for a dual-port BRAM for a Xilinx FPGA:

```
"xilinx": [
{
  "name": "xilinx_bram_2rw",
  "description": "7-Series Dual-Port BRAM",
  "cost-per-bit": 3
  "width": 36,
  "height": 1024,
  "read_ports": 0,
  "write_ports": 0,
  "read_write_ports": 2,
  "features": [ "WriteReadForward", "Sync" ]
},
```

As described in Section 4.4.5, we also design a bespoke equality saturation technique for memory decompilation. In a departure from conventional equality saturation, we implement this technique entirely in SQL (`sqlite3` in Python). The relational database approach lends itself well to representing netlists. Equality saturation frameworks are AST-based and generally do not support cycles, whereas netlists do not resemble ASTs and due to registers, contain cycles. The customizations we added to the equality saturation technique—subsumption, "scoped" rewrites, and anti-unification—were easily implemented as SQL expressions.

4.5.2 Micro-Benchmark Experiments

We answer RQ 1 and RQ 2 through a series of micro-benchmark experiments demonstrating Elephant's support for automated memory technology mapping and memory inference via decompilation.

Memory design in Elephant

To answer RQ 1, we port a five-stage RISC-V processor, originally implemented in PyRTL, to use memories in Elephant. The processor has three memory blocks: a register file, an instruction memory, and a data memory with a write mask.

Specifying the memories in Elephant, we shrink the memory implementation from 130 lines of code to *3 lines of code*, one line for each of the three memory interfaces. Be-

```
dmem = MemBlock(bitwidth=32, addrwidth=32)
rdata = dmem[addr]
with conditional_assignment:
  with mask_mode == MaskMode.BYTE:
    with offset == 0:
      to_write |= wdata[:8].zero_extended(len(rdata)) | (~(Const("32'hff")) & rdata)
    with offset == 1:
      to_write |= concat(wdata[0:8],
                  Const("8'h0")).zero_extended(len(rdata)) | (~(Const("32'hff00")) & rdata)
    with offset == 2:
      to_write |= concat(wdata[0:8],
                  Const("16'h0")).zero_extended(len(rdata)) | (~(Const("32'hff0000")) & rdata)
    with otherwise:
      to_write |= concat(wdata[0:8],
                  Const("24'h0")).zero_extended(len(rdata)) | (~(Const("32'hff000000")) & rdata)
  with mask_mode == MaskMode.SHORT:
    with offset == 0:
      to_write |= wdata[:16].zero_extended(len(rdata)) | (~(Const("32'hffff")) & rdata)
    with offset == 2:
      to_write |= concat(wdata[0:16], Const("16'h0")) | (~(Const("32'hffff0000")) & rdata)
  with otherwise:
    with offset == 0:
      to_write |= wdata
    with otherwise:
      to_write |= rdata
dmem[addr] <<= MemBlock.EnabledWrite(to_write, enable)
```

Figure 4.5: Original PyRTL implementation of the data memory, including write masking logic, for a RISC-V core.

cause PyRTL memory blocks do not natively support write masks, the original design manually implemented the masking logic. Of note is the original implementation of the data memory and write masking logic, shown in Figure 4.5. We can express this same logic in one line of Elephant:

```
dmem = Elephant.Memory([ReadPort(addr, rdata)], [WritePort(addr, wdata, mask)])
```

Porting the design to Elephant also comes with the added benefit of automatically targeting four additional backends besides simulation. Two backends, for the Basejump STL and synthesizable BRAMs, directly generate module instantiations in Verilog. The other two backends, Vivado Tcl and OpenRAM, generate configurations which themselves must be processed in their respective memory compiler. Table 4.1 shows that these backends generate valid memory configurations that successfully target the given technologies. We validate the configurations by running them through the appropriate toolchain (depending on the backend), and check that the memory mapped correctly to the technology.

Decompilation-based memory inference

To answer RQ 2, we compare our decompilation-based memory lifting technique against state-of-the-art memory inference in Yosys and a state-of-the-art proprietary synthesis toolchain. Note that the comparison to memory inference in Yosys and the proprietary tool is not completely one-to-one because these techniques work through syntactic pattern matching in HDL code, whereas our technique works at the netlist level. Nonetheless, we report if memory inference from these tools works over the original HDL code that generated the netlists we run through our memory decompiler. We consider a set of six micro-benchmarks coming from netlists synthesized from open-

Table 4.1: Memory mapping validation tests from ELEPHANT across a range of port configurations (**1rw**, **1r1w**, and **2rw**) and memory sizes, targeting the Vivado Tcl and OpenRAM backends. The port notation '**1r1w**' indicates a memory with 1 distinct read and write port, whereas '**1rw**' indicates 1 port which can either read or write (but not at the same time). $\checkmark_{Tcl}$ stands for validated mapping for the Vivado Tcl backend. $\checkmark_{OpenRAM}$ stands for validated mapping for the OpenRAM backend.

Memory Size	**1rw**	**1r1w**	**2rw**
32 × 16	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$
32 × 32	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$
32 × 64	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$
32 ×128	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$

Table 4.2: Memory decompilation results comparing against memory inference in Yosys and a state-of-the-art proprietary synthesis toolchain.

Design	**Gates (thousands)**	**Memory**	**Time (seconds)**	**Yosys inferred?**	**Proprietary inferred?**
`bsg_cache`	92 k	`1rw`: (8 × 256) ×8	3 s	∅	∅
`bsg_fifo`	34 k	`1r1w`: (64 × 256)	0.7 s	∅	✓
`bsg_fifo`	527 k	`1r1w`: (512 × 512)	23 s	∅	✓
`nerv`	8 k	`2r1w`: (32 × 32)	0.1 s	✓	✓
`pico`	10 k	`2r1w`: (32 × 32)	0.1 s	✓	✓
`sparc_ffu`	27 k	`1rw`: (78 × 128)	0.4 s	✓	∅

source hardware design suites such as OPDB [107] and the Basejump STL [83]. The selection of micro-benchmarks shows the range of memories our technique decompiles, as they are deployed in a variety of designs from caches to cores.

Table 4.2 presents the results for memory decompilation in our technique versus memory inference from Yosys and a state-of-the-art proprietary toolchain. Noteworthy are the `bsg_cache`, `bsg_fifo`, and `sparc_ffu` designs. Here **we successfully identify memories where either Yosys or the proprietary synthesis tool, or both, did not**. Because the state of the art infers memories through syntactic pattern matching, memory inference is sensitive to the implementation. Memory decompilation, by contrast, works on the structure of the netlist.

4.5.3 Large-Scale Case Studies

To answer RQ 3, we show how Elephant plus memory decompilation enables technology re-targeting on real-world open-source SoC designs with many memory blocks. We consider three larger case studies from OPDB with multiple expected memory blocks. Table 4.3 presents the memory decompilation and re-targeting results. These results demonstrate how the memory decompilation technique scales to large designs, with the L2 cache totaling 1.5 million gates. Using the original source code as a reference, the first three decompiled memory blocks for the L2 cache correspond to the port configurations and dimensions of the data, tag, and state memory blocks.

Next, we consider BlackParrot, an open-source RISC-V multicore SoC [108]. Table 4.4 presents the results from decompiling an entire BlackParrot core, totaling 11.8 million gates. Memory decompilation took 373 seconds, and identified 92 distinct memory blocks. It is noteworthy that memory decompilation successfully identifies all expected memory blocks in the netlist.

The BlackParrot core itself is split between a front end and back end (although in the netlist, there is no such distinction). Memories in the front end include a BHT (Branch History Table), a BTB (Branch Target Buffer), RAS (Return Address Stack), and 3 smaller queues as part of the PC generation module, as well as 3 memories that make up the instruction cache. Memories in the back end include an integer register

Table 4.3: Large-scale case studies from OPDB [107]. The benchmarks are an L1.5 cache, an L2 cache, and entire SPARC core. Since the benchmarks contained multiple memory blocks each, the table reports them separately in the "Decompiled" column. The decompilation times are totals for each module including all decompiled memories. The table marks $\checkmark_{Tcl}$ or $\checkmark_{OpenRAM}$ to indicate successful re-targeting to Vivado Tcl or OpenRAM, respectively.

Design	**Gates**	**Decompiled**	**Time (s)**	**Re-targeted**
`OPDB l1.5 cache`	242 k	`1r1w`: (158 × 512) `1r1w`: (116 × 128) `1r1w`: (8 × 128) `1r1w`: (8 × 128) `1r1w`: (16 × 4)	10 s	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$
`OPDB l2 cache`	1.5 m	`1r1w`: (144 × 4096) `2r1w`: (170 × 256) `1r1w`: (64 × 16) `1r1w`: (16 × 4)	109 s	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$
`SPARC core`	769 k	`1r1w`: (544 × 256) `1r1w`: (576 × 128) `1r1w`: (120 × 128) `1r1w`: (78 × 128) `1r1w`: (151 × 32)	48 s	$\checkmark_{Tcl}$, $\checkmark_{OpenRAM}$

file, a floating-point register file, and 3 memories as part of the data cache. The data memory block that is part of the data cache is decompiled as 64 memories of size 512×8, due to the original memory using a byte mask.

Table 4.4: BlackParrot memory decompilation results. The "Corresponding Modules" column maps the decompiled memories to a memory blocks in the source code based on the dimensions and portedness.

Decompiled Memories	**Corresponding Modules**
32×14, 64×184, 512×64 (×8)	Instruction Cache (data, tag, stat)
64×15, 64×184, 512×8 (×64)	Data Cache (data, tag, stat)
512×8, 64×50, 16×43	BHT, BTB, RAS
32×66 (`2r1w`, `3r1w`)	Integer and Floating Point Register Files
8×80, 8×174, 4×114	Preissue FIFO, Issue queue, Cmd queue
32×20 (×3), 32×67, 8×15 (×3), 4×10	Uncatergorized
Core Size: 11.8 million gates	**Time**: 373 seconds

4.6 Related Work

Due to the "forward" and "backward" nature of the memory mapping and re-targeting problem, our work touches on past research in hardware design and reverse engineering.

4.6.1 Hardware Decompilation

Part of this work is a continuation of work in hardware decompilation. Prior work focused on recognizing repeated logic in a netlist and decompiling it into loops at the HDL level, producing more compact HDL code and speeding up simulation time [5]. The hardware loop rerolling technique uses suffix trees to find loop candidates and sketch-guided program synthesis to reroll candidates into valid loops that are semantically equivalent to the original netlist. Our work in this chapter presents another design-focused application of hardware decompilation: technology re-targeting for memories. Departing from previous techniques in hardware decompilation, this work does not use suffix trees for netlist analysis or program synthesis for code generation, and as a result scales to netlists with millions of cells.

4.6.2 Netlist Reverse Engineering

The memory decompilation technique introduced in this work builds off of previous research in reverse engineering sequential components in netlists. The prior work falls in two categories: (1) register aggregation, and (2) memory identification. However, in contrast to reverse engineering, our work takes steps beyond just *identification* of register and memory components, and also automatically generates semantically equivalent HDL code. Automated memory technology mapping (and re-targeting) requires working with a programmatic representation for compilation (and decompilation).

Register Aggregation

Register aggregation is the problem of finding multi-bit registers from one-bit D-flip flops in a gate-level netlist [86, 109]. DANA is a netlist reverse engineering framework and presents a general solution for the register aggregation problem [109]. Their technique uses a data-flow analysis to group D-flip flops into multi-bit registers. DANA generates a flip-flop dependency graph and groups flip-flops together based on shared clock and control signals. Grouping proceeds based on predecessor and successor analysis in the dependency graph. Our register aggregation technique for memory decompilation can be viewed as a specialization of DANA's technique geared towards the structure of registers found in memories.

Memory Identification

There is prior reverse engineering work that describes a technique for identifying small RAMs and register files in netlists [110, 86]. Our memory decompilation technique mirrors this work's high-level algorithm for outlining the main boundary of the memory block's logic. Due to the regular structure of the synthesized logic for memory blocks, there is a clear pattern to identifying the multiplexer and decoder trees for a memory block's read and write port logic. However, our technique can identify memory blocks with richer semantics that previous work cannot—distinguishing more read/write port configurations, write masking, and other memory block features used in common SoC components such as caches and branch predictors.

4.6.3 Rewriting in HDLs

Existing HDLs use rewrite-driven strategies to elaborate hardware designs written in high-level programming languages [111, 112, 103, 25]. Many of these languages target Verilog, which does not address the technology mapping problem. ELEPHANT presents a new category of HDL that focuses specifically on the memory mapping problem while also utilizing the "elaboration-through-execution" strategy. Besides code compilation, rewriting techniques have also been applied to other EDA tasks such as synthesis and optimization [113, 114, 115, 116, 117, 118], with recent work using an efficient rewriting technique called equality saturation [104, 100].

4.6.4 Memory Compilers

Memories in commodity ASIC products are near-universally generated from commercial SRAM compilers. Memory options and configuration ranges may vary significantly between technology nodes or even simply across foundries. Because of this, designers spend great effort optimizing configurations as SRAM often comprises the lion's share of the modern ASIC area. These are provided as part of a standard PDK during a tapeout and so come with significant costs as well as usage restrictions. Synopsys provides Generic Memory Compiler [119] to eligible universities but restricts the technology to educational PDKs.

In contrast, OpenRAM [120] generates SRAM views in realistic technologies. However, designers must manually optimize configurations as with commercial offerings. Our work builds upon this by formally identifying functional configuration knobs to enabling automated design sweeps of generated memory configurations.

4.7 Conclusion

This chapter presented ELEPHANT, a memory design language for automated memory technology mapping. Through a rewrite-driven elaboration semantics, we show how ELEPHANT provides chip developers an abstract memory interface to map to many memory technologies without littering their code base with `ifdef` blocks. For existing designs, we also present a decompilation-based memory lifting technique which enables automated technology *re-targeting*. We show that ELEPHANT effectively targets five backends for three different technology platforms—for simulation, ASIC, and FPGA. We evaluate memory decompilation over a set of hardware design benchmarks showing that our technique identifies memories where state-of-the-art memory inference does not. Further, we demonstrate technology re-targeting on real-world open-source SoC designs.

Chapter 5

Conclusions

The "golden age" of computer architecture [2] offers an exciting opportunity to advance the design of novel, open computing architectures and specialized hardware. However, we can only realize this if we improve the languages and tools that chip designers use. In this thesis, I move the field forward by integrating formal methods into open-source language-driven hardware design tools, opening two new areas in the chip design space—**control logic synthesis** and **hardware decompilation**—and enabling novel design processes based around increasing developer agility with correctness guarantees.

As control logic synthesis and hardware decompilation are both new areas, there are many future directions to pursue. With control logic synthesis, there is future work to develop new techniques that can synthesize control for more control structures, such as microcode for complex instructions. Handling larger microarchitectures with more optimizations will also require advancing the state-of-the-art in automated reasoning techniques to push past the limitations of CEGIS-style program synthesis.

For hardware decompilation, there are many future directions in terms of recovering more kinds of programming abstractions such as finite state machines, protocols, and control logic. One long-term goal for this work is an end-to-end decompiler which composes many decompilation passes together to recover as many abstractions as possible from a netlist. Further, there is additional work in improving hardware decompilation for other hardware design tasks. For instance, a large problem in industrial settings are developers needing to understand and modify highly tuned and low-level RTL code. Lifting this low-level RTL code to "behavioral" HDL code would deliver enormous value for developers in terms of making code bases more *explainable* (where future work would need to rigorously define "explainable").

Going forward, I envision new programming languages and compilers enhanced with automated reasoning techniques deployed across the hardware–software stack for emerging computing architectures—from system specification, to the hardware–software interface, all the way down to the physical design level. Along each of these layers, I believe we must work across disciplines to ensure programmer usability while balancing performance metrics with correctness. Achieving this vision will require advancing the state-of-the art in formal methods and continuing to approach the hardware design process through the perspective of programming languages abstractions.

Bibliography

[1] H. 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.

[2] J. L. Hennessy and D. A. Patterson, *A new golden age for computer architecture*, *Commun. ACM* **62** (Jan., 2019) 48–60.

[3] Z. D. Sisco, A. D. Alex, Z. Ma, Y. Aghamohammadi, B. Kong, B. Darnell, T. Sherwood, B. Hardekopf, and J. Balkind, *Control logic synthesis: Drawing the rest of the owl*, in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ASPLOS '24, (New York, NY, USA), p. 63–78, Association for Computing Machinery, 2025.

[4] Z. D. Sisco, J. Balkind, T. Sherwood, and B. Hardekopf, *A position on program synthesis for processor development*, in *LATTE '22: Workshop on Languages, Tools, and Techniques for Accelerator Design at ASPLOS 2022*, 3, 2022.

[5] Z. D. Sisco, J. Balkind, T. Sherwood, and B. Hardekopf, *Loop rerolling for hardware decompilation*, *Proc. ACM Program. Lang.* **7** (jun, 2023).

[6] A. Shademan, R. S. Decker, J. D. Opfermann, S. Leonard, A. Krieger, and P. C. W. Kim, *Supervised autonomous robotic soft tissue surgery*, *Science Translational Medicine* **8** (2016), no. 337 337ra64–337ra64, [https://www.science.org/doi/pdf/10.1126/scitranslmed.aad9398].

[7] A. Cui, *The next frontier in cyberwar: Embedded devices*, *GCN* (2022) [https://gcn.com/cybersecurity/2022/02/next-frontier-cyberwar-embedded-devices/362545/].

[8] lowRISC, “Opentitan.” `https://docs.opentitan.org/`, 2022.

[9] J. R. Burch and D. L. Dill, *Automatic verification of pipelined microprocessor control*, in *Proceedings of the 6th International Conference on Computer Aided Verification*, CAV '94, (Berlin, Heidelberg), p. 68–80, Springer-Verlag, 1994.

[10] M. N. Velev and R. 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, (New York, NY, USA), p. 112–117, Association for Computing Machinery, 2000.

[11] R. Hosabettu, G. Gopalakrishnan, and M. Srivas, *Verifying advanced microarchitectures that support speculation and exceptions*, in *Computer Aided Verification* (E. A. Emerson and A. P. Sistla, eds.), (Berlin, Heidelberg), pp. 521–537, Springer Berlin Heidelberg, 2000.

[12] R. Jhala and K. L. McMillan, *Microarchitecture verification by compositional model checking*, in *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, (Berlin, Heidelberg), p. 396–410, Springer-Verlag, 2001.

[13] J. Sawada and W. 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** (2002), no. 2 187–222.

[14] P. Manolios and S. K. Srinivasan, *A refinement-based compositional reasoning framework for pipelined machine verification*, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **16** (2008), no. 4 353–364.

[15] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, *End-to-end verification of processors with isa-formal*, in *Computer Aided Verification* (S. Chaudhuri and A. Farzan, eds.), (Cham), pp. 42–58, Springer International Publishing, 2016.

[16] A. Lööw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. Fox, *Verified compilation on a verified processor*, in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, (New York, NY, USA), p. 1041–1053, Association for Computing Machinery, 2019.

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

[18] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik, *Template-based Synthesis of Instruction-Level Abstractions for SoC Verification*, in *Proceedings of the Conference on Formal Methods in Computer-Aided Design*, pp. 160–167, 2017.

[19] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, *Instruction-level abstraction (ila): A uniform specification for system-on-chip (soc) verification*, *ACM Trans. Des. Autom. Electron. Syst.* **24** (dec, 2018).

[20] B.-Y. Huang, S. Ray, A. Gupta, J. M. Fung, and S. Malik, *Formal Security Verification of Concurrent Firmware in SoCs using Instruction-Level Abstraction for Hardware*, in *Proc. Design Automation Conference*, p. 91, 2018.

[21] H. Zhang, C. Trippel, Y. A. Manerkar, A. Gupta, M. Martonosi, and S. 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*, pp. 1–10, 2018.

[22] B.-Y. Huang, H. Zhang, A. Gupta, and S. 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*, pp. 351–357, 2019.

[23] H. Zhang, W. Yang, G. Fedyukovich, A. Gupta, and S. Malik, *Synthesizing environment invariants for modular hardware verification*, in *Verification, Model Checking, and Abstract Interpretation* (D. Beyer and D. Zufferey, eds.), (Cham), pp. 202–225, Springer International Publishing, 2020.

[24] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, *ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS*, in *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, Jan., 2019. Proc. ACM Program. Lang. 3, POPL, Article 71.

[25] 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)*, pp. 1–7, 2017.

[26] E. Torlak and R. 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, (New York, NY, USA), p. 135–152, Association for Computing Machinery, 2013.

[27] E. Torlak and R. Bodik, *A lightweight symbolic virtual machine for solver-aided host languages*, *SIGPLAN Not.* **49** (June, 2014) 530–541.

[28] B.-Y. Huang, "Imdb-archive." `https://github.com/PrincetonUniversity/IMDb-Archive`, 2018.

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

[30] H. Lu, Y. Xing, A. Gupta, and S. Malik, *Soc protocol implementation verification using instruction-level abstraction specifications*, *ACM Trans. Des. Autom. Electron. Syst.* **28** (oct, 2023).

[31] C. Wolf, "Yosys open synthesis suite." `http://www.clifford.at/yosys/`, 2024.

[32] J. Bornholt and E. Torlak, *Finding code that explodes under symbolic evaluation*, *Proc. ACM Program. Lang.* **2** (Oct., 2018).

[33] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. 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, (New York, NY, USA), p. 225–242, Association for Computing Machinery, 2019.

[34] K. Ryan and C. Sturton, *Sylvia: Countering the path explosion problem in the symbolic execution of hardware designs*, in *2023 Formal Methods in Computer Aided Design (FMCAD)*, pp. 110–121, 2023.

[35] A. Becker, D. Novo, and P. Ienne, *Automated circuit elaboration from incomplete architectural descriptions*, in *2013 Asilomar Conference on Signals, Systems and Computers*, pp. 391–395, IEEE, 2013.

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

[37] A. Ardeshiricham, Y. Takashima, S. Gao, and R. 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, (New York, NY, USA), p. 1623–1638, Association for Computing Machinery, 2019.

[38] K. v. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala, *Solver-aided constant-time hardware verification*, in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, (New York, NY, USA), p. 429–444, Association for Computing Machinery, 2021.

[39] N. Bruns, V. Herdt, and R. Drechsler, *Processor verification using symbolic execution: A RISC-V case-study*, in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, 2023.

[40] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. 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, (New York, NY, USA), p. 41–54, Association for Computing Machinery, 2017.

[41] A. Athalye, M. F. Kaashoek, and N. Zeldovich, *Verifying hardware security modules with Information-Preserving refinement*, in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, (Carlsbad, CA), pp. 503–519, USENIX Association, July, 2022.

[42] A. Athalye, A. Belay, M. F. Kaashoek, R. Morris, and N. Zeldovich, *Notary: A device for secure transaction approval*, in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, (New York, NY, USA), p. 97–113, Association for Computing Machinery, 2019.

[43] Y. Yang, T. Bourgeat, S. Lau, and M. Yan, *Pensieve: Microarchitectural modeling for security evaluation*, in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, (New York, NY, USA), Association for Computing Machinery, 2023.

[44] D. Zagieboylo, C. Sherk, E. Suh, and A. 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), Association for Computing Machinery, 6, 2022.

[45] 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.*, pp. 69–70, 2004.

[46] R. Gonzalez, *Xtensa: a configurable and extensible processor*, *IEEE Micro* **20** (2000), no. 2 60–70.

[47] C. D. Systems, *TIE language—the fast path to high-performance embedded SoC processing*, tech. rep., Cadence Design Systems, Inc., San Jose, CA, USA, 2016.

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

[49] R. Brayton and A. Mishchenko, *ABC: An academic industrial-strength verification tool*, in *Computer Aided Verification* (T. Touili, B. Cook, and P. Jackson, eds.), (Berlin, Heidelberg), pp. 24–40, Springer Berlin Heidelberg, 2010.

[50] J. S. Zhang, S. Sinha, A. Mishchenko, R. K. Brayton, and M. Chrzanowska-Jeske, *Simulation and satisfiability in logic synthesis*, *Computing* **7** (2005) 14.

[51] D. Kroening, *Computing over-approximations with bounded model checking*, in *Proceedings of the Third International Workshop on Bounded Model Checking (BMC 2005)*, vol. 144, pp. 79–92, January, 2006.

[52] V. D'Silva, M. Purandare, and D. Kroening, *Approximation refinement for interpolation-based model checking*, in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, vol. 4905 of *Lecture Notes in Computer Science*, pp. 68–82, Springer, 2008.

[53] R. Mukherjee, D. Kroening, and T. Melham, *Hardware verification using software analyzers*, in *IEEE Computer Society Annual Symposium on VLSI*, pp. 7–12, IEEE, 2015.

[54] M. Mann, A. Irfan, F. Lonsing, Y. Yang, H. Zhang, K. Brown, A. Gupta, and C. Barrett, *Pono: A flexible and extensible smt-based model checker*, in *Computer Aided Verification* (A. Silva and K. R. M. Leino, eds.), (Cham), pp. 461–474, Springer International Publishing, 2021.

[55] C. Mattarei, M. Mann, C. Barrett, R. G. Daly, D. Huff, and P. Hanrahan, *CoSA: Integrated verification for agile hardware design*, in *2018 Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–5, 2018.

[56] A. Goel and K. Sakallah, *Avr: Abstractly verifying reachability*, in *Tools and Algorithms for the Construction and Analysis of Systems* (A. Biere and D. Parker, eds.), (Cham), pp. 413–422, Springer International Publishing, 2020.

[57] YosysHQ, "Symbiyosys." `https://github.com/YosysHQ/sby`, 2022.

[58] A. Dobis, T. Petersen, H. J. Damsgaard, K. J. Hesse Rasmussen, E. Tolotto, S. T. Andersen, R. Lin, and M. Schoeberl, *Chiselverify: An open-source hardware verification library for chisel and scala*, in *2021 IEEE Nordic Circuits and Systems Conference (NorCAS)*, pp. 1–7, 2021.

[59] D. Lustig, M. Pellauer, and M. Martonosi, *Verifying correct microarchitectural enforcement of memory consistency models*, *IEEE Micro* **35** (2015), no. 3 72–82.

[60] D. Lustig, G. Sethi, M. Martonosi, and A. 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, (New York, NY, USA), p. 233–247, Association for Computing Machinery, 2016.

[61] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, *CCICheck: Using μhb graphs to verify the coherence-consistency interface*, in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 26–37, 2015.

[62] Y. A. Manerkar, D. Lustig, M. Martonosi, and M. Pellauer, *RTLcheck: Verifying the memory consistency of RTL designs*, in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), p. 463–476, Association for Computing Machinery, 2017.

[63] Y. Hsiao, D. P. Mulligan, N. Nikoleris, G. Petri, and C. 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, (New York, NY, USA), p. 679–694, Association for Computing Machinery, 2021.

[64] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, *Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations*, in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 209–216, Nov, 2017.

[65] S. Beamer, *A case for accelerating software rtl simulation*, *IEEE Micro* **40** (2020), no. 4 112–119.

[66] M. K. Ganai and A. Kuehlmann, *On-the-fly compression of logical circuits*, in *International Workshop on Logic Synthesis*, 2000.

[67] B. Su, S. Ding, and L. Jin, *An improvement of trace scheduling for global microcode compaction*, in *Proceedings of the 17th Annual Workshop on Microprogramming*, MICRO 17, p. 78–85, IEEE Press, 1984.

[68] B. Su, S. Ding, and J. Xia, *URPR—An extension of URCR for software pipelining*, in *Proceedings of the 19th Annual Workshop on Microprogramming*, MICRO 19, (New York, NY, USA), p. 94–103, Association for Computing Machinery, 1986.

[69] G. Stiff and F. Vahid, *New decompilation techniques for binary-level co-processor generation*, in *Proceedings of the 2005 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '05, (USA), p. 547–554, IEEE Computer Society, 2005.

[70] E.-W. Hu, B. Su, and J. Wang, *Instruction level loop de-optimization*, in *Computer and Information Science 2015* (R. Lee, ed.), (Cham), pp. 221–234, Springer International Publishing, 2016.

[71] R. C. O. Rocha, P. Petoumenos, B. Franke, P. Bhatotia, and M. O'Boyle, *Loop rolling for code size reduction*, in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 217–229, 2022.

[72] T. Ge, Z. Mo, K. Wu, X. Zhang, and Y. Lu, *Rollbin: Reducing code-size via loop rerolling at binary level*, in *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2022, (New York, NY, USA), p. 99–110, Association for Computing Machinery, 2022.

[73] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, *Lava: Hardware design in Haskell*, *SIGPLAN Not.* **34** (Sept., 1998) 174–184.

[74] A. Mycroft and R. Sharp, *Higher-level techniques for hardware description and synthesis*, *International Journal on Software Tools for Technology Transfer* **4** (2003), no. 3 271–297.

[75] J. O'Donnell, *Overview of Hydra: A concurrent language for synchronous digital circuit design*, *International Journal of Information* **9** (March, 2006) 249–264.

[76] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, *Introducing Kansas Lava*, in *Implementation and Application of Functional Languages* (M. T. Morazán and S.-B. Scholz, eds.), (Berlin, Heidelberg), pp. 18–35, Springer Berlin Heidelberg, 2010.

[77] B. Baker, *On finding duplication and near-duplication in large software systems*, *Proceedings of 2nd Working Conference on Reverse Engineering* (1995) 86–95.

[78] T. Kamiya, S. Kusumoto, and K. Inoue, *CCFinder: a multilinguistic token-based code clone detection system for large scale source code*, *IEEE Transactions on Software Engineering* **28** (2002), no. 7 654–670.

[79] A. Solar-Lezama, *Program sketching*, *International Journal on Software Tools for Technology Transfer* **15** (2013), no. 5 475–495.

[80] J. Stoye and D. Gusfield, *Simple and flexible detection of contiguous repeats using a suffix tree*, *Theoretical Computer Science* **270** (2002), no. 1 843–856.

[81] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in *Combinatorial Pattern Matching* (A. Amir, ed.), (Berlin, Heidelberg), pp. 181–192, Springer Berlin Heidelberg, 2001.

[82] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, *Combinatorial sketching for finite programs*, in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (New York, NY, USA), p. 404–415, Association for Computing Machinery, 2006.

[83] M. B. Taylor, *BaseJump STL: SystemVerilog needs a standard template library for hardware design*, in *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[84] L. Tang and S. Davidson, "BSG Micro Designs." `https://github.com/bsg-idea/bsg_micro_designs`, 2019.

[85] U. Berkeley, *Berkeley logic interchange format (BLIF)*, *Oct Tools Distribution* **2** (1992) 197–247.

[86] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascon, W. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik, *Reverse engineering digital circuits using structural and functional analyses*, *IEEE Transactions on Emerging Topics in Computing* **2** (jan, 2014) 63–80.

[87] W. Snyder, "Verilator." `https://www.veripool.org/verilator/`, 2024.

[88] N. Rubanov, *A high-performance subcircuit recognition method based on the nonlinear graph optimization*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **25** (2006), no. 11 2353–2363.

[89] T. Meade, Y. Jin, M. Tehranipoor, and S. Zhang, *Gate-level netlist reverse engineering for hardware security: Control logic register identification*, in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1334–1337, 2016.

[90] Y. Shi, C. W. Ting, B. Gwee, and Y. Ren, *A highly efficient method for extracting FSMs from flattened gate-level netlist*, in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 2610–2613, 2010.

[91] T. Meade, S. Zhang, and Y. Jin, *Netlist reverse engineering for high-level functionality reconstruction*, in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 655–660, 2016.

[92] W. Li, Z. Wasson, and S. A. Seshia, *Reverse engineering circuits using behavioral pattern mining*, in *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 83–88, 2012.

[93] W. Li, A. Gascon, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia, *WordRev: Finding word-level structures in a sea of bit-level gates*, in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 67–74, 2013.

[94] A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanović, and S. Malik, *Template-based circuit understanding*, in *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 83–90, 2014.

[95] M. Soeken, B. Sterin, R. Drechsler, and R. Brayton, *Simulation graphs for reverse engineering*, in *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 152–159, 2015.

[96] B. Cakir and S. Malik, *Reverse engineering digital ics through geometric embedding of circuit graphs*, *ACM Trans. Des. Autom. Electron. Syst.* **23** (July, 2018).

[97] J. Portillo, T. Meade, J. Hacker, S. Zhang, and Y. Jin, *RERTL: Finite state transducer logic recovery at register transfer level*, in *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pp. 1–6, 2019.

[98] T. Zhang, J. Wang, S. Guo, and Z. Chen, *A comprehensive FPGA reverse engineering tool-chain: From bitstream to RTL code*, *IEEE Access* **7** (2019) 38379–38389.

[99] C. Nandi, M. Willsey, A. Anderson, J. R. Wilcox, E. Darulova, D. Grossman, and Z. Tatlock, *Synthesizing structured cad models with equality saturation and inverse transformations*, in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, (New York, NY, USA), p. 31–44, Association for Computing Machinery, 2020.

[100] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, *egg: Fast and extensible equality saturation*, *Proc. ACM Program. Lang.* **5** (jan, 2021).

[101] C. Wolf, "Memory handling." `https://yosyshq.readthedocs.io/projects/yosys/en/latest/using_yosys/synthesis/memory.html`, 2024.

[102] Xilinx, "Ultrascale architecture memory resources." `https://docs.amd.com/v/u/en-US/ug573-ultrascale-memory-resources`, 2024.

[103] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, *Chisel: Constructing hardware in a Scala embedded language*, in *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, (New York, NY, USA), p. 1216–1225, Association for Computing Machinery, 2012.

[104] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, *Equality saturation: a new approach to optimization*, *SIGPLAN Not.* **44** (jan, 2009) 264–276.

[105] C. G. Nelson, *Techniques for program verification*. Stanford University, 1980.

[106] D. Cao, R. Kunkel, C. Nandi, M. Willsey, Z. Tatlock, and N. Polikarpova, *babble: Learning better abstractions with e-graphs and anti-unification*, *Proc. ACM Program. Lang.* **7** (jan, 2023).

[107] G. Tziantzioulis, T.-J. Chang, J. Balkind, J. Tu, F. Gao, and D. Wentzlaff, *OPDB: A scalable and modular design benchmark*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **41** (2022), no. 6 1878–1887.

[108] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, T. Guarino, A. Joshi, M. Oskin, and M. B. Taylor, *Blackparrot: An agile open-source risc-v multicore for accelerator socs*, *IEEE Micro* **40** (2020), no. 4 93–102.

[109] N. Albartus, M. Hoffmann, S. Temme, L. Azriel, and C. Paar, *DANA - universal dataflow analysis for gate-level netlist reverse engineering*, *IACR Transactions on Cryptographic Hardware and Embedded Systems* **2020** (Aug, 2020) 309–336.

[110] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, *Reverse engineering digital circuits using functional analysis*, in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1277–1280, 2013.

[111] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, *C?ash: Structural descriptions of synchronous hardware using haskell*, in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pp. 714–721, 2010.

[112] C. Baaij and J. Kuper, *Using rewriting to synthesize functional languages to digital circuits*, in *Trends in Functional Programming* (J. McCarthy, ed.), (Berlin, Heidelberg), pp. 17–33, Springer Berlin Heidelberg, 2014.

[113] Y. Pi, H. Zou, T. Li, W. Qu, and H. Wan, *Esfo: Equality saturation for firrtl optimization*, in *Proceedings of the Great Lakes Symposium on VLSI 2023*, pp. 581–586, 2023.

[114] K.-W. Ho, S.-T. Chung, T.-F. Chen, Y.-W. Fan, C. Cheng, C.-H. Liu, and J.-H. R. Jiang, *Wolfex: Word-level function extraction and simplification from gate-level arithmetic circuits*, in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2023.

[115] S. Coward, G. A. Constantinides, and T. Drane, *Automatic datapath optimization using e-graphs*, in *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, pp. 43–50, 2022.

[116] A. Wanna, S. Coward, T. Drane, G. A. Constantinides, and M. D. Ercegovac, *Multiplier optimization via e-graph rewriting*, *arXiv preprint arXiv:2312.06004* (2023).

[117] E. Ustun, I. San, J. Yin, C. Yu, and Z. Zhang, *Impress: Large integer multiplication expression rewriting for fpga hls*, in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 1–10, IEEE, 2022.

[118] G. H. Smith, Z. D. Sisco, T. Techaumnuaiwit, J. Xia, V. Canumalla, A. Cheung, Z. Tatlock, C. Nandi, and J. Balkind, *There and back again: A netlist's tale with much egraphin'*, in *Workshop on Languages, Tools, and Techniques for Accelerator Design*, 2024.

[119] R. Goldman, K. Bartleson, T. Wood, V. Melikyan, and E. Babayan, *Synopsys' educational generic memory compiler*, in *10th European Workshop on Microelectronics Education (EWME)*, pp. 89–92, 2014.

[120] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, *Openram: An open-source memory compiler*, in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–6, 2016.