University of California
Santa Barbara

Programming Language Techniques for Improving ISA and HDL Design

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

Doctor of Philosophy
in
Computer Science

by

Michael Alexandre Christensen

Committee in charge:

Professor Ben Hardekopf, Chair
Professor Timothy Sherwood
Professor Jonathan Balkind

December 2021

The Dissertation of Michael Alexandre Christensen is approved.

_____

Professor Timothy Sherwood

_____

Professor Jonathan Balkind

_____

Professor Ben Hardekopf, Committee Chair

December 2021

Programming Language Techniques for Improving ISA and HDL Design

by

Michael Alexandre Christensen

Dedicated to my wife, Crystal, and my children, Brooklyn, Rhys, and Ira, for the joy they bring to my life.

Acknowledgements

The past six years have been the best of my life. They were also a lot of work, and I have many people to thank for helping me get here:

Ben Hardekopf For teaching me how to do research, giving me the freedom to discover my interests, and being supremely patient as I had three children along the way.

Tim Sherwood For your unparalleled advice and encouragement. You helped me believe that my ideas were good and that I could finish my PhD.

Jonathan Balkind For your invaluable insights into where the field is and where it's headed. You expanded my vision of what's possible when we use PL to solve computer architecture problems.

Rich Wolski For your incredible passion for systems. I'm grateful to have been able to learn from, TA for, and work with you during the first years of my PhD.

Joseph McMahan For being a great role model and even greater friend. I feel extremely lucky to have been able to work alongside you on Zarf starting my first year—thanks for everything.

Kyle Dewey For being a mentor with a work ethic like no other. You were an anchor and example, always willing to lend a commiserating ear through the lows and rejoice in the highs of my PhD experience.

George Tzimpragos For all the time you spent teaching me about the world of superconductor electronics. Your creativity and the depth and breadth of your knowledge continually amaze me.

Mehmet Emre For your enviable ability to grok it all. We never have a conversation where I don't learn something new from you. Project Neptune forever.

Lawton Nichols For teaching me the importance of consistency and perseverance. You motivate me to continue running, learning, and trying new things.

Miroslav (Mika) Gavrilov For being the sweetest, most caring person I know. You're still Uncle Mika to my children, and they now know the cosmic entities of Lovecraft (and their alphabet) thanks to you.

The PL Lab Harlan Kringen, Zach Sisco, Madhukar Kedlaya, Davina Zamanzadeh, and everyone not already mentioned, past and present, for the energy and creativity you brought to the lab.

The Arch Lab Deeksha Dangwal, Jennifer Volk, Weilong Cui, and everyone not already mentioned, past and present, for helping a PL person feel welcome and for guiding me through the details.

John Shalf and George Michelogiannakis For the opportunity to work on the interesting computer architecture problems of superconductor electronics.

Christophe Giraud-Carrier, Scott Burton, and Ryan Farrell For introducing me to research and guiding my decision to attend graduate school.

Janis Smith For being the best mother-in-law anyone could ask for.

My Parents For your examples of hard work and for the many sacrifices you made that enabled me to pursue my passions.

Crystal For your unwavering love and support through all of this. You're my best friend and continual source of strength, and I'm grateful to get to navigate this life with you.

Curriculum Vitæ
Michael Alexandre Christensen

Education

2015 – 2021	Ph.D. in Computer Science, University of California, Santa Barbara
	• M.S. in Computer Science awarded in March 2021
2007 – 2013	B.S. in Computer Science, Brigham Young University, Provo
	• Minor in Mathematics

Courses TA'd

Winter 2020	UCSB CS 64: Computer Organization and Logic Design
Fall 2019	UCSB CS 170: Operating Systems
Spring 2019	UCSB CS 138: Formal Languages and Automata
Winter 2017	UCSB CS 162: Programming Languages
Winter 2016	UCSB CS 162: Programming Languages
Fall 2015	UCSB CS 64: Computer Organization and Logic Design
Winter 2013	BYU CS 478: Machine Learning and Data Mining

Experience

2015 – 2021	Research Assistant, UCSB (Programming Languages Lab)
2020 – 2021	Research Assistant, Lawrence Berkeley National Lab (Computer Architecture Group)
Summer 2020	Software Engineer Intern, Facebook (Hack Language Team)
Summer 2019	Software Engineer Intern, Facebook (Disaster Recovery Team)
2014 – 2015	Software Engineer II, Dell EMC (NetWorker Team)
2013	Research Assistant, BYU Data Mining Lab
Summer 2012	Software Engineer Intern, SerialTek

Publications

Michael Christensen, Georgios Tzimpragos, Harlan Kringen, Jennifer Volk, Timothy Sherwood, Ben Hardekopf. "PyLSE: A Pulse-Transfer Level Language for Superconductor Electronics," *Under review*.

Michael Christensen, Timothy Sherwood, Jonathan Balkind, Ben Hardekopf. "Wire Sorts: A Language Abstraction for Safe Hardware Composition," *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (**PLDI**), June 2021. Virtual, Canada.

Michael Christensen, Joseph McMahan, Lawton Nichols, Jared Roesch, Timothy Sherwood, Ben Hardekopf. "Safe Functional Systems through Integrity Types and Verified Assembly," *Theoretical Computer Science*, 2021.

Joseph McMahan, **Michael Christensen**, Kyle Dewey, Ben Hardekopf, Timothy Sherwood. "Bouncer: Static Program Analysis in Hardware," *Proceedings of the 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture* (**ISCA**), June 2019. Phoenix, AZ, USA.

Joseph McMahan, **Michael Christensen**, Lawton Nichols, Jared Roesch, Sung-Yee Guo, Ben Hardekopf, and Timothy Sherwood. "An Architecture for Analysis," *IEEE Micro: Top Picks from the 2017 Computer Architecture Conferences* (**IEEE Micro - Top Picks**), vol. 38, no. 3, pp 107-115, May/June 2018.

Joseph McMahan, **Michael Christensen**, Lawton Nichols, Jared Roesch, Sung-Yee Guo, Ben Hardekopf, and Timothy Sherwood. "An Architecture Supporting Formal and Compositional Binary Analysis," *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (**ASPLOS**), April 2017. Xi'an, China.

Awards and Service

June 2021	Outstanding Teaching Assistant, Computer Science Department, UCSB (2020 – 2021 Academic Year)
June 2016	Student Volunteer Co-Captain, PLDI
2008 – 2010	Volunteer Church Representative, The Church of Jesus Christ of Latter-Day Saints (San José, Costa Rica)

Abstract

Programming Language Techniques for Improving ISA and HDL Design

by

Michael Alexandre Christensen

Despite all the effort spent in testing, analyzing, and formally verifying software, a program is ultimately only as correct as the underlying hardware on which it runs. As processors become more performant, their microarchitectures become increasingly complex; this complexity often manifests in instruction set architectures (ISAs) that are bloated, imprecise, and therefore unamenable to formal verification. The ISA itself is realized as a hardware implementation written in a hardware description language (HDL). Unfortunately, modern HDLs lack the expressive, composable programming abstractions we've come to expect of traditional high-level programming languages, hampering innovation and correct-by-construction hardware design. Furthermore, the unique characteristics of emerging technologies like superconductor electronics (SCE) require us to rethink the HDLs we use and retool the entire design, simulation, and verification stack.

This thesis shows how various programming language techniques, applied to the realm of computer architecture and hardware design, help address these issues. I show that abstraction, formal semantics, and type theory can be used to create an ISA that is precise, concise, and amenable to formal reasoning by both human and machine. I also show how HDLs can better support composability via formalized notions of intermodular communication and dependency. Finally, I show that we can improve SCE behavioral modeling and system design using a new automata-based pulse-transfer level language.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Our programs are only as correct as the machines on which they run. This correctness is often taken for granted, with the software realm siloed off from the hardware realm and often oblivious to the intricacies of the processor running the code. However, if we are to truly tackle the spectre of full-stack program verification, we must adequately include our hardware in the discussion. In this thesis, I argue that to do so, we must change how we approach hardware design both *above and below the microarchitecture* as well for emerging technologies *beyond* CMOS.

Above the Microarchitecture The separation of concerns between hardware and software has given the hardware realm the freedom to focus on creating machines that meet particular power, efficiency, area, and security targets, all while interacting with software via the same interface. This interface, the instruction set architecture (ISA), serves as both an *abstraction* and a *contract* by presenting a set of assembly instructions and a specification (often in thousands of pages of plain prose and pseudocode) of how each instruction affects machine behavior. As processors have become increasingly performant, their microarchitectures have become increasingly large and complex;

unfortunately, this complexity has permeated the ISA abstraction boundary, manifesting as ISAs that are buggy, imprecise, ill-defined, and hard to reason about, model, and simulate.

Below the Microarchitecture ISAs are in turn implemented as processor microarchitectures using a hardware description language (HDL); these processors are just one example of *intellectual property* (IP) blocks—reusable hardware components, packaged into IP catalogs, that are then connected together. Systems on chip (SOCs) and manycore processors integrate hundreds of these components, and as design teams become larger and more remote, it becomes more important that these IPs, especially proprietary IPs, have detailed and precise interface specifications. Naively connecting hardware blocks together leads to bugs which are difficult to diagnose when looking at the interfaces alone and which are especially difficult to debug after synthesis. Thus, composability of IP becomes a major issue, and modern HDLs lack effective abstractions for dealing with this intermodular composability and communication beyond simple abstractions like the "module" associating "input" and "output" wires with each other via gates. Modern high-level programming languages, on the other hand, have many mechanisms supporting effective modularity, abstraction, and dependency specification, and we can use these techniques to be more precise about the surprisingly complex requirements imposed on the use of data and what compositions lead to well-defined digital designs.

And Beyond As we enter a post-Moore, post-Dennard scaling era in search of additional power, speed, and efficiency, we must to look beyond CMOS toward emerging technologies like superconductor electronics (SCE). The unique characteristics of SCE, such as its pulse-based information encoding and stateful gate primitives, require im-

proved language abstractions and that we rethink the entire design, simulation, and verification stack. Programming language theory again provides us insight into how we can use better mathematical foundations, such as automata theory, for improved SCE languages and simulation frameworks.

We need to ensure the correctness of the entire stack, from source code to silicon[1] die, and this thesis seeks to show that the precision of programming language theory can be effectively applied on the hardware side. This brings me to my thesis statement:

1.1 Thesis Statement

The application of programming language principles like abstraction and formal semantics can make it easier to write verifiably correct and secure hardware, tangibly improving both the interface which software uses to communicate with the machine (the *instruction set architecture*) and the means by which we design the machine itself (the *hardware description language*).

1.2 Organization of This Document

This thesis is structured as follows:

Chapter 2 This chapter discusses **Zarf**, an ISA resembling the untyped lambda calculus designed to bring the assembly closer to frameworks used for formal program analysis and reasoning, and which has been implemented as an FPGA prototype running an embedded medical application. I show that the architecture allows for the formal verification of multiple properties of the end-to-end system, including a proof

[1] Niobium for SCE.

of correctness of the assembly-level implementation of the core algorithm, the integrity of trusted data via a non-interference proof, and a guarantee that our prototype meets critical timing requirements. It is based on work published in ASPLOS 2017 (Citation: [1]; DOI: 10.1145/3037697.3037733; © 2017 ACM), IEEE Micro Top Pick 2018 (Citation: [2]; DOI: 10.1109/MM.2018.032271067; © 2018 IEEE), and the Journal of Theoretical Computer Science 2021 (Citation: [3]; DOI: 10.1016/j.tcs.2020.09.039; © 2021 Elsevier) and which appeared as a portion of the PhD thesis of Joseph McMahan [4], co-author on the aforementioned publications. It is based on work supported by the National Science Foundation under Grant No. 1239567, 1162187, and 1563935.

Chapter 3 This chapter discusses **Bouncer**, which extends Zarf with let-polymorphism to make it easier to correctly write and reason about Zarf binaries. Using special-purpose hardware, I show that we can use static analysis to prevent all program binaries with memory errors, invalid control flow, and other undesirable properties from ever being loaded onto the embedded program store. We can check this in a streaming and verifiably non-bypassable way, directly in hardware, resulting in a system that is small, efficient, and which guarantees freedom from many security and safety concerns. It is based on work published in ISCA 2019 (Citation: [5]; DOI: 10.1145/3307650.3322256; © 2019 ACM) and which also appeared as a portion of the aforementioned thesis by Joseph McMahan. It is based on work supported by the National Science Foundation under Grants No. 1740352, 1730309, 1717779, 1563935, 1350690, and a gift from Cisco Systems.

Chapter 4 This chapter discusses **Wire Sorts**, an abstraction that makes it impossible to create certain classes of erroneous digital design and makes it easier to express interface requirements and compose IP. Using a taxonomy of sorts to soundly abstract even

complex combinational dependencies of arbitrary hardware modules, I show that we can facilitate modularity in digital design by escalating problematic aspects of module input/output interaction to the language-level interface specification. I also show how these sorts have been formalized and proven sound and demonstrate that they can be applied and even inferred automatically at scale via an examination of a variety of large open-source digital designs and libraries. It is based on work published in PLDI 2021 (Citation: [6]; DOI: 10.1145/3453483.3454037; © 2021 ACM) and supported by the National Science Foundation under Grants No. 1763699 and 1717779. Source code for this is available as an accompanying artifact at https://zenodo.org/record/4695169.

Chapter 5 This chapter discusses **PyLSE**, a new pulse-transfer level language for superconductor electronics built on the theory of automata. PyLSE enables the precise specification of gate semantics via a transition system-based Python embedded domain-specific language (DSL). I show that it is a effective framework for easily composing cells into larger designs and that it facilitates the identification of timing and logic errors through a mix of dynamic checks and sound static analysis. I also show how PyLSE can be formalized mathematically, demonstrating its capabilities through the creation, simulation, and verification of a selection of SCE designs. It is based on work that has been submitted to a conference and is currently under review. I am the first author on this work, and my co-authors are Georgios Tzimpragos, Harlan Kringen, Jennifer Volk, Timothy Sherwood, and Ben Hardekopf.

Chapter 2

Zarf: An Architecture Supporting Formal and Compositional Binary Analysis

2.1 Introduction

Embedded devices are ubiquitous, with many now playing roles that support human health, well-being, and safety. The critical nature of these systems — automotive, medical, cryptographic, avionic — is at odds with the increasing complexity of embedded software overall: even simple devices can easily include an HTTP server for monitoring purposes. Traditional processor interfaces are inherently global and stateful, making the task of isolating and verifying critical application subsets a significant challenge. Architectural extensions have been proposed that enhance the power, performance, and functionality of systems, but in contrast, we cover the first architecture designed with formal program analysis as a core motivating principle.

High-level, functional languages offer a remarkable ability to reason about the behav-

ior of programs, but are often unsuited to low-level embedded systems, where reasoning must be done at the assembly level to give a full picture of the code that will actually execute. At a high level, in a language designed for verification, reasoning typically requires relying on a language run-time that can be prohibitive for resource-constrained or real-time embedded systems, and/or require the assumption that thousands of lines of untrusted code in the language stack are correct.

Another approach is to directly model the processor interface by giving formal semantics to the ISA. However, reasoning about binary behavior on traditional architectures is difficult and often left incomplete. Unless *all* program components and architectural behaviors are included, any piece outside the expected model could mutate a piece of machine state and violate the assumptions of the verification effort. Even assuming that never happens, using a verified compiler, assuming other modules are correct, using only a subset of the ISA and assuming the rest is unused, program-specific reasoning is still difficult — i.e., reasoning about C still means reasoning about pointers, memory mutation, and countless imperative, effectful behaviors.

We propose a system where the critical code can execute at the assembly level in a way that is very similar to the underlying computational model upon which proof and reasoning systems are already built. Under such a mode of computation, properties such as isolation, composition, and correctness can be reasoned about incrementally, rather than monolithically. However, instead of requiring a complete reprogramming of all software in a system, we instead examine a novel system architecture consisting of two cooperating layers: one built around a traditional imperative ISA, which can execute arbitrary, untrusted code, and one built around a novel, complete, purely functional ISA designed specifically to enable reasoning about behavior at the binary level. Application behaviors that are mission critical can be hoisted piecemeal from the imperative to the functional world as needed.

Our proposed system, the Zarf Architecture for Recursive Functions, observes the following properties:

1. The functional ISA, "Zarf," is devoid of all global or mutable state, and provides a compact, complete, and mathematical semantics for the behavior of instructions;

2. The imperative ISA is strictly separated from the functional ISA, connected only via a communication channel through which the system components can pass values;

3. The subset of the application which operates on Zarf can be verified and reasoned about without regard to the operation of the imperative components, meaning that *only* the critical components need to be ported and modeled;

4. Reasoning on the functional ISA is provably composable — i.e., two separate pieces can be statically shown to never interfere with each other.

To demonstrate the usefulness of this platform, we develop, model, and test a sample application which implements an Implantable Cardio-Defibrillator (ICD) — an embedded medical device which is implanted in a patient's chest cavity, monitors the heart, and administers shocks under certain conditions to prevent or counter cardiac arrest. Though ICDs provide life-saving treatment for patients with serious arrhythmia, these devices, along with other embedded medical devices, have seen thousands of recalls due to dangerous software bugs [7, 8]. By leveraging this two-layer approach, we are able to formally verify the correctness of a low-level implementation of the core functions in Coq and directly extract executable assembly code without needing software runtimes. The ISA semantics allow us to construct an integrity type system and formally prove that the rest of the code never corrupts the inputs or outputs of the critical functions. Furthermore, the functional abstraction built in to the binary code allows

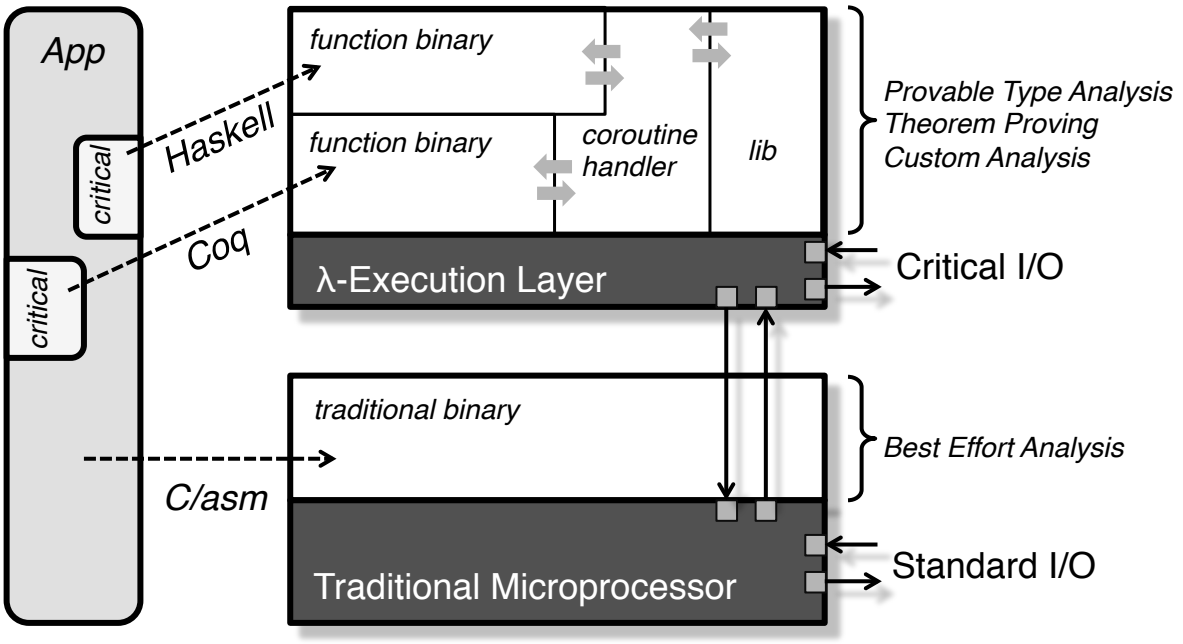


Figure 2.1: High-level Zarf system architecture: by dividing the system into two hardware realms — one that provides a precise, mathematical semantics for reasoning about program behavior, and the other a standard imperative core for legacy software — we can formally verify and otherwise reason about critical subsets of applications without needing to model and verify the entire program.

us to bound worst-case execution time, even in the face of garbage collection. Taken altogether, we have an embedded medical application whose core components have been proven correct, where non-interference is guaranteed, where real-time deadlines are assured to be met, and where C code can execute arbitrary auxiliary functions in parallel for monitoring. The high-level system architecture is shown in Figure 2.1.

Given the significant amount of related efforts in verification and ISA design, we begin by summarizing how our work differs from previous efforts in the fields of verification and architecture (Section 2.2). We then describe the Zarf platform in more detail and describe a hardware implementation, which runs the application on an FPGA (Section 2.3). Details of our embedded ICD software application and the ways it can leverage the properties of the Zarf platform are described next (Section 2.4),

followed by a precise definition of Zarf's semantics (Section 2.5). We then discuss the verification of multiple properties of the critical sub-components of the ICD, covering correctness, timing, and non-interference (Section 2.6). Finally, we evaluate this system architecture and approach, presenting hardware resource requirements of the novel ISA, and examine the performance loss of the verified components when compared to an unverified C alternative (Section 2.7), and conclude (Section 2.8).

2.2 Related Work

2.2.1 Verification

Our dual ISA approach, where one is untrusted and the other trusted, draws in part on Rushby's work on security kernels [9]. He separates machine components into virtual "regimes" and proves isolation. Having done so, Rushby can then show that security is maintained when introducing clearly defined and limited channels of communication whose information flow can be tracked. Our imperative and functional ISAs behave as separate components, communicating only through a specified, dedicated channel, thus eliminating any insecure information flow — via memory contamination, for example.

Our security type system draws from the work done on the Secure Lambda (SLam) calculus by Heintze and Riecke [10] and its further development by Abadi et al. in their Core Calculus of Dependency [11]. It also draws inspiration from Volpano [12] et al., who created a type system for secure information flow for an imperative block-structured language. By showing that their type system is sound, they show the absence of flow from high-security data to lower-security output, or similarly, that low-security data does not affect the integrity of higher-security data. Other seminal work on secure information flow via the formulation of a type system include Denning [13], Goguen

[14], Pottier's information flow for ML [15], and Sabelfeld and Myers's survey on language-based information flow security [16].

Productive, expressive high-level languages that are also purely functional are excellent source platforms for Zarf. Even languages like Haskell, though, can have occasional weaknesses that can lead to runtime type-errors. Subsets such as Safe Haskell [17] shore up these loopholes, and provide extensions for sandboxing arbitrary untrusted code. Zarf provides isolation guarantees at the ISA level and does not require runtimes, but relies on languages like Safe Haskell for source code development.

Previous work on ISA-level verification has often involved either simplified or incomplete models of the architecture. These can be in the form of new "idealized" assembly-like languages: Yu et al. [18] use Coq to apply Hoare-style reasoning for assembly programs written in a simple RISC-like language. They also provide a certified memory management library for the machine they describe. Chlipala presents Bedrock, a framework that facilitates the implementation and verification of low-level programs [19], but limits available memory structures.

Kami [20] is a platform for specifying, implementing, and verifying hardware designs in Coq. By making the language of design the same as the language of verification, the gap that traditionally exists between high-level specification and actual hardware implementation is minimized. A key distinction between Kami and our work is that Zarf focuses on building a processor specifically to make software verification easier, while Kami focuses on the task of specifying and building verifiable hardware. The two are very complementary — you can build a traditional imperative machine using Kami (e.g. their pipelined RISC-V processor implementation), and you can build a Zarf machine using traditional hardware design.

Verification has also been done for subsets of existing machines. For example, a "substantial" subset of the Motorola MC68020 interface is modeled and used to

mechanically prove the correctness of quicksort, GCD, and binary search [21]; other examples include a formalization of the SPARC instruction set, including some of the more complex properties, such as branch delay slots [22]; and subsets of x86 [23]. One of the biggest efforts to date has been a formal model of the ARMv7 ISA using a monadic specification in HOL4 [24]. Moore developed Piton, a high level assembly language and a verified compiler for the FM8502 microprocessor [25], which complemented the verification work done on the FM8502 implementation [26]. These are large efforts because of the difficulty in reasoning about imperative systems. At higher levels of abstraction, entire journal issues have been devoted to works on Java bytecode verification [27].

In addition to proofs on machine code for existing machines, it is also possible to define new assembly abstractions that carry useful information. Typed assembly as an intermediate representation was previously identified as a method for Proof-Carrying Code [28], where machine-checked proofs guarantee properties of a program [29]. Typed assemblies and intermediate representations have seen extensive use in the verification community [30, 19, 31, 32] and have been extended with dependent types [33], allowing for more expressive programs and proofs at the assembly level.

Verified compilers are a popular topic in the verification community [34, 35, 36, 37], the most well-known example being CompCert [38], a verified C compiler. Verified compilers are usually equipped with a proof of semantics preservation, demonstrating that for every output program, the semantics match those of the corresponding input program. A verified compiler does not provide tools for, nor simplify the process of doing, program-specific reasoning. One needs a secondary tool-chain for reasoning about source programs, such as the Verified Software Toolchain (VST) [39] for CompCert. These frameworks often have a great cost, mandating the use of sophisticated program logics, such as higher-order separation logic in VST, in order to fully reason

about possible program behaviors.

Further, in many systems, it's possible that not all source code is available; without being able to reason about binary programs, guarantees made on a piece of the source program (and preserved by the verified compiler) may be violated by other components. Extensions to support combining the output of verified compilers, such as separate compilation and linking, are still an active research area [40, 41]. As work on verified compilers requires a semantic model of the ISA, it is complemented by our work, which gives complete and formal semantics for an ISA.

Previous work at the intersection of verification and biological systems has attempted to improve device reliability through modeling efforts. This includes work that formulates real-time automata models of the heart for device testing [42], formal models of pacing systems in Z notation [43], quantitative and automated checking of the interaction of heart-pacemaker automata to verify pacemaker properties [44], and semi-formal verification by combining platform-dependent and independent model checking to exhaustively check the state space of an embedded system [45]. Our work is complemented by verification works such as these that refine device specification by taking into account device-environment interactions.

2.2.2 Architecture

The SECD Machine [46] is an abstract machine for evaluating arithmetic expressions based in the lambda calculus, designed in 1963 as a target for functional language compilers. It describes the concept of "state" (consisting of a Stack, Environment, Control, and Dump) and transitions between states during said evaluation. Interpreters for SECD run on standard, imperative hardware. Hardware implementations of the SECD Machine have been produced [47], which explore the implementation of SECD

at the RTL and transistor level, but present the same high-level interface. The SECD hardware provides an abstract-machine semantics, indicating how the machine state changes with each instruction. Our verification layer makes machine components fully transparent, presenting a higher-level small-step operational semantics, where instructions affect an abstract environment, and a big-step semantics, which immediately reduces each operation to a value. These latter two versions of the semantics are more compact, precise, and useful for typical program-level reasoning.

The SKI Reduction Machine [48] was a hardware platform whose machine code was specially designed to do reductions on simple combinators, this being the basis of computation. Like our verification layer, it was garbage-collected and its language was purely applicative. The goal was to create a machine with a fast, simple, and complete ISA. The choice to use the "simpler" SKI model means that machine instructions are a step removed from the typically function-based, mathematical methods of reasoning about programs. Our functional ISA, while also simple and complete, chooses somewhat more robust instructions based on function application; though the implementation is more complicated, modern hardware resources can easily handle the resulting state machine, giving a simple ISA that is sufficiently high-level for program reasoning.

The most famous work on hardware support for functional programming was on Lisp Machines [49, 50, 51]. Lisp machines provided a specialized instruction set and data format to efficiently implement the most common list operations used in functional programming. For example, Knight [51] describes a machine with instructions for Lisp primitives such as CAR and CADR, and also for complex operations like CALL and MOVE. While these these machines partially inspired this work, Lisp Machines are not directly applicable to the problem at hand. Side-effects on global state at the ISA level are critical to the operation of these machines, and while fast function calls are supported, the stepwise register-memory-update model common to more traditional

ISAs is still a foundation of these Lisp Machine ISAs. In fact, several commercial Lisp Machine efforts attempted to capitalize on this fact by building Lisp Machines as a thin translation layer on top of other processors.

Flicker also dealt with architectural support for a smaller TCB in the presence of untrusted, imperative code, but did so with architectural extensions that could create small, independent, trusted bubbles within untrusted code [52]. Our architecture is almost inverted, with a trusted region providing the main control, calling out to an untrusted core as needed. Previous works such as NoHype [53] dealt with raising the level of abstraction of the ISA and factoring software responsibilities into the hardware. Our verification layer shares some of these characteristics, but deals with verification instead of virtualization, as well as being a complete, self-contained, functional ISA.

Previous work has explored the security vulnerabilities present in many embedded medical devices, as well as zero-power defenses against them [54, 55, 56]. The focus of our work is analysis and correctness properties, and we do not deal with security.

2.3 Hardware Architecture and ISA

Our system relies on two separate layers, running two different ISAs, connected only by a data channel. This allows one of the layers to be specialized to the execution of machine code with 1) a compact, precise, and complete semantics highly amenable to proofs, and 2) the ability to compose verified pieces safely. It is entirely possible that all code in the system be written to be purely functional and run on Zarf: the ISA for this layer is complete. However, embedded devices often contain a mix of software, including legacy code or nice-to-have features that do not affect the application's behavior, such as relaying data and diagnostic information to outside receivers. With a two-layer approach, we can run imperative code that is orthogonal to the operation of

critical application components while still connecting with the vetted, functional code in a structured way. This, in turn, allows code to be formally verified piecemeal, with functions "raised" into Zarf as deemed necessary.

The following subsections describe the interface and construction of Zarf, including the reasons we take an approach much closer to the lambda calculus underlying most software proof techniques, how we capture this style of execution in an instruction set, the semantics for that instruction set, and more practical considerations such as I/O, errors, and ALU functions.

2.3.1 Design Goals

Normal, imperative architectures have been difficult to model, and the task of composing verified components is still an open problem [40, 41]. We identify the following features as undesirable and counterproductive to the goal of assembly-level verification:

1. Large amounts of global machine state (memory, stack, registers, etc.) directly accessible to instructions, all of which must be modeled and managed in every proof, and which inhibit modularity: state may be modified by code you haven't seen.

2. The mutable nature of machine state, which prevents abstraction and composition when reasoning about functions or sets of instructions.

3. A large number of instructions and features: a complete model must incorporate all of them (e.g., fully modeling the behavior of the ARMv7 was 6,500 lines of HOL4 [24]).

4. Arbitrary control flow, which often requires complex and approximate analyses to soundly determine possible control flows [57].

5. Unenforced function call conventions, meaning one must prove that every function respects the convention.

6. Implicit instruction semantics, such as exceptions where "jump" becomes "jump and update registers on certain conditions."

To avoid these traits, we design an interface that is small, explicit in all arguments, and completely free of state manipulation and side effects — with the exception of I/O, which is necessary for programs to be useful. Without explicit state to reference (memory and registers), standard imperative operations become impossible, and we must raise the level of abstraction. Instead of imperative instructions acting as the building blocks of a program, our basic unit is the *function*. This is a major departure from a typical imperative assembly, where the notion of a "function" is a higher-level construct consisting of a label, control flow operations, and a calling convention enforced by the compiler — but which has no definition in the machine itself. By bringing the definition of functions to the ISA level, they become not just callable "methods" that serve to separate out independent routines, but are actually strict functions in the mathematical sense: they have no side effects, never mutate state, and simply map inputs to outputs. This change allows us to attach precise and formal semantics to the ISA operations.

2.3.2 Description and Semantics

Zarf's functional ISA is effectively an **a**) untyped, **b**) lambda-lifted, **c**) administrative normal form (ANF) lambda calculus. Those limitations are a result of the

$$x \in \mathit{Variable} \quad n \in \mathbb{Z} \quad \mathit{fn}, \mathit{cn} \in \mathit{Name}$$

$$\begin{aligned}
p \in \mathit{Program} &::= \overrightarrow{\mathit{decl}}\ \textbf{fun}\ \text{main}\ \textbf{=}\ e \\
\mathit{decl} \in \mathit{Declaration} &::= \mathit{cons} \mid \mathit{func} \\
\mathit{cons} \in \mathit{Constructor} &::= \textbf{con}\ \mathit{cn}\ \vec{x} \\
\mathit{func} \in \mathit{Function} &::= \textbf{fun}\ \mathit{fn}\ \vec{x}\ \textbf{=}\ e \\
e \in \mathit{Expression} &::= \mathit{let} \mid \mathit{case} \mid \mathit{res} \\
\mathit{let} \in \mathit{Let} &::= \textbf{let}\ x\ \textbf{=}\ \mathit{id}\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e \\
\mathit{case} \in \mathit{Case} &::= \textbf{case}\ \mathit{arg}\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e \\
\mathit{res} \in \mathit{Result} &::= \textbf{result}\ \mathit{arg} \\
\mathit{br} \in \mathit{Branch} &::= \mathit{cn}\ \vec{x}\ \Rightarrow\ e \mid n\ \Rightarrow\ e \\
\mathit{id} \in \mathit{Identifier} &::= x \mid \mathit{fn} \mid \mathit{cn} \mid \mathit{op} \\
\mathit{arg} \in \mathit{Argument} &::= n \mid x \\
\mathit{op} \in \mathit{PrimOp} &::= + \mid - \mid \times \mid \div \mid = \\
&\quad \mid < \mid \leq \mid \wedge \mid \vee \mid \bar{\wedge} \mid \underline{\vee} \\
&\quad \mid \oplus \mid \ll \mid \gg \mid \texttt{sra} \mid \neg \\
&\quad \mid \texttt{getint} \mid \texttt{putint}
\end{aligned}$$

Figure 2.2: Abstract syntax of Zarf's functional ISA. A program is a set of function and constructor declarations, where functions are composed solely of `let`, `case`, and `result` expressions, and constructors are tuples with unique names. Case expressions contain branches and serve as the mechanism for both control flow and deconstruction of constructor forms. An arrow over any metavariable (e.g. $\vec{x}$) signifies a list of zero or more elements. *op* refers to a function that is implemented in hardware (such as ALU operations); though the execution of the function invokes a hardware unit instead of a piece of software, the functional interface is identical to program-defined functions.

implementation being done in real hardware: **a**) to avoid the complexity of a hardware typechecker, the assembly is untyped[1]; **b**) because every function must live somewhere in the global instruction memory, only top-level declarations of functions are allowed (lambda-lifted); **c**) because the instruction words are fixed-width with a static number of operands, nested expressions are not allowed and every sub-expression must be bound to its own variable (ANF). The abstract syntax of Zarf assembly is given in Figure 2.2.

All words in the machine are 32-bits. Each binary program starts with a magic word, a word-length integer N stating how many functions are contained in the program, and then a sequence of N functions. Each function starts with an informational word that lets the machine know the "fingerprint" of the function (including the number of arguments expected and how many locals will be used) and a word-length integer M to specify that the body of the function is M words long. The remaining M words of the function are then composed entirely of the individual instructions of the machine.

Each function, as it is loaded, is given a unique and sequential identifier. These function identifiers are the only globally visible state in the system and serve as both a kind of name and a kind of pointer back to the code. Other functions can refer to, test, and apply arguments to function identifiers. There are two varieties of function identifiers: those that refer to full functions that contain a body of code, and "constructors," which have no body at all. Constructors are essentially stub functions and cannot be executed. However, just like other functions, you can apply arguments to them. These special function identifiers thus can serve as a "name" for software data types, where arguments are the composed data elements. (In more formal terms, you can use our constructors to implement algebraic data types.)

[1]The original ISA definition as presented in the paper was untyped; in later work, we extended the ISA to include type annotations and a hardware typechecker.

The words defining the body of a function are built out of just three instructions: `let`, `case`, and `result`, which we will describe below. Unlike RISC instructions, `let` and `case` can be multiple words long (depending on the number of arguments and branches, respectively). However, unlike most CISC instructions, each piece of the variable length instruction is also word-aligned and trivial to decode.

Zarf has no programmer-visible registers or memory addresses, but instructions will still need to reference particular data elements. Instructions can refer to data by its source and index, where the source is one of a predefined set — e.g., *local* and *arg*, which serve a purpose similar to the stack on a traditional machine. The *local* and *arg* indices might be analogous to stack offsets, while the actual addresses themselves are never visible.

The primary ways of generating Zarf assembly are via extraction from Coq and writing it by hand. We also have a Haskell compiler that supporting a subset of basic Haskell constructs. In our experience, Zarf assembly code resembles a typical functional programming language like desugared Haskell or OCaml, and the resultant expressibility makes directly writing assembly relatively easy; the user doesn't need to worry about memory address calculations, maintaining register or stack state across function calls, or the myriad other things that make programming traditional ISAs tedious and error-prone. For more information on automatic Coq extraction, see our discussion of the ICD implementation in Section 2.6.

Figure 2.5 gives the complete ISA behavior using a big-step semantics, which explains how each instruction reduces to a value. This semantics uses eager evaluation for simplicity; though the current hardware implementation uses lazy semantics, the difference is not observable in our application because I/O interactions are localized to a specific function and always evaluated immediately. The semantics use assembly keywords for readability; Figure 2.3 shows how the assembly maps one-to-one with

the binary encoding, and Figure 2.7 shows how low-level Coq code can be directly converted to our assembly.

2.3.3 Instruction Set

The `let` instruction applies a function to arguments and assigns it a *local* identifier. The first word in the `let` instruction indicates a function identifier or closure object and the number of argument words that follow. Note that unlike a function "call", `let` does not immediately change the control flow or force evaluation of arguments; rather it creates a new structure in memory (closure) tying the code (function identifier) to the data (arguments), which, when finally needed, can actually be evaluated (using lazy evaluation semantics). Additionally, the `let` instruction allows partial application, meaning that new functions (but not function identifiers) can be dynamically produced by applying a function identifier to some, but not all, of its arguments.

The `case` instruction provides pattern-matching for control flow. It takes a value, then makes a set of equality comparisons, one for each "pattern" provided. The first word of the case instruction indicates a piece of data to evaluate. As we need an actual value, this is the point in execution that forces evaluation of structures created with `let` — however, it is evaluated only enough to get a value with which comparisons can be made; specifically, until it results in either an integer or a constructor object[2]. The words following the instruction encode patterns (`pattern_literal` and `pattern_cons`) against which to match the case value. If the case value exactly equals the literal value or function (i.e. constructor) identifier, execution proceeds with the next instruction; otherwise, it skips the number of words specified in the pattern argument. A matching `pattern_else` is required for every `case` which will be executed when no other matches

[2]More precisely, evaluation of that argument will always produces a result in Weak Head-Normal Form (WHNF), but never a lambda abstraction.

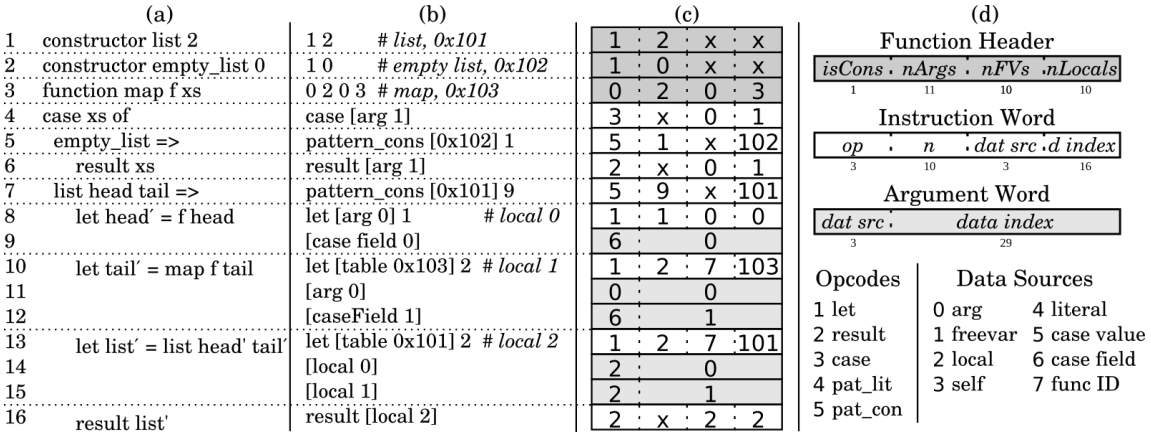


Figure 2.3: How the high-level assembly instructions are directly compiled into a Zarf binary for execution. This example shows the `map` function, along with the `list` constructors, in (a) high-level untyped assembly, (b) machine assembly, and (c) binary. (**a**) The standard linked-list definition requires just two constructors: a `list` is either empty or a 2-element struct containing a head (a value) and a tail (a list) [lines 1-2]. The function `map` takes a function and a list as arguments [line 3]; it builds a new list, applying the function to each list element. If the argument list is empty, it returns an empty list [lines 5-6]. Otherwise, if the list matches against the head/tail constructor [line 7], it applies the function it was given to the list element [lines 8-9], calls `map` recursively on the list tail [lines 10-12], builds a new list [lines 13-15], and yields that new list [line 16]. The `else` branch is not shown. (**b**) In the lowering to machine assembly, names are replaced with local indices, addressing a value on the locals stack (e.g., list' becomes local 2 [line 13]). Function allocations are broken up so that each argument occupies its own word. (**c**) The binary is a direct mapping from the assembly in (b), simply translating ops to opcodes and data sources to integer identifiers. 'x' indicates an unused field. (**d**) Binary encoding. Each word of the binary is either the start of a function, the start of an instruction, or an argument word in a `let` instruction. With no architecturally visible state, data is accessed with a scoped system where the program identifies source and index; all data references use the same source/index pattern.

are found (and demarcates the end of the `case` instruction encoding). Case/pattern sequences not adhering to the encoding described are malformed and invalid — e.g., you cannot skip to the middle of a branch, or have a case without an else branch.

The `result` instruction is a single word, indicating a single piece of data that the current function should yield. Every branch of every function must terminate with a result instruction (disallowing re-convergent branches means the simple pattern-skip mechanism is all that is necessary for control flow). Functions that do not produce a value do not make sense in an environment without side effects, and so are disallowed. After a `result`, control flow passes to the `case` instruction where the function result was required.

We realize that this is a departure from traditional hardware instructions and suggest reference to Figure 2.3 to help ground our descriptions in a concrete example. Figure 2.3 shows a small function, `map`, written in high-level assembly, machine assembly, and encoded as a binary. A more thorough description of the semantics of each of these instructions is found in Section 2.5.

2.3.4 Built-In Functions, I/O, and Errors

ALU operations are, for the most part, already purely mathematical functions — they just map inputs to an output. The Zarf functional ISA is built around the notion of function calls, so no new mechanism or instructions are needed to use the hardware ALU. Invoking a hardware "add" is the same as invoking a program-supplied function. In our prototype, function indices less than 256 (0x100) are reserved for hardware operations; the first program-supplied function, main, is 0x100, with functions numbered up from there. During evaluation, if the machine encounters a function with an index less than 0x100, it knows to invoke the ALU instead of jumping to a space in instruction memory.

The only two functions with side-effects in the system, input and output, are also primitive functions. The input function takes one argument (a port number) and returns a single word from that port; the output function takes two arguments, a port and a value, and writes its result to the port, returning the value written. Since data dependencies are never violated in function evaluation, software can ensure I/O operations always occur in the right order even in a pure functional environment by introducing artificial data dependencies; this is the principle underlying the I/O monad [58, 59], used prominently in languages like Haskell.

In a purely functional system there are no side effects, and thus no notion of an "exception". For program-defined functions, this just requires that every branch of every case return a value (that value could be a program-defined error). However, some invalid conditions resulting from a malformed program can still occur at runtime. To respect the purely functional system, these must cause a non-effectful result that is still distinguishable from valid results. Our solution is to define a "runtime error constructor" in the space of reserved functions. Every function, both hardware- and software-defined, can potentially return an instance of the error constructor. The ISA semantics are undefined in these error cases, because it's very easy to avoid — compiling from any Hindley-Milner typechecked language will guarantee the absence of runtime type errors [60, 61].

2.4 System Software

This section describes the software architecture across the two realms (functional and imperative) of the system, and provides an overview of the ICD and the functional coroutines.

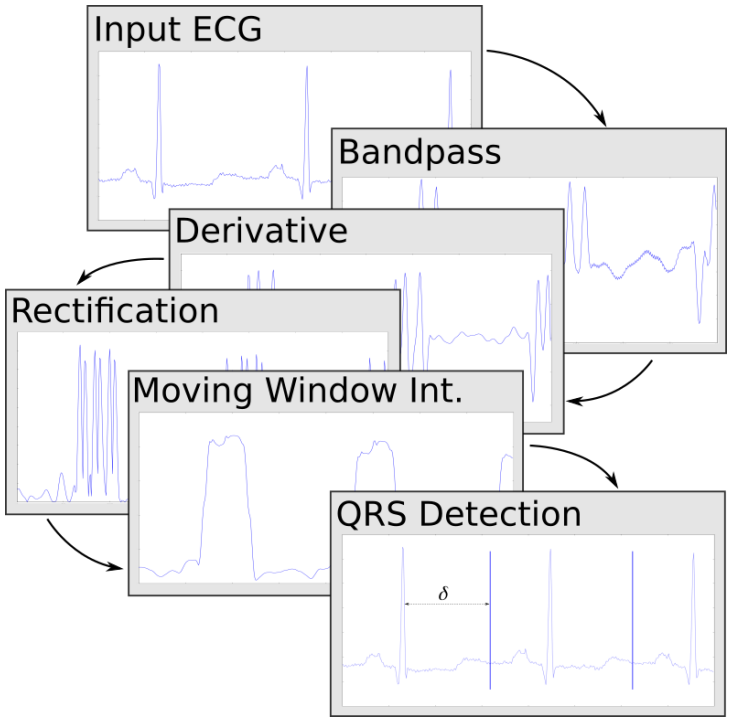


Figure 2.4: The ECG takes input signals sampled at 200 Hz and filters them multiple times, after which the peaks are classified and the rate of heartbeat determined. These values are fed to an ATP (antitachycardia pacing) procedure, which decides if ventricular tachycardia is occurring based on current and previous heart rate, and administers pacing shocks to prevent acceleration and ventricular fibrillation (a form of cardiac arrest).

2.4.1 Functional vs. Imperative

As our system is composed of two small and separate computational layers, the software is split across two different ISAs. For existing applications, or applications prototyped for existing platforms, the decision of which components to migrate to Zarf represents a trade-off of increased abstraction and verification capability for additional development effort and some decrease in performance. Section 2.7 provides some quantitative worst-case bounds for this trade-off.

Zarf runs a small microkernel based on cooperative coroutines [62, 63] to handle the scheduling and communication of different software components. This allows us to more easily group and reason about code in terms of higher-level behaviors — i.e., the small surface area of each coroutine means they can be considered (and occasionally verified) in blocks, as collections of functions with a single specification and interface. The cooperative nature of the system is a design choice that allows us to avoid interrupts, which would complicate proofs of a single coroutine's behavior. Timing analysis (section 2.6.2) ensures each coroutine always returns control.

Zarf enables reasoning about these coroutines at the assembly and binary level. Section 2.6 demonstrates different properties that can be verified. The integrity type system allows a developer to statically prove that a given set of coroutines (and the microkernel itself) will execute in cooperation without one coroutine corrupting values important to another. This *composability* of verification is extremely difficult on traditional architectures, as the global and mutable nature of all state makes it quite easy for any software component to affect any other.

The imperative layer — which can be any embedded CPU, but for our purposes is a Xilinx MicroBlaze processor — runs whatever pieces of the software are not placed on Zarf. This allows for monitoring software, low-level drivers, communication protocols,

and other complex, imperative code to exist and run without requiring modeling or pure-functional implementations. As this area of the system is untrusted and unverified, anything on which the critical components depend should be rewritten to run on Zarf.

In our sample application, three application coroutines are run on Zarf: one that handles the core ICD application, an I/O routine that handles the timing of reading the values from the patient's heart and outputting when shocks should occur, and a routine that sends values to the monitoring software on the imperative layer. The system operates in real-time, reading a single value from the heart, running ECG and ICD processing, and communicating the resulting value back out. In our application, the monitoring software tracks the number of times treatment occurs, and, when prompted from its communication channel, will output that number. This imperative software could be arbitrarily complex and handle more complicated monitoring and diagnosis, communication drivers to communicate with the outside world, or other features; as it is a standard imperative core, any embedded C code can be easily compiled for it with an off-the-shelf compiler.

2.4.2 ICD

ICDs are small, battery-powered, embedded systems which are implanted in a patient's chest cavity and connect directly with the heart. For patients with arrhythmia and at risk for heart failure, an ICD is a potentially life-saving device. Currently, the primary use of ICDs is to detect dangerous arrhthymias (such as ventricular tachycardia, or VT) and administer pacing shocks (anti-tachycardia pacing, or ATP). These shocks help prevent the acceleration in heart rate leading to ventricular fibrillation, a form of cardiac arrest.

From 1990 to 2000, over 200,000 ICDs and pacemakers were recalled due to software

issues [7]. Between 2001 and 2015, over 150,000 implanted medical devices were recalled by the FDA because of life-threatening software bugs [8]. However, ICDs are credited with saving thousands of lives; for patients who have survived life-threatening arrhthymia, ICDs decrease mortality rates by 20-30% over medication [64, 65, 66]. Currently, around 10,000 new patients have an ICD implanted each month [67], and around 800,000 people are living with ICDs [68].

The core of our ICD is an embedded, real-time ECG algorithm that performs QRS[3] detection on raw electrocardiogram data to determine the timing between heartbeats. We work off of an established real-time QRS detection algorithm [69], which has seen wide use and been the subject of studies examining its performance and efficacy [70]. An open-source update of several versions of the algorithm [71] is available; we use the results of this open-source work as the basis of our algorithm's specification as well as the C alternative. After the ECG algorithm detects the pacing between heartbeats, the ATP function checks for signs of ventricular tachycardia and, if found, administers a series of pacing shocks. We implement the VT test and ATP treatment published in [72].

The I/O coroutine is passed the output of the previous iteration of the ICD coroutine. A hardware timer is used to ensure that I/O events occur at the correct frequency. When the correct time has elapsed (5 ms), the I/O coroutine outputs the given value and reads the next input value. It yields this value to the microkernel.

This input is then passed through to the ICD coroutine, which implements a series of filter passes to detect the spacing between QRS complexes (Figure 2.4 illustrates the ECG filter passes). If 18 of the last 24 beats had periods less than 360 ms (corresponding to a heart rate greater than 167 bpm), the ICD coroutine moves into a

[3]The "QRS complex" is made up of the rapid sequence of Q, R, and S waves corresponding to the depolarization of the left and right ventricles of the heart, forming the distinctive peak in an ECG.

treatment-administering state, where it outputs three sequences of eight pulses at 88% of the current heartrate, with a 20 ms decrement between sequences. This is designed to prevent continued acceleration and restore a safe rhythm.

The monitoring software, which runs on the MicroBlaze, receives the output of the ICD coroutine each cycle. A command can be given on the diagnostic input channel for the software to output the number of times treatment has occurred.

I/O events occur at a fixed frequency of 200 Hz. Timing analysis in Section 2.6.2 confirms that, after an input event, the entire cycle of each coroutine running and yielding, including garbage collection, is able to conclude well within the 5 ms window, meaning that the entire system is always able to meet its real-time deadline.

2.5 ISA Semantics

Zarf has the core goal of providing concise, mathematical semantics for its hardware ISA. These can be found in Figure 2.5, which gives the complete ISA behavior using a big-step semantics, explaining how each instruction reduces to a value. This semantics uses eager evaluation for simplicity; though the current hardware implementation uses lazy semantics, the difference is not observable in our application because I/O interactions are localized to a specific function and always evaluated immediately.

The semantics are discussed in more detail in the following subsections. Note that terms introduced in the abstract syntax (Figure 2.2) are used in the semantics. Each rule (or helper function) is applicable in a different case, depending on what is under evaluation; the scenarios are all mutually exclusive, meaning that there is always exactly one rule that can (and should) be applied at every step.

$$c \in \mathit{Constructor} = \mathit{Name} \times \overrightarrow{\mathit{Value}} \qquad \mathit{clo} \in \mathit{Closure} = (\lambda \vec{x}.e) \times \overrightarrow{\mathit{Value}}$$
$$v \in \mathit{Value} = \mathbb{Z} \uplus \mathit{Constructor} \uplus \mathit{Closure} \qquad \rho \in \mathit{Env} = \mathit{Variable} \to \mathit{Value}$$

$$\frac{\vdash e \Downarrow v}{\vdash \overrightarrow{decl}\ \textbf{fun}\ \text{main} = e \Downarrow v}\ (\textsc{program}) \qquad \frac{v = \rho(arg)}{\rho \vdash \textbf{result}\ arg \Downarrow v}\ (\textsc{result})$$

$$\frac{\vec{v}_1 = \rho(\overrightarrow{arg}) \quad v_2 = \texttt{applyCn}(cn, \vec{v}_1) \quad \rho[x \mapsto v_2] \vdash e \Downarrow v_3}{\rho \vdash \textbf{let}\ x = cn\ \overrightarrow{arg}\ \textbf{in}\ e \Downarrow v_3}\ (\textsc{let-con})$$

$$\frac{\begin{array}{c} fn \notin \{\textbf{getint}, \textbf{putint}\} \quad \textbf{fun}\ fn\ \vec{x}_2 = e_2 \in \overrightarrow{decl} \\ \vec{v}_1 = \rho(\overrightarrow{arg}) \quad v_2 = \texttt{applyFn}((\lambda \vec{x}_2.e_2, []), \vec{v}_1, \rho) \quad \rho[x_1 \mapsto v_2] \vdash e_1 \Downarrow v_3 \end{array}}{\rho \vdash \textbf{let}\ x_1 = fn\ \overrightarrow{arg}\ \textbf{in}\ e_1 \Downarrow v_3}\ (\textsc{let-fun})$$

$$\frac{v_1 = \rho(x_2) \quad \vec{v}_2 = \rho(\overrightarrow{arg}) \quad v_3 = \texttt{applyFn}(v_1, \vec{v}_2, \rho) \quad \rho[x_1 \mapsto v_3] \vdash e \Downarrow v_4}{\rho \vdash \textbf{let}\ x_1 = x_2\ \overrightarrow{arg}\ \textbf{in}\ e \Downarrow v_4}\ (\textsc{let-var})$$

$$\frac{\vec{v}_1 = \rho(\overrightarrow{arg}) \quad v_2 = \texttt{applyPrim}(op, \vec{v}_1) \quad \rho[x \mapsto v_2] \vdash e \Downarrow v_3}{\rho \vdash \textbf{let}\ x = op\ \overrightarrow{arg}\ \textbf{in}\ e \Downarrow v_3}\ (\textsc{let-prim})$$

$$\frac{(cn, \vec{v}_1) = \rho(arg) \quad (cn\ \vec{x}\ \Rightarrow\ e_1) \in \overrightarrow{br} \quad \rho[\vec{x} \mapsto \vec{v}_1] \vdash e_1 \Downarrow v_2}{\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v_2}\ (\textsc{case-con})$$

$$\frac{n = \rho(arg) \quad (n\ \Rightarrow\ e_1) \in \overrightarrow{br} \quad \rho \vdash e_1 \Downarrow v}{\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v}\ (\textsc{case-lit})$$

$$\frac{\begin{array}{c} ((cn, \overrightarrow{v_1}) = \rho(arg) \quad (cn\ \vec{x}\ \Rightarrow\ e_1) \notin \overrightarrow{br}) \vee (n = \rho(arg) \quad (n\ \Rightarrow\ e_1) \notin \overrightarrow{br}) \\ \rho \vdash e_2 \Downarrow v_2 \end{array}}{\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v_2}\ (\textsc{case-else})$$

$$\frac{\begin{array}{c} n_2 \text{ is input from port } n_1 \\ \rho[x \mapsto n_2] \vdash e \Downarrow v \end{array}}{\rho \vdash \textbf{let}\ x = \textbf{getint}\ n_1\ \textbf{in}\ e \Downarrow v}\ (\textsc{getint}) \qquad \frac{\begin{array}{c} n_1 \text{ is a port} \\ n_2 = \rho(arg) \quad \rho[x \mapsto n_2] \vdash e \Downarrow v \end{array}}{\rho \vdash \textbf{let}\ x = \textbf{putint}\ n_1\ arg\ \textbf{in}\ e \Downarrow v}\ (\textsc{putint})$$

Figure 2.5: Big-step semantics for Zarf's functional ISA. It is a ternary relation on an environment; a **let**, **case**, or **result** expression; and the value to which it evaluates. Evaluation begins with the main function's body. $\rho[x \mapsto v]$ returns an updated copy of the environment with x mapped to v. **getint** gets an integer from a specified port, and **putint** puts an integer onto a specified port; both are the only mechanisms for I/O.

$$\texttt{applyFn}((\lambda\vec{x}_1.e, \vec{v}_1), \vec{v}_2, \rho) =$$

$$\begin{cases} v & \text{if } |\vec{v}_2| = 0,\ |\vec{v}_1| = |\vec{x}_1|, \text{ and} \\ & \quad \rho[\vec{x}_1 \mapsto \vec{v}_1] \vdash e \Downarrow v \\ (\lambda\vec{x}_1.e, \vec{v}_1) & \text{if } |\vec{v}_2| = 0 \text{ and } |\vec{v}_1| < |\vec{x}_1| \\ \texttt{applyFn}((\lambda\vec{x}_1.e, \vec{v}_1 :+ \texttt{hd}(\vec{v}_2)), \texttt{tl}(\vec{v}_2), \rho) & \text{if } |\vec{v}_2| > 0 \text{ and } |\vec{v}_1| < |\vec{x}_1| \\ \texttt{applyFn}((\lambda\vec{x}_2.e', \vec{v}_3), \vec{v}_2, \rho) & \text{if } |\vec{v}_2| > 0,\ |\vec{x}_1| = |\vec{v}_1|, \text{ and} \\ & \quad \rho[\vec{x}_1 \mapsto \vec{v}_1] \vdash e \Downarrow (\lambda\vec{x}_2.e', \vec{v}_3) \end{cases}$$

$$\texttt{applyCn}(cn, \vec{v}) = \begin{cases} (cn, \vec{v}) & \text{if } (\textbf{con } cn\ \vec{x}) \in \overrightarrow{decl} \text{ and } |\vec{v}| = |\vec{x}| \\ (\lambda\vec{x}.\textbf{let } c = cn\ \vec{x} \textbf{ in result } c, \vec{v}) & \text{if } (\textbf{con } cn\ \vec{x}) \in \overrightarrow{decl} \text{ and } |\vec{v}| < |\vec{x}| \end{cases}$$

$$\rho(arg) = \begin{cases} n & \text{if } arg = n \\ v & \text{if } arg = x \text{ and } (x \mapsto v) \in \rho \end{cases}$$

$$\texttt{applyPrim}(op, \vec{v}_1) = \begin{cases} v & \text{if } |\vec{v}_1| = \texttt{arity}(op) \text{ and} \\ & \quad v = \texttt{eval}(op, \vec{v}_1) \\ (\lambda\vec{x}_1.\textbf{let } x_2 = op\ \vec{x}_1 \textbf{ in result } x_2, \vec{v}_1) & \text{if } |\vec{v}_1| < \texttt{arity}(op) \text{ and} \\ & \quad |\vec{x}_1| = \texttt{arity}(op) \end{cases}$$

Figure 2.6: Big-step semantics helpers for Zarf's functional ISA. `applyFn` (`applyCn`) performs function (constructor) application. $\vec{x} :+ y$ appends y to the end of $\vec{x}$, creating a new list. $|\vec{x}|$ means length of the list $\vec{x}$. `eval` returns the value of applying a primitive operation to arguments. Because functions are lambda-lifted, our version of closures track the list of values to be applied upon saturation, rather than an entire environment like normal closures.

2.5.1 Names and Programs

A `Constructor` is a unique name and a list of zero or more values. Constructors serve as software data types, as a simple system for building up more complex data objects. The name indicates the "type" of the constructor, encoded statically as a unique integer, which the machine uses at runtime to distinguish constructors of different types.

A `Closure` is a function object, tying a function to a list of zero or more values, which are the arguments that have already been supplied. Closures allow for the dynamic construction of function objects from statically defined functions: e.g., applying the argument `1` to the static binary function `add` creates a new closure, which expects one argument, that performs the function $\lambda x.x + 1$.

A `Value` is either an integer, a constructor, or a closure. The machine uses one bit at runtime to track which values are primitives and which are objects (either constructors or closures), and identifies constructor types with their name (unique type integer), but is otherwise untyped.

An `Environment` is a semantic entity mapping variables (local names) to values (integers, constructors, and closures). The semantics are high-level and do not specify how the machine should implement the behavior of the environment, but function arguments and local variables fall into the environment of each function. The set of functions (declared at the top level) are stored in a list of declarations $\overrightarrow{decl}$, which is essentially a map from the function's name to its parameter list and body.

The `PROGRAM` rule states that there should be a set of zero or more function and constructor declarations, and one function `main` with a body expression e. Given the declarations and main function, and application of the semantic rules, we can reduce e to a value.

2.5.2 Result

The RESULT rule states that, given the current function environment and a `result` instruction with argument *arg*, we can reduce the current function execution to a single value *v* using the environment to look up *arg*, if *arg* is a variable, or simply returning it, if *arg* is a number.

2.5.3 Let

A `let` instruction will be reduced using one of four rules: LET-FUN, LET-CON, LET-VAR, or LET-PRIM. The first is used for static program function application; i.e., applying arguments to a program-defined function (which excludes I/O and hardware functions). Similarly, the second is used for static constructor application; i.e., applying arguments to a program-defined constructor. The third is used to apply arguments to a runtime value, which will be a closure expecting additional arguments. The final is application on primitive (hardware ALU) functions. I/O functions (`getint` and `putint`) have separate rules.

LET-FUN is used when the instruction under evaluation is a `let` instruction applying zero or more arguments to a program-defined function fn. Its premises state that the function should not be `getint` or `putint`, that the function should be defined in the program declarations, that the arguments should all be reducible to a sequence of values $\overrightarrow{v_1}$ using the current function environment, that application of the `applyFn` helper rule on the body of fn with arguments $\overrightarrow{v_1}$ will result in a value v_2, and finally, that binding a new local variable to v_2 will allow us to reduce the remainder of the instructions in the current function to a value.[4] The final premise of the rule ($\rho[x_1 \mapsto v_2] \vdash e_1 \Downarrow v_3$)

[4] As these are big-step semantics, rules indicate how expressions reduce directly to values, rather than giving step-by-step instructions for performing the reduction. In evaluating LET-FUN, for example, the rule does not instruct you to "call" into the indicated function, but rather just states that the function reduces to a value (as it must, eventually), then uses the value; as we are writing mathematical expressions, the

continues the execution: e_1 is the remainder of the instructions, where the environment now includes a mapping from x_1 to the newly calculated value v_2. A premise of this format occurs in every rule for non-terminal instructions (everything but `PROGRAM` and `RESULT`); the semantics treat the instructions as recursive, such that each instruction "points" to the rest of the instructions.

The premises for `LET-CON` are very similar to those of `LET-FUN`, with the primary difference coming in `applyCn`. While function applications can be oversaturated, we disallow oversaturation of constructor applications for simplicity; we haven't found this overly restrictive at all in practice. Otherwise, the rules behaves similarly, storing the value that results from applying arguments to a constructor into the environment ρ before continuing to evaluate the next expression e to a value v_3. In this way, closures and constructors are structurally the same; both are function identifiers with a sequence of arguments. The difference is that closures are already fully evaluated.

`LET-VAR` is used when a `let` instruction applies arguments to a dynamic (runtime) value x_2. The premises state that x_2 should be reducible via the environment to a value v_1, that the sequence of arguments are reducible to a sequence of values $\overrightarrow{v_2}$, that applying the arguments $\overrightarrow{v_2}$ to the object v_1 with the `applyFn` operation will result in a value v_3, and that binding a local variable to that value will allow us to reduce the remainder of the instructions to a value. The `applyFn` is written to only accept closures as its first arguments; in calling it, there is an implicit premise that v_1 is a closure object. x_2 reducing to any other type of value (an integer or constructor) is a runtime error, and can occur only if the program is not well-typed.

`LET-PRIM` is similar to `LET-FUN`, but is used only when the static function is a primitive operation. These functions must be treated differently because there is no program

reduction is assumed to occur immediately. One can still "execute" the semantics by stepping through, evaluating operands as necessary using the rules in places that the semantics simply reduce immediately to a value.

definition for `add`, or `sub`, or `mult`; during machine execution, the hardware ALU handles them. The semantics define ALU operations the same way as the machine (as 32-bit modular mathematical operations). The premises of `LET-PRIM` contain no function declaration; in addition, they invoke `applyPrim` instead of `applyFn`. Otherwise, the application is similar: the function call reduces to a value, and binding that value to a local variable will allow us to reduce the remainder of the instructions in the function under evaluation.

2.5.4 I/O

`GETINT` is the rule for the hardware-defined input function; it is invoked when a `let` instruction uses `getint` in a program, which takes a single argument n_1. The premises state that reading a value from port n_1 should return an integer n_2; binding that value to a local variable, we can proceed with evaluation.

`PUTINT` is the rule for the output function (also hardware-defined); it takes two values: a port number n_1, and an *arg* that should be output. The premises use the environment to reduce *arg* to a value; not stated (as it is a side effect) is that the hardware sends this integer value to the indicated port. The integer sent is also used as the value to which the instruction reduces, so it is bound to a local variable, and evaluation proceeds.

2.5.5 Case

We use two rules for the evaluation of case instructions (`CASE-CON` and `CASE-LIT`, for constructor and integer scrutinees, respectively); in addition, the rule `CASE-ELSE` handles the "else" branch for each of the two case varieties.

These `CASE` rules are invoked when a `case` instruction is under evaluation; which

rule is used depends on what the scrutinee reduces to: if it is a constructor, `CASE-CON` is used, while `CASE-LIT` handles integers. A `case` instruction includes a series of zero or more branches, each of which has a constructor name or integer as a guard, and one `else` branch. Exactly one branch will always execute. Since constructor "names" become unique integers, constructor and integer matches must be encoded differently to distinguish which variety each branch is meant to match.

The first premise of `CASE-CON` indicates that the scrutinee *arg* must reduce to a constructor; the second says to take the expression from the branch with the matching constructor name and use that for the remainder of the function, which will reduce to a value. `CASE-LIT` is similar, but requires the *arg* to reduce to an integer, and takes the expression from the branch that attempts to match exactly that integer.

The `CASE-ELSE` rule is invoked when no matching branch is found for the `case` instruction (indicated in the disjuncted premises); there, it indicates to take the expression from the `else` branch in the instruction and use that for the remainder of the function.

2.5.6 `apply` Helper Functions

`applyFn` is perhaps the most complicated rule, because it is where currying is handled — different actions must be taken if too few, too many, or just enough arguments are supplied to a function application. The helper rule takes a closure and a list of zero or more values, which will be arguments to the closure.

1. If zero arguments are supplied, and there are exactly enough values already saved and ready to be applied in the closure, then simply feed those values as the arguments to the function, reducing the function body to a value, and return that.

2. If zero arguments are supplied, and there are not enough saved values in the closure, return the same closure.

3. If at least one argument is supplied, and there are not enough saved values in the closure, recursively call into `applyFn`, taking the first argument from the list and appending it to the list of saved values. Either the argument list will run out before the function is saturated (resulting in case 2), or the function will eventually be saturated (resulting in case 1 or 4).

4. If at least one argument is supplied, and there are exactly enough saved values already in the closure, then we evaluate the closure (which must, if the program is well-typed, result in another closure), then recursively call `applyFn` with the new closure and the same argument list.

`applyCn` handles applications of arguments to constructors. The first rule is if exactly enough arguments are applied to the constructor application; in that case, a constructor containing those values is built and returned. The second handles the case where fewer values were supplied than expected (partially saturating a constructor); in this case, a closure is returned to capture the values already supplied, and when additional values are applied, it will invoke `applyCn` again to check if the constructor has all fields necessary to be built. We note that this appears to be dynamically creating syntax (the rule indicates placing a `let` instruction into the created closure), but as every constructor has a finite number of fields, these functions are all known statically.

`applyPrim` handles evaluation of primitive operations. The first case is invoked when the correct arguments have been supplied for that particular operation, and simply evaluates the operation according to the rule for 32-bit modular arithmetic, returning the answer. Similar to `applyCn`, we must account for the case where the function did not receive enough arguments; as there, we create a new closure to capture the arguments supplied, and the operation can be evaluated once all arguments are received (the second case).

The $\rho(arg)$ = ... helper function is for convenience, simply stating that if *arg* is an integer, return that value; otherwise, it must be a name mapped to some value in the environment, in which case that value should be returned.

2.6 Verification

We separate the verification of the embedded ICD application into three parts: verification of the correctness of the ICD coroutine, a timing analysis to show that the assembly meets timing requirements in the worst case, and a proof of non-interference between the trusted ICD coroutine and untrusted code outside of it.

2.6.1 Correctness

We first implement a high-level version of the application's critical algorithms (the ECG filters and ATP procedure) in Gallina, the specification language of the Coq theorem prover [73], using this version as our specification of functional correctness. This specification operates on streams — a datatype that represents an infinite list — by taking a stream as input and transforming it into an output stream. By sticking to a high-level, abstract specification, we can be more confident that we have specified the algorithm correctly. An ICD implementation cannot operate on streams, as all data is not immediately available; instead, it takes a single value, yields a single value, and then repeats the process.

The form of the correctness proof is by refinement: first, we create a Coq implementation of the ICD algorithm that is "lower-level" than the Coq specification. This lower-level implementation operates on machine values rather than streams, isolates function applications to let expressions, and avoids the use of "if-then-else" expressions, among other trivially-resolved differences. We then create an extractor that converts

(a)

```
CoFixpoint threshold_rec_hl
    (xs: Stream Z) (pBCnt tmpPeak: Z) ...
        : (Stream Z) :=
  let x := Str_nth 0 xs in
  match filter_pks_hl pBCnt tempPeak x with
    | (x', preBlankCt', tempPeak') => ...
```

⇓ **(b)**

```
CoFixpoint threshold_rec_ll
    (x pBCnt tmpPeak ... : Z) :=
  let fres := filter_pks_ll pBCnt tmpPeak x in
  match filterres with
    | mk_tuple3 preBlankCt' tmpPeak' x' => ...
```

⇓ **(c)**

```
fun threshold_rec x pBCnt TmpPeak ... =
  let fres = filter_pks pBCnt tmpPeak x in
  case fres of {
    tuple3 preBlankCt' tmpPeak' x' => ...
```

Figure 2.7: Extraction of verified application components, summarized for a small excerpt. (**a**) The high-level Coq specification is written to operate on `Streams` (infinite lists); values are pulled from the front of the stream. (**b**) An intermediate version is written in Coq which operates on integers instead of streams, and unfolds nested operations so each function call and arithmetic operation takes one line. This intermediate version is proven equivalent in Coq to the high-level specification — meaning that repeated recursive application of (b) will always output the same sequence of values as (a). (**c**) A simple extractor just replaces the keywords in (b) to produce valid assembly code that can run on Zarf.

this lower-level Coq code directly into executable Zarf functional assembly code (see Figure 2.7). If, for all possible input streams, we can prove that the output stream produced by the high-level Coq specification is the same sequence of values produced by the lower-level implementation, we can conclude that the program we run on Zarf is faithful to the high-level Coq specification. This proof of equivalence between the two Coq implementations is done by induction and coinduction over the program, showing that if output has matched up to point N, and the computation of value N is equivalent, then value $N + 1$ will be equivalent as well. As compared to extracting for an imperative architecture, we avoid needing to compile functional operations to an imperative ISA and do not require a large software runtime — or any software runtime at all. The translation simply replaces Coq keywords with Zarf assembly keywords, which is possible because the low-level Coq specification is in the A-normal form that Zarf requires. For example, the Coq keyword `CoFixpoint` would be textually replaced with `fun`, `match` replaced with `case`, etc.

We begin constructing our proof by first defining the relevant datatypes and expressions of the ISA as inductively defined mathematical objects:

```
Inductive data : Set :=
  | local : nat -> data
  | arg   : nat -> data
  | caseValue : data
  | caseField : nat -> data
  | literal : Z -> data
  | function  : string -> data
  | functionApplication : string -> list data -> data.

Inductive zarf_inst : Set :=
  | apply' : data -> list data -> zarf_inst
```

```
  | case  : data -> list alt -> zarf_inst
  | ret   : data -> zarf_inst
with alt : Set :=
  | literalAlt : Z -> list zarf_inst -> alt
  | constructorAlt : string -> list zarf_inst -> alt.

Inductive zarf_table : Set :=
  | function_table : string -> nat -> list zarf_inst -> zarf_table
  | constructor_table : string -> nat -> zarf_table.
```

We're then able to define a small interpreter that executes the instruction semantics (several constructors have been omitted for brevity and clarity of presentation):

```
Inductive zarf_run_function' : string -> list data -> list data -> data ->
    list zarf_inst -> data -> Prop :=
| H_apply' x res ys zs (args:list data) f locals caseVal :
    zarf_run_function' f args (locals ++ execApply x ys args locals caseVal) caseVal
        zs res -> zarf_run_function' f args locals caseVal (apply' x ys :: zs) res
...
| H_matchesLiteral xrep z zs alts d res (l:list zarf_inst) f args locals caseVal x :
    getData x args locals caseVal = Some xrep ->
    matchesCase xrep (literalAlt d l) = true ->
    zarf_run_function' f args locals xrep l res ->
    zarf_run_function' f args locals caseVal
        (case x (literalAlt d l :: z :: alts) :: zs) res
| H_ret x args locals caseVal f xrep zs : getData x args locals caseVal = Some xrep
    -> zarf_run_function' f args locals caseVal (ret x :: zs) xrep .

Inductive zarf_run' (t : zarf_table) (args : list data) : data -> Prop :=
| Run_function f arity insts res : t = function_table f arity insts ->
                zarf_run_function' f args [] (function "error") insts res ->
                zarf_run' t args res.
```

The `zarf_run_function'` interpreter is made up of several cases, each of which correspond to rules in the big-step semantics defined in Figure 2.5. For example, the first case, `H_apply'`, defines what function application means. It says that if we add a thunk to our local environment that is the result of executing the apply operation (`locals + execApply x ys args locals caseVal`), and *then* execute the rest of the instructions `zs`, we have successfully executed an apply statement followed by the rest of the instructions `zs`.

We declare several axioms asserting the correctness of the ISA's built-in functions and their equivalence to built-in Coq operations. These built-in functions (like add, multiply, etc., labelled *PrimOp* in Figure 2.2) are ultimately the base operations employed by all user-defined functions, and we reference them during the proof of correctness of the higher-level ECG algorithms later on. We also define a few axioms related to list and pair construction, whose correspondence to the user-made Zarf function equivalents are trivial. Here are a few assorted examples:

```
Axiom cons_rep : forall l' A (x':A) xs', l' == x' :: xs' -> exists x xs,
    l' = (functionApplication "cons" [x; xs]) /\ x == x' /\ xs == xs'.
Axiom snd_same : forall (A B : Type) x (x' : A*B), x == x' ->
    functionApplication "snd" [x] == snd x'.
Axiom add_same : forall x y x' y', x == x' -> y == y' -> x + y ==
    functionApplication "add" [x'; y'].
```

We can then begin defining and proving things about non-builtin functions, like append, reverse, index, and various matrix multiplication operations. Here's an example of defining the simplest user-defined function, `id`, which is used often in Zarf assembly for assigning a constant numeric value to a variable (because `let` expressions purely operate over function identifiers (see Figure 2.2)). Proof bodies are omitted for brevity:

```
Definition zarf_id : zarf_table :=
```

```
  function_table "id" 1 [ret (arg 0)].

Axiom id_eta : forall x res, zarf_run' zarf_id [x] res ->
    functionApplication "id" [x] = res.

Theorem equivalence_id :
  forall A (x:A) x', x == x' ->
    exists res, zarf_run' zarf_id [x'] res /\ res == x.

Lemma id_same :
  forall (A : Type) x (x' : A), x == x' ->
    functionApplication "id" [x] = x.
```

We then proceed to define several ECG helper functions, like the low pass filter, at both a high-level stream-based abstraction and the lower-level implementation which operates on machine values, and verify their equivalence (and thus the correctness of the machine's implementation of the specification):

```
(* Define Zarf implementation of the low pass filter as a series of Zarf instr. *)
Definition zarf_lpf : zarf_table :=
  function_table "lpf" 0 [ apply' (function "lpf_rec") ...<arguments>...].
Definition zarf_lpf_rec : zarf_table :=
  function_table "lpf_rec" 13 [
    apply' (function "mult") [arg 1; literal 2];
    ...
    apply' (function "tuple2") [local 7; local 6];
    ret (local 8)
  ].

(* Define the high-level Coq stream-based specification of the low pass filter *)
CoFixpoint low_pass_filter_rec (xs: Stream Z) (ys: list Z) : (Stream Z) := ...
```

```
(* Declare that the result of the running the "lpf_rec" Zarf function
is the same that results from running the low-pass filter function
through our Coq-defined Zarf interpreter *)
Axiom lpf_rec_eta : forall xs' res, zarf_run' zarf_lpf_rec xs' res ->
    functionApplication "lpf_rec" xs' = res.

(* Define the non-stream-based LPF function *)
CoFixpoint low_pass_filter_rec2
  (y2 y1 x10 x9 x8 x7 x6 x5 x4 x3 x2 x1 x0: Z) : Tuple2 Z :=
  let y1m2 := Z.mul 2 y1 in
  let x5m2 := Z.mul 2 x5 in
  let t0 := Z.sub y1m2 y2 in
  ...

(* Prove the two implementations are equivalent *)
Theorem same_low_pass_filter_rec_and_rec2 :
  forall xs ret1 ret2
         y2 y1 x10' x9' x8' x7' x6' x5' x4' x3' x2' x1' x0',
    (forall i, i >= 0 -> i <= 10 -> Str_nth i xs =
         nth i [x10';x9';x8';x7';x6';x5';x4';x3';x2';x1';x0'] 0%Z)%nat ->
    low_pass_filter_rec xs [y2;y1] = ret1 ->
    low_pass_filter_rec2 y2 y1 x10' x9' x8' x7' x6' x5' x4' x3' x2' x1' x0' = ret2
    -> StreamTuple2Same 10 xs ret1 ret2.
...
```

After we've proven the `same_low_pass_filter_rec_and_rec2` theorem, we can convert the low-level low pass filter implementation directly into Zarf assembly. In this example, the assembly version of `low_pass_filter_rec2` would look like:

```
fun lpf_rec y2 y1 x10 x9 x8 x7 x6 x5 x4 x3 x2 x1 x0 =
    let y1mult2 = mult y1 2 in
```

```
let x5mult2 = mult x5 2 in
let t0 = sub y1mult2 y2 in
...
```

The preceding snippets of code have been just a sample of the entire set of proofs we wrote. The full proofs of correctness of the assembly-level critical ECG and ATP functions take under 2,500 lines of Coq. The implementations are converted line-for-line into Zarf assembly code, which is combined with assembly for the microkernel and other coroutines.

In total, the Trusted Code Base for the correctness proof includes: the hardware, the Coq proof assistant, and the small extractor that converts the low-level Coq code into Zarf functional assembly code. All other code is untrusted and may be incorrect, and the proof will still hold. The high-level ISA and clearly-defined semantics make this very small TCB possible, allowing the exclusion of language runtimes, compilers, and associated tooling that is frequently present in the TCB in verification efforts.

2.6.2 Timing

With a knowledge of how the Zarf hardware executes each instruction, we create worst-case timing bounds for each operation. In general, in a functional setting, unbounded recursion makes it impossible to statically predict execution time of routines. Though our application uses infinite recursion to loop indefinitely, the goal is to show that each iteration of the loop meets the real-time deadline; within that loop, each coroutine is executed only once, and no functions call into themselves. This allows us to compute a total worst-case execution time for the sum of all the instructions by extracting the worst-case route through the hardware state machine to execute each possible operation. For example, applying two arguments to a primitive ALU function

and evaluating it has a maximum runtime of 30 cycles — this includes the overhead of constructing an object in memory for the call, performing a function call, fetching the values of the operands, performing the operation, marking the reference as "evaluated" and saving the result, etc. In an average case, only a fraction of the possible overhead will actually be invoked (see Section 2.7 for CPI averages).

Hardware garbage collection is a complicating factor on timing. GC can be configured to run at specific intervals or when memory usage reaches a certain limit; for our application, to guarantee real-time execution, the microkernel calls a hardware function to invoke the garbage collector once each iteration. To reason about how long the garbage collection takes, we bound the worst-case memory usage of a single iteration of the application loop. The hardware implements a semispace-based trace collector, so collection time is based on the live set, not how much memory was used in all. For the trace-collector state machine, each live object takes N+4 cycles to copy (for N memory words in the object), and it takes 2 cycles to check a reference to see if it's already been collected. We bound the worst-case by conservatively assuming that all the memory that is allocated for one loop through the application might be simultaneously live at collection time, and that every argument in each function object may be a reference which the collector will have to spend 2 cycles checking.

From the static analysis, we determine that the worst execution of the entire loop is 4,686 cycles, not including garbage collection. Garbage collection is bounded by a worst-case of 4,379 cycles, making a total of 9,065 cycles to run one iteration of system — or 181.3 μs on our FPGA-synthesized prototype running at 50 MHz, falling well-within the real-time deadline of 5 ms.

2.6.3 Non-Interference

Because the ICD coroutine has been proven correct (Section 2.6.1), we treat its output as trusted. This output must then travel through the rest of the cooperative microkernel until it reaches the outside world via the I/O coroutine's `putint` primitive. In order to guarantee the integrity of this data (meaning it is never corrupted nor influenced by less-trusted data), we rely on a proof of non-interference. Non-interference means that "values of variables at a given security level $\ell \in \mathcal{L}$ can only influence variables at any security level that is greater than or equal to ℓ in the security lattice $\mathcal{L}$" [74]. In a standard security lattice, **L** (low-security) $\sqsubseteq$ **H** (high-security), meaning that high-security data does not flow to (or affect) low-security output. In our application, however, we are concerned with integrity; our lattice is composed of two labels, **T** (trusted) and **U** (untrusted), organized such that **T** $\sqsubseteq$ **U**. Therefore, our integrity non-interference property is that untrusted values cannot affect trusted values [16].

To prove this about Zarf, we create a simple integrity type system that provides a set of typing rules to determine and verify the integrity type of each expression, function, and constructor in a program. After providing trust-level annotations in a few places and constraining the normal Zarf semantics slightly to make type-checking much easier, we can run a type-checker over the resulting Zarf code to know whether it maintains data integrity. We extend the original Zarf syntax to allow for these type annotations, as follows:

$$\ell, \text{PC} \in \mathit{Label} ::= \textbf{T} \mid \textbf{U}$$

$$\tau \in \mathit{Type} ::= \textbf{num}^{\ell} \mid (cn, \vec{\tau}) \mid (\vec{\tau} \rightarrow \tau)$$

$$\mathit{func} \in \mathit{Function} ::= \textbf{fun}\ \mathit{fn}\ x_1 : \tau_1, \ldots, x_n : \tau_n\ : \tau\ \textbf{=}\ e$$

$$\mathit{cons} \in \mathit{Constructor} ::= \textbf{con}\ cn\ x_1 : \tau_1, \ldots, x_n : \tau_n$$

Specifically, following the spirit of Abadi et al. [11] and Simonet [75], types are inductively defined as either labeled numbers, or functions and constructors composed of other types. Our proof of soundness on this type system follows the approach done in work by Volpano et al. [12]. We show that if an expression e has some specific type τ and evaluates to some value v, then changing any value whose type is less-trusted than e's type results in e evaluating to the same value v; thus, we show that arbitrarily changing untrusted data cannot affect trusted data. We prove soundness case-wise over the three types of expressions in our language, combining our evaluation semantics with our security typing rules.

Integrity Type System

The integrity type system is found in Figure 2.8. Our integrity lattice is composed of two elements, **T** and **U** (trusted and untrusted, respectively), such that **T** < **U** (opposite of a normal *security* lattice). We extended the Zarf ISA by requiring function and constructor type annotations. Constructors, which previously were untyped, are now singleton types: each constructor declaration defines a type, but that constructor is the sole inhabitor of the type. This restriction eliminates **case** expressions as sources of control flow when casing on a constructor type (since we know statically that the **case** expression's scrutinee will be a single unique value, and therefore also statically know which branch will be taken); note that this also eliminates the consideration of the **else** branch in a case expression on a constructor type. Instead, **case** expressions in this lightly-typed Zarf are solely for binding a constructor's internal values to variables (via deconstruction). Though this causes a loss in expressive power in the general case (constructors must be singleton types), our microkernel was designed without the need for this type of control flow.

case expressions whose scrutinee is a number, however, still allow for control flow

$$\ell, \text{PC} \in \mathit{Label} ::= \mathbf{T} \mid \mathbf{U} \quad \tau \in \mathit{Type} ::= \mathbf{num}^{\ell} \mid (cn, \vec{\tau}) \mid (\vec{\tau} \to \tau)$$
$$\Gamma \in \mathit{Env} = \mathit{Identifier} \to \mathit{Type}$$

$$\frac{\Gamma[i_1 \mapsto \tau_1, \ \ldots, i_n \mapsto \tau_n] \vdash e : \tau}{\Gamma \vdash (\mathbf{fun}\ \mathit{fn}\ i_1 : \tau_1, \ldots, i_n : \tau_n\ : \tau = e) : (\tau_1, \ \ldots, \ \tau_n) \to \tau} \ (\textsc{func})$$

$$\frac{\tau_1 = \Gamma(id) \quad \overrightarrow{\tau_2} = \Gamma(\overrightarrow{arg}) \quad \tau_3 = \texttt{applyType}(\tau_1, \overrightarrow{\tau_2}) \quad \Gamma[x \mapsto \tau_3] \vdash e : \tau_4}{\Gamma \vdash \mathbf{let}\ x = id\ \overrightarrow{arg}\ \mathbf{in}\ e : \tau_4} \ (\textsc{let})$$

$$\frac{(cn, \vec{\tau}_1) = \Gamma(arg) \quad (cn\ \vec{x}\ \Rightarrow\ e_1) \in \overrightarrow{br} \quad \Gamma[\vec{x} \mapsto \vec{\tau}_1] \vdash e_1 : \tau_2}{\Gamma \vdash \mathbf{case}\ arg\ \mathbf{of}\ \overrightarrow{br}\ \mathbf{else}\ e_0 : \tau_2} \ (\textsc{case-cons})$$

$$\frac{\begin{array}{c}\mathbf{num}^{\ell} = \Gamma(arg) \quad \Gamma \vdash e_1 : \tau_1 \ \ldots\ \Gamma \vdash e_n : \tau_n \quad \Gamma \vdash e_0 : \tau_0 \\ \tau = (\tau_0 \sqcup \tau_1 \sqcup \ldots \sqcup \tau_n) \bullet \ell\end{array}}{\Gamma \vdash \mathbf{case}\ arg\ \mathbf{of}\ n_1 \Rightarrow e_1\ \ldots\ n_n \Rightarrow e_n\ \mathbf{else}\ e_0 : \tau} \ (\textsc{case-lit})$$

$$\frac{\tau = \Gamma(arg)}{\Gamma \vdash \mathbf{result}\ arg : \tau} \ (\textsc{result}) \qquad \frac{\Gamma[x \mapsto \mathbf{num}^{\mathbf{T}}] \vdash e : \tau}{\Gamma \vdash \mathbf{let}\ x = \mathbf{getint}\ n\ \mathbf{in}\ e : \tau} \ (\textsc{getint})$$

$$\frac{\mathbf{num}^{\ell} = \Gamma(arg) \quad \Gamma[x \mapsto \mathbf{num}^{\ell}] \vdash e : \tau}{\Gamma \vdash \mathbf{let}\ x = \mathbf{putint}\ n\ arg\ \mathbf{in}\ e : \tau} \ (\textsc{putint})$$

Figure 2.8: Integrity typing rules. A type is inductively defined as either a labelled number, a singleton constructor, or a function constructed of these types. The type environment maps variables, function, and constructor names to types. Since all functions are annotated with their types, type checking proceeds by ensuring that the return type of a function is the same as the type deduced by checking the function's body expression with the function's parameter types added to the type environment. $\sqcup$ denotes the join of two types, and $\bullet$ denotes the joining of a type's integrity label with another.

$$\texttt{applyType}((\vec{\tau}_1 \to \tau), \vec{\tau}_2) =$$
$$\begin{cases} \tau & \text{if } |\vec{\tau}_1| = 0 \text{ and } |\vec{\tau}_2| = 0 \\ (\vec{\tau}_1 \to \tau) & \text{if } |\vec{\tau}_1| > 0 \text{ and } |\vec{\tau}_2| = 0 \\ \texttt{applyType}(\vec{\tau}_3 \to \tau, \vec{\tau}_4) & \text{if } \vec{\tau}_1 = \tau_1 :: \vec{\tau}_3,\ \vec{\tau}_2 = \tau_2 :: \vec{\tau}_4, \text{ and } \tau_2 \leq \tau_1 \\ \texttt{applyType}(\vec{\tau}_3 \to \tau_4, \vec{\tau}_2) & \text{if } |\vec{\tau}_1| = 0,\ |\vec{\tau}_2| > 0, \text{ and } \tau = (\vec{\tau}_3 \to \tau_4) \end{cases}$$

$$\Gamma(\text{n}) = \textbf{num}^{\text{PC}} \quad \Gamma(id) = \begin{cases} (\vec{\tau} \to (cn, \vec{\tau})) & \text{if } id = cn \text{ and } \textbf{con}\ cn\ \vec{x}{:}\vec{\tau} \in \overrightarrow{decl} \\ (\vec{\tau} \to \tau) & \text{if } id = fn \text{ and } \textbf{fun}\ fn\ \vec{x}{:}\vec{\tau}\ {:}\tau = e \in \overrightarrow{decl} \\ \tau & \text{if } id = x \text{ and } (x \mapsto \tau) \in \Gamma \end{cases}$$

Figure 2.9: Integrity typing rules helpers. Γ is a helper function that gets the type of an argument, and `applyType` applies a function type to argument types. Applying a helper function that takes one argument to a list of arguments is shorthand for mapping that function over the list.

$$\textbf{num}^{\ell} \sqcup \textbf{num}^{\ell'} = \textbf{num}^{\ell \sqcup \ell'}$$
$$(cn, \vec{\tau}) \sqcup (cn, \vec{\tau}) = (cn, \vec{\tau})$$
$$(\vec{\tau} \to \tau) \sqcup (\vec{\tau'} \to \tau') = (\vec{\tau} \sqcup \vec{\tau'} \to \tau \sqcup \tau')$$
$$\textbf{num}^{\ell} \bullet \ell' = \textbf{num}^{\ell \sqcup \ell'}$$

Figure 2.10: Joining two types. The $\bullet$ operator is used to join a type's label with another label; if the type that the label is being joined with is not a **num**, the label will be joined with each of the type's inner types until a base **num** is reached. Joining two lists of types is equal to the pairwise join of their elements. Constructor join is trivial because constructors are singletons whose type never changes, and only equal constructors can be compared.

$$\frac{\ell \sqsubseteq \ell'}{\mathbf{num}^{\ell} \leq \mathbf{num}^{\ell'}}\ (\textsc{num}) \qquad \frac{\vec{\tau'} \leq \vec{\tau} \quad \tau \leq \tau'}{(\vec{\tau} \to \tau) \leq (\vec{\tau'} \to \tau')}\ (\textsc{func})$$

$$\frac{}{(cn, \vec{\tau}) \leq (cn, \vec{\tau})}\ (\textsc{cons}) \qquad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ (\textsc{tran})$$

Figure 2.11: Subtyping rules. One type is a subtype of another if their base types are equal and, in the case of the base num type, the first's label is lower in the integrity lattice than the other. A list is a subtype of another if pairwise each element of the first is a subtype of the corresponding element in the other list.

(since the value of an **num** is *not* known ahead of time); therefore, the type of this form of case expression is the join of all of its branch types. The type of the scrutinee, which is significant in a security analysis, is here irrelevant — there are no implicit flows for integrity. Because we do not use union types, another small restriction we enforce is that each branch in a case expression must result in the same base type (i.e. all must either type-check to a **num**, $(cn, \vec{\tau})$, or $\vec{\tau} \to \tau$), such that we may join them together properly (see Figure 2.10).

The integrity label associated with a **num** depends on the integrity level of the code that created it: untrusted code can only create numbers of type $\mathbf{num}^{\mathbf{U}}$, while trusted code can create trusted numbers (which can be treated as untrusted numbers via subtyping; see Figure 2.11). Primitive operations (add, subtract, etc.) are treated as named functions contained within the set of declarations $\overrightarrow{decl}$. The type of primitive operators is dependent on the trust level of the caller: for example, the type of add is $\mathbf{num}^{\ell_1} \to \mathbf{num}^{\ell_2} \to \mathbf{num}^{\ell_1 \sqcup \ell_2 \sqcup \mathrm{PC}}$, where PC represents the trust level of the current program location (we assume its value can be tracked and changed outside of the type system proper). This all implies that untrusted code cannot use the primitive operations to create any type of trusted value (regardless of the types of the numbers an untrusted caller uses), thus restricting untrusted code's ability to obtain trusted values to (1) the

`getint` function (which in our application is data straight from the heart monitor) and (2) by calling trusted functions which return trusted values.

This system will verify integrity for a value with singular endpoints — i.e., for the code being checked, it is received at one point and sent at one point. More complex annotations and treatment of values, like an arbitrary number of mutually untrusted but critical values passing through an arbitrary number of trusted and untrusted regions, can be guaranteed with this type-system via piecewise checking. By guaranteeing each link in the chain one-at-a-time, the integrity of the chain is verified.

The soundness proof of the integrity type system proceeds by cases on the three forms of expressions.

Lemma 1 (Case Expression Soundness)**.** *If* $\forall x\ e_0\ e_1\ arg_1\ arg_2\ \tau_0\ \tau_1\ cn_1\ \vec{\tau}_1\ v_1\ v_2\ \rho\ \Gamma$, *where*

1. $e_0 = (\textbf{\textit{case}}\ \text{arg}_1\ \textbf{\textit{of}}\ \overrightarrow{br}\ \textbf{\textit{else}}\ e_1)$
2. $\Gamma \vdash e_0 : \tau_0 \ \wedge\ \rho \vdash e_0 \Downarrow v_1 \ \wedge\ \rho(x) = arg_1$
3. $\Gamma \vdash arg_2 : \tau_1 \ \wedge\ \tau_1 > \tau_0 \wedge \rho \vdash arg_2 \Downarrow\ v_2$

then $\rho[x \mapsto v_2] \vdash e_0 \Downarrow v_1$.

Proof. 1. If $\Gamma \vdash arg_1 : \textbf{num}^{\ell}$, then $\forall n\ e_2\ \ldots\ e_m\ ,\ e_0 = (\textbf{case}\ n\ \textbf{of}\ n_2 \Rightarrow e_2\ \ldots\ n_m \Rightarrow e_m\ \textbf{else}\ e_1)$, and either $\ell = \textbf{T}$ or $\ell = \textbf{U}$. We show that regardless of arg_2's level when it is of type **num**, it cannot be changed and therefore e_0's value doesn't change.

   (a) If $\exists \ell_1 \in \tau_0$ s.t. $\ell_1 = \textbf{T}$, then by typing rule CASE-LIT and the rule for join, n's integrity label is **T**. Therefore, arg_1 cannot both equal n and be arbitrarily changed to some expression arg_2 because it is not an expression whose type label is less trusted than the type of the entire expression (i.e. $\textbf{num}^{\textbf{T}} \not\geq \tau_0$). Thus we cannot replace arg_1 with arg_2, so in this case the value of e_0 remains

the same, as desired. Since e_1 through e_{m+1} are expressions whose soundness with respect to the type system can be considered separately through Lemmas 1, 2, and 3, we do not consider them here.

(b) If $\exists \ell_1 \in \tau_0$ s.t. $\ell_1 = \mathbf{U}$, then by our definition of the $\mathbf{T} - \mathbf{U}$ integrity lattice, there can be no values whose type is greater than τ_0 (arg_1 included) that we can change. Therefore, e_0 remains unchanged, satisfying our conclusion.

2. If $\Gamma \vdash arg_1 : (cn, \vec{\tau_1})$, then $\forall cn_3 \ \ldots \ cn_n \ \vec{x_3} \ \ldots \ \vec{x_n} \ e_3 \ \ldots \ e_n$,
$e_0 \ = \ (\textbf{case} \ (cn, \vec{\tau_1}) \ \textbf{of} \ cn_3 \ \vec{x_3} \Rightarrow e_3 \ \ldots \ cn_n \ \vec{x_n} \Rightarrow e_n \ \textbf{else} \ e_1)$.

- We know by the operational semantics (restricted to accommodate this type system, with singleton constructor types) that which branch we case on is determined entirely by the constructor that arg_1 evaluates to, and **not** the values contained within that constructor. Therefore, changing the expressions within any constructor will result in the same branch being taken, such that e_0 evaluates to the branch's right-hand-side expression. Therefore, we cannot choose to replace arg_1 with another arbitrary arg_2 when $\Gamma \vdash arg_1 : (cn, \vec{\tau_1})$.
- Let $(cn_3 \ \vec{x_3} \Rightarrow e_3)$ be the matching branch (where $cn = cn_3$). Based on the previous bullet point, we know that changing the expressions of any other branches will not change the value of the entire case expression, so we focus on this particular branch as an example. We must show that $\forall \tau_3, \exists x_3 \in \vec{x_3}$ s.t. $\Gamma \vdash x_3 : \tau_3 > \tau_0$, changing the value that x_3 maps to in ρ does not change the value that e_3 evaluates to; that is, $\rho[x_3 \mapsto v_3] : e_3 \Downarrow v$, where $\rho(arg_2) = v_3$. Since e_3 is an expression, its soundness is either covered by Lemma 1 (by induction) or Lemmas 2 or 3.

□

Lemma 2 (Result Expression Soundness). *If* $\forall x\ e_0\ arg_1\ arg_2\ \tau_1\ \tau_2\ v_1\ \rho\ \Gamma$, *where*

1. $e_0 = (\textbf{\textit{result}}\ arg_1) \wedge \rho \vdash e_0 \Downarrow v_1$
2. $\Gamma \vdash arg_1 : \tau_1 \wedge arg_2 = \rho(arg_1)$
3. $\rho \vdash arg_2 : \tau_2 \wedge \tau_2 > \tau_1 \wedge \rho \vdash arg_2 \Downarrow v_2$

then $\rho[(x \mapsto v_2)] \vdash e_0 \Downarrow v_1$

Proof. The **result** expression is used for wrapping a value into a single expression containing that value. Therefore, changing the value of arg_1 to arg_2 would change the resultant value v_1 that e_0 is given, contradicting our result. As another point, by the typing rule RESULT, **result**'s type is precisely the type of arg_1, meaning there are no values within e_0 to change that would not cause us to violate (3) above. Therefore, the value of arg_1 must equal the value of arg_2 such that value of e_0 cannot change. □

Lemma 3 (Let Expression Soundness). *If* $\forall e_0\ e_1\ x\ id\ arg_1\ arg_2\ \overrightarrow{arg_3}\ \overrightarrow{arg_4}$ $\tau_1\ \tau_2\ v_1\ v_2\ v_3\ \vec{v}_3\ \vec{v}_4\ v_5\ \rho\ \Gamma$, *where*

1. $e_0 = (\textbf{\textit{let}}\ x = id\ \overrightarrow{arg_3}\ \textbf{\textit{in}}\ e_1)$
2. $\Gamma \vdash e_0 : \tau_1 \ \wedge\ \rho \vdash e_0 \Downarrow v_1$
3. $\Gamma \vdash id : \vec{\tau} \rightarrow \tau$
4. $\rho(\overrightarrow{arg_3}) = \vec{v}_3$
5. $\Gamma(arg_1) = \tau_2 \ \wedge\ \tau_2 > \tau_1$
6. $arg_0 \in \overrightarrow{arg_3} \ \wedge\ \overrightarrow{arg_4} = \overrightarrow{arg_3} - arg_0 + arg_1$
7. $\rho(\overrightarrow{arg_4}) = \vec{v}_4$

8. $(id \in \overrightarrow{cons} \wedge \texttt{applyCn}(id, \vec{v}_3) = v_2) \vee (\texttt{applyFn}(id, \vec{v}_3, \rho) = v_2)$

9. $(id \in \overrightarrow{cons} \wedge \texttt{applyCn}(id, \vec{v}_4) = v_3) \vee (\texttt{applyFn}(id, \vec{v}_4, \rho) = v_3)$

10. $v_2 = v_3$

11. $\rho[x \mapsto v_3] \vdash e_1 \Downarrow v_5$

then $v_1 = v_5$.

Proof. By cases on *id*:

1. If *id* is a primitive function (add, multiply, etc.), then $v_2 \neq v_3 \iff \overrightarrow{arg}_3 \neq \overrightarrow{arg}_4$. By the typing rule of primitives, the type τ that the function returns is the least upper bound of all of its arguments, including arg_1, meaning by definition, both the value and type of the primitive operation are entirely dependent on all arguments. Therefore, there cannot exist an arg_2 that allows us to substitute it for arg_1 whose type is less trusted than τ without changing the entire value v_1.

2. If *id* is a constructor, then *id* has the type $\overrightarrow{\tau} \rightarrow (cn, \vec{\tau})$. *id*'s return type is determined statically and does not change throughout program execution. Therefore, there does not exist a subexpression in $\overrightarrow{arg}_3$, or more generally, in e_0, that can changed without changing the type of the constructor, which would contradict our having the same values after evaluation.

3. If *id* is a non-recursive function composed solely of case and result expressions and applications of primitive functions and constructors used in let expressions, then by (1), (2), Lemmas 1 and 2 and induction on Lemma 3, we know *id* must be sound. By extension, if *id* calls a function that fulfills these requirements, one can unfold the called function's contents in order to see that the resultant value v_2 satisfies this case.

4. If *id* is a recursive function or calls a function which calls *id* (i.e. mutual recursion), it is possible that the function call never terminates and therefore never results in a single value. The soundness of e_0 must then be guaranteed via induction on possible expressions, proven in the previous lemmas. We know statically that the type of *id* is of the form $\vec{\tau} \rightarrow \tau$, so we are guaranteed via simplification rules in the `apply` helper functions that types of $\overrightarrow{arg}_3$ must be equal to or subtypes of $\vec{\tau}$, or otherwise our operational semantics would get stuck. By induction, any recursive calls made in e_1 must also satisfy this lemma, meaning that the actual arguments $\overrightarrow{arg}_3$ are used properly, otherwise e_0 wouldn't type check to type τ_1 by getting stuck.

By proving that v_2's value does not change when less-trusted values change, we can safely continue with the evaluation of e_1, which will be a **case**, **result**, or **let**, all of which are handled in Lemma 1, Lemma 2, and Lemma 3, respectively. □

Theorem 1 (Integrity Type System Soundness)**.** *Our integrity type system is sound if, given some expression* $\boldsymbol{e}$ *of type* τ *which evaluates to some value* $\boldsymbol{v}$*, we can show that we can arbitrarily change any (or all) expressions in* $\boldsymbol{e}$ *which are less trusted than* τ *so that* $\boldsymbol{e}$ *still evaluates to* $\boldsymbol{v}$*; i.e., untrusted data does not affect trusted data.*

Formally, if $\forall e_1\ e_2\ e_3\ e_4\ \tau_1\ \tau_2\ \tau_3\ v\ \rho\ \Gamma$*, where*

1. $\rho \vdash e_1 \Downarrow v\ \wedge\ \Gamma \vdash e_1 : \tau_1$
2. $e_2 \in \texttt{subexprs}(e_1)\ \wedge\ \Gamma \vdash e_2 : \tau_2\ \wedge\ \tau_2 > \tau_1$
3. $\Gamma \vdash e_3 : \tau_3\ \wedge\ \tau_3 \geq \tau_2$

then $\rho \vdash e_1[e_3/e_2] \Downarrow v$

Proof. There are just three types of expressions: **let**, **case**, and **result**. By Lemma 1, we show that **case** expressions (the vehicle for control-flow) are sound. By Lemma 2, we show that **result** expressions are sound. Likewise, by Lemma 3, we show that **let** expressions (the vehicle for function application) are sound. Thus, we have exhaustively shown soundness of all expressions. Furthermore, we can see that when these expressions are composed according to the abstract syntax, with the additional typing annotations and a few restrictions, any well-typed Zarf program has the property of non-interference with respect to integrity, even while using a simplistic type system such as that explained here. □

2.6.4 Programmer Responsibility

We have demonstrated that there are varying degrees of responsibility a Zarf programmer can take when writing their application, each involving greater effort. The first is doing the minimum: the programmer writes their program in Zarf assembly. A major advantage of Zarf is that the application automatically gains the benefits of memory and control-flow safety inherent in the ISA, properties that other ISAs don't easily offer. Any well-formed application that runs on Zarf gets these properties without any additional programmer involvement.

The second degree of responsibility that can be taken is writing the application's specification in Coq and automatically lowering it to Zarf to prove its correctness. This approach involves a non-trivial amount of proof-writing, but since the ISA resembles the language of verification very closely, we argue that the amount of work involved relative to doing so over other imperative ISAs is significantly less. Since high-level specification and verification of critical applications is common practice, this level of programmer responsibility is not usual. Gladly, however, any future proof efforts might

not need to be entirely application-specific. Given the exercise of proving the correctness of the Zarf implementation of the ICD algorithm, we now have a set of theorems and proofs showing the equivalence between common user-made Zarf functions and Coq versions. It is conceivable that verification in Coq of future Zarf applications could reuse this underlying work.

The third degree of responsibility involves proving additional properties over the system, beyond the aforementioned safety and correctness. We demonstrated this by laying a security type system over the ISA, somewhat restricting it (like all type systems are wont to do) in exchange for the added property of non-interference. Because the process of writing a type system and checker are sufficiently general, we can see additional type systems or analyses being made over the base Zarf ISA relatively easily.

Finally, there is the issue of determining which parts of an application should go into each hardware execution realm. Zarf has two execution realms due in part to the assumption that users might want to include legacy or high performance, non-critical code; this code can run on the imperative ISA. However, any activity providing critical functionality for safe operation should happen in the functional processor. Veridrone [76] is an example of another project beyond our ICD that might benefit from this approach; that project uses both a lower-performance core safety control system and a higher-performance unverified version that is more energy-efficient and allows for smoother flying.

2.7 Evaluation

To validate our designs, we download the Zarf hardware specification onto a Xilinx Artix-7 FGPA and run our sample application. For a comparison, we also run a completely unverified C version of the application on a Xilinx Microblaze on the same

Resource	Zarf	MicroBlaze
LUTs	4,337	1,840
FFs	2,779	1,556
Cycle Time	20ns (50 MHz)	10ns (100 MHz)

Table 2.1: Resource usage of Zarf and basic MicroBlaze (3-stage pipeline), the two layers of Zarf, when synthesized for a Xilinx Artix-7 FPGA. In total, the logic of Zarf uses 29,980 gates.

FPGA. Hardware synthesis results are summarized in Table 2.1.

The hardware description of Zarf is more complex than a simple embedded CPU, with 66 total states of control logic (4 deal with program loading, 15 with function application, 18 with function evaluation, and 29 with garbage collection). In all, the combinational logic takes 29,980 primitive gates (roughly the size of a MIPS R3000), or 4,337 LUTs when synthesized for an Artix-7 FPGA (less than 7% of the available logic resources). Estimated on 130nm, the combinational logic takes up .274 mm^2. Though larger than very simple embedded CPUs, Zarf is still quite a bit smaller than many common embedded microcontrollers.

From a dynamic trace of several million cycles, the ICD application exhibited the following average CPI for each instruction type. `Let` instructions had an average of 5.16 arguments and took on average 10.36 cycles. `Case` instructions averaged at 10.59 cycles; each branch head in a `case` takes exactly 1 cycle to check if the branch matches. `Results` took 11.01 cycles on average. The total dynamic CPI across the trace was 7.46 (or 11.86 if garbage collection time is included). Approximately one third of the dynamic instructions were branch heads.

The C version of the ICD application on the MicroBlaze takes fewer than one thousand cycles for each iteration of the application. The analysis in section 2.6.2 discusses the worst-case runtime of the Zarf application, which is around 9,000 cycles or 180 μs (though much faster in the typical case). This is in addition to a longer cycle time

(see Table 2.1). When compared to the carefully optimized and tiny MicroBlaze, our experimental prototype uses approximately twice the hardware resources, and the application is around 20x slower in the worst case than Microblaze in the common case — but is still over 25 times faster than it needs to be to meet the critical real-time deadlines, all while adding invaluable guarantees about the correctness of the most critical application components and assurance of non-interference between separate functions.

2.8 Conclusion

As computing continues to automate and improve the control of life-critical systems, new techniques which ease the development of formally trustworthy systems are sorely needed. The system approach demonstrated in this work shows that deep and *composable* reasoning directly on machine instructions is possible when the architecture is amenable to such reasoning. Our prototype implementation of this concept uses Zarf to control the operation of critical components in a way that allows assembly-level verified versions of critical code to operate safely in close partnership with more traditional and less-verified system components without the need to include run-times and compilers in the TCB. We take a holistic approach to the evaluation of this idea, not only demonstrating its practicality through an FPGA-implemented prototype, but furthermore showing the successful application of three different forms of static analysis at the assembly level of Zarf.

As we move to increasingly diverse systems on chip, heterogeneity in semantic complexity is an interesting new dimension to consider. A very small core supporting highly critical workloads might help ameliorate critical bugs, vulnerabilities, and/or excessive high-assurance costs. A core executing the Zarf ISA would take up roughly

0.002% of a modern SoC. Our hope is that this work will begin a broader discussion about the role of formal methods in computer architecture design and how it might be embraced as a part, rather than an afterthought, of the design process.

Chapter 3

Bouncer: Static Program Analysis in Hardware

3.1 Introduction

The demand for more connectivity and richer interactions in everyday objects means that everything from light bulbs to thermostats now contains general-purpose microprocessors for carrying out fairly straightforward and low-performance tasks. Left unanalyzed, these systems and their associated software stacks can be expected to hold a seemingly endless collection of opportunities for attack. Static analysis provides powerful tools to those wishing to understand or limit the set of behaviors some software might exhibit. By facilitating sound reasoning over the set of all possible executions, this type of analysis can identify important classes of behavior and prevent them from ever happening. If embedded system developers simply *never* released software that failed, such that those well-analyzed applications were the *only* things to ever execute on platforms under our control, many of the bugs and vulnerabilities that plague our life would be eliminated. Unfortunately, realizing this in practice has proven incredibly

hard due to pressure to market, pressure to reduce cost, and the delayed and stochastic cost associated with vulnerabilities and bugs.

While larger software companies might be more trusted to rigorously verify their software releases, the embedded systems market has a long and heavy tail of providers with a much wider distribution of expertise and resources at their disposal. When we bring an embedded device into our home or business, how can we have confidence that the software running there (which depends on chains of control well outside our ability to observe) is "above the bar" for us? Seemingly innocuous issues, for example passing a string instead of an integer, can open the door for an attacker to gain root privileges and serve as a base for other attacks (exactly this happened already in a class of WiFi routers [77]). Similar attacks targeting embedded devices and firmware updates have succeeded on everything from printers [78] to thermostats [79].

The basic research question we ask in this paper is: is it possible to make forms of static analysis an *intrinsic* part of executing on a microprocessor? In other words, we examine a machine that will *guarantee at the hardware level* that any and all code executing on it is bound to the constraints imposed by a given static program analysis. This moves the decision to do a proper analysis away from those that push software updates (who may be making decisions about updates many years removed from the original purchase) to the decision to purchase and deploy a particular hardware device itself.

Such a machine would reject any attempt to load it with code that fails to meet the specified "bar," independent of who wrote it, who signed it, how it was managed, or where the software came from. The trust one could put in aspects of execution on such a processor could be independent of measurement, attestation, or other active third-party evaluation. By doing the checks in hardware, we can make them intrinsic to the device's functionality: the checks will be fully live right from power-up; the checks

will require no dependency on other software on the system functioning correctly (zero TCB); and if properly designed, they will be directly wired into the operation of the system, making them provably impossible to bypass.

As this is the first approach to propose and evaluate fully-hardware implemented static analysis there are two big open questions: a) is it even possible to do a useful static analysis in hardware, and b) what would the costs of such an analysis be in terms of time or area? We answer these questions through the hardware development of a new module, the Binary Exclusion Unit (which we call "the bouncer" more informally), capable of scanning and rejecting program binaries right as they are streamed onto the device. Specifically, we make the following contributions:

- We introduce hardware static binary analysis and show that it can be implemented in a way that can never be circumvented through some clever manipulation of software (e.g. a compromised set of keys, a bug in the operating system, or a change in the boot ordering).

- We describe a method of static analysis co-design where the checking algorithm is modified to be more amenable to hardware implementation while maintaining correctness and efficiency.

- We demonstrate that the analysis, in conjunction with the functional ISA, ensures all executions are free of memory and type errors and have guaranteed control flow integrity.

- We evaluate the functioning of the system with a complete RTL implementation (synthesizable Verilog) of the checker and processor interoperating with gate-level simulation.

- Finally, we show that the resulting system is efficient both in terms of hardware

resources required and performance, and describe how program transformations can make it even more so.

We elaborate on the motive of our work (Section 3.2), present our hardware static analysis in the form of a new hardware/software co-designed type system and prove its soundness (Section 3.3), outline the checking algorithm implementing the type system (Section 3.4), and design type annotations that can be easily encoded into the machine binary and provide a hardware implementation of the typechecker (Section 3.5). We prove the non-bypassability of the circuit in Section 3.6, something that would be extremely difficult to achieve for a software solution. Next, we provide hardware synthesis figures, evaluate update-time overhead, and show how to manage worst-case examples (Section 3.7). Finally, we discuss related work (Section 3.8) and conclude.

3.2 Hardware Static Analysis

In building a static analysis hardware engine directly into an embedded microcontroller, one of the big advantages of customization is that at the hardware level we can see, *either physically through inspection or through analysis at the gate or RTL level*, exactly how information is flowing through a system to introduce safety or security mechanisms that are truly non-bypassable. No software can change the functioning of the system at that level. However, doing static analysis at the level of machine code is no easy task — even for software.

Fortunately, there are some great works to draw inspiration from. Previous work has used types to aid in assembly-level analysis; specifically TAL [80] and TALx86 [81] have created systems where source properties are preserved and represented in an idealized assembly language (the former) or directly on a subset of x86 (the latter). Working up the stack from assembly, other prior works attempt to prove properties

and guarantee software safety at even higher levels of abstractions. We seek to take these software ideas and find a way to make them *intrinsic* properties of the physical hardware for embedded systems where needed.

In this work we draw upon the opportunity afforded by architectures that have already been designed with ease of analysis in mind. Specifically, we leverage the Zarf ISA, a purely functional, immutable, high-level ISA and hardware platform used for binary reasoning, which is suitable for execution of the most critical portions of a system [1]. At a high level, the Zarf ISA consists of three instructions: `Let` performs function application and object allocation, applying arguments to a function and creating an object that represents the result of the call. `Case` is used for both pattern-matching and control flow. One cases on a variable, then gives a series of patterns as branch heads; only the branch with the matching pattern is executed. Patterns can be constructors (datatypes) or integer values, depending on what was cased on. `Result` is the return instruction; it indicates what value is returned at the end of a function. Branches in case statements are non-reconvergent, so each must end in a result instruction.

A big advantage of this ISA for static analysis is that it has a compact and precise semantics. If we could could guarantee the physical machine would always execute only according to these semantics (e.g. always respecting call/return behavior, using the proper number of arguments from the stack, etc.) we would end up with a system that has some very desirable properties. In Section 3.7 we show that these include verifiable control flow integrity, type safety, memory safety, and others; e.g., ROP [82] is impossible, programs never encounter type errors, and buffer overruns can never happen.

Unfortunately, the semantics of any language govern the behavior of execution only for "well-formed" programs. When we are talking about machine code, as opposed to programming languages, things are a little trickier, because machines are expected to

read instruction bits from memory and execute them faithfully as they arrive. As we describe in more detail below, checking membership in the language of well-formed Zarf programs is actually something that requires some sophistication and would be difficult to do at run-time. Even though there are just three instructions, Zarf binaries support casing, constructors, datatypes, functions, and other higher-level concepts as first-class citizens in the architecture. Our goal is to correctly implement these checks statically and show that the only binaries that can *ever* execute on this machine pass this static analysis.

3.2.1 The Analysis Implemented

While one could, in theory, capture every possible deviation from the Zarf semantics with a set of run-time checks in hardware, actually catching every possible thing that can go wrong quickly grows in complexity. An advantage of static checking over dynamic checks is that once the binaries are analyzed, no additional energy and time costs are required during execution. For an embedded system that runs the same code continuously, any small static cost is amortized rather quickly. As we will show later, in fact the static analysis can actually be done in a *single streaming pass* over the executable. However, just to see the scope of the problem it is useful to enumerate some of the dynamic checks that would be required to achieve the same objective as our hardware static analysis.

Table 3.1 lists ways that programs can fail and costs that are incurred if one were to dynamically check for errors on the platform. There are 21 different ways for the hardware to throw errors, the great majority of which require keeping some significant bookkeeping to actually check. At the very least, we would need to keep extra information on number of arguments, number of local variables, number of recently cased

Possible failure:	**Meaning:**
malformed instruction	Bit sequence does not correspond to a valid instruction.
fetch out-of-bounds arg	Accessing argument *N* when there are fewer than *N* arguments.
fetch out-of-bounds local	Accessing local *N* when there are fewer than *N* locals allocated.
fetch out-of-bounds field	Accessing field *N* when there are fewer than *N* fields in the case'd constructor.
fetch invalid source	Bit sequence does not correspond to a valid source.
apply arguments to literal	Treating a literal value as a function and passing arguments to it.
apply arguments to constructor	Treating a saturated constructor as a function and passing arguments to it.
application with too many args	Passing more arguments than a function can handle, even if it returns other functions.
application on invalid source	Invalid source designation for function in application.
oversaturated error closure	Passing arguments to an error closure.
oversaturated primitive	Passing more arguments than a primitive operation can handle.
passing non-literal into primitive op	Passing an object (constructor or closure) into a primitive operation.
case on undersaturated closure	Trying to branch on the result of a function that cannot be evaluated.
unused arguments on stack	Oversaturating a function and branching on the result when not all arguments have been consumed.
matching a literal instead of a pattern	Branching on a function that returns a constructor, but trying to match an integer.
invalid skip on literal match	Instruction says to skip *N* words on failed match, but that location is not a branch head.
no else branch on literal match	Incomplete case statement because of lack of else branch.
matching a pattern instead of a literal	Branching on a function that returns an integer, but trying to match a constructor.
incomplete constructor set in case statement	Incomplete case statement because not all possible constructors are present.
invalid skip on pattern match	Instruction says to skip *N* words on a failed match, but that location is not a branch head.
no else branch on pattern match	Incomplete case statement because of lack of else branch.

Table 3.1: Summary of 21 conditions that require dynamic checks in the absence of static type checking. With our approach, checking is achieved ahead of time, in a single pass through the program; energy and time are not wasted with repeated error checking. No information needs to be tracked at runtime, and the only runtime hardware check is for out-of-memory errors. All of the listed errors are guaranteed by our type system to not occur.

constructor fields, and runtime tags on heap objects to distinguish between closures and constructors — all of which the hardware would need to track at runtime. Crucially, this information must be incorruptible and inaccessible to the software for the dynamic checks to be sound. If software is able to access and corrupt this information, it compromises the integrity of the dynamic checks. In general, guaranteeing that the set of dynamic checks are always occurring, i.e. not bypassed, can be very difficult. With a hardware-implemented static analysis, we are able to formally prove that our checks cannot be bypassed (outlined in Section 3.6). In addition to the hardware implementation overhead of these checks, reasoning about software behavior in the face of dynamic checks becomes more difficult as well if error states are returned. Programmers that wish to handle errors due to code that fails such checks are forced to reason about every situation that can arise (e.g. what if this function encounters an oversaturated primitive, or cases on an undersaturated closure, and so on.). Instead, by performing the checks statically, all software components understand that any other component with which they might interact on the system is subject to the same analysis as their own code.

3.2.2 The Bouncer Architecture

Given that we can develop a unit to actually perform the desired static analysis, a big question is where it fits into the actual micro-controller design. Figure 3.1 shows how a static analysis engine (the Binary Exclusion Unit) fits into an embedded system at a high level: all incoming programs are vetted by the checker before being written to program storage, ensuring that all code that the core executes conforms to the type system's proven high-level guarantees. During programming mode, as a binary image is loaded into the core, the checker has write access to the program store and can use data memory as a working space. The Binary Exclusion Unit can thus be used as a

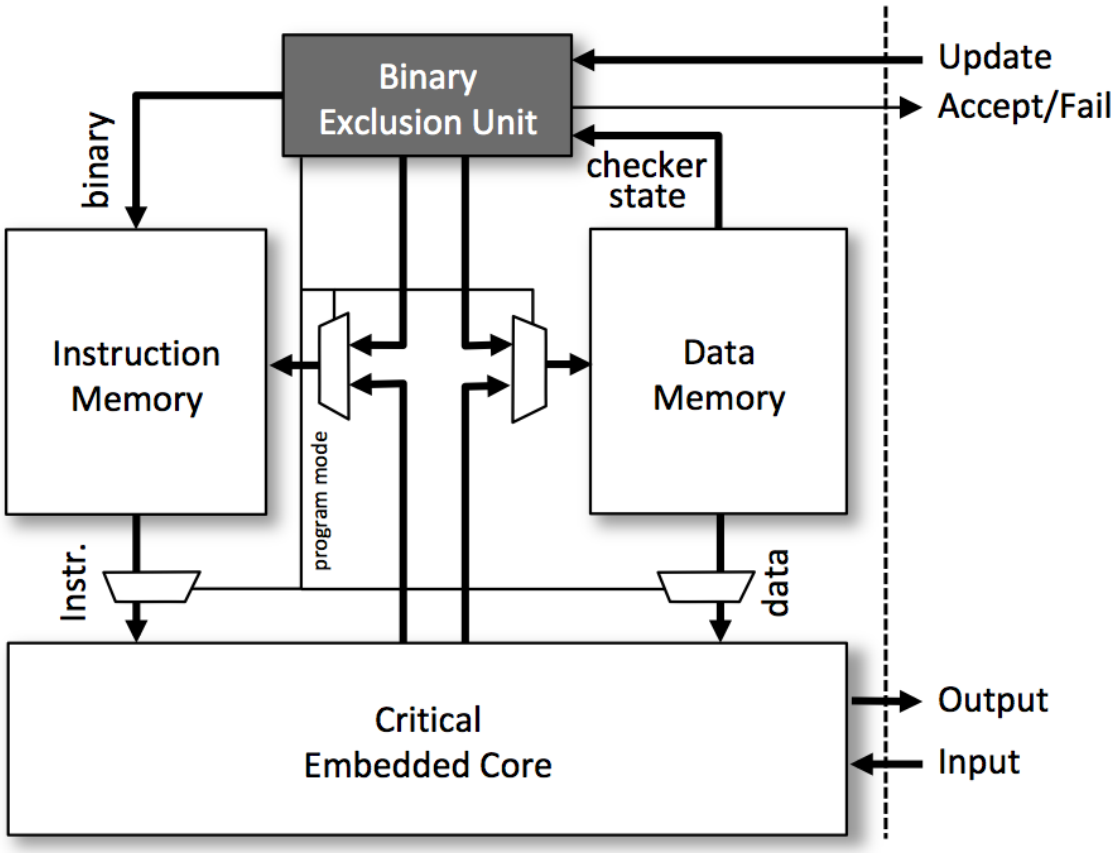


Figure 3.1: The Binary Exclusion Unit works as a gatekeeper, only allowing input binary programs if they pass the static analysis. When in "Programming" mode, the core is halted while the program is fed to the checker; if it passes, it is written to the system instruction memory. The checker makes use of the core's data memory, which is otherwise unused during system programming. At run-time, the checker is disabled and consumes no resources. Programs that pass static analysis are guaranteed to be free of memory errors, type errors, and control flow vulnerabilities. The checker is non-bypassable; all input binaries are subject to the inspection.

runtime guard, checking programs right before execution when they are loaded into memory, or as a program-time guard, checking programs when they are placed into program storage (flash, NVM, etc.).

Only once the programming mode is complete do the instruction and data memory become visible. The upshot of catching errors this way is that this gives you feedback at programming time, before a device is deployed, that the binary contains errors. It further ensures that when reprogramming occurs in the field, malicious or malformed code that exploits interactions outside of the ISA semantics will never be loaded.

In either case, checking works the same way: each word of the binary is examined one at a time in a linear pass over the program as it is fed through the Binary Exclusion Unit. It is trivial to verify that the BEU is the only unit given access to write to the memory — the more interesting discussion, covered later, is the verification that the only way though the BEU is via a static analysis.

3.3 Static Analysis Strategy

While many different static analysis approaches might be implemented in hardware in the way we described in the sections above, to embody these ideas in a hardware prototype we need a specific analysis specification and implementation. Here we draw inspiration from TAL [80], and use types to clearly and completely specify allowed behavior. By extending the Zarf ISA with types, passing a portion of that type information along with the binary, and then performing the static analysis to check those types, we know the program conforms to the allowed behaviors. This new type-extended Zarf ISA is, unlike untyped Zarf, based on the polymorphic lambda calculus. Figure 3.2 describes the abstract syntax of the typed ISA; note that there are four types: integers, functions, type variables, and datatypes (which are similar to algebraic datatypes found

$$x \in \mathit{Variable} \quad n \in \mathbb{Z} \quad \mathit{fn}, \mathit{tn}, \mathit{cn} \in \mathit{Name} \quad \oplus \in \mathit{PrimOp}$$
$$\alpha \in \mathit{GenericTypeVariable} \quad \beta \in \mathit{RigidTypeVariable}$$

$$\begin{aligned}
P \in \mathit{Program} &::= \overrightarrow{\mathit{data}}\ \overrightarrow{\mathit{func}} \\
\mathit{data} \in \mathit{Datatype} &::= \textbf{data}\ \mathit{tn}\ \vec{\alpha}\ \textbf{=}\ \overrightarrow{\mathit{cons}} \\
\mathit{cons} \in \mathit{Constructor} &::= \textbf{con}\ \mathit{cn}\ \vec{\tau} \\
\mathit{func} \in \mathit{Function} &::= \textbf{fun}\ \mathit{fn}\ \overrightarrow{x : \tau}\ \tau\ \textbf{=}\ e \\
e \in \mathit{Expression} &::= \mathit{let} \mid \mathit{case} \mid \mathit{res} \\
\mathit{let} \in \mathit{Let} &::= \textbf{let}\ x\ \textbf{=}\ n\ \textbf{in}\ e \mid \textbf{let}\ x\ \textbf{=}\ \mathit{id}\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e \\
\mathit{case} \in \mathit{Case} &::= \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br} \mid \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e \\
\mathit{res} \in \mathit{Result} &::= \textbf{result}\ \mathit{arg} \\
\mathit{br} \in \mathit{Branch} &::= \mathit{cn}\ \vec{x} \Rightarrow e \mid n \Rightarrow e \\
\mathit{id} \in \mathit{Identifier} &::= \mathit{fn} \mid \mathit{cn} \mid \oplus \mid x \\
\mathit{arg} \in \mathit{Argument} &::= n \mid x \\
\tau \in \mathit{Type} &::= \textbf{Int} \mid \mathit{dt} \mid \mathit{ft} \mid T \\
\mathit{dt} \in \mathit{Datatype} &::= \mathit{tn}\ \vec{\tau} \\
\mathit{ft} \in \mathit{FuncType} &::= \vec{\tau} \to \tau \\
T \in \mathit{TypeVar} &::= \alpha \mid \beta
\end{aligned}$$

Figure 3.2: Typed Zarf abstract syntax. An arrow over any metavariable signifies a list of zero or more elements, except for a datatype's constructor list, which must be non-empty.

$$\Gamma \in Env = Variable \to Type \quad C \in ConstraintSet = \mathcal{P}(Type \times Type)$$
$$\sigma \in Substitution = TypeVar \to Type \quad b \in Bool = \textbf{true} + \textbf{false}$$

Figure 3.3: Semantic domains for the Zarf static semantics. See Figure 3.4 for the typing rules.

in languages like Haskell and ML). Both functions and datatypes are declared at the top level; since the ISA is lambda-lifted, the introduction of universally-quantified type variables ranging over a function body or datatype is limited to the top level as well, simplifying the ISA's type system.

Our static analysis requires that type information be encoded into the binary, but we note specifically that the Binary Exclusion Unit discards these annotations when finished, leaving a (safe and certified) standard binary program in protected core memory. To qualify as a typed Zarf program, a binary must declare types of all top-level functions and make all (data) constructors members of a datatype. With this, all types will be tracked and checked, including type variables for polymorphism, facilitating local type inference within the bodies of functions.

3.3.1 Static Semantic Rules

The type system in Figure 3.4 describes, using a set of inference rules, what it means for a Zarf binary to be well-typed; the associated static semantic domains are found in Figure 3.3. In this section we define what each inference rule means formally.

FUNC-RET

Rule FUNC-RET checks functions that have zero parameters. It does so by determining the type τ_{r2} of the function body via the inference rule over e, comparing it against the expected return type τ_{r2} via the call to `princType`. If the result of that function call

Functions $\boxed{\vdash func : \tau}$

$$\frac{\tau_{r1} = \texttt{makeRigid}(\tau_r) \quad \vdash e : \tau_{r2} \quad \texttt{princType}([\tau_{r1}, \tau_{r2}]) = \tau_{r1}}{\vdash \textbf{fun}\ fn\ []\ \tau_r\ \textbf{=}\ e : \tau_r}\ (\textsc{func-ret}) \qquad \frac{(\vec{\tau}_{p1} \rightarrow \tau_{r1}) = \texttt{makeRigid}(\vec{\tau}_p \rightarrow \tau_r) \quad \vec{x} \mapsto \vec{\tau}_{p1} \vdash e : \tau_{r2} \quad \texttt{princType}([\tau_{r1}, \tau_{r2}]) = \tau_{r1}}{\vdash \textbf{fun}\ fn\ \overrightarrow{x : \tau_p}\ \tau_r\ \textbf{=}\ e : \vec{\tau}_p \rightarrow \tau_r}\ (\textsc{func-params})$$

Expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\texttt{idTy}(\Gamma, id) = \tau_i \qquad \alpha = \texttt{freshGenTV} \qquad \Gamma_1 = \Gamma[x_1 \mapsto \alpha] \quad \texttt{map}(\texttt{argTy}(\Gamma_1), \overrightarrow{arg}) = \vec{\tau}_a \quad \texttt{applyType}(\tau_i, \vec{\tau}_a, [], \alpha) = \tau_1 \quad \Gamma[x_1 \mapsto \tau_1] \vdash e : \tau}{\Gamma \vdash \textbf{let}\ x_1\ \textbf{=}\ id\ \overrightarrow{arg}\ \textbf{in}\ e : \tau}\ (\textsc{let-var})$$

$$\frac{\Gamma[x \mapsto \textbf{Int}] \vdash e : \tau}{\Gamma \vdash \textbf{let}\ x\ \textbf{=}\ n\ \textbf{in}\ e : \tau}\ (\textsc{let-int}) \qquad \frac{\texttt{argTy}(\Gamma, arg) = \tau}{\Gamma \vdash \textbf{result}\ arg : \tau}\ (\textsc{result})$$

$$\frac{\Gamma(x) = dt \quad \overrightarrow{cons} = \texttt{getCons}(dt) \quad \texttt{allConsPres}(\overrightarrow{cons}, \overrightarrow{br}) = \textbf{true} \quad \vec{\tau} = \texttt{brTypes}(\Gamma, \overrightarrow{br}, \overrightarrow{cons}) \quad \texttt{princType}(\vec{\tau}) = \tau}{\Gamma \vdash \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br} : \tau}\ (\textsc{case-con})$$

$$\frac{\Gamma(x) = dt \quad \overrightarrow{cons} = \texttt{getCons}(dt) \quad \vec{\tau} = \texttt{brTypes}(\Gamma, \overrightarrow{br}, \overrightarrow{cons}) \quad \Gamma \vdash e : \tau_e \quad \texttt{princType}(\tau_e :: \vec{\tau}) = \tau}{\Gamma \vdash \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e : \tau}\ (\textsc{case-con-else})$$

$$\frac{\Gamma(x) = \textbf{Int} \quad \overrightarrow{(n_i \Rightarrow e_i)} \in \overrightarrow{br} \quad \Gamma \vdash e_i : \tau_i \quad \Gamma \vdash e : \tau_e \quad \texttt{princType}(\tau_e :: \vec{\tau}_i) = \tau}{\Gamma \vdash \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e : \tau}\ (\textsc{case-int})$$

Figure 3.4: Zarf static semantics (typing rules). See Figure 3.2 for the abstract syntax and Figure 3.3 for the static semantic domains. `map`, `filter`, and `concatMap` refer to their standard definitions. `delete` removes the first instance of an element from a list. We use the notation $\Gamma[x \mapsto \tau]$ to represent the creation of an updated map Γ where the new entry $x \mapsto \tau$ has been added to the old map; similarly, $\Gamma[\vec{x} \mapsto \vec{\tau}]$ denotes mapping the first variable in $\vec{x}$ to the first type in $\vec{\tau}$, etc. for all point-wise pairs. Unless otherwise stated, the lengths of both lists must be the same. We use the notation $O(\cdot)$ to indicate an "Option" type: essentially a set guaranteed to contain zero or one values. We use the symbol $\bullet$ to represent the absence of a value where an "Option" type is required. We differentiate between metavariables with subscripts that can be a combination of letters or numbers; for example, τ and τ_1 represent distinct metavariables denoting types. See Sections 3.3.1 and 3.3.2 for descriptions of each rule and helper function, respectively.

doesn't equal the expected type, the inference rule fails, indicating a type error.

FUNC-PARAMS

Rule FUNC-PARAMS checks functions that have one or more parameters. It does so by mapping each function's parameters to its declared types before checking the body. `makeRigid` universally quantifies all type variables in the type declaration across the body. Like rule FUNC-RET, it uses `princType` to check that the function body's expected type matches its inferred type, with failure indicating a type error.

LET-VAR

Rule LET-VAR applies a type to zero or more arguments using the helper `applyType` to get the principal type of the application. Functions may be partially-applied, and mapping the bound variable to a fresh type variable allows for recursive definitions. It determines the type of the next subexpression expression e in an updated environment $\Gamma[x_1 \mapsto \tau_1]$.

LET-INT

Rule LET-INT performs constant assignment, simply updating the environment with a binding from identifier x to the type **Int**.

CASE-CON

Rule CASE-CON is used when scrutinizing a datatype. It gets the list of constructors associated with a particular datatype, replacing all type variables in its constructors' fields with any type variable instantiations found in the datatype. It uses the helper `brTypes` to get the type of each branch.

CASE-CON-ELSE

Rule CASE-CON-ELSE is similar to CASE-CON, but used when all constructors of the datatype aren't present. In this case, an else branch is required so that the entire expression can always evaluate to a value with a type τ.

CASE-INT

Rule CASE-INT is used when scrutinizing an integer. It compares branch body types for equality, and like rule CASE-CON-ELSE, an else branch is required, as it is not feasible to have branches for every possible integer value.

RESULT

Rule RESULT is the base case, simply producing the type of a bound variable or integer. As **result** is the only expression without a subexpression, it is guaranteed to be the last expression of all branches of a function.

3.3.2 Static Semantic Helper Functions

The rules used in type system in Figures 3.4 and described in the previous section use a variety of auxiliary functions for clarity in defining the semantics. In this section we define what each helper function does formally.

`applyType`

Performs constraint generation, unification (`unify`), and substitution (`substitute`) to get the principal type of an application. When no arguments are applied, the type of this helper's first parameter is returned, thus allowing the `Let` instruction to apply an integer, datatype, or generic type as well as function types. Note that because the type

returned by `applyType` is the principal type of the variable to which it is bound in the `Let` instruction, no constraints are propagated to any instructions that follow, limiting the amount of information that needs to be tracked throughout typechecking, as well as making error reporting of ill-typed applications more accurate.

`allConsPres`

`allConsPres` checks if all the constructors have a matching branch. Note that it is *not* ill-typed for the pattern branch to provide more binders than the constructor has fields; it's only ill-typed if those extraneous binders are *used* in the branch body. In the helper `brTypes`, when the added binders are mapped to their types, it is implied that the mapping is guaranteed to be no larger than the number of types in the corresponding constructor. If an extraneous binder is used in the branch body, it will not appear in the environment because it wasn't added, and the type system will determine the branch body to be ill-typed, as expected. This relaxation is needed here because the machine replaces all the binder references with a field index; if the field is never used in the body of the branch, then it would not appear in the binary, and therefore we cannot try to over-constrain it here because the binary would not produce an error during execution.

Also note that there is no requirement that $\overrightarrow{br}$ contain exactly the same number of pattern branches as the number of constructors; it is okay for there to be more branches that constructors for the given datatype. In the helper function `brTypes`, we ensure that each pattern branch matches a constructor in the cased-on datatype; therefore, the implication is that having duplicate pattern branches is acceptable, as long as they match a constructor of the scrutinized datatype.

$$\texttt{allConsPres} \in \overrightarrow{\mathit{Constructor}} \times \overrightarrow{\mathit{Branch}} \to \mathit{Bool}$$

$$\texttt{allConsPres}([], \_) = \textbf{true}$$

$$\texttt{allConsPres}((\textbf{con}\ cn\ \vec{\tau}) :: \overrightarrow{cons}, \overrightarrow{br}) = (cn\ \_ \Rightarrow \_) \in \overrightarrow{br} \wedge \texttt{allConsPres}(\overrightarrow{cons}, \overrightarrow{br})$$

applyHelper

applyHelper generates constraints between a function application's parameters and arguments, taking care to handle over-application appropriately. It does so by recursively iterating through a list of parameter and argument types, generating the constraint that the current parameter must equal the current argument. As function types are uncurried in this formalism, it is used to determine when to continue applying extra arguments to the current function's return type, which must also be a function type.

$$\texttt{applyHelper} \in \mathit{FuncType} \times \overrightarrow{\mathit{Type}} \times \mathit{ConstraintSet} \times \mathit{TypeVar} \to \mathit{Type}$$

$$\texttt{applyHelper}(\tau_p :: \vec{\tau}_p \to \tau_r, \tau_a :: \vec{\tau}_a, C_1, \alpha) = \texttt{applyType}(\tau_f, \vec{\tau}_a, C_2, \alpha)$$

where

$$C_2 = \{\tau_p = \tau_a\} \cup C_1$$

$$\tau_f = \begin{cases} \tau_r & \text{if } \vec{\tau}_p = [] \\ \vec{\tau}_p \to \tau_r & \text{otherwise} \end{cases}$$

applyType

applyType describes application of a type to a (possibly empty) list of argument types, performing constraint generation, unification (unify), and substitution (substitute) to get the principle type of an application. When applying a function type to a non-empty list of argument types, it calls applyHelper, which performs the step of constraint

generation so that this function can verify that the application is valid via unification. This and the associated helper functions are needed because of the uncurried presentation of functions in the abstract syntax, which reflects their representation in the machine more closely. Note that it is considered an error to try to apply a non-function type to a non-empty list of argument types. When no arguments are applied, the type of this helper's first parameter is returned, thus allowing the `Let` instruction to apply an integer, datatype, or generic type as well as function types.

The argument α is used to denote the type variable that the variable in the current let binding was bound to before calling this function. This is necessary to handle the case where an argument's type is unknown when application begins because it is being recursively defined. For example, in `let ones = Cons 1 ones in ...`, `ones` is being recursively defined and therefore will not have a type fully determined until after application; upon a call to `applyHelper`, the parameter `ones` will have type α (where α is fresh) in the environment. After the process of constraint generation and unification is completed during this application, if α is found in the substitution, that means it was used as an argument during application and we can then check that its use corresponds correctly with the result of the entire application.

$$\texttt{applyType} \in \mathit{Type} \times \overrightarrow{\mathit{Type}} \times \mathit{ConstraintSet} \times \mathit{TypeVar} \to \mathit{Type}$$

$$\texttt{applyType}(\tau_1, \vec{\tau}_a, C, \alpha) = \begin{cases} \tau_2 & \text{if } \vec{\tau}_a = [] \\ \texttt{applyHelper}(\vec{\tau}_p \to \tau_r, \vec{\tau}_a, C, \alpha) & \text{if } \vec{\tau}_a \neq [] \wedge \tau_1 = \vec{\tau}_p \to \tau_r \end{cases}$$

where

$$\sigma = \texttt{unify}(C)$$

$$\tau_2 = \texttt{substitute}(\sigma, \tau_1)$$

$$\textbf{true} = (\alpha \notin \mathit{dom}(\sigma)) \vee ((\alpha \mapsto \tau_\alpha) \in \sigma \wedge \texttt{substitute}(\sigma, \tau_\alpha) = \tau_2)$$

`argTy`

`argTy` determines the type of an argument; if the argument is a variable, it looks up its mapping in the type environment. All arguments must be literals (integers) or previously bound in the environment.

$$\texttt{argTy} \in \mathit{Env} \times \mathit{Argument} \to \mathit{Type}$$

$$\texttt{argTy}(\Gamma, \mathit{arg}) = \begin{cases} \textbf{Int} & \text{if } \mathit{arg} = n \\ \tau & \text{if } \mathit{arg} = x \wedge (x \mapsto \tau) \in \Gamma \end{cases}$$

`brTypes`

`brTypes` typechecks a list of branch bodies, mapping the branch's binders to the matching constructor's field types. It does so by iterating over a list of pattern branches, evaluating the branch's body expression in an environment where the pattern's binders have been mapped to the matching constructors field types. Note that the number of variables in a pattern branch need not equal the number of fields in the matching constructor; any variables without a matching field (because too many variables were supplied in pattern) are ignored. This is okay because any reference to those omitted variables (which shouldn't be allowed) will be caught during typechecking of the branch's body expression, giving an ill-typed result due to a bad environment lookup, as desired. This rule also shows that each pattern branch must have a matching constructor in the list of constructors; it is considered ill-typed for a pattern matching a constructor of a different datatype to be present in the list of branches of the current case.

$$\texttt{brTypes} \in \mathit{Env} \times \overrightarrow{\mathit{Constructor}} \times \overrightarrow{\mathit{PatCons}} \to \overrightarrow{\mathit{Type}}$$

$$\texttt{brTypes}(\Gamma, \overrightarrow{\mathit{cons}}, []) = []$$

$$\texttt{brTypes}(\Gamma, \overrightarrow{\mathit{cons}}, (\mathit{cn}\ \vec{x} \Rightarrow e) :: \overrightarrow{br}) = \tau :: \vec{\tau}$$

where

$$\textbf{true} = (\textbf{con}\ \mathit{cn}\ \vec{\tau} \in \mathit{cons})$$

$$\Gamma[\vec{x} \mapsto \vec{\tau}] \vdash e : \tau$$

$$\vec{\tau} = \texttt{brTypes}(\Gamma, \overrightarrow{\mathit{cons}}, \overrightarrow{br})$$

`BuiltinTypes`

`BuiltinTypes` maps primitive operator identifiers ($\oplus$ as shown in the abstract syntax) to the types of the primitive operations they represent. For example, + maps to the function type [**Int**, **Int**] $\to$ **Int**. The set of builtin operators and their types is fixed, straightforward, and thus omitted here for brevity's sake.

`Datatypes`

`Datatypes` is simply the list of datatypes extracted out of the top-level program definition.

`Functypes`

`Functypes` is a map from function and constructor identifiers to their respective types, created from the list of function and datatype declarations that constitute a program. $\overrightarrow{\mathit{data}}$ and $\overrightarrow{\mathit{func}}$ are the list of datatypes and functions, respectively, that constitute the contents of a program (see Figure 3.2).

$$\texttt{Functypes} \in \mathit{Operator} \to \mathit{Type}$$

$$\texttt{Functypes} = \vec{\tau}_c +\!\!+ \vec{\tau}_f +\!\!+ \vec{\tau}_b$$

where

$$\vec{\tau}_c = \texttt{concatMap}(\texttt{createConsTys}, \overrightarrow{data})$$

$$\vec{\tau}_f = \texttt{map}(\texttt{createFuncTy}, \overrightarrow{func})$$

$$\vec{\tau}_b = \texttt{BuiltinTypes}$$

createConsTys

createConsTys is a helper function for Functypes, creating types for each of the constructors that are part of a datatype. If a constructor has fields, its type is a function type where those fields are the parameter types and the datatype of which it is a member is the return type. If the constructor doesn't have fields, its type is just the datatype of which it is a member.

$$\texttt{createConsTys} \in \mathit{Datatype} \to (\mathit{Name} \to \mathit{Type})$$

$$\texttt{createConsTys}(\textbf{data}\ tn\ \vec{\alpha}\ \texttt{=}\ \overrightarrow{cons}) = \texttt{map}(\texttt{createConTy}(tn\ \vec{\alpha}), \overrightarrow{cons})$$

createConTy

createConTy is a helper function for createConTy, creating a types a constructor that is a part of a datatype.

$$\texttt{createConTy} \in \mathit{Datatype} \times \mathit{Constructor} \to (\mathit{Name} \to \mathit{Type})$$

$$\texttt{createConTy}(dt, \textbf{con}\ cn\ \vec{\tau}_c)$$

$$\begin{cases} cn \mapsto (\vec{\tau}_c \to dt) & \text{if } |\vec{\tau}_c| \geq 1 \\ cn \mapsto dt & \text{otherwise} \end{cases}$$

`createFuncTy`

`createFuncTy` is a helper function for `Functypes`, creating a type for a functions defined in the program.

$$\texttt{createFuncTy} \in \mathit{Function} \to (\mathit{Name} \to \mathit{Type})$$

$$\texttt{createFuncTy}(\textbf{fun}\ \mathit{fn}\ \overrightarrow{x : \tau}\ \tau\ \texttt{=}\ e) =$$

$$\begin{cases} \mathit{fn} \mapsto (\vec{\tau} \to \tau) & \text{if } |\overrightarrow{x : \tau}| \geq 1 \\ \mathit{fn} \mapsto \tau & \text{otherwise} \end{cases}$$

`freshTypes`

`freshTypes` is a helper function for `idTy`, replacing all of a type's generic type variables with fresh generic type variables, consistently across the type.

$$\texttt{freshTypes} \in \mathit{Type} \to \mathit{Type}$$

$$\texttt{freshTypes}(\tau_1) = \tau_2$$

where

$$\vec{\alpha} = \texttt{filter}(\texttt{isGeneric}, \texttt{getTyVars}(\tau_1))$$

$$\sigma = \vec{\alpha} \mapsto \overrightarrow{\texttt{freshGenTV}}$$

$$\tau_2 = \texttt{substitute}(\sigma, \tau_1)$$

`freshGenTV`

`freshGenTV` gets a fresh uniquely-identifiable generic type variable.

`freshRigTV`

`freshRigTV` gets a fresh uniquely-identifiable rigid type variable.

getCons

getCons retrieves the constructors associated with a datatype and replaces any type parameters in those constructors' field types with any type variable instantiations recorded in the datatype.

$$\texttt{getCons} \in \mathit{Datatype} \to \overrightarrow{\mathit{Constructor}}$$

$$\texttt{getCons}(\mathit{tn}\ \vec{\tau}_t) = \overrightarrow{\mathit{cons}}_2$$

where

$$(\textbf{data}\ \mathit{tn}\ \vec{\alpha}_t\ \texttt{=}\ \overrightarrow{\mathit{cons}}_1) \in \texttt{Datatypes}$$

$$\sigma = \texttt{zip}(\vec{\alpha}_t, \vec{\tau}_t)$$

$$\overrightarrow{\mathit{cons}}_2 = \texttt{map}(\texttt{instCon}(\sigma), \overrightarrow{\mathit{cons}}_1)$$

getTyVars

getTyVars gets the set of type variables used in a type. unions takes the union of each set in a list.

$$\texttt{getTyVars} \in \mathit{Type} \to \mathcal{P}(\mathit{TypeVar})$$

$$\texttt{getTyVars}(\tau) =$$

$$\begin{cases} \{T\} & \text{if } \tau = T \\ \{\vec{T}\} & \text{if } \tau = \mathit{dt}\ \vec{T} \\ \texttt{unions}(\texttt{map}(\texttt{getTyVars}, \vec{\tau}_p)) \cup \texttt{getTyVars}(\tau_r) & \text{if } \tau = \vec{\tau}_p \to \tau_r \end{cases}$$

idTy

idTy allows let-polymorphism by replacing non-rigid type variables with fresh ones. It does so by getting the type associated with an identifier from the type environment, if present; otherwise, it looks up the identifier in the set of declared function types.

Afterwards, it uses `freshTypes` to replace all generic type variables in the type with fresh new ones. This function is used during the **let** expression for getting unique instances of types so that type variables do not inadvertently clash during the process of unification and substitution (let-polymorphism).

$$\texttt{idTy} \in \mathit{Env} \times \mathit{Identifier} \rightarrow \mathit{Type}$$

$$\texttt{idTy}(\Gamma, id) = \begin{cases} \texttt{freshTypes}(\tau) & \text{if } id = x \wedge (x \mapsto \tau) \in \Gamma \\ \texttt{freshTypes}(\tau) & \text{if } id = op \wedge \tau = \texttt{Functypes}(op) \end{cases}$$

instCon

`instCon` instantiates a constructor by replacing the type variables in its field types with its mapping in the constraint list.

$$\texttt{instCon} \in \mathit{Substitution} \times \mathit{Constructor} \rightarrow \mathit{Constructor}$$

$$\texttt{instCon}(\sigma, \textbf{con}\ cn\ \vec{\tau}) = \textbf{con}\ cn\ \texttt{substitute}(\sigma, \vec{\tau})$$

makeRigid

`makeRigid` converts all generic type variables in a type into consistently-renamed rigid type variables.

$$\texttt{makeRigid} \in \mathit{Type} \rightarrow \mathit{Type}$$

$$\texttt{makeRigid}(\tau_1) = \tau_2$$

where

$$\vec{\alpha} = \texttt{getTyVars}(\tau_1)$$

$$\sigma = \vec{\alpha} \mapsto \overrightarrow{\texttt{freshRigTV}}$$

$$\tau_2 = \texttt{substitute}(\sigma, \tau_1)$$

princType

princType determines if a list of types all refer to the same type, determining the most general type that matches all of them. For example, it is used at the end of the case typing rule to check that all branch bodies have a compatible type. This check is different than a normal check for equality because it takes into account type variables.

$$\texttt{princType} \in \overrightarrow{Type} \to Type$$

$$\texttt{princType}(\tau_h :: []) = \tau_h$$

$$\texttt{princType}(\tau_1 :: \tau_2 :: \vec{\tau}_{tl}) = \texttt{princType}(\tau_3 :: \vec{\tau}_{tl})$$

where

$$\sigma = \texttt{unify}(\{\tau_1 = \tau_2\})$$

$$\tau_3 = \texttt{substitute}(\sigma, \tau_2)$$

substitute

substitute takes a constraint set and a type, recursively replacing all type variables present as keys in the constraint set with their associated value.

$$\texttt{substitute} \in ConstraintSet \times Type \to Type$$

$$\texttt{substitute}(C, \tau) = \begin{cases} \tau_2 & \text{if } \tau = T \wedge (T = \tau_1) \in C \wedge \tau_2 = \texttt{substitute}(C, \tau_1) \\ tn\ \vec{\tau}_2 & \text{if } \tau = tn\ \vec{\tau}_1 \wedge \vec{\tau}_2 = \texttt{map}(\texttt{substitute}(C), \vec{\tau}_1) \\ \vec{\tau}_{p2} \to \tau_{r2} & \text{if } \tau = \vec{\tau}_{p1} \to \tau_{r1} \wedge \\ & \quad \vec{\tau}_{p2} = \texttt{map}(\texttt{substitute}(C), \vec{\tau}_{p1}) \wedge \\ & \quad \tau_{r2} = \texttt{substitute}(C, \tau_{r1}) \\ \tau & \text{otherwise} \end{cases}$$

`unify`

`unify` performs the standard process of unification, iterating over each constraint in the constraint set and creating a substitution that contains a mapping from type variables to types. Any cases unlisted in this function imply that the function fails in that case. Regarding rigid type variables: the rigid types β_1 and β_2 are successfully unified if and only if β_1 equals β_2. This is because in the context of checking a function, rigid type variables cannot be replaced or unified with any other concrete type or polymorphic type variable. For compound types like functions and data types, unification proceeds recursively.

Similar to map creation, we use the notation $\{\vec{\tau}_1 = \vec{\tau}_2\}$ to denote the creation of a constraint set where the first element of $\vec{\tau}_1$ is paired with the first element of $\vec{\tau}_2$, the second element of $\vec{\tau}_1$ paired with the second element of $\vec{\tau}_2$, etc.

$\texttt{unify} \in \mathit{ConstraintSet} \rightarrow \mathit{Substitution}$

$\texttt{unify}(\emptyset) = \{\}$

$\texttt{unify}(\{\tau_1 = \tau_2\} \cup C_1) =$

$$\begin{cases} \texttt{unify}(C_1) & \text{if } \tau_1 = \tau_2 \\ \sigma[\alpha \mapsto \tau_2] & \text{if } \tau_1 = \alpha \wedge \alpha \notin \texttt{getTyVars}(\tau_2) \wedge \\ & \quad C_2 = \texttt{updateConstraints}(C_1, \{\alpha = \tau_2\}) \wedge \sigma = \texttt{unify}(C_2) \\ \sigma[\alpha \mapsto \tau_1] & \text{if } \tau_2 = \alpha \wedge \alpha \notin \texttt{getTyVars}(\tau_1) \wedge \\ & \quad C_2 = \texttt{updateConstraints}(C_1, \{\alpha = \tau_1\}) \wedge \sigma = \texttt{unify}(C_2) \\ \texttt{unify}(C_2) & \text{if } \tau_1 = tn\ \vec{\tau}_1 \wedge \tau_2 = tn\ \vec{\tau}_2 \wedge |\vec{\tau}_1| = |\vec{\tau}_2| \wedge \\ & \quad C_2 = C_1 \cup \{\vec{\tau}_1 = \vec{\tau}_2\} \\ \texttt{unify}(C_2) & \text{if } \tau_1 = \vec{\tau}_{p1} \rightarrow \tau_{r1} \wedge \tau_2 = \vec{\tau}_{p2} \rightarrow \tau_{r2} \wedge |\vec{\tau}_{p1}| = |\vec{\tau}_{p2}| \wedge \\ & \quad C_2 = C_1 \cup \{\vec{\tau}_{p1} = \vec{\tau}_{p2}\} \cup \{\tau_{r1} = \tau_{r2}\} \end{cases}$$

updateConstraints

updateConstraints iterates through a constraint set, replacing all type variables in each constraint by their mapped types, if present in the substitution. This is a helper function used only by unify.

$\texttt{updateConstraints} \in \mathit{ConstraintSet} \times \mathit{Substitution} \to \mathit{ConstraintSet}$

$\texttt{updateConstraints}(\emptyset, \_) = \emptyset$

$\texttt{updateConstraints}(\{\tau_1 = \tau_2\} \cup C_1, \sigma) = \{\tau_3 = \tau_4\} \cup C_2$

where

$\tau_3 = \texttt{substitute}(\sigma, \tau_1)$

$\tau_4 = \texttt{substitute}(\sigma, \tau_2)$

$C_2 = \texttt{updateConstraints}(\sigma, C_1)$

3.3.3 Properties and Proofs

Two formal properties, when combined, can guarantee that the machine never has to create and return an error object. The first is progress, which says that if a term is well-typed, then there is always a way to continue evaluating it according to the semantic rules; the second is preservation, which says that if a term is well-typed, evaluating it will result in a well-typed term. Taken together, we have a guarantee that there will always be an applicable semantic rule to evaluate each step of the program, which means that we never encounter anything outside of our semantic definitions and never run into type or memory errors.

We prove progress and preservation in a straightforward way, via induction on the typing rules and the dynamic semantics, giving a brief overview below.

Lemma 4 (Apply Type)**.** $\texttt{applyType}(\tau_i, \vec{\tau}_a, C, \alpha)$ *returns the principal type of an application of a type to zero or more arguments.*

Proof. `applyType` generates a constraint for each parameter and argument until the list of arguments $\vec{\tau}_a$ is exhausted. Unifying these constraints to produce a substitution, it then determines the principal type of the application; this proof relies on standard

proofs on principal type generation.

Lemma 5 (Progress of Functions)**.** *Assuming the correct arguments are given, executing the body of a well-typed function*

fun *fn* $\overrightarrow{x : \tau}\ \tau$ **=** *e produces a value of type* $\tau \neq$ `Error`*, when the body terminates.*

Proof. The rule func-params checks a function body e in an environment Γ that maps the parameters in $\vec{x}$ to their declared types in $\vec{\tau}$. Any type variables in those parameter types and the return type are replaced with fresh rigid type variables, implying that those parameters cannot be specialized within the function body (i.e they are universally quantified across the entire function). Rule func-ret follows similarly, except that the initial environment used to typecheck the body e is empty. We must show that $\Gamma \vdash e : \tau$, that is, that the function body evaluates to a non-error value of type τ. The proof proceeds by induction on the derivation of e and using Lemma 4:

Case (let-var: $e =$ **let** x_1 **=** $id\ \overrightarrow{arg}$ **in** e_1). Let τ_i be the type of id; the helper function `idTy` looks up its entry in Γ if present, otherwise id is function or constructor name and its type is statically known by referring to the program's function type list. If $\overrightarrow{arg} = []$, then by induction $\Gamma[x_1 \mapsto \tau_i] \vdash e_1 : \tau$. Otherwise, by Lemma 4, $\texttt{applyType}(\tau_i, \vec{\tau}_a, [], \alpha) = \tau_1$ returns x_1's principal type (where $\vec{\tau}_a$ and α are the types of the arguments and x_1, initially, respectively). By induction then, $\Gamma[x_1 \mapsto \tau_1] \vdash e_1 : \tau$.

Case (let-int: $e =$ **let** x **=** n **in** e_1). Then $\Gamma[x \mapsto$ **Int**$] \vdash e_1 : \tau$ by induction, since $n :$ **Int**.

Case (case-con: $e =$ **case** x **of** $\overrightarrow{br}$). Let $(x \mapsto dt) \in \Gamma$, so that the value of x is a constructor named cn with fields of type $seqs\tau$. By definition, all constructors that are members of type dt must have a matching branch $cn_i\ \vec{x}_i \Rightarrow e_i inbr$. Additionally, each matching branch must contain at least as many binders $\vec{x}_i$ as fields in its constructor, such that all variable references in the branch body e_i are valid. (All of these requirements are handled in the helper function `allConsPres`.) The helper function `brTypes` then proceeds to

get the type of each branch body, in the environment Γ extended with bindings from the pattern binders to the constructor field types for that particular constructor. By induction, those bodies $\vec{e}_i$ return types $\vec{\tau}_i$, all of which must be unifiable as a type τ. As **case** is the only source of control flow, requiring case completeness ensures that machine cannot attempt to go to an invalid location or reach an error state.

Case (CASE-CON-ELSE: $e = \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_e$). This case proceeds as in CASE-CON above, except the restriction that all branches for the scrutinized datatype be present is relaxed. Instead, an else expression e_e is required to maintain control-flow safety.

Case (CASE-INT: $e = \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_e$). Let $(x \mapsto \textbf{Int}) \in \Gamma$, so that the value of x is n. Since all branches $\overrightarrow{n_i \Rightarrow e_i} \in \overrightarrow{br}$ may be a match, the typing rule requires all branch bodies result in a unifiable type. Additionally, since the number of values in the set of integers is enormous, the rule requires an else expression e_e be provided to handle the non-matching case. As the **case** expression is the only source of control flow in the system, and because we require an else branch when scrutinizing an integer, the machine cannot go to invalid location or produce an error at this stage. By induction, $\Gamma \vdash e_e : \tau_e$ and $\Gamma \vdash e_i : \tau_i$ for all branch bodies, which must be unifiable to the type τ.

Case (RESULT: $e = \textbf{result}\ arg$). If $arg = n$, then $\tau = \textbf{Int}$. If $arg = x$, then $(x \mapsto \tau) \in \Gamma$, stored previously via the rules LET-* or because it was an argument to the function. As this is the base instruction, it is always the last instruction in each branches of a function. Therefore its type will be the type of the function itself (and must match all other **result** types of the function, checked in the FUNC-* rules).

Theorem 2 (Progress of Programs)**.** *Let P be a well-typed program composed of a list of datatypes $\overrightarrow{data}$ and functions $\overrightarrow{func}$. Let $\left(\textbf{fun}\ \texttt{main}\ \overrightarrow{x : \tau}\ \tau\ \texttt{=}\ e\right) \in \overrightarrow{func}$ be the entry point to P where execution begins. Then P either halts and returns a value of type $\tau \neq \texttt{Error}$, or it continues execution indefinitely.*

Proof. By Lemma 5 and rule FUNC-PARAMS, we know that
fun `main` $\overrightarrow{x : \tau}$ τ `=` e has type τ (similarly for functions without parameters, using rule FUNC-RET). Since a hardware error value of type `Error` is created when the machine encounters an invalid state during evaluation, and Lemma 5 says that a well-typed function does not lead to an invalid state, P returns a value of type $\tau \neq$ `Error` when it terminates.

3.4 Algorithm for Analysis

As mentioned earlier, the Binary Exclusion Unit (BEU) can be used as a runtime guard, checking programs right before execution when they are loaded into memory, or as a program-time guard, checking programs when they are placed into program storage (flash, NVM, etc.). In either case, checking works the same way: each word of the binary is examined one at a time as it streams through. Central to this process is the embedded Type Reference Table (TRT), which is copied from the binary into the checker's memory and contains the type information for the binary. This serves as a reference during all stages of the checking process and will be extended during the checking of each function as local variables are introduced. Later, when the BEU arrives at a new function, it consults the function signature, which provides type information for the arguments and the return type of the function. Each instruction in the function is then scanned word-by-word, guaranteeing type safety of each instruction according to the static semantics (Figure 3.4). Checking can fail at any step of the process: e.g., a function might expect an `Integer` but is passed a `List`, or the `add` function, which expects two `Integer` arguments, is given three. A single type violation causes the entire program to be rejected. The steps required to check each instruction class are described in more detail below:

Let — When a `Let` instruction is encountered, we first check for special-case operations: applying no arguments to something will always result in the same thing, so we can simply assign the result to that type and do no further checking. Assuming the `Let` does have arguments, the checker then gets the type of the function and creates an alias of it in a new TRT entry. The point of the alias is to make each type variable unique — e.g., the same type `List a` (a list of elements of type "a") used in two places may not be using the same type for "a", so the separate usages should have separate type variables. In order to allow recursive `Let` operations, a type variable is assigned to the result of the operation; when all the arguments have been processed, that variable will be set equal to what's left. The checker goes through each argument, one at a time, and unifies its type with the function's expected type. This creates a list of constraints that, along with the constraint on the resulting type variable, are checked altogether as the last step. If there are no inconsistencies in the constraint set, the operation was valid, and a new valid type is produced for the local variable.

Because type inference is relatively simple, we chose to forgo type annotations on each function application that indicate the result of the operation. Instead, the checker uses function-local type inference to figure out the return type of each function application. Because function calls (`Let` instructions) make up the majority of the instructions in a binary, the absence of annotations on each one results in much smaller binary sizes for typed binaries.

Special care must be taken in `Let` instructions when the resulting type is a function, and when the function being applied has a function in its return type. The former requires creating a new TRT entry for the function; the latter requires a special "unfolding" routine to begin applying arguments to the function in the return type. Both of these are reasons that the `Let` section of the hardware checker has so many states (Table 3.2).

Case — `Case` instructions are much more straightforward. The checker simply saves some type information on what the program is casing on, which is used in later instructions. Specifically, the primary task is to get the type of the scrutinee (the thing being cased on) and save a reference both to the particular variable's type and the root program datatype (assuming the variable is a constructor, not an integer). For example, this way branches will know that a `List` was cased on, not a `Tuple`, and know that the particular variable was a `List Int` as opposed to a `List Char`.

`Pattern_literal` branch heads are quite simple: the case head must be an integer, and the value specified in the instruction must be an integer.

`Pattern_con` branch heads are one of the more complex things to check. We have to reconcile the generic type of the indicated pattern (constructor) with the specific type of the variable that we're matching against. To do this, the checker must get the function type specified in the pattern head, then alias it in a new TRT entry. Then we must generate the constraint that the return type of the function is the same as the type of the scrutinee — this ensures that the type variables in this entry will be constrained to be the same as those in the original scrutinee. Constraints can then be checked, yielding a map with which the variables can be recursively replaced to the correct types. Finally, a pointer is set to where the fields of the constructor begin (if applicable). When we are done, we have direct, usable information on the type of each field in the constructor, which can be used by following instructions.

In addition, we must keep track of which constructors we've seen in this case statement; that way, when we get to the end of the `Case`, we'll know if all of the constructors of that type were present or not. A `Case` statement must either contain an `else` branch or use all constructors of the scrutinee's type.

(a)

```
data List[a] = Cons a List[a] | Nil

fun map :: ((a) -> b, List[a]) -> List[b]
fun map f list =
case list of {
  Nil => let ret = Nil in ret
  Cons x xs =>
    let head = f x in
    let tail = map f xs in
    let ret = Cons head tail in
    ret
}
```

(b)

```
0 0x40000000    Error
1 0x40000000    Int
2 0x40010002    List a (1 TV, 2 constructors)
3 0xa0000002    Function of 1 arg
4 0xc0000001    arg: Int
5 0xc0000001    return type: Int
6 0xa0000003    Function of 2 args
7 0xc0000001    arg: Int
8 0xc0000001    arg: int
9 0xc0000001    return type: Int
10 0x80000401   Derived Type on List (List a)
11 0xe0000000   arg: Type variable 0
12 0xa0000003   Function of 2 args (Cons)
13 0xe0000000   arg: Typevar 0 (a)
14 0xc000000a   arg: List a
15 0xc000000a   return type: List a
16 0xa0000002   Function of 1 arg (a -> b)
17 0xe0000000   arg: Typevar 0
18 0xe0000001   arg: Typevar 1
19 0x80000401   Derived Type on List (List b)
20 0xe0000001   arg: Type variable 1
                Function Signatures:
25 0xa0000001   Function of no args (main)
26 0xc0000001   return type: Int
27 0xa0000003   Function of 2 args (Cons)
28 0xe0000000   arg: Typevar 0
29 0xc000000a   arg: Reference to 10
30 0xc000000a   return type: Reference to 10
31 0xa0000001   Function of no args (Nil)
32 0xc000000a   return type: Reference to 10
33 0xa0000003   Function of 2 args (map)
34 0xc0000010   arg: Reference to 16
35 0xc000000a   arg: Reference to 10
36 0xc0000013   return type: Reference to 19
```

Figure 3.5: An example Type Reference Table (TRT) for the function `map`. The original program is shown in (**a**), while (**b**) shows the actual binary type information that the assembler produces (annotated for human understanding). This type information is included at the head of the binary file, leaving the program unchanged. The first section lists types used in the signatures of the program, while the second section contains type information for the parameters and return type of each function. The type system is polymorphic and uses function-local type inference.

3.5 BEU Implementation

At a high level, the BEU is a hardware implementation of a pushdown automaton (PDA) and is structured as a state-machine with explicit support for subroutine calls. While there a numerous bookkeeping structures required, we must take care to access a single structure at a time to ensure we do not create structural hazards. The final analysis hardware is the result of a chain of successive lowerings from a high-level static semantics ending with a concrete state chart that we could then implement with minimal and straighforward hardware. First, a bit-accurate software checker was made that checked binary files. Then, a cycle-accurate software pushdown automata was written from that refined specification. From that program, the leap to real hardware was somewhat straightforward (see Section 3.7 for synthesis results). The full details of the checker cannot hope to fit in this paper, so we concentrate here on the strategy used at a high level and a couple of details to give a better sense for the full design.

The first challenge in implementing this analysis is how to encode the type information into the binary. As discussed in the prior section, we put this information at the head of each binary in the form of the TRT. To get a sense of what that actually looks like in a real implementation, Figure 3.5 shows an example TRT for the function `map`. This information is discarded after checking, leaving a standard, untyped binary, which executes with normal performance.

At the bit-level, we see only a sequential series of bytes. Therefore, all type information must be encoded into a single list. To avoid unnecessary complexity, we make all entries in the TRT fixed-width 32-bit words. An entry can be either 1) a program-specified datatype or built-in type[1], or 2) a derived type based on another type. Entries of type 2 can have one or more argument words, which we refer to as "typewords."

[1]The Zarf ISA includes integers and an error datatype built-in.

"Derived" here means that the entry contains references to other types in the table. This manifests as either a type applied to some type variables or as a function. For example, `List` is specified as a program datatype with one type variable, then derived type entries can create the types `List a`, `List Int`, etc, where `a` and `Int` are typewords following the derived type entry.

The second challenge in bringing the typechecker to a low level is dealing with recursive types. Implicitly, types in the system may be arbitrarily nested: for example, one could declare a `List` of `Tuple`s of `List`s of `Int`s. During the checking process, the hardware typechecker must be able to recursively descend through a type in order to make copies, do comparisons, and validate types. Because of this, the Binary Exclusion Unit cannot be expressed as a simple state machine — a stack is required for recursive operations (and hence the pushdown automaton).

Data structures used in the higher-level checking, like maps, need to be converted to structures native to hardware: they must flattened into a list, which can be stored in memory. In some cases, this requires a linear scan to check for the presence of some elements, such as checking case completeness — but those lists tend to be small, containing just one entry for each constructor of a given datatype. We found that all of the structures could be represented as lists with stack pointers, except in the case of the type variable map used in the recursive replace procedure, which required two lists (one to check for membership in the map, the second with values at the appropriate indices).

To create the control structure of the PDA, we started by implementing a software-level checker, broken into a set of functions implemented with discrete steps, where each step cannot access more than one array in sequence (in hardware, the arrays will become memories, which we do not want strung together in a single cycle). While, given our space constraints, it is difficult to describe the system in detail, the number of

Figure 3.6: The full state machine portion of the hardware Binary Exclusion Unit, which typechecks a binary, polymorphic type system. The machine makes use of several subroutines, so some states are missing incoming or outgoing edges; these correspond to jumps to and from a subroutine.

Purpose	**Number of States**
Initialization	21
Function signatures	15
Dispatch	6
Let checking	37
Return checking	3
Case checking	7
Literal pattern checking	1
Constructor pattern checking	21
Following references	6
Type variable (TV) counting	12
Recursive TV replacement	12
Recursive TV aliasing	26
Generating constraints	19
Checking constraints	21
Total	**207**

Table 3.2: Number of states devoted to the various parts of the Binary Exclusion Unit's state machine. Checking function calls, allowing for polymorphic functions with type variables, and constraint checking were the most complex behaviors, making up most of the states.

states for each part of the analysis is a reasonable proxy for complexity. The resulting state machine has 207 states and they are broken down by purpose in Table 3.2. We summarize them briefly here, with number of states denoted in parentheses. The initialization stage reads the program and prepares the type table (21 states). Function heads are checked to ensure the argument count matches the provided function signature, and bookkeeping is done to note the types of each argument and the return type (15). Dispatch decides which instruction is executed next and handles saving and restoring state as necessary for `Case` statements (6). `Let` (37), `Result` (3), `Case` (7), `Pattern_literal` (1), and `Pattern_con` (21) are checked as outlined in Section 3.4.

Because types can be recursively nested, a type entry in the TRT can reference other types; a set of states is devoted to following references to find root types as needed (6 states). To handle this, the state machine implements something akin to subroutines.

A routine executes at the beginning of each function that counts the number of type variables used in the signature (these type variables are "rigid" within the scope of the function and cannot be forced to equal a concrete type) (12). Another routine recursively replaces type variables to make one type entry match the variables in another; it allows pattern heads to be forced to the same type as the variable in the `Case` instruction (12). The aliasing subroutine recursively walks a type and maps its type variables to a "fresh" set (26). This allows, for example, each usage of `List a` to have "`a`" refer to a different type. Part of the complexity of this task is keeping track of the variables already seen and what they map to so that a variable is not accidentally mapped twice to different values. Constraint generation takes two type entries and, based on the entries, branches and generates the appropriate constraint for the constraint set indicating that the entries should be equal (19).

Finally, we have the constraint checking routine (21). This is invoked at the end of each `Let` instruction, as well as after a `Result`. Constraint propagation proceeds by taking one constraint from the set, which consists of a pair of types, then walking all the remaining constraints in the set and replacing all occurrences of the first type with the second. In this way, for each unique constraint, one type is eliminated from the constraint set. If at some point two different concrete types (like `Int` and `List`) are found to be equal, the set is inconsistent and typechecking fails, rejecting the program. Similarly, if ever a rigid type variable (a type variable used in the function signature) is found to be equal to a concrete type, typechecking fails. This second fail condition ensures that functions with polymorphic type signatures are, in fact, polymorphic. Without it, one could write a function that takes "`a`" and returns "`a`", which *should* work for all types, but in fact only works for integers.

As we developed our software and hardware checkers, we used a software fuzzing technique to generate 200,184 test cases based on prior techniques in program testing

[83]. Rather than generating random bits, which would not meaningfully exercise the checker, we encode the type system's semantics with logic programming and run them "in reverse" to generate, rather than check, well-typed programs. By performing mutations on half of these programs, we also generate ill-typed benchmarks. In all 200,184 generated test cases, the simulated hardware RTL has 100% agreement with the high-level checker semantics. The tests provide 100% of coverage of all 207 states of the checker.

While the resulting analysis engine is complex, one could certainly reuse parts of the analysis for other sets of properties and automated translation would be an interesting direction for future work. The software model is 1,593 lines of Python, while the hardware RTL is 1,786 lines (requiring extra code for declarations and the simulation test harness). Synthesis results are found in Section 3.7.

3.6 Provable Non-bypassability

The hardware static analysis we developed has a variety of states governing when it is active, how it initializes, and so on. An important point of this paper is the non-bypassibility of these checks, but we need to know that some sequence of inputs cannot be given to the checker that causes outputs to write to memory that have not been checked by analysis. To solve this problem, we can create an assertion and employ the Z3 SMT solver [84] to check it for us. Z3 is well-suited to our task because of its ability to represent logical constructs and solve propositional queries. In addition, because we can directly represent the circuit in Z3 at the logic level (gates), we do not have to operate at higher levels of abstraction and risk the proof not holding for the real hardware.

We actually translate our entire analysis circuit into a large Z3 expression. Then, we

add two constraints: the first says that, at some point in the operation of the circuit, it output the "passed" (meaning well-typed) signal, while the second says that at no point did the hardware enter the checking states. If the conjunction of the expressions is unsatisfiable, there is no way to get a "pass" signal without undergoing checking (and the program will never be loaded if it fails checking). Around 30 of the states deal with program loading, initialization, etc., and perform no checking; our proof guards against, for example, situations in which some clever manipulation of the state machine moves it from initialization directly to passing, or otherwise manages to circumvent the checking behavior of the state machine.

In the most direct strategy, we use the built-in `bitvec` Z3 type for wires in the circuit, with gates acting as logical operations on those bitvectors. Memories are represented as arrays. Arrays in Z3 are unbounded, but because we address the array with a bitvector, there is an implicit bound enforced that makes the practical array non-infinite.

A straightforward approach to handling sequential operation of the analysis is to duplicate the circuit once for each cycle we wish to explore. The cycle number is appended to the name of each variable to ensure they are unique. Obviously, because the entire circuit is duplicated for each cycle, this method does not scale well — both in terms of memory usage and the time it takes to determine satisfiability. Checking non-bypassability for numbers of cycles up to 32 took under 2 minutes and used less than 1 GB of RAM. Checking for 64 cycles used almost 16 GB and did not terminate within four days.

To make the SMT query approach scalable, we employ Z3's theory of arrays. Instead of representing each wire as a bitvector, duplicated once for each cycle, we represent it as an array mapping integers to bitvectors: the integer index indicates the cycle, while the value at the index is the value the wire takes in that cycle. There is then one array for each wire in the circuit, and one array of arrays for each memory in the circuit (the first

array represents the memory in each cycle, while the internal array gives the state of the memory in that cycle). Logical expression (gates) can then be represented as *universal quantifiers* over the arrays. For example, an AND expression might look like, `ForAll(i, wire3[i] == wire1[i] & wire2[i])`. This constrains the value of `wire3` for all cycles. Sequential operations are easy, simply referring to the previous index where necessary for register operations, e.g. `ForAll(i, reg1[i] == reg1_input[i-1])`. To bound the number of cycles, we add constraints to each universal quantifier that `i` is always less than the bound; this prevents Z3 from trying to reason about the circuit for steps beyond `i`.

Solving satisfiability with arrays took under two minutes and under one GB of RAM, no matter what bound we placed on the cycle count — in fact, even when *unbounded*, Z3 was still able to demonstrate our hardware analysis bypassibility assertion was unsatisfiable — i.e., the circuit is non-bypassable.

3.7 Evaluation

3.7.1 Checking Benchmarks

To understand if real-world programs can be efficiently typed and checked with our system, we implement a subset of the benchmarks from MiBench [85]. These tended to be much longer and more complex programs when compared to the randomly-generated ones. While the fuzzer's programs averaged 50-65 instructions per program, the embedded benchmarks range from 500 to over 7,000 and represent code structures drawn from real-world applications, such as hashes, error detection, sorting, and IP lookup. In addition to the MiBench programs, a standard library of functions was checked, as well as a synthetic program combining all the other programs (to see the

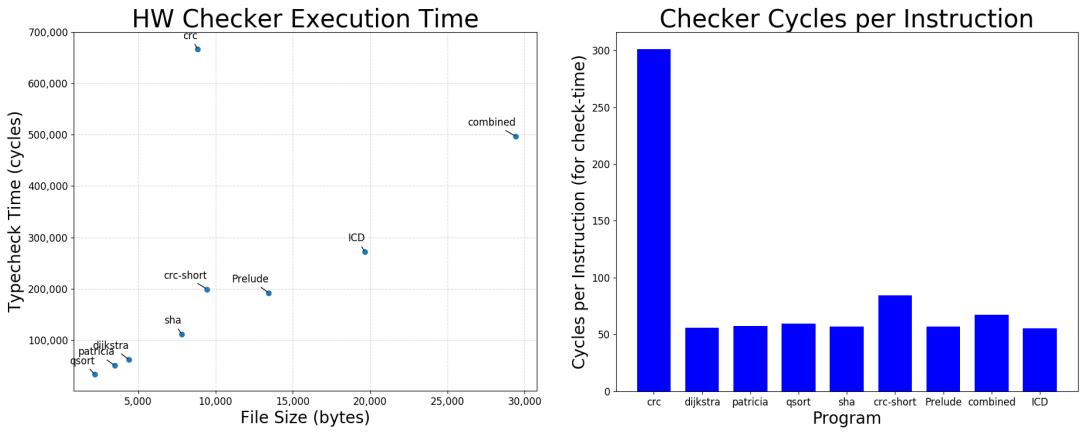


Figure 3.7: BEU evaluation for a set of sample programs drawn from MiBench, an embedded benchmark suite. For most programs, complete binary checking will take 150-160 cycles per instruction. LEFT: Time for hardware checker to complete, in cycles, as a function of the input program's file size. RIGHT: The same checking time, divided over the number of instructions in each program. Though the stock CRC32 has the longest typecheck time, an automatic procedure can modify the program to lower the checking time while preserving program semantics, noted as CRC-short.

characteristics of longer programs).

Figure 3.7 shows how long typechecking took for the benchmark programs as a function of their code size. A linear trend is clearly visible for most of the programs, but one stands out from the pack: the CRC32 error detection function. The default CRC32 implementation is, in fact, a pathological case for our checking method as it is dominated by a single large function in the program. This function constructs a lookup table used elsewhere and is fully unrolled in the code. No other benchmark had a function nearly as large. The typecheck algorithm, while linear in program length (it checks in a single pass), is quadratic in function length and type complexity[2]. This insight not only explains the anomalous behavior of the initial CRC32 program, but provides a clear solution: break up the large function.

We test this hypothesis by breaking up CRC32 and re-checking it. While the task of

[2]"Type complexity" refers to how many base types are in a type; i.e., the length of its member types, recursively.

breaking up a function in a traditional imperative programming language is complicated by the large amounts of global and implicit state, and would be even harder to perform at level of assembly, in a pure functional environment every piece of state is explicit. This makes the process not only easier, but even possible to fully automate. When we look at CRC32 specifically, the state, passed directly from one instruction to the next for table composition, can be captured in a single argument. We perform this transformation on our CRC32 program to break table construction across 26 single-argument functions, producing the CRC-short data point in the graphs in Figure 3.7. It still stands slightly above average because the table-construction functions are still above the average function length; recursively applying the breaking procedure could easily reduce the gap further.

While function length is an important aspect of checking time, with some care it can be effectively managed, and in the end all of the programs examined can be statically analyzed in hardware at a rate greater than 1 instruction per 100 cycles. This rate is more than fast enough to allow checking to happen at software-update-time, and could perhaps even be used at load-time, depending on the sensitivity of the application to startup latency.

3.7.2 Practical Application to an ICD

In addition to the benchmarks described above, we additionally provide results for a complete embedded medical application that was typed and checked; specifically, an ICD, or implantable cardioverter-defibrillator[3]. The ICD code was the largest single program examined (only the synthetic, combined program was larger). Its complexity

[3] An ICD is a device planted in a patient's chest cavity, which monitors the heart for life-threatening arrhythmias. In the case one is detected, a series of pacing shocks are administered to the heart to restore a safe rhythm.

required the use of multiple cooperating coroutines, managed by a small microkernel that handled scheduling and communication. Despite its length and complexity, it had the best typecheck characteristics of any of our test programs, with its cycles-per-instruction figure falling just below the average at 55.2. The process of adding types to the application was relatively simple, taking approximately 2 hours by hand.

Attempted Attack	**Result**
Binary that reads past the end of an object to access arbitrary memory	Hardware refuses to load binary due to type error "field count mismatch"
Binary that passes an argument to a function of the wrong type to cause unexpected behavior	Hardware refuses to load binary due to type error "not expected type"
Binary that writes past the end of an object to corrupt memory	Hardware refuses to load binary due to “application on non-function type”
Binary that passes too few arguments to a function to attempt to corrupt the stack	Hardware refuses to load binary due to "undersaturated call"
Binary that uses an invalid branch head to try and make arbitrary jump	Hardware refuses to load binary due to type error "branch type mismatch"
Binary that jumps past the end of a case statement to enable creation of ROP gadgets	Hardware refuses to load binary due to "invalid branch target"
Jump past the end of a function to create ROP gadgets	Hardware refuses to load binary due "invalid branch target"

Table 3.3: A list of some of the erroneous code that may be present in a binary (tested in our ICD application) and how the BEU identifies it as an error. Some of these errors, such as reading off the end of an object, writing beyond the end of an object, and jumping to arbitrary code points, are sufficient to thwart common attacks, like buffer overflow and ROP.

Since the ICD represents the largest and most complex program, as well as the exact type of program the BEU is designed to protect, we attempt to introduce a set of errors in the program to demonstrate the ability of the BEU to ensure integrity and security.

Some of the errors are designed to crash the program; some are designed to hijack control flow; others are designed to read privileged data. The list of attempted attacks and how the BEU caught them are shown in Table 3.3.

In an unchecked system, passing an invalid function argument, writing past the end of an object, and passing an invalid number of function arguments could all lead to undefined behavior or system crashes. While past work could establish that a specific piece of code would not do these things independent of the device, this work establishes these properties for the device itself, applying to all programs that can potentially execute — it is simply impossible to load a binary that will allow these errors to manifest. To establish that this was indeed the case, Table 3.3 shows the result of our attempts to produce these behaviors: a type error, a function application error, and an undersaturated call error, respectively. Reading past the end of an object in an attempt to snoop privileged data was thwarted by detecting a type error dealing with field count mismatches. Control-flow hijacks, like using an invalid branch head, jumping past the end of a case statement, and jumping past the end of a function, were caught by a type mismatch in the first case and the detection of an invalid branch target in the latter two.

Though not exhaustive, these attacks show the resilience of the system to injected errors when compared to an unchecked alternative and demonstrate its practicality in the face of real errors and attempted attacks.

3.7.3 Synthesis Results

Synthesized with Yosys, the hardware typechecker logic uses 21,285 cells (of which 829 are D Flip Flops, the equivalent of approximately 26 32-bit registers). Mapped to the open-source VSC 130nm library, it is .131 mm^2, with a clock rate of 140.8 MHz. Scaled to 32nm, it is approximately .0079 mm^2. As an addition to an embedded system

or SoC, it provides only a tiny increase in chip area, and requires no power at run-time (having already checked the loaded program).

Assuming the checker can use the system memory, it requires no additional memory blocks; if not, it needs a memory space at least as large as the input binary type information, and space linear in the size of the program's functions.

The worst-case checking rate was 301 cycles per instruction for a pathological program; even a program of 450,000 lines with worst-case checking performance can be checked in under a second at the computed clock speed of 140 MHz on 130nm.

3.8 Related Work

Typed Assembly

When dealing with typed assembly, the most prominent works are TAL [80] and its extensions TALx86 [81], DTAL [86], STAL [87], and TALT [88]. In TAL, they demonstrate the ability to safely convert high-level languages based on System F (e.g. ML) into a typed target assembly language, maintaining type information through the entire compilation process. Their target typed assembly provides several high-level abstractions like integers, tuples, and code labels, as well as type constructors for building new abstractions.

TALx86 is a version of IA32, extending TAL to handle additional basic type constructors (like records and sums), recursive types, arrays, and higher-order type constructors. They use dependent types to better support arrays; the size of an array becomes part of its type, and they introduce singleton types to track integer values of arbitrary registers or memory words. TAL provides a way to ensure that high-level properties like type- and memory-safety are preserved after compiler transformations and optimizations have taken place.

Unlike TAL, our type system was co-designed with hardware checking in mind — a distinction that greatly impacts the type system design. It allows for binary encoding of types and empowers the target machine, rather than the program authors, to decide if a program is malformed. TAL requires a complex, compile-time software typechecker, as opposed to our small, load-time hardware checker. Our type system operates on an actual machine binary and not an intermediate language.

The eventual target of TALx86 is untyped assembly code (assembled by their MASM assembler into x86). The types are not carried in the binary and are not visible to the device that ultimately runs the code. Though useful, a device cannot trust that the program it has been given has been vetted; therefore, bad binaries can still run on TAL's target machines.

Our work's most significant contribution, the Binary Exclusion Unit (BEU), overcomes this problem. The BEU, a hardware typechecker for the system capable of rejecting malformed programs, is an integral, non-bypassable part of the machine; if typechecking fails, execution cannot begin. To our knowledge, this is the only hardware module that performs typechecking on binary programs. We leave expansion of the BEU to other ISAs for future work, but note that the complexity of the TAL type system indicates that a hardware implementation would be significantly more work and overhead on an imperative ISA.

Architecture and Programming Languages

In SAFE [89], the authors develop a machine design that dynamically tracks types at the hardware level. Using these types along with hardware tags assigned to each word, their system works to prove properties about information-flow control and non-interference. They claim that the generic architecture of their system could facilitate efforts related to memory and control-flow safety in further work.

There has also been important work in binary analysis, which seeks to recover information from arbitrary binaries to make sound and useful observations. For example, Code Surfer [90] is a tool that analyzes executables to observe run-time and memory usage patterns and determine whether a binary may be malicious. Work on binary type reconstruction in particular seeks to recover type information from binaries. In one work [91], they recover high-level C types from binaries via a conservative inference-based algorithm. In Retypd [92], Noonan et al. develop a technique for inferring complex types from binaries, including polymorphic and recursive structures, as well as pointer, subtyping, and type qualifier information. Caballero et al. [93] provide a survey of the many approaches to binary type inference and reconstruction.

Static safety via on-card bytecode verification in a JavaCard [94] is an interesting line of work with a similar goal to our approach. However, a hardware implementation can be verified non-bypassable in a way that is much harder to guarantee for software. The Java type system is known to both violate safety [95, 96] and be undecidable [97] which makes it a far more difficult target for static analysis and, we would argue, nearly impossible to implement in hardware directly.

At the intersection of hardware and functional programming, previous works have synthesized hardware directly from high-level Haskell programs [98], even incorporating pipelined dataflow parallelism [99]. Run-time solutions to help enforce memory management for C programs have been proposed at the software level [100], as well as in hardware-enforced implementations [101, 102]; these provide run-time, rather than static, checks.

Other work has used formal methods to find and enforce properties at the hardware level to help ensure hardware and software security [103], while others have shows the effectiveness of hardware-software co-analysis for exploring and verifying information flow properties in IoT software [104].

3.9 Conclusion

While the micro-controller design in this paper might be an extremely non-traditional example, going so far as to have proofs of the properties that hold and rejecting non-conforming programs outright, it opens the door to other work that limits hardware functionality in meaningful and helpful ways without entirely giving up programmability. The result of our effort is a Binary Exclusion Unit that can easily fit into embedded systems or perhaps even serve as an element in a heterogeneous system-on-chip, providing a hardware-based solution that cannot be circumvented by software. Our approach prevents all malformed binaries from ever being loaded (let alone run), and ensures that all code loaded onto the machine is free from memory errors, type errors, and erroneous control flow. It requires neither special keys/attestation nor trust in any part of the system stack (a size zero TCB), providing its guarantees with static checks alone (no dynamic run-time checking is needed).

This approach has many non-traditional moving parts, from the function-oriented microprocessor at its heart, to the higher-level instruction set semantics, to the engine that performs static analysis in hardware. Rather than work on an architecture simulator, we built both the processor and the hardware checking engine in RTL, both to provide synthesis results and to demonstrate the feasibility of actually building such a thing. We have proofs of correctness for our approach at the algorithm level and gate-level proofs of non-bypassability. We coded and typed not just a set of benchmarks, but also a more complete medical application, which we then tried to break in order to show that such an approach works in practice as well as in theory. The final design is surprisingly small, taking no more than .0079 mm^2, and is capable of performing our static analysis on binaries at an average throughput of around 60 cycles per instruction. We believe this is the first time any binary static analysis has been implemented in

the machine hardware, and we think it opens an interesting new door for exploration where properties of the software running on a physical platform are enforced by the platform itself.

Chapter 4

Wire Sorts: A Language Abstraction for Safe Hardware Composition

4.1 Introduction

In our current era of diminished transistor scaling, the need for higher levels of energy efficiency and performance is greater than ever. The quest to achieve these goals calls for more people to be able to participate in the creation of accelerators and other digital hardware designs. It has become common for hardware designers to utilize commercial libraries (known as Intellectual Property or IP catalogs) to get hold of the most efficient or performant hardware components. At the same time, open-source hardware has begun to emerge as a viable development strategy, drawing parallels to open-source software, due to the commercial benefits of exploiting free and open components. This new development paradigm raises questions of how hardware developers can best compose their components and treat their underlying implementations as opaque.

Modern high-level programming languages have many mechanisms that work in

support of effective modularity and abstraction; for example, one might place requirements on data (e.g. arguments) at an interface (e.g. function call) through a type system. Most hardware description languages (HDLs), in contrast, have comparatively little support for these features. The interface of the primary unit of abstraction, a module, is typically described simply as "wires" which, in turn, may be refined as "input" or "output." However, we find experimentally across hundreds of designs that these interfaces actually carry surprisingly complex requirements not just on how the data are to be used or interpreted but even on what compositions leads to well-defined digital designs. The goal of our work is to turn a programming language eye to this problem: to be mathematically precise in the definition of wired interfaces and ultimately give more support to hardware designers seeking modularity, abstraction, and better compositional guarantees at the HDL level.

We wish to support a scenario where (1) separate hardware designers can independently create a set of hardware modules according to some connection protocol using an HDL, and the HDL can *automatically* infer relevant properties about the input and output wires for each module in isolation; (2) a hardware designer can treat these modules as opaque components without knowledge of their internals, wiring them together into a circuit such that the HDL provide guarantees based on the properties of the modules' input and output wires; and (3) the number of design "surprises" discovered late in the development cycle due to intermodular incompatibilities is significantly reduced.

Such a scenario is increasingly not just desirable but strictly necessary. In the traditional design methodology where a whole chip may be designed by a single company or team who can agree on interfaces in advance and readily inspect modules' internals, establishing modules' compositionality was straightforward. However, this is a much stickier problem in a world of IP-driven design where the user of a module may have

no knowledge of the module's internals, perhaps even working with an obfuscated or encrypted design [105]. IP catalog designers today lack any clear specification of the module-level connection properties needed to ensure well-composed designs. Thus, it is incredibly easy to create a design that assumes something about an up- or downstream interface which only becomes apparent after the *full design* has been completed at the RTL level. Discovering such an issue late in the process can be a serious issue because the exact cycle a data value is produced might need to change to accommodate a different interface. While this sounds easy in theory, traditional RTL design practices are fragile to timing changes, and fixing problems might mean significant surgery to control state machines, the recoordination of multiple producers or consumers, or even failure to meet a latency goal. As we ask a broader set of engineers to engage in the hardware design process, whether to understand tradeoffs in an AI accelerator design or deploy computation into an FPGA in the cloud, we need languages that help steer effort towards realizable designs and reduce the number of "surprises" (i.e. failures) typically only found at the very last stages of implementation (at synthesis time).

The specific property that we focus on in this work is what we are calling *well-connectedness*; we formally specify the property in Section 4.3.4 but informally, it implies that the final circuit does not contain any combinational loops.[1] Combinational loops are a sign of a broken design (except in certain rare circumstances) and must be avoided. Such loops are easy to spot once all components have been fully implemented and then synthesized into a netlist (one need only look for cycles in the netlist graph) but are hidden through the *entire* process of design at the HDL level, especially when they cross module boundaries and require reasoning about multiple modules' internal structures.

[1]This is a necessary but not sufficient condition for overall correctness. For example, we are not concerned in this paper about checking that a specific protocol is being correctly followed. Our techniques could potentially also reason about properties related to timing and circuit layout, but we leave these for future work.

This is a real problem we have encountered in our experiences writing digital hardware designs, motivating us to find better ways via programming language abstraction and enforcement.

This problem of avoiding combinational loops at the HDL module level is surprisingly subtle, requiring that designers reason about a number of non-obvious corner cases. Well-connectedness cannot always be guaranteed by looking at pair-wise module interconnections but is in fact a property of the entire circuit requiring information about all modules at once. Nevertheless, we show that it is possible to annotate module interfaces at the HDL level for each module independently such that the well-connectedness of a given combination of modules can be automatically proven by only looking at these interface annotations. We further show that if a full implementation of the design is already available, such as for legacy code, we can *automatically* infer annotations directly from the design. These annotations in turn radically lessen the number of interfaces where "surprises" might occur, allowing designers to focus their attention more effectively. The specific contributions of this paper are:

- We are the first to apply a modular static analysis to the problem of ensuring the correct compositionality of hardware modules in arbitrary RTL, via a global property which we define as *well-connectedness*.
- We prove this property is achievable in a *modular* way via a mathematical specification of wire dependencies, developing a novel taxonomy of sorts: **to-sync**, **to-port**, **from-sync**, and **from-port**.
- We embody these properties, and the analysis they enable, in a usable and scalable tool that completely prevents the late discovery of combinational loops. We further propose an extension to the analysis to protect synchronous memory semantics through composition.

- We analyze more than 500 parameterized hardware modules to quantify, for the first time, the diversity of expectations placed on module interfaces found in the wild. Across three independent projects (BaseJump STL, OpenPiton, and a RISC-V implementation) our analysis is able to automatically infer the correct wire sorts to enable composability in less than 31 seconds. Our analysis is 2.6–33.9x faster at finding intermodular loops than standard cycle detection during synthesis.

In Section 4.2 we describe related approaches to modular hardware design and why they do not solve the problem addressed in this paper. In Section 4.3 we describe the formalisms related to well-connectedness and the wire properties of **to-sync**, **to-port**, **from-sync**, and **from-port**, and discuss how to apply these formalisms to analyze module connections. In Section 4.4 we discuss details of the added annotation language and checker tool implementation. In Section 4.5 we evaluate our technique and tool by applying the checker to a series of standalone modules as well modularly-constructed circuits, and conclude in Section 4.6 we conclude.

4.2 Motivation and Related Work

To demonstrate the problem, we use the example of a simple first-in first-out (FIFO) queue using the ready-valid protocol, as shown in Figure 4.1. The role of the FIFO queue is to accept input data from one module (at the consumer endpoint), buffer that data inside its internal state, and then send the data to another module (at the producer endpoint) upon request. The consumer endpoint consists of a set of wires: $data_{\texttt{in}}$ contains the data being sent to the FIFO; $valid_{\texttt{in}}$ determines whether the incoming signals on $data_{\texttt{in}}$ represent valid input from the connected module; and $ready_{\texttt{out}}$ is an outgoing signal indicating whether the FIFO is ready to accept input (i.e., it isn't full). Similarly, the producer endpoint consists of another set of wires: $data_{\texttt{out}}$ contains the

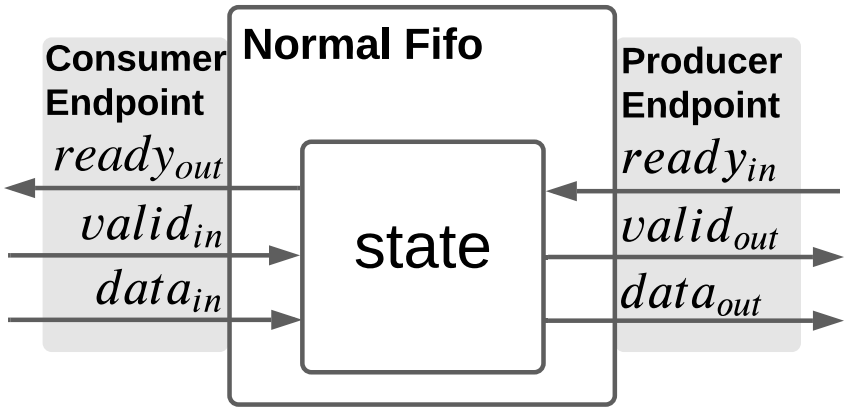


Figure 4.1: Normal FIFO queue. The consumer endpoint receives data from one module, and the producer endpoint sends data to another module.

data being produced by the FIFO and read from another connected module; $valid_{\mathtt{out}}$ determines whether the outgoing signals on $data_{\mathtt{out}}$ represent valid data from the FIFO (i.e., it isn't empty); and $ready_{\mathtt{in}}$ is an incoming signal indicating whether the connected module is ready to receive data from this FIFO.

We have left the internals of the FIFO opaque (as they may realistically be to a user); the details do not matter for our purposes except to note that each FIFO endpoint is combinationally independent of the other. In other words, every path between the endpoints is interrupted by some state inside the FIFO, so that an action at one endpoint cannot affect the other endpoint within a single cycle.

A FIFO queue of this kind is often called a "universal interface" because it can be placed between any two modules without danger of ill effects due to timing issues. However, for various reasons (such as efficiency) a normal FIFO queue may not be appropriate. A *forwarding* FIFO improves efficiency by allowing data entering in one clock cycle to be immediately sent out in the same clock cycle if the FIFO is empty. An abstract depiction of this module is shown in Figure 4.2.

The important points for our purposes are that: (1) the module interface (i.e., the ready-valid endpoint specification) is unchanged from the normal FIFO, so that from a

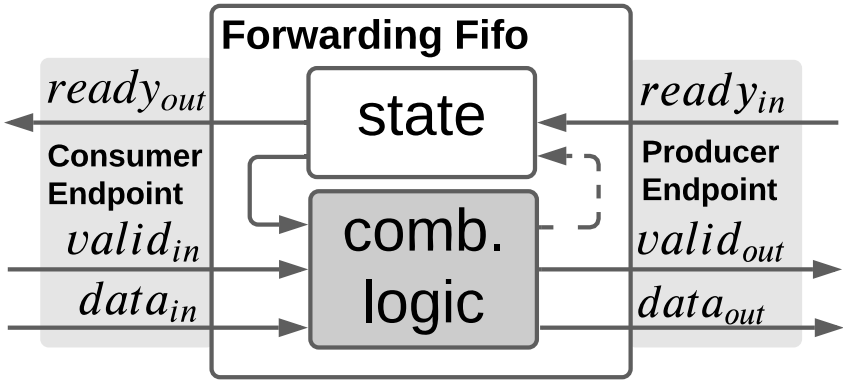


Figure 4.2: Forwarding FIFO queue.

module connection standpoint the two are indistinguishable; and (2) the endpoints are no longer combinationally independent because there is a combinational path from one endpoint to the other, enabling the data forwarding that is the whole point of the new module. Here's a closer look at the combinational logic used for assigning to $valid_{\mathtt{out}}$ across the two FIFO modules (where $count_{\mathtt{reg}}$ is a register containing the number of enqueued elements):

- **Normal:**

  $valid_{\mathtt{out}} ::= (count_{\mathtt{reg}} > 0)$

- **Forwarding:**

  $valid_{\mathtt{out}} ::= (count_{\mathtt{reg}} > 0) \vee (valid_{\mathtt{in}} \wedge ready_{\mathtt{out}})$

This combinational dependence between the endpoints means that designers may inadvertently cause a combinational loop when they wire modules together. In fact, the problem may not even arise due to direct interactions between the queue and the modules connected to its endpoints, but rather due to indirect interactions mediated by yet other modules. We show an example of a problematic circuit in Figure 4.3.

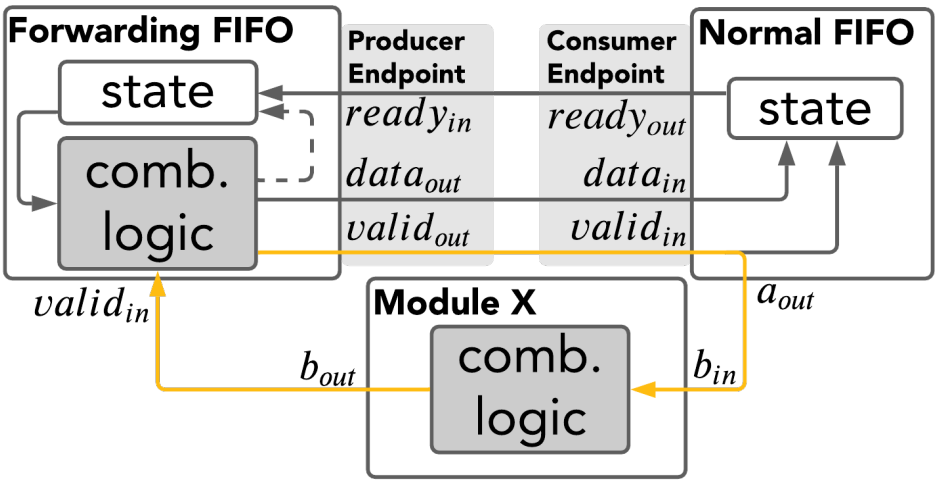


Figure 4.3: Forwarding FIFO connected to other modules causing a combinational loop (in yellow). Only pertinent IO ports have been shown for each module.

Here we have three modules: a normal FIFO, a forwarding FIFO, and some module X. In this contrived example, the normal FIFO sends a signal to module X that is some combinational function of its $valid_{\texttt{in}}$ wire (here, a direct connection); module X sends some combinational function of its input to the forwarding FIFO's $valid_{\texttt{in}}$, which (as previously discussed) is a combinational input to the forwarding FIFO's $valid_{\texttt{out}}$, which in turn is wired to the normal FIFO's $valid_{\texttt{in}}$. If the forwarding FIFO were instead a normal FIFO (which at a module connection level looks the same) then this would be fine, but since it is not this circuit contains a combinational loop. Detecting and understanding the cause of the loop requires reasoning about the internal details of three different modules.

We note that detecting the existence of the combinational loop is simple once the HDL program has been synthesized to a netlist: simply perform a standard cycle detection algorithm. Verilog [106] synthesis tools such as the linter in Verilator [107] and the Yosys synthesis suite [108] and HDLs such as Chisel [109] and PyRTL [110, 111] can provide warnings about loops during synthesis. However, relying on loop detection

at synthesis time has several drawbacks. First, gate-level netlists take a long time to produce and are significantly larger (47X larger in one example we studied), since high-level and multi-bit operations have been transformed into sets of simple 1-bit primitive gates. Second, these detection systems aren't infallible: under certain combinations of flags or optimizations, tools like Yosys fail to detect loops or silently delete them, "successfully" synthesizing the offending circuit. Third, once the loop is detected after synthesis, it is entirely up to the designer to trace the synthesized loop back to the relevant modules and interactions in the original HDL program.

The RTL, on the other hand, has fewer gate dependencies to analyze while still representing the same dataflow graph. Going up one level of abstraction, the behavioral level describes the same system algorithmically, making it even easier to take advantage of high-level constructs for determining dependency. Thus, our goal is to raise the level of abstraction for detecting loops up to the HDL module level in order to give the designer maximum information and context, to avoid loops more easily, and to detect loops sooner in the design process. An apt analogy is the difficulty in trying to determine the cause of an assembly-level link time error versus one presented at the source level; we aim to do the latter for HDLs.

There do exist HDL-level tools to check certain kinds of properties, for example SystemVerilog Assertions (SVA) [112], Property Specification Language (PSL)/Sugar [113, 114], and Open Verification Library (OVL) [115]. These frameworks facilitate the specification of temporal relationships between wires, which are checked via simulation or model checking rather than statically at design time. These tools can express properties about the relative order in which things occur but not the reasons why they occur. Since our analysis is concerned with the exact causes of events (i.e., combinational dependencies between wires), we believe from our experience using these tools that they are not suitable for our purpose.

There is additionally a long history of using higher-level abstractions to describe hardware formally [116, 117] and of using richer type systems [118] and functional programming techniques [119, 120, 121, 122]. DSLs like Murφ [123] and Dahlia [124] target specific use cases like protocol descriptions or improved accelerator design, while high-level synthesis (HLS) techniques [125, 126] translate subsets of C/C++ to RTL. Other HDLs [127] like PyMTL [128], Cλash [129], Pi-Ware [130], HardCaml [131], BlueSpec [132], and Kami [20] also use modern programming language techniques to overcome some of the issues that arise when writing in traditional HDLs [133, 134]; like many of them, we focus on improving the register-transfer level design process by creating better and more expressive abstractions.

4.2.1 BaseJump STL

The closest work to our own is BaseJump STL [135, 136]. Their work discusses the requirements for creating a library of hardware modules (analogous, in their words, to the C++ standard template library) and introduces some informal terminology to help describe module interfaces and promote properties such as well-connectedness. They draw upon the principles of latency-insensitive hardware design [137, 138, 139, 140] but aim for a less restrictive model.

BaseJump STL informally defines the notions of **helpful** and **demanding** module interface endpoints (such as the ready-valid endpoints from the previous FIFO example). The distinction is based on whether an endpoint is able to offer up data without "waiting" for input. For the ready-valid protocol, a **helpful** producer offers $valid_{\texttt{out}}$ upfront while a **demanding** producer waits for $ready_{\texttt{in}}$ before computing and emitting $valid_{\texttt{out}}$. Similarly, a **helpful** consumer offers $ready_{\texttt{out}}$ upfront while a **demanding** consumer waits for $valid_{\texttt{in}}$ before computing and emitting $ready_{\texttt{out}}$. BaseJump STL creates a taxonomy of

interface connections based on the various combinations of **helpful** and **demanding** endpoints. They note that the only unsafe combination is a **demanding-demanding** connection, which would directly lead to a combinational loop.

The problem with BaseJump STL's approach is that it considers module endpoint connections in isolation: the notion of dependence inherent in the **demanding** and **helpful** classifications only considers wires that directly participate in the connection. However, this isn't sufficient to guarantee detection of combinational loops, as we have shown with our previous example of a problematic circuit in Figure 4.3. In that example, the forwarding FIFO's producer endpoint is considered **helpful** because $valid_{\texttt{out}}$ is offered without needing to wait on $ready_{\texttt{in}}$. The normal FIFO's consumer endpoint is considered **helpful** because $ready_{\texttt{out}}$ doesn't wait on $valid_{\texttt{in}}$. According to BaseJump STL's model, these modules have a **helpful-helpful** connection and are therefore safe. But as we have demonstrated, the design is actually faulty due to the third module in the circuit and how it interacts with the connection between the forwarding and normal FIFOs.

*We discovered the issues with BaseJump STL's notions of **helpful** and **demanding** endpoints when we attempted to formalize them and prove that they were adequate to detect combinational loops at the HDL module level of abstraction.* Our experience led us to conclude that in order to guarantee well-connectedness, we need to: (1) be able to reason about module endpoints based on wire dependencies between the input and output wires within a module; and (2) using only the resulting endpoint annotations, reason about an entire circuit at the module level to resolve possible loops introduced by interactions between multiple modules.

4.3 Wire Sorts and Well-Connectedness

In this section we define our notion of wire sorts, formalize the property of well-connectedness using these sorts, and prove a set of properties that can be used to demonstrate that a circuit composed of independently designed modules is well-connected. Finally, we show exactly how our definitions contrast to BaseJump STL's notions of **helpful** and **demanding** endpoints and how our approach avoids the problems that BaseJump STL encounters.

4.3.1 Defining Basic Domains

We formally define a set of basic domains that collectively comprise a circuit composed of independent modules, so that we can precisely define wire sorts and well-connectedness and prove that a well-connected circuit has no combinational loops. Our formalisms and techniques apply to synchronous digital designs, and we assume for simplicity that there is a single clock driving all stateful elements (both are properties of the most commonly found designs).

A *wire* is denoted by w_σ where $\sigma \in \{\texttt{const}, \texttt{reg}, \texttt{in}, \texttt{out}, \texttt{basic}\}$. A constant wire $w_{\texttt{const}}$ produces a 0 or 1, an input wire $w_{\texttt{in}}$ serves as input into a module, and an output wire $w_{\texttt{out}}$ serves as output from a module. Registers are stateful elements that are latched each cycle according to the same shared clock; the $w_{\texttt{reg}}$ wires represent the outputs of these registers. Basic wires $w_{\texttt{basic}}$ are used to connect or combine these wires together via nets. A *net* is a tuple $(\overrightarrow{w_\sigma}, w_\sigma, op)$ representing a gate, with multiple wires $\overrightarrow{w_\sigma}$ coming into the gate, a single wire w_σ coming out of the gate, and a bitwise logical operation op denoting the type of gate such that $w_\sigma = op(\overrightarrow{w_\sigma})$.

A *module M* is a tuple $(\overrightarrow{w_{\texttt{in}}}, \overrightarrow{w_{\texttt{out}}}, \overrightarrow{net})$ composed of sets of input wires, output wires, and nets representing a directed acyclic graph (DAG); in this DAG, the nets are nodes,

and the outputs of the nets are the forward edges in the graph. The input and output wires form the module's external interface. Given a module $M = (\overrightarrow{w_{\texttt{in}}}, \overrightarrow{w_{\texttt{out}}}, \overrightarrow{net})$ we will use the shorthand *M.inputs*, *M.outputs*, and *M.nets* to mean $\overrightarrow{w_{\texttt{in}}}$, $\overrightarrow{w_{\texttt{out}}}$, and $\overrightarrow{net}$, respectively.

A *circuit* C is a tuple $(\overrightarrow{M}, \overrightarrow{(w_{\texttt{out}}, w_{\texttt{in}})})$ composed of a set of modules *C.modules* and the connections *C.conns* between their inputs and outputs. Given $M_1, M_2 \in C.modules$ and two wires $w_{\texttt{out}} \in M_1.outputs$ and $w_{\texttt{in}} \in M_2.inputs$, we use $w_{\texttt{out}} \rightarrow_C w_{\texttt{in}}$ to mean that $w_{\texttt{out}}$ is directly connected to $w_{\texttt{in}}$, i.e., $(w_{\texttt{out}}, w_{\texttt{in}}) \in C.conns$. We define the function $\texttt{module}(w_{\texttt{in}}, C) = M$ iff $w_{\texttt{in}} \in M.inputs \wedge M \in C.modules$. Without loss of expressiveness, we assume that one module's outputs are always connected directly to another module's inputs.[2] Note that a circuit C and its set of modules *C.modules* can essentially define a larger module composed of *submodules*. A circuit composed of many of these "supermodules" connected together in turn makes an even larger module, ad infinitum. Thus the intra- and intermodular analyses we discuss in the following sections are fully generalizable to the notion of submodules common in popular HDLs.

4.3.2 Defining Combinational Reachability

We define two different levels of combinational reachability: one intra-modular that can be computed for each module independently and one inter-modular that involves the entire circuit.

Given a module M containing a wire w_σ, we define the *combinationally reachable set* $\texttt{reachable}(M, w_\sigma)$ as the set of wires reachable from w_σ in *M.nets* without going through any wire $w_{\texttt{reg}}$; in other words, the transitively reachable wires that don't go through any registers (state).

We can now define two terms that will be important for determining combinational

[2] If there is any extra-modular logic between modules, one can wrap that logic into its own module to trivially meet this condition.

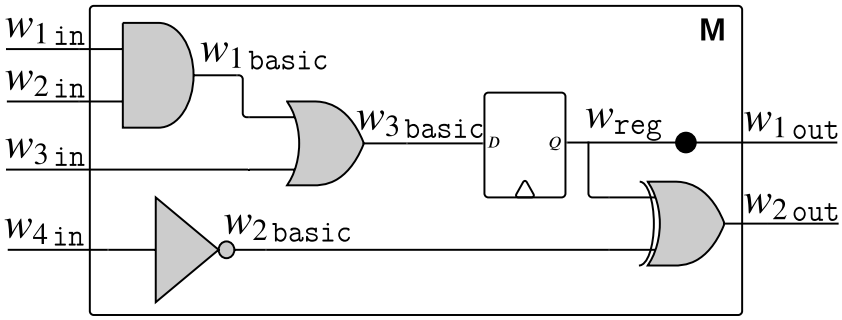
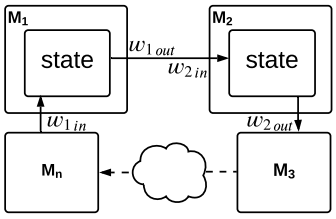
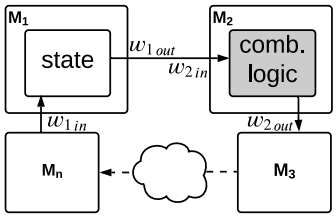


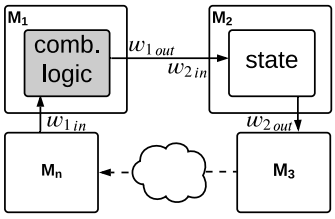
Figure 4.4: Example for computing the output-port-set and input-port-set of a module M. The output-port-set of input $w_{4\texttt{in}}$ is $\{w_{2\texttt{out}}\}$ and $\emptyset$ for the other inputs. The input-port-set of $w_{2\texttt{out}}$ is $\{w_{4\texttt{in}}\}$ and $\emptyset$ for $w_{1\texttt{out}}$.



(a) A **from-sync** wire $w_{1\texttt{out}}$ connected to a **to-sync** wire $w_{2\texttt{in}}$.



(b) A **from-sync** wire $w_{1\texttt{out}}$ connected to a **to-port** wire $w_{2\texttt{in}}$.



(c) A **from-port** wire $w_{1\texttt{out}}$ connected to a **to-sync** wire $w_{2\texttt{in}}$.

Figure 4.5: Connections between **to-sync** or **from-sync** wires cannot result in combinational loops.

reachability at the module level without needing the internal details of the relevant modules: **output-port-set** and **input-port-set**. The output-port-set is relevant for module inputs: given module M and input $w_{\texttt{in}}$, the output-port-set $\texttt{output-ports}(M, w_{\texttt{in}})$ is the set of module output wires that are combinationally reachable from that input wire. In other words, $\texttt{output-ports}(M, w_{\texttt{in}}) = \texttt{reachable}(M, w_{\texttt{in}}) \cap M.outputs$. Similarly, the input-port-set is relevant for module outputs: for an output wire $w_{\texttt{out}}$ of module M, the input-port-set $\texttt{input-ports}(M, w_{\texttt{out}})$ is the set of module input wires that combinationally reach that output wire. In other words, $\texttt{input-ports}(M, w_{\texttt{out}}) = \{w_{\texttt{in}} \mid w_{\texttt{in}} \in M.inputs, w_{\texttt{out}} \in \texttt{output-ports}(M, w_{\texttt{in}})\}$. These sets need only be computed once per module definition (regardless of how many instantiations are used in a circuit).

To illustrate these definitions consider the module diagram in Figure 4.4. In this module, which we'll call M, the output-port-set of input $w_{4\texttt{in}}$ is $\texttt{output-ports}(M, w_{4\texttt{in}}) = \{w_{2\texttt{out}}\}$, while the output-port-set of each of the inputs $w_{1\texttt{in}}, w_{2\texttt{in}}, w_{3\texttt{in}}$ is $\emptyset$. The input-port-set of output $w_{1\texttt{out}}$ is $\emptyset$, while the input-port-set of $w_{2\texttt{out}}$ is $\texttt{input-ports}(M, w_{2\texttt{out}}) = \{w_{4\texttt{in}}\}$.

Given a circuit composed of multiple modules along with the output-port-set and input-port-set for each input and output wire of each module, we can compute inter-modular combinational loops without needing to inspect the internals of any module. The transitive forward reachability of any output wire amounts to a fixpoint computation involving the output-port-sets of the modules in the circuit; while tracing a path from between wires, if a module input wire is encountered, skip over its module's internal logic by continuing with the output wires in its output-port-set. We use $w_1 \rightsquigarrow_C w_2$ to denote that wire w_1 transitively affects wire w_2 in circuit C and call $\rightsquigarrow_C$ the *TransitivelyAffects* relation. This computation is shown in Algorithm 1.

Algorithm 1: TransitivelyAffects

Data: $w_{\texttt{out}}, w_{\texttt{in}}, C$
Result: `True` if output wire $w_{\texttt{out}}$ of one module affects input wire $w_{\texttt{in}}$ of another module in circuit C, otherwise `False`.

```
1  begin
2      a ← ∅;
3      c ← {w_1in | (w_out, w_1in) ∈ C.conns};
4      while c ≠ ∅ do
5          w_1in ← pop(c);
6          if w_1in = w_in then
7              return True;
8          if w_1in ∈ a then
9              continue;
10         M ← module(w_1in, C);
11         forall w_1out ∈ output-ports(M, w_1in) do
12             c ← c ∪ {w_2in | (w_1out, w_2in) ∈ C.conns};
13         a ← a ∪ {w_1in};
14     return False;
```

4.3.3 Wire Sorts

We can now formally define the novel set of sorts for module input and output wires, a key contribution of this paper. An input wire $w_{\texttt{in}}$ is **to-sync** if $\texttt{output-ports}(M, w_{\texttt{in}}) = \emptyset$ and is **to-port** otherwise. An output wire $w_{\texttt{out}}$ is **from-sync** if $\texttt{input-ports}(M, w_{\texttt{out}}) = \emptyset$ and is **from-port** otherwise. The **to-sync**, **to-port**, **from-sync**, or **from-port** designation of a wire is its **sort**, and this set of sorts is sufficient to label all module ports. In Figure 4.4, the sort of input wires $w_{1\texttt{in}}, w_{2\texttt{in}}, w_{3\texttt{in}}$ is **to-sync** while the sort of $w_{4\texttt{in}}$ is **to-port**. Of the outputs, the sort of $w_{1\texttt{out}}$ is **from-sync** while the sort of $w_{2\texttt{out}}$ is **from-port**.

Note that an input wire of sort **to-sync** cannot be involved in a combinational loop, nor can an output wire of sort **from-sync**. By definition, these wires terminate or originate in some stateful or constant-valued element, and therefore module interface wires of these sorts can be freely connected to other modules safely without regard to

the connected module's interface wire sorts or the rest of the circuit. This leads us to our first property.

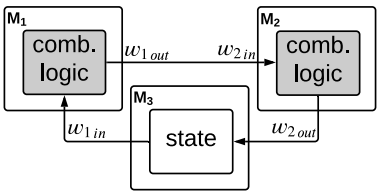
Property 1. *Two connected wires w_{out} and w_{in} cannot be involved in a combinational loop if w_{out} is **from-sync** or w_{in} is **to-sync**.*

Summary. Given a module M_1 such that $w_{\texttt{out}} \in M_1.outputs$, if $w_{\texttt{out}}$ is **from-sync**, then $\texttt{input-ports}(M_1, w_{\texttt{out}}) = \emptyset$, meaning it does not combinationally depend on any module input. Similarly, given a module M_2 such that $w_{\texttt{in}} \in M_2.inputs$, if $w_{\texttt{in}}$ is **to-sync**, then $\texttt{output-ports}(M_2, w_{\texttt{in}}) = \emptyset$, meaning it does not combinationally affect any module output. □

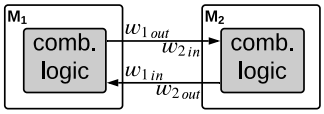
In Figure 4.5a, **from-sync** wire $w_{1\texttt{out}}$ is connected to **to-sync** wire $w_{2\texttt{in}}$, while in Figure 4.5b, it is connected to **to-port** wire $w_{2\texttt{in}}$. We can see that it doesn't matter what sort of input $w_{1\texttt{out}}$ connects to, since there is at least one stateful element shielding $w_{1\texttt{out}}$ from being fed into itself combinationally: the stateful elements of M_1 in both figures and additionally the stateful elements of M_2 in Figure 4.5a. In both cases, it doesn't matter what modules $M_3 \ldots M_n$ may do or any other output M_1 may have that could possibly feed into them. Similarly, in Figure 4.5c, because **from-port** wire $w_{1\texttt{out}}$ is connected to **to-sync** wire $w_{2\texttt{in}}$, we can know even without analyzing the entire circuit that this particular connection is safe.

4.3.4 Defining Well-Connectedness

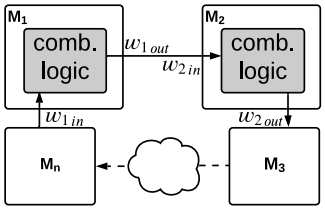
There are cases, like our previous example of a forwarding FIFO queue in Section 4.2, where it doesn't make sense to require that module interface wires be only **to-sync** or **from-sync**. Relaxing this requirement means we cannot rely solely on Property 1 for establishing safety between wires, and so we must more precisely define our notion of inter-wire safety as follows:



(a) A **from-port** wire $w_{1\texttt{out}}$ connected to a **to-port** wire $w_{2\texttt{in}}$, which through interactions with other modules can be shown to not result in a combinational loop.



(b) A **from-port** wire $w_{1\texttt{out}}$ connected to a **to-port** wire $w_{2\texttt{in}}$ resulting in a combinational loop.



(c) A **from-port** wire $w_{1\texttt{out}}$ connected to a **to-port** wire $w_{2\texttt{in}}$, which cannot be determined to be well-connected until the entire circuit has been completed.

Figure 4.6: Connections between **from-port** or **to-port** wires might result in combinational loops.

Definition 1 (Wire Well-Connectedness)**.** *Given a circuit C and two modules* $M_1, M_2 \in C.\textit{modules}$ *(where* M_1 *may be* M_2*), an output wire* $w_{\texttt{out}} \in M_1.\textit{outputs}$*, and an input wire* $w_{\texttt{in}} \in M_2.\textit{inputs}$ *such that* $w_{\texttt{out}} \rightarrow_C w_{\texttt{in}}$*,* $w_{\texttt{out}}$ *is* ***well-connected*** *to* $w_{\texttt{in}}$ *iff* $\forall w_1 \in \texttt{input-ports}(M_1, w_{\texttt{out}})$*,* $\forall w_2 \in \texttt{output-ports}(M_2, w_{\texttt{in}})$*,* $w_2 \not\rightsquigarrow_C w_1$*.*

It is straightforward to show that it satisfies our desired safety property:

Property 2. *The connection between two wires* $w_{\texttt{out}}$ *and* $w_{\texttt{in}}$ *that are well-connected to one another does not introduce a combinational loop.*

Summary. By definition, all of the input wires w_1 in M_1 that combinationally affect $w_{\texttt{out}}$ are present in its input-port-set. Likewise, by definition, all of the output wires w_2 in M_2 that are combinationally affected by $w_{\texttt{in}}$ are in its output-port-set. If it is impossible to transitively trace any output wire w_2 through the nets it combinationally affects to any input wire w_1 that $w_{\texttt{out}}$ is awaiting, then no combinational loop has been introduced by $w_{\texttt{out}} \rightarrow w_{\texttt{in}}$. □

We illustrate this property in Figure 4.7 below.

Any wires of sort **to-port** or **from-port** are potential problems, so we cannot in general determine safety without inspecting the entire circuit. For example, Figure 4.6a and

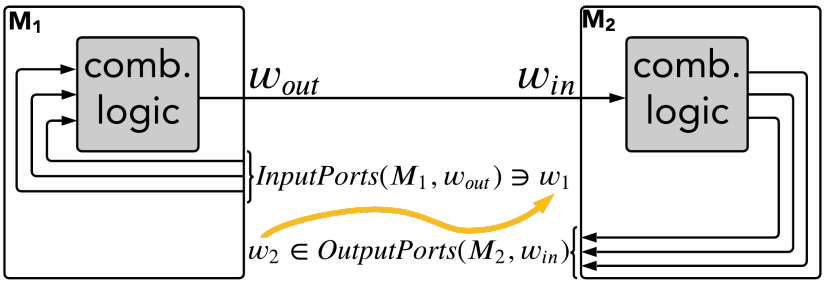


Figure 4.7: Illustration of the Wire Well-Connectedness definition. Given a circuit C, well-connectedness for a connection $(w_{\texttt{out}}, w_{\texttt{in}}) \in C.conns$ occurs when there does not exist an output port w_2 in $w_{\texttt{in}}$'s output-port-set that is transitively connected ($\rightsquigarrow_C$) to any wire w_1 in $w_{\texttt{out}}$'s input-port-set.

Figure 4.6b both show two connected modules with a **from-port** output wire connected to a **to-port** input wire, but in the former case it does not result in a combinational loop while in the latter it does. Note, however, that we still do not need to inspect the internals of any modules as long as we know the sorts of their interface wires.

We can distinguish between the examples in Figure 4.6a and Figure 4.6b by defining a safe class of connections to **from-port** sorts, called **safely from-port**:

Definition 2 (Safely From-Port Wires)**.** *A **from-port** output wire $w_{\texttt{out}}$ connected to a **to-port** input wire $w_{\texttt{in}}$ is called **safely from-port** with respect to $w_{\texttt{in}}$ if $w_{\texttt{out}}$ and $w_{\texttt{in}}$ are well-connected according to Definition 1.*

A **safely from-port** output wire combinationally depends on some module input wires (and hence its value is not valid until those inputs have propagated to the output wire later in the clock cycle) but still guarantees the absence of combinational loops with respect to certain connected wires. In Figure 4.6a, the dependent output wire $w_{1\texttt{out}}$ is **safely from-port** with respect to $w_{2\texttt{in}}$, and hence the overall circuit is well-connected since $w_{1\texttt{out}}$ is not connected to anything else. In contrast, in Figure 4.6b the **from-port** output wire $w_{1\texttt{out}}$ is *not* **safely from-port** and hence the overall circuit is *not* well-connected.

Determining whether a wire is **safely from-port** or not requires the complete circuit in order to compute the *TransitivelyAffects* relation. Figure 4.6c demonstrates this fact. We define a circuit composed of a set of modules such that all module interface wires are connected to be a **complete circuit**. A **well-connected circuit** is a complete circuit that has no combinational loops. This definition brings us to our final property:

Property 3 (Circuit Well-Connectedness)**.** *A complete circuit is well-connected if and only if all **from-port** output wires in the circuit are **safely from-port** with respect to the **to-port** input wires to which they are connected.*

Summary. The forward implication is that in a complete, well-connected circuit C, all **from-port** output wires are **safely from-port**. By definition, a well-connected circuit does not contain any combinational loops. If there exists some module M_1's **from-port** output wire $w_{\texttt{out}}$ that is *not* **safely from-port**, then by the definition of **safely from-port** (Definition 2) either:

1. Wire $w_{\texttt{out}}$ is not connected to any other wire. But this contradicts the fact that the circuit must be complete.

2. Wire $w_{\texttt{out}}$ is connected to wire $w_{\texttt{in}}$ of some module M_2 and there exist wires $w_1 \in \texttt{input-ports}(M_1, w_{\texttt{out}})$, $w_2 \in \texttt{output-ports}(M_2, w_{\texttt{in}})$ such that $w_2 \rightsquigarrow_C w_1$. By the definition of $\rightsquigarrow_C$ this means that there is a combinational loop in the circuit. But this contradicts that the circuit is well-connected.

Therefore by contradiction the forward implication holds. The reverse implication is that if all **from-port** output wires are **safely from-port**, then the complete circuit is well-connected. Since the circuit is complete, every input and output wire is connected to some output or input wire, respectively. For a given connection, if either the output wire is **from-sync** or the input wire is **to-sync** then they cannot be part of a combinational

loop. So the only case that we need to worry about is if the output wire $w_{\texttt{out}}$ is **from-port** and the input wire $w_{\texttt{in}}$ is **to-port**. Assuming that $w_{\texttt{out}}$ is **safely from-port**, this means that by Definition 2 it must be true that $w_{\texttt{out}}$ and $w_{\texttt{in}}$ are well-connected according to Definition 1. This property directly implies that these wires cannot be part of a combinational loop. Therefore the forward direction holds. □

4.3.5 Putting It All Together

Given the definitions and properties stated above, we can divide checking a circuit for well-connectedness into three stages:

- **Stage 1.** At the time each module is designed, automatically compute the sort of each input and output wire. Annotate each wire with its sort and, for a **from-port** or **to-port** wire, its input-port-set and output-port-set, respectively.
- **Stage 2.** When modules are connected during circuit design, any connections involving a **from-sync** or **to-sync** wire can be marked as safe immediately.
- **Stage 3.** Either periodically during circuit construction (useful when using interactive HDLs with a tight feedback loop) or only once when the circuit is completed: for each **from-port** output wire connected to a **to-port** input wire, check whether the output wire is **safely from-port** with respect to the input wire.

This process neatly encapsulates the necessary information about the module's internal details into its interface and allows for checking well-connectedness in the final circuit while treating each module as a opaque.

4.3.6 Comparison to BaseJump STL

We can relate the informal notions given by BaseJump STL (described in Section 4.2) to our more precise definitions given here and thereby pin down exactly where the BaseJump STL notions become problematic. BaseJump STL says that an endpoint is **demanding** if it needs the other endpoint's input signal ($valid_{\texttt{in}}$ for the consumer endpoint, $ready_{\texttt{in}}$ for the producer endpoint) before computing its own output signal ($ready_{\texttt{out}}$ for the consumer endpoint, $valid_{\texttt{out}}$ for the producer endpoint) and is **helpful** otherwise.

Using our definitions, we can formulate these notions precisely. We are given a module M with producer endpoint ($ready_{\texttt{in}}, valid_{\texttt{out}}, data_{\texttt{out}}$) and consumer endpoint ($ready_{\texttt{out}}, valid_{\texttt{in}}, data_{\texttt{in}}$). The producer endpoint is **helpful** iff $ready_{\texttt{in}} \notin \texttt{input-ports}(M, valid_{\texttt{out}})$, otherwise it is **demanding**. This says nothing about the presence or absence of M's other inputs in $\texttt{input-ports}(M, valid_{\texttt{out}})$, meaning $valid_{\texttt{out}}$ could be **from-port** and thus potentially cause a loop due to other module connections. The consumer endpoint is **helpful** iff $valid_{\texttt{in}} \notin \texttt{input-ports}(M, ready_{\texttt{out}})$, otherwise it is **demanding**; again, this does not preclude $ready_{\texttt{out}}$ from being **from-port**.

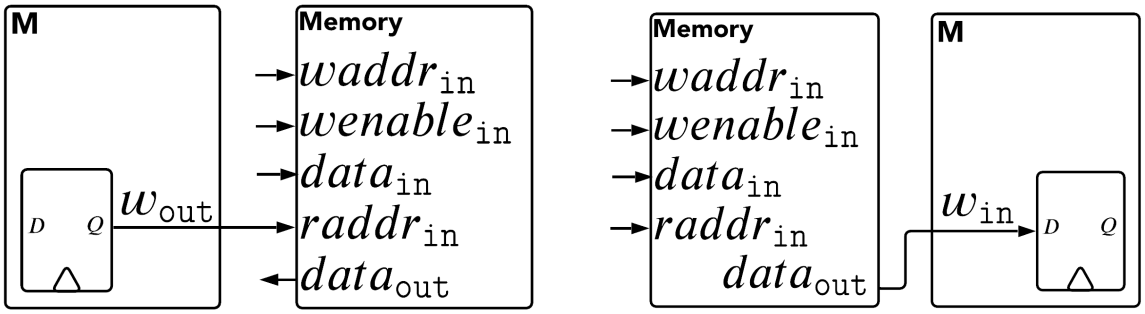
According to the BaseJump STL work, the only potentially problematic connection is between two **demanding** endpoints; all other types of connections are safe while **demanding-demanding** connections should be forbidden. However, according to our analysis above this is not a sufficient condition for correctness. It is possible (as demonstrated in Section 4.2) for a **helpful-helpful** connection to create a combinational loop; this is because the **helpful** and **demanding** endpoint classifications focus only on direct connections between two modules and do not consider the possibility of combinational loops via other modules not directly involved in the connection.

4.3.7 Extension to Synchronous Memory Reads

The basic set of domains described in Section 4.3.1 omits mention of memories. Memories are a special case in digital logic; their semantics partially depend upon whether they are synchronous or asynchronous. Synchronous memories are often preferable in order to be able to synthesize a design into efficient hardware, but using them imposes additional conditions on the design. For example, one class of synchronous memories requires that the read operations are able to start at the *beginning of the clock cycle*. What this often means is that the designer must make sure that the read address port, $raddr_{\texttt{in}}$, is fed directly from a register.

Take as an example the module-memory interconnection in Figure 4.8a. At first glance, this condition requires that any external module's output wire $w_{\texttt{out}}$ connected to $raddr_{\texttt{in}}$ be **from-sync**. However, this still doesn't meet the required conditions for synchronous memories; our definition of **from-sync** allows combinational logic to exist between the source register from which the **from-sync** data originates and its destination. In order for the data on $w_{\texttt{out}}$ to be available immediately at the beginning of the clock cycle, it must not go through any combinational logic at all (since all gates have propagation delay), and so we find that we must create a **from-sync** subsort, which we'll call **from-sync-direct**.

A **from-sync-direct** output wire $w_{\texttt{out}}$ is simply one where $\texttt{reachable}(M, w_{\texttt{out}}) = \emptyset$. By our definition of `reachable` in Section 4.3.2, this means that $w_{\texttt{out}}$ is connected only directly to registers, with no intermediate combinational logic. In Figure 4.4, wire $w_{1\texttt{out}}$ could thus be labelled **from-sync-direct** and qualify as being able to be connected to a synchronous memory's input wires. Its data is available at the start of the clock cycle because its signal doesn't need to propagate through any attached combinational logic. A **from-sync** wire that isn't **from-sync-direct** is said to be **from-sync-indirect**.



(a) The read address line $raddr_{\texttt{in}}$ of certain synchronous memories must be connected to **from-sync-direct** wires like $w_{\texttt{out}}$.

(b) Other synchronous memories require $data_{\texttt{out}}$ be connected to **to-sync-direct** input wires like $w_{\texttt{in}}$ feeding directly to a register.

Figure 4.8: Wire sorts for synchronous memories.

There are other forms of memories where synchronous requirements are placed on certain *outputs*, rather than inputs. In these memories, the designer must ensure that the $data_{\texttt{out}}$ wire is fed directly into a register, as shown in Figure 4.8b. This naturally leads to an input subsort for describing such conditions, which we call **to-sync-direct**; a **to-sync-direct** input wire $w_{\texttt{in}}$ is one where $\texttt{reachable}(M, w_{\texttt{in}}) = \emptyset$. A **to-sync** wire that isn't **to-sync-direct** is said to be **to-sync-indirect**.

By providing these additional sorts, designers can communicate the interface requirements of modules using synchronous memories, making libraries of hardware components more accessible and easier to use. Furthermore, the particular aspect of timing information provided by these new subsorts can be used as part of clock cycle determination and place-and-route post-synthesis of standard EDA workflows. Thus, this sort taxonomy, now at for inputs: **to-sync** (with its subsorts **to-sync-direct** and **to-sync-indirect**) and **to-port**; and for outputs: **from-sync** (with its subsorts **from-sync-direct** and **from-sync-indirect**) and **from-port**, has a wide range of applications and can be potentially expanded even further.

4.4 Implementation of Modular Well-Connectedness Checks

We augmented the PyRTL HDL [110] to implement lightweight annotations and design-time checks according to the formal properties that we have described of the original four wire sorts (**to-sync**, **to-port**, **from-sync**, and **from-port**). PyRTL does not natively support a module abstraction, so we first modified the language by adding a `Module` class that isolates a modular design and exposes an interface consisting of input and output wires.[3]

Our formalism made two simplifying assumptions. First, it assumed that all logic is contained inside modules. For developer convenience, we eased this restriction to allow for arbitrary logic to exist between modules. We tweaked the *TransitivelyAffects* relationship ($\rightsquigarrow_C$) to account for combinational paths through this extra-modular logic. Second, it treated all wires as one bit in width. At the HDL level, it is much more convenient to group related one-bit wires, especially input and output ports, into single n-bit wire vectors. For native PyRTL designs (but not BLIF import), the output-port-set or input-port-set of each port wire vector becomes the union of the output-port-set or input-port-set of its constituent wires; thus we are overly conservative because single-bit dependencies are not tracked, but maintain soundness by continuing to be able to detect all combinational loops.

The well-connectedness implementation itself consists of (1) a sort inferencer that automatically computes the sorts of a module's input and output wires at module design time; (2) lightweight syntactic annotations that allow a designer to (optionally) specify what they believe the sorts should actually be; and (3) a whole-circuit checker that automatically triggers when needed to verify that a circuit composed of multiple modules is well-connected.

[3]Our PyRTL modifications and the complete implementation of our tool are available at https://github.com/pllab/PyRTL/tree/pldi-2021.

The computed sorts are then checked against any existing designer annotations to ensure that the computed sorts match the designer's expectations; any unascribed ports are labeled with their computed sorts. We require sort ascriptions, where the output-port-set or input-port-set are fully specified by the user, for all the ports of opaque modules, since there is no internal logic to use for calculating these sorts. Once a module has its wire sorts, these sorts make it quicker to determine intermodular connections because they facilitate re-use: every instantiation of the same module in the larger design reuses the same wire sort information.

An interesting question during circuit design, as modules are being composed, is *when* exactly to check well-connectedness. We would like to highlight problems as early as possible instead of waiting until the entire circuit is complete. However, we also want to minimize the cost of constantly checking well-connectedness during the design process. As such, our tool can either check for well-connectedness after all modules have been connected or instead whenever a newly formed connection between two modules meets the following condition: the connection's forward combinational reachability set includes a **to-port** input, and its backward combinational reachability set contains a **from-port** output. This condition is cheap to track by saving and updating information about each wire's reachability as wires are added to the design, and it guarantees that (1) a check is never done unless a problem could potentially be found; and (2) an actual problem is found as soon as possible.

4.5 Evaluation

We evaluate our tool in five parts: (1) an application to a number of SystemVerilog modules provided by the BaseJump STL; (2) an application to several components from the OpenPiton Design Benchmark; (3) a case study applying the tool to the design of a

multithreaded RISC-V CPU; (4) a comparison of our tool to standard cycle detection during synthesis; and (5) a discussion of the scalability and asymptotic complexity of the tool.

4.5.1 BaseJump STL Modules

We begin with an evaluation of a number of the SystemVerilog modules provided by the industrial-strength BaseJump STL library. This evaluation serves as a baseline sanity check allowing us to verify that we can successfully assign the correct sorts to module interfaces.

We ran our annotation framework successfully on 144 unique modules from the BaseJump STL as found in the BSG Micro Designs repository [141], a repository containing a large number of BaseJump STL modules parameterized and converted into Verilog. Each module was instantiated one to four times to test various combinations of its parameters (e.g. data bit width, address width, queue size, etc.), so that we analyzed 533 modules in total. Since our technique is currently only applicable to synchronous, single-clock designs, we were unable to analyze 5 modules that relied on asynchronous or multi-clock constructs.

We converted each top-level module and their submodules into the flattened BLIF format[142] via Yosys version 0.9 and imported the result into PyRTL. The average size of each module in BLIF was 1.7 MB, with an average number of primitive gates of 19,981, an average number of inputs and outputs per module of 6, and an average time for inferring all of the interface sorts for each module of 361 milliseconds. We ran all of these experiments using PyRTL on a computer with a 1.9 GHz Intel Xeon E5-2420 processor and 32 GB 1333 MT/s DDR3 memory.

A representative subset of these BaseJump STL modules is shown in Table 4.1:

Table 4.1: Wire sorts of module ports for a subset of BaseJump STL; TS = **to-sync**, TP = **to-port**, FS = **from-sync**, FP = **from-port**. Every module also has a `reset` input wire whose sort is **to-sync**. The time listed is cumulative time to annotate all the wire sorts.

Module	Prim. Gates	Time (s)	Inputs			Outputs		
			Wire Name	Sort	Output Port Set	Wire Name	Sort	Input Port Set
First-In First-Out Queue	148,272	2.669	`data_i`	TS	∅	`data_o`	FS	∅
			`yumi_i`	TS	∅	`ready_o`	FS	∅
			`v_i`	TS	∅	`v_o`	FS	∅
Parallel-In Serial-Out Shift Reg.	53,637	0.606	`valid_i`	TS	∅	`valid_o`	FS	∅
			`data_i`	TS	∅	`data_o`	FS	∅
			`yumi_i`	TP	{`ready_o`}	`ready_o`	FP	{`yumi_i`}
Serial-In Parallel-Out SR	1,617,698	18.752	`yumi_cnt_i`	TS	∅	`ready_o`	FS	∅
			`valid_i`	TP	{`valid_o`}	`valid_o`	FP	{`valid_i`}
			`data_i`	TP	{`data_o`}	`data_o`	FP	{`data_i`}
Cache DMA	4,440	0.051	`data_mem_data_i`	TS	∅	`data_mem_data_o`	FS	∅
			`dma_data_i`	TS	∅	`dma_data_o`	FS	∅
			`dma_data_v_i`	TS	∅	`dma_data_v_o`	FS	∅
			`dma_data_yumi_i`	TS	∅	`dma_data_ready_o`	FS	∅
			`dma_pkt_yumi_i`	TP	{`done_o`}	`dma_pkt_v_o`	FP	{`dma_cmd_i`}
			`dma_way_i`	TP	{`data_mem_w_mask_o`}	`data_mem_addr_o`	FP	{`dma_addr_i`}
			`dma_addr_i`	TP	{`data_mem_addr_o`, `dma_pkt_o`}	`data_mem_v_o`	FP	{`dma_cmd_i`}
						`data_mem_w_mask_o`	FP	{`dma_way_i`}
			`dma_cmd_i`	TP	{`done_o`, `dma_pkt_o`, `dma_pkt_v_o`, `data_mem_v_o`}	`dma_pkt_o`	FP	{`dma_addr_i`, `dma_cmd_i`}
						`done_o`	FP	{`dma_cmd_i`, `dma_pkt_yumi_i`}
						`data_mem_w_o`	FS	∅
						`dma_evict_o`	FS	∅
						`snoop_word_o`	FS	∅

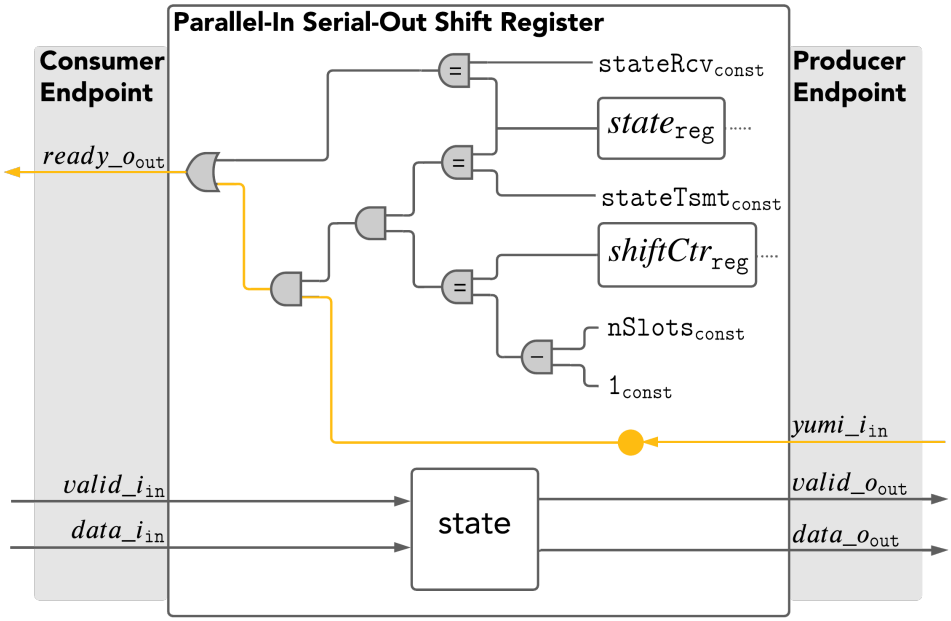


Figure 4.9: Tracing a path through the parallel-in serial-out (PISO) shift register shows the dependency between `yumi_i` and $ready\_o_{\texttt{out}}$, making connections involving either of their respective endpoints potentially unsafe.

a first-in first-out queue, a parallel-in serial-out shift register, a serial-in parallel-out shift register, and cache DMA. Information on the sizes, wire sort annotations, and annotation time for all 533 modules is found in the supplementary material.

We highlight in particular the parallel-in serial-out (PISO) shift register as an interesting case (see Figure 4.9). Three of its four inputs are **to-sync**, while `yumi_i` is **to-port** (specifically, its output-port-set contains the output `ready_o`). We can see the details by looking at the logic for output `ready_o`:

$$\begin{aligned} ready\_o_{\texttt{out}} ::= (state_{\texttt{reg}} = \texttt{stateRcv}) \vee \\ ((state_{\texttt{reg}} = \texttt{stateTsmt}) \wedge \\ (shiftCtr_{\texttt{reg}} = \texttt{nSlots} - 1) \wedge yumi\_i_{\texttt{in}}) \end{aligned}$$

According to the BaseJump STL paper, the consumer endpoint of this module

(to which *ready_o*$_{\texttt{out}}$ belongs) is **helpful** because *ready_o*$_{\texttt{out}}$ does not combinationally depend on *valid_i*$_{\texttt{in}}$ (an input wire in the consumer endpoint). Thus, according to them, this module can safely be connected to any other module. However, our own analysis more precisely shows that while it is true that *ready_o*$_{\texttt{out}}$ doesn't depend on other consumer endpoint wires, it does require other module input (in particular *yumi_i*$_{\texttt{in}}$, which is part of the producer endpoint). This fact means that the PISO module connections may or may not be safe depending on the sorts of the interfaces to which it is directly or transitively connected.

Notably, after personal correspondence in which we reported the issue, the authors of the BaseJump STL PISO module updated it so that, according to our terms, *yumi_i*$_{\texttt{in}}$ is now **to-sync** and *ready_o*$_{\texttt{out}}$ is now **from-sync**.[4] This shows that designers care about the precise behavior of these interfaces and that an analysis that annotates wire sorts and verifies their interconnections is a useful thing to have.

4.5.2 OpenPiton Modules

We also used our analysis on a completely separate body of work: the OpenPiton Design Benchmark (OPDB), based on the OpenPiton manycore research platform [143, 144]. OPDB is interesting because it provides modules of a variety of scales and with different configuration options pre-generated per module. We were also interested in these OpenPiton designs due to anecdotes from the developers of OpenPiton related to issues they experienced with compositionality. In one instance, developing a like-for-like replacement for an existing component led to combinational loops that went undetected until final integration and synthesis due to minor mismatches in interfaces and test configurations. In another, a hardware generator produced combinational

[4]See https://github.com/bespoke-silicon-group/basejump_stl/commit/67830f05ffce1333c7b790600530da0681af74fe

Table 4.2: Size (in primitive gates), wire sort inference time (in seconds), and number of IO ports of 17 OPDB modules.

Module	Prim. Gates	Time (s)	Ports
dynamic_node	29,918	0.759	35
fpu	168,525	1.456	16
ifu_esl	15,602	1.362	40
ifu_esl_counter	310	0.001	5
ifu_esl_fsm	2,299	0.040	34
ifu_esl_htsm	524	0.012	30
ifu_esl_lfsr	213	0.001	6
ifu_esl_rtsm	170	0.005	24
ifu_esl_shiftreg	208	0.001	4
ifu_esl_stsm	267	0.016	26
l2	1,088,384	15.128	16
l15	1,518,073	30.176	71
pico	36,479	0.245	24
sparc_ffu	104,966	0.723	77
sparc_mul	20,702	0.260	7
space_exu	320,397	10.203	132
sparc_tlu	650,364	8.753	214

loops for only particular values of a parameter designed to change the size of a module, and those loops would require the composition of as many as seven modules to come into existence.

To process the OPDB designs, we followed the same Yosys Verilog-to-BLIF synthesis step as with the BaseJump STL designs, excluding some with asynchronous or multi-clock constructs. Our selected OPDB designs include a floating-point unit, network-on-chip router, and two caches, among others. Table 4.2 shows the OPDB designs we selected, their sizes in number of primitive gates, the time taken to infer the wire sorts of the design, and the number of input/output ports. Of the 17 designs we processed, the average number of gates was 232,788, while the smallest (`ifu_esl_rtsm`) had just 170 gates and the largest (`l15`) had more than 1.5 million gates. The designs had an average of 44 ports with the fewest ports (`ifu_esl_stsm`) being just 4, while the design

Table 4.3: A comparison of cycle detection during synthesis (Yosys) versus our tool using wire sorts on large OPDB designs. Each unique module type only needs to be analyzed once; additional (non-unique) instantiations reuse the calculated sorts. The number of primitive gate differs from Table 4.2 because these are unflattened, and thus unoptimized, designs.

Module	Prim. gates (hier. BLIF)	Cycle det. time (s)		Speedup	Sort infer. time (s)	Submodules	
		Yosys	Ours			Total	Unique
`fpu`	189,452	46.42	3.11	14.92x	0.845	3530	118
`sparc_ffu`	105,688	11.30	1.00	11.30x	0.397	208	51
`sparc_exu`	331,452	22.81	8.65	2.63x	0.989	737	92
`sparc_tlu`	761,538	108.54	5.82	18.64x	0.813	777	128
`l2`	1,176,219	361.04	10.64	33.93x	13.80	157	45
`l15`	1,549,475	643.45	20.81	30.92x	7.86	68	26

with the most (`sparc_tlu`) had 214. The larger scale of these designs also skews to a longer average wire sort inference time, at 4.067 seconds, with a minimum of 0.001s and a maximum of 30.176s. We describe the asymptotic complexity of this operation in Section 4.5.5.

4.5.3 RISC-V CPU

For a more holistic case study, we implemented a multithreaded single-cycle RISC-V [145, 146] CPU (RV32I base integer instruction set) in PyRTL. The CPU consists of 11 modules in total; the total number of primitive gates for the entire design, configured for five threads and five pipeline stages, is 229,011 gates. Our tool spent an average of 13.5 milliseconds on each module inferring its interface sorts; it took on average 162.7 milliseconds to determine all of the sorts, with a lower bound of 148.9 milliseconds and an upper bound of 194.2 milliseconds, at a rate of 298 nanoseconds per primitive gate. Once all the modules were connected, it was able to correctly check all the inter-module connections in an average of 67.1 milliseconds, with a lower bound of 62.5 milliseconds and an upper bound of 77.3 milliseconds. Information on the sizes, wire

sort annotations, and annotation times for the RISC-V submodules can be found in the supplementary material.

4.5.4 Comparison to Loop Detection During Synthesis

In our final analysis, we compared the efficiency of doing cycle detection at the HDL level via wire sorts versus at the netlist level during synthesis. Finding broken designs in the wild is difficult because most designers don't publish broken designs. So instead, we altered the OPDB Verilog designs slightly by introducing multi-module loops, importing the largest of them in their hierarchical BLIF format into PyRTL where our intermodular analysis is done. We then timed how long (1) Yosys takes to find the cycle during synthesis, (2) our tool takes to determine all interface sorts, and (3) our tool takes to check for intermodular loops given these sorts. We found that Yosys took longer to synthesize and find loops than our tool. It was also not straightforward to get Yosys to tell us these loops exist: depending on the options given, it would optimize them out or convert them to something else entirely without warning. Our results are found Table 4.3.

In actual use, we expect the user to write their designs in a modular fashion in a high-level HDL that can be analysed directly to begin with and to provide wire sort ascriptions if wanted. This experiment favored synthesis over our technique because it relied on importing a BLIF file, which has a few downsides. The Verilog-to-BLIF process converts N-bit ports into N 1-bit ports, meaning the number of ports increased by a factor equal to the average port bitwidth. The conversion also creates a module instance for each unique set of parameters used; since BLIF doesn't offer information that a module instantiation differs from another only by some parameter, those count as additional unique modules whose sorts must be calculated.

Despite this, annotating all modules with their I/O sorts was relatively quick, and detecting loops via intermodular connections using these sorts was **2.6–33.9x** faster than trying to find them during synthesis at the pure netlist level. We expect that by analyzing the design in its original form (e.g. Verilog or PyRTL), where the wires stay bundled together and parameterized module instances can be abstracted over, this speedup would increase significantly. This is exemplified by our RISC-V case study mentioned in Section 4.5.3, which was written entirely in PyRTL.

4.5.5 Complexity and Scalability

We describe the asymptotic complexity of the two analysis phases in order to demonstrate their scalability.

Module Wire Sort Inference

Sort inference takes place once per module definition. For a given module $M = (\overrightarrow{w_{\texttt{in}}}, \overrightarrow{w_{\texttt{out}}}, \overrightarrow{net})$, we must compute the transitive closure of combinationally reachable output wires for each $w_{\texttt{in}} \in \overrightarrow{w_{\texttt{in}}}$. Thus the total complexity of computing the sorts for all input wire sorts is $O(|\ \overrightarrow{w_{\texttt{in}}}\ | \cdot |\ edges\ |)$, where $edges = \bigcup_{net \in M.nets} \{\overrightarrow{w_\sigma} \mid (\overrightarrow{w_\sigma}, w_\sigma, op) = net\} \cup M.outputs$. Since the **to-port**/**from-port** relationship is symmetric, the wire sorts for outputs can be computed using the previously computed input wire sorts without traversing the module's internal wires again.

Circuit Well-Connectedness

The phase to check circuit well-connectedness uses the wire sorts computed by the module wire sort inference, and it operates only on the module interfaces without caring how large or complex any individual module might be. It only needs to be run

Table 4.4: The number of annotations per sort. TS = To-Sync, TP = To-Port, FS = From-Sync, FP = From-Port.

Source	Modules	Inputs		Outputs	
		TS	TP	FS	FP
BaseJump STL	144	233	211	178	197
OpenPiton DB	17	347	113	245	56
RISC-V	11	14	33	3	33
Total	**172**	**594**	**357**	**426**	**286**

once, after the circuit is complete. The algorithm iterates over each pair of inter-module input-output connections checking them against the *TransitivelyAffects* relationship ($\rightsquigarrow_C$).

Since each input port is connected to only one incoming output wire from another module, the number of connections is equal to the total number of input ports across the circuit. Given a circuit C and arbitrary wires $w_{\texttt{out}}$, $w_{\texttt{in}}$ in the circuit, the worst-case scenario is when the path from $w_{\texttt{out}}$ to $w_{\texttt{in}}$ traces through every inter-module connection before finally reaching the combinational loop. Thus, the *TransitivelyAffects* computation has a worst-case complexity of $O(|\ C.conns\ |)$. Since we do this check for *each* connection pair in $C.conns$, the total worst-case complexity is $O(|\ C.conns\ |^2)$.

Distribution of Wire Sorts

We found that sort annotations that our tool assigned to the module ports were widely distributed, as shown in Table 4.4. Across all 172 modules, **to-sync** inputs make up 62.5% of module inputs, compared to 37.5% for **to-port** inputs. **from-sync** outputs make up 59.8% of module outputs, compared to 40.2% for **from-port** outputs.

The foremost goal of this work was to reduce the number of "late surprises" in the design process. In these designs, 38.7% of the ports raise the possibility of a "late surprise" loops because they are **to-port** or **from-port**. For the remaining 61.3%, our

technique has the additional advantage of making the checking process faster, by eliminating individual wires, or in the case of modules with entirely **to-sync**/**from-sync** IO, entire modules that need to be included in the cycle detection analysis.

4.6 Conclusion

We have presented an approach to creating hardware modules in isolation while tracking enough information to make checking their well-connectedness in an entire design feasible and user-friendly. BaseJump STL's informal approach of commenting ready-valid endpoints as **helpful** or **demanding** is a step in the right direction at classifying modules with information to help in connecting them at circuit design time in a plug-and-play fashion, but as we show it falls short in being able to prevent combinational loops.

Our solution is to provide wire-level information via a taxonomy of sorts: **to-sync**, **to-port**, **from-sync**, and **from-port**, allowing for modules to be written in isolation effectively and still safely connected without knowing their internals. We implemented our approach in a hardware description language and analyzed real-world designs (BaseJump STL and the OpenPiton Design Benchmark) as well as a multithreaded RISC-V CPU implementation, showing that our approach is feasible, effective, and efficient.

Chapter 5

PyLSE: A Pulse-Transfer Level Language for Superconductor Electronics

5.1 Introduction

Superconductor electronics (SCE) are a promising emerging technology for the post-Moore era due to their low power dissipation, energy-efficient interconnects, and sub-attojoule ultra-high-speed switching [147]. However, the physical properties that make SCE so promising also make them difficult to design for. In particular, SCE use a *pulse-based*, rather than a voltage level-based, information encoding. This, along with the *stateful* nature of superconducting cells [148] and the lack of a uniformly agreed-upon efficient translation from design to implementation, makes it necessary to develop unique logic gates and design rules [149, 150, 151]. As a result, the superconducting realm lacks tools and workflows for the rapid prototyping and testing of microarchitectures and SCE-based applications (see Figure 5.1).

The primary question we seek to answer in this paper is: *what is a suitable abstraction for precisely defining the functional and timing behavior of SCE designs*? Our solution is

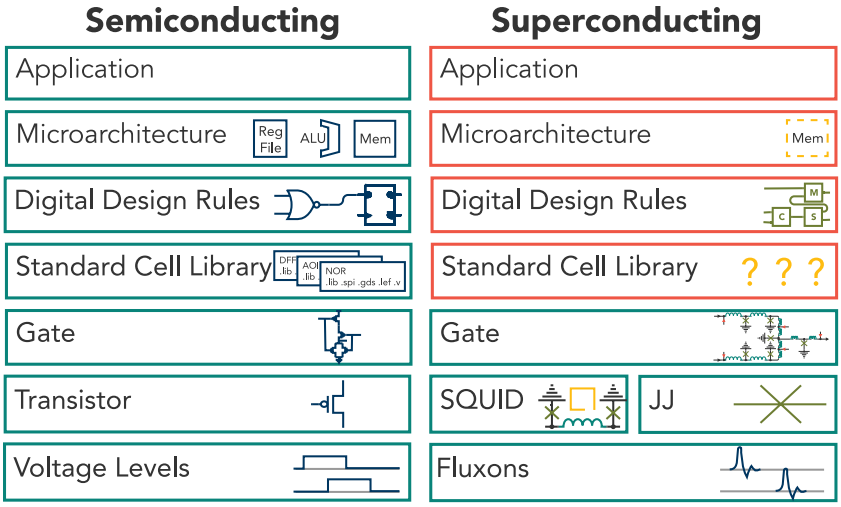


Figure 5.1: Unlike the semiconducting realm, superconducting lacks standard cell libraries and standardized digital design rules, making it difficult to rapidly build and test microarchitectures and larger applications.

to completely depart from existing hardware description languages (HDLs), taking a bottom-up approach to build a new Python [152] embedded domain-specific language (DSL), called PyLSE (**Py**thon **L**anguage for **S**uperconductor **E**lectronics). We argue that PyLSE is well tailored to the unique needs of SCE, making it easier to create and compose cells into correct, scalable systems.

Inspired by the theory of automata [153], we propose a custom finite state machine (FSM) abstraction, which we call a PyLSE Machine, to precisely describe the functional and timing behavior of SCE cells. This FSM abstraction eliminates the need for complex and error-prone conditional assignments, commonly found in state-of-the-art approaches [154], and forms the core our PyLSE language. Through this abstraction, we also develop a new link between SCE and the theory of Timed Automata [155], which enables the integration of PyLSE with modern formal verification tools like the UPPAAL model checker [156]. Overall, the main contributions of this paper are:

- We create the PyLSE Machine, a language abstraction for the formalization of the

functional and timing semantics of pulse-based circuits (Section 5.3).

- We create PyLSE, a lightweight transition system-based Python DSL for the rapid prototyping of pulse processing systems, modeled as networks of PyLSE Machines (Section 5.4).
- We automate the translation of PyLSE Machines to Timed Automata (Section 5.4).
- We build a multi-level framework for the simulation and analysis of PyLSE Machine systems, which also allows for the integration of abstract behavioral software models, fostering agile development (Section 5.4).
- We evaluate PyLSE's capabilities through a series of comparisons with state-of-the-art approaches, dynamic checks of SCE designs with stochastic timing behaviors, and formal verification using UPPAAL (Section 5.5).

5.2 Defining Computation on Pulses

5.2.1 Functional Behavior

Complementary metal-oxide-semiconductor (CMOS) is the dominant technology of today's computing landscape. The majority of the tools used to design and fabricate modern hardware assume an underlying digital logic basically corresponding to the functionality of basic gatesand n- and p-type field-effect transistors[157]. Because the CMOS fabrication process is so well-understood, engineers encapsulate the functional, timing, and physical layout characteristics of these gates into *standard cell libraries* [158]. In turn, digital designers and computer architects use abstractions of these cells to create consistent digital design rules that underpin the semantics of hardware description languages [106, 159] and which are used to create microarchitectures and larger

(a) CMOS: Steady voltage levels encode information; thus, wires are considered stateful and the gates can be stateless.

(b) SFQ: Transient pulses encode information; thus, wires are considered stateless and the gates should be stateful.

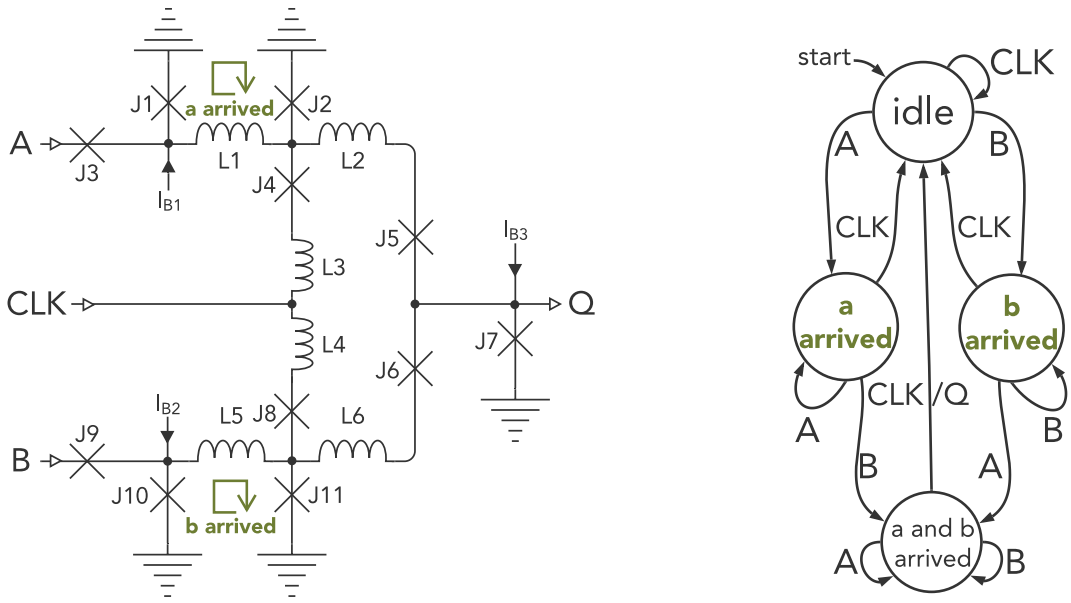
Figure 5.2: Comparing information in CMOS and SFQ.

applications (see the left side of Figure 5.1).

Superconductor electronics, on the other hand, exploit the unique properties of superconductivity [160, 161] to perform computation through the carefully orchestrated consumption and emission of tiny pulses of magnetic flux. In this realm, *Josephson junctions* (JJs), rather than transistors, serve as the basic switching device. While the quantum nature of such flux exchange is central to the device operation, the computation performed is strictly classical. Information is moved from logic element to logic element in the same "feed-forward" way as traditional digital logic. However, the use of picosecond-scale *pulses* of single flux quanta (SFQ), rather than sustained voltage levels, to carry information between logic elements has myriad downstream effects. In CMOS, information is "held" in the wires connecting gates and registers together; that is, the voltage level corresponding to a logical 0 or 1 persists as long as the input remains unchanged (see Figure 5.2a). In SFQ [162], information is encoded via short-lived pulses and does *not* persist on the wires; instead, cells[1] must be designed to "remember" that a particular input has arrived (see Figure 5.2b).

The SCE community has traditionally relied on low-level analog models for the design and analysis of basic SCE cells. Figure 5.3a is one such example, showing the

[1]We use "cell" and "gate" interchangeably throughout.



(a) Schematic (b) Mealy machine

Figure 5.3: Schematic and Mealy machine description of the Synchronous And Element. Labels `a arrived` and `b arrived` in the schematic are locations of superconducting loops holding state and roughly correspond to states in the Mealy machine.

low-level circuit schematic of the Synchronous And Element[2]. The mechanism these SCE cells use to "remember" input arrival by storing flux quantum is the inferferometer, also known as *superconducting quantum interference devices* (*SQUIDs*) [163]; labels a arrived and b arrived indicate the location of these SQUIDS. At a high level, this cell functions as follows. The arrival of a pulse on `A` causes flux to be stored in the SQUID labelled `a arrived`; similarly, a pulse on `B` stores flux in SQUID `b arrived`). When a pulse arrives on `CLK`, the flux in either SQUID is read out to another loop involving `J7`; if both `a arrived` and `b arrived` have been set, the combined flux causes `J7` to switch producing an output on output `Q`. The flux in the SQUIDs has been expended, causing the cell to return back to its initial state and receive pulses anew.

With the burgeoning interest of digital designers in SCE, the area is leaving the

[2]While the details of its construction using inductances and bias currents is beyond the scope of this paper, we include it show the physical manifestation of state being held in each cell, which informs our discussion of automata.

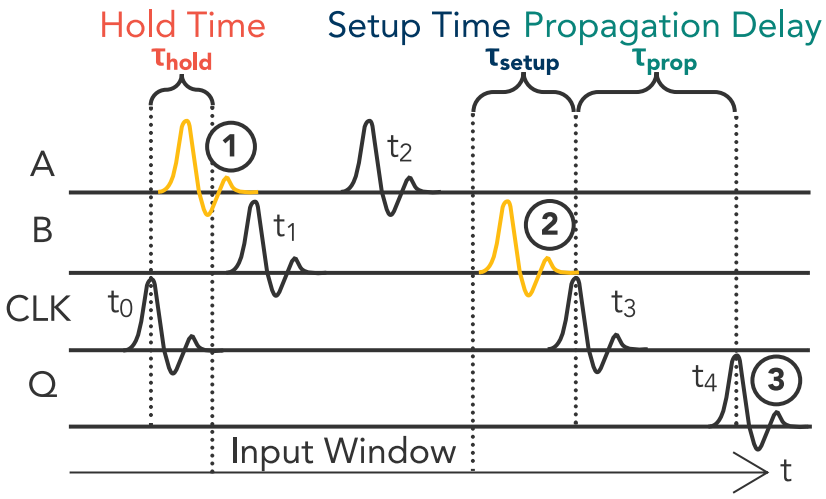


Figure 5.4: Waveform for the Synchronous And Element showing the timing constraints that must be met for correct operation. Pulses arriving during the hold time ① or setup time ② are erroneous. Assuming their absence, a pulse is produced some propagation delay ③ after a clock pulse.

confines of domain experts, increasing the need for new abstractions more suitable for the scalable design and analysis of SCE systems. One abstraction commonly used to explain the stateful behavior of SCE cells is the Mealy machine [164]. which are deterministic finite state transducers mapping state-input symbol tuples to new states and output symbols [165]. This model has been used extensively to model SCE cells [166, 167, 150, 151, 168, 162] like the Synchronous And Element in Figure 5.3, allowing digital designers to hide the low-level circuit details of Figure 5.3a in favor of the simpler, easier-to-use functional model of Figure 5.3b.

5.2.2 Timing Behavior

The schematic shown in Figure 5.3a comes with particular timing behavior as well as constraints on said behavior, which we visualize using the waveform in Figure 5.4. One such behavior is *propagation delay*, defined as the time it takes after a particular

pulse for an output to appear (see moment ③). Two such constraints are *setup time* and *hold time* [169, 170], defined as the intervals before and after the clock, respectively, where no pulses are expected to arrive. The combination of these latter two times help define an "input window" for legal inputs; pulses arriving outside this window (such as at moments ① or ②) run the risk of getting lost or dropped, or causing the cell to enter a metastable state.

Because Mealy machines lack an explicit notion of time, they fall short when constraints on the relative arrival times of inputs must be part of the functional description. These and other timing restrictions need to be carefully thought through and should be captured as early in the design process as possible. The time it takes for pulses to propagate through a system is a *first class concern* for not only analog circuit designers, but digital designers and architects as well. A good language abstraction must therefore be centered around a notion of time and provide mechanisms for (1) easily defining timing constraints and (2) verifying the absence of violations in the system.

5.3 A Language Abstraction for Superconductor Electronics

5.3.1 Overview of the PyLSE Machine

There are four key pieces of information that must be captured in a new SCE abstraction:

1. The time it takes to transition between states
2. The priority order that a given input should be handled if more than one arrives simultaneously

3. How long it takes for an output to appear once fired

4. Constraints on when it is legal to receive inputs

We believe that the Mealy machine should be the base of this new abstraction, but with the timing deficiencies covered in Section 5.2 resolved by augmenting the machine's edges. These augmented edges are composed of three parts: the **Trigger** (in turn composed of an input, priority, and transition time), the **Firing Outputs** (associating each output with its firing delay), and **Past Constraints** (for specifying conditions on a cell's past history of inputs); the details of each are found in Figure 5.5. We require that the machine be *fully-specified* (i.e. that for all states, there are edges labelled for all inputs) and call a machine composed of states and these augmented edges a *PyLSE Machine*.

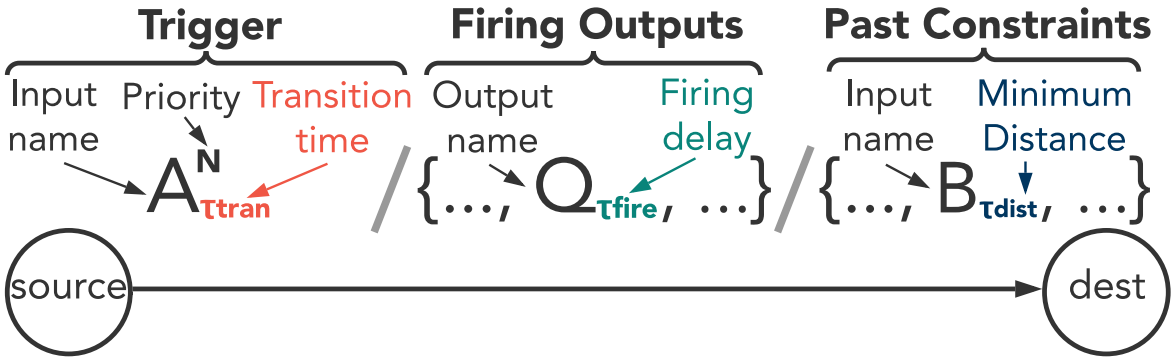


Figure 5.5: Anatomy of a PyLSE Machine transition. The arrival of an input pulse on wire `A` **Trigger**s the transition from the *source* to *dest* state. This transition has priority N over other simultaneously-triggered transitions originating from *source* and takes $\tau_{\texttt{tran}}$ time to complete; during this period, receiving any inputs is illegal. A pulse for each output `Q` in the **Firing Outputs** set appears on their associated output wire some $\tau_{\texttt{fire}}$ time units later. Finally, according to the **Past Constraints**, if it's been less than $\tau_{\texttt{dist}}$ since the last time an input `B` was received during a previous transition, it is an error. $\texttt{A}^{N}_{\tau_{\texttt{tran}}}$ is shorthand for $\texttt{A}^{N}_{\tau_{\texttt{tran}}}/\varnothing/\varnothing$.

To show how the proposed extension transforms a plain Mealy machine, we will focus on the Synchronous And Element. Figure 5.6 shows the PyLSE Machine for the

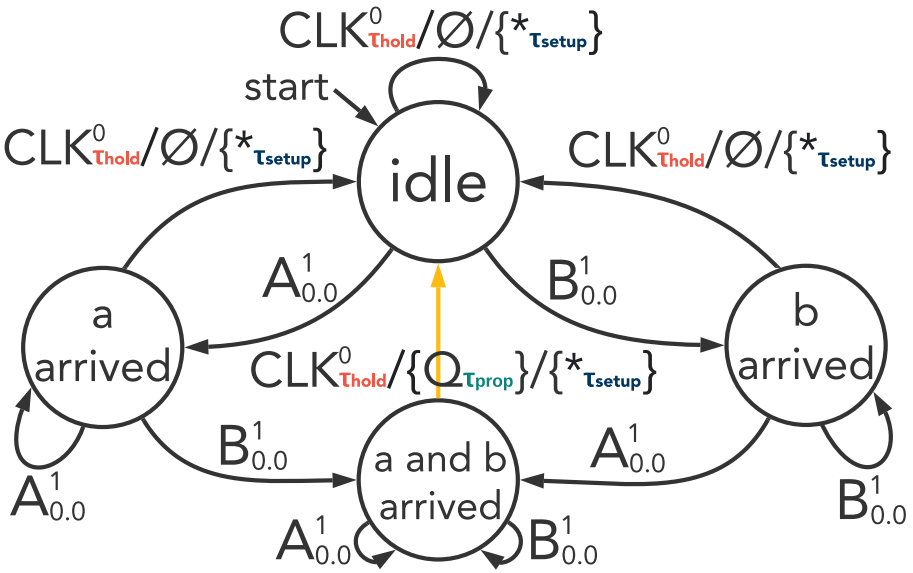


Figure 5.6: PyLSE Machine for the Synchronous And Element. Using transition time, we can model the hold time $\tau_{\texttt{hold}}$ and using the past constraints, we can model the $\tau_{\texttt{setup}}$ time. Firing delay directly models the propagation delay $\tau_{\texttt{prop}}$

Synchronous And Element, expanded from the original Mealy machine representation in Figure 5.3b. We'll dissect the edge (highlighted in gold) moving from state `a and b arrived` to `idle` ($\texttt{CLK}^0_{\tau_{\texttt{hold}}}/\{Q_{\tau_{\texttt{prop}}}\}/\{*_{\tau_{\texttt{setup}}}\}$) and show how we can use particularly subtle and useful features of Figure 5.5 to represent the setup and hold time constraints and propagation delay of this particular cell.

Transition Time The **Trigger** portion shows that this transition occurs when `CLK` is received and that it takes $\tau_{\texttt{hold}}$ time units to complete. We make it illegal to receive other inputs during a transitionary period, so that an edge's transition time can be used directly to model the *hold time* constraint of Figure 5.4, by setting $\tau_{\texttt{tran}} := \tau_{\texttt{hold}}$.

Priorities: The example edge has the highest priority (with a priority of `0` shown in the **Trigger**), meaning that if the machine received `A`, `B`, and `CLK` simultaneously, it would handle the transition associated with `CLK` first, transition to `idle`, and then make an

arbitrary choice between `A` and `B`, since both have the same labeled priority of 1 leaving `idle`. While it is practically impossible to arrange for SFQ pulses to purposefully arrive "simultaneously," it is not uncommon to consider *idealized* models of gates which have a delay of either zero, or some small integer. Priorities let the designer identify and explicitly handle cases of simultaneous arrival in a deterministic manner when desired.

Multi-Output: We require a *set* of outputs, rather than a single one, in order to accurately model SFQ cells with more than one output. In the **Firing Outputs** portion of the example edge being discussed, the singleton set $\{Q_{\tau_{\texttt{prop}}}\}$ indicates that a single output should fire and take $\tau_{\texttt{prop}}$ time units to appear; thus, we can model the cell's *propagation delay* by an edge's firing delay, setting $\tau_{\texttt{fire}} \coloneqq \tau_{\texttt{prop}}$.

Constraints on the Past: The **Past Constraints** portion says that when a `CLK` pulse arrives, it is an error to have received a pulse on any input (indicated by the * in this example) within the last $\tau_{\texttt{setup}}$ time units. We note that the hold time constraint (represented as transition time) could also have instead been placed in the past constraints section of the edges leaving idle, i.e. $\texttt{A}^{1}_{0.0}$ leaving `idle` could have instead been written as $\texttt{A}^{1}_{0.0}/\varnothing/\{\texttt{CLK}_{\tau_{\texttt{hold}}}\}$ and the transition times involving $\tau_{\texttt{hold}}$ replaced with 0.0; however, we feel that using a transition time, which imposes a future-looking constraint on what inputs can't be received, sometimes more easily reflects the mental model of the digital designer, and so offer both forms. Thus, we can model the *setup time* constraint shown in Figure 5.4 by setting $\tau_{\texttt{dist}} \coloneqq \tau_{\texttt{setup}}$ on this and all other edges whose triggering input is `CLK`.

5.3.2 Formalization of the PyLSE Machine

We will now precisely define PyLSE Machines, their semantics, and how they interact in larger designs:

Definition 3 (PyLSE Machine)**.** *A finite state machine with timed, prioritized transitions, an output set, and past constraints, which we call a PyLSE Machine, is a tuple* $M = \langle Q, q_{init}, \Sigma, \Lambda, \delta, \mu, \theta \rangle$ *where*

$$q \in Q \text{ is a set of states}$$

$$q_{init} \in Q \text{ is the initial state}$$

$$\sigma \in \Sigma \text{ is a set of input symbols}$$

$$\lambda \in \Lambda \text{ is a set of output symbols}$$

$$\delta : Q \times \Sigma \to Q \times \mathbb{N} \times \mathbb{R} \text{ is the transition function}$$

$$\mu : Q \times \Sigma \to \mathcal{P}(\Lambda \times \mathbb{R}) \text{ is the output function}$$

$$\theta : Q \times \Sigma \to \mathcal{P}(\Sigma \times \mathbb{R}) \text{ is the past constraints function}$$

We write $M.\Sigma$ *to extract* Σ*, and likewise* $M.\Lambda$ *for* Λ*. We call a system modeled using a PyLSE Machine a pulse-based system.*

The first three domains, Q, Σ, and Λ, are similar to a typical Mealy machine definition. The transition function δ maps a state and input symbol to the next state it should transition to, a natural number corresponding to the priority of that transition, and a real number corresponding to the physical time it takes to complete. The output function, μ, maps tuples of states and inputs to *sets* of tuples consisting of output symbols and the time it takes for them to appear (i.e. a firing delay). The past constraints function θ maps the current state and input to a set of input—real number tuples, each indicating

Transition Relation $\boxed{\rightarrow_{tran} \subseteq \mathrm{K} \times \Sigma \times \mathbb{R} \times (\mathrm{K} \cup \{q_{\mathtt{err}}\})}$

$$\frac{\langle q_{\mathtt{next}}, \_, \tau_{\mathtt{tran}} \rangle = \delta(q_{\mathtt{curr}}, \sigma) \quad \tau_{\mathtt{arr}} \geq \tau_{\mathtt{done}} \quad \forall \langle \sigma', \tau_{\mathtt{dist}} \rangle \in \theta(q_{\mathtt{curr}}, \sigma), \tau_{\mathtt{arr}} \geq \Theta[\sigma'] + \tau_{\mathtt{dist}}}{\kappa_{\langle q_{\mathtt{curr}}, \tau_{\mathtt{done}}, \Theta \rangle} \xrightarrow{\langle \sigma, \tau_{\mathtt{arr}} \rangle}_{tran} \kappa_{\langle q_{\mathtt{next}}, \tau_{\mathtt{tran}} + \tau_{\mathtt{arr}}, \Theta[\sigma \mapsto \tau_{\mathtt{arr}}] \rangle}} \text{ (Normal-}\kappa\text{)}$$

$$\frac{\tau_{\mathtt{arr}} < \tau_{\mathtt{done}}}{\kappa_{\langle q_{\mathtt{curr}}, \tau_{\mathtt{done}}, \_ \rangle} \xrightarrow{\langle \sigma, \tau_{\mathtt{arr}} \rangle}_{tran} q_{\mathtt{err}}} \text{ (Error-}\kappa\text{ Tran)} \qquad \frac{\exists \langle \sigma', \tau_{\mathtt{dist}} \rangle \in \theta(q_{\mathtt{curr}}, \sigma), \tau_{\mathtt{arr}} < \Theta[\sigma'] + \tau_{\mathtt{dist}}}{\kappa_{\langle q_{\mathtt{curr}}, \tau_{\mathtt{done}}, \Theta \rangle} \xrightarrow{\langle \sigma, \tau_{\mathtt{arr}} \rangle}_{tran} q_{\mathtt{err}}} \text{ (Error-}\kappa\text{ Cons)}$$

Dispatch Relation $\boxed{\rightarrow_{disp} \subseteq \mathrm{K} \times (\mathcal{P}(\Sigma) \times \mathbb{R}) \times \mathrm{K} \times (\mathcal{P}(\Sigma) \times \mathbb{R}) \times \mathcal{P}(\Lambda \times \mathbb{R})}$

$$\frac{\begin{array}{c}\sigma \in argmin(\pi_2(\delta(q_{\mathtt{curr}}, \sigma')) \quad outs = \mu(q_{\mathtt{curr}}, \sigma) \\ \kappa_{\langle q_{\mathtt{curr}}, \_, \_ \rangle} \xrightarrow[\sigma' \in \vec{\sigma}]{\langle \sigma, \tau_{\mathtt{arr}} \rangle}_{tran} \kappa_{next} \quad \vec{\sigma}_{rest} = \vec{\sigma}/\sigma\end{array}}{\left\langle \kappa_{\langle q_{\mathtt{curr}}, \_, \_ \rangle}, \langle \vec{\sigma}, \tau_{\mathtt{arr}} \rangle \right\rangle \rightarrow_{disp} \left\langle \kappa_{next}, \langle \vec{\sigma}_{rest}, \tau_{\mathtt{arr}} \rangle \right\rangle \Big| outs} \text{ (Disp)}$$

Trace Relation $\boxed{\downarrow_{trace} \subseteq \mathrm{K} \times (\mathcal{P}(\Sigma) \times \mathbb{R})^* \times (\mathcal{P}(\Lambda \times \mathbb{R}))^*}$

$$\frac{\left\langle \kappa, \langle \vec{\sigma}, \tau_{\mathtt{arr}} \rangle \right\rangle \rightarrow_{disp} \left\langle \kappa', xs \right\rangle \Big| outs \quad \left\langle \kappa', xs \right\rangle \downarrow_{trace} \left\langle \kappa'', outs' \right\rangle \quad outs'' = outs + outs'}{\left\langle \kappa, \langle \vec{\sigma}, \tau_{\mathtt{arr}} \rangle \right\rangle \downarrow_{trace} \left\langle \kappa'', outs'' \right\rangle} \text{ (Trace-Cont)} \qquad \frac{}{\left\langle \kappa, \langle \varnothing, \_ \rangle \right\rangle \downarrow_{trace} \left\langle \kappa, \varnothing \right\rangle} \text{ (Trace-Done)}$$

Network Relation $\boxed{\rightarrow_{net} \subseteq \mathcal{P}(\mathrm{K}) \times (\Sigma \times \mathbb{R})^* \times \mathcal{P}(\mathrm{K})(\Sigma \times \mathbb{R})^*}$

$$\frac{\begin{array}{c}\left\langle \langle \vec{\sigma}, \tau_{\mathtt{arr}} \rangle_M, ps' \right\rangle = \mathtt{getSimPulses}(ps) \quad \kappa_M \in \vec{\kappa} \\ \langle \kappa_M, \langle \vec{\sigma}, \tau_{\mathtt{arr}} \rangle_M \rangle \downarrow_{trace} \langle \kappa'_M, outs \rangle \quad \vec{\kappa}' = \vec{\kappa}[\kappa'_M / \kappa_M] \quad ps'' = ps' + outs\end{array}}{\langle \vec{\kappa}, ps \rangle \rightarrow_{net} \langle \vec{\kappa}', ps'' \rangle} \text{ (Network-Cont)}$$

$$\frac{\forall \langle \sigma, \tau_{\mathtt{arr}} \rangle \in ps.\ \sigma \in C.\Lambda}{\langle \_, ps \rangle \rightarrow_{net} \langle \_, ps \rangle} \text{ (Network-Done)}$$

Figure 5.7: Semantics of the Transition, Dispatch, and Trace relation of the PyLSE Machine $\langle Q, q_0, \Sigma, \Lambda, \delta, \mu \rangle$ as well as the Network relation for larger composite designs. $\pi_i(\langle ..., x_i, ... \rangle) = x_i$ is standard tuple projection. $\Theta[\sigma \mapsto \tau]$ produces an updated mapping where σ now maps to τ. We use $S\,[y/x]$ to denote y replacing x in S. The helper function `getSimPulses` extracts the pulse heap ps into the earliest set of simultaneous pulses destined for the same PyLSE Machine and the rest for later use. If both x and y are heaps of pulses, we use $x + y$ to denote merging them into a single ordered heap.

a precondition needed in order for the given transition to be allowed to proceed.

The transition semantics of our PyLSE Machine is found in Figure 5.7. To help define the semantics, we define the configuration $\kappa \in \mathrm{K} = Q \times \mathbb{R} \times \Theta$, parameterized over a current state $q \in Q$, a real-valued time $\tau_{\texttt{done}}$, and a mapping $\Theta : \Sigma \to \mathbb{R}$ that maps each input to the last time it was seen; this is written as $\kappa_{\langle q, \tau_{\texttt{done}}, \Theta \rangle}$, with the $\tau_{\texttt{done}}$ being used to represent the end of the unstable period during which time the machine is transitioning. The initial configuration is $\kappa^{M}_{init} = \kappa_{\langle q_{\texttt{init}}, 0, \{\sigma \mapsto -\infty | \sigma \in M.\Sigma\} \rangle}$.

Transition Relation Given the current configuration $\kappa_{\langle q_{\texttt{curr}}, \tau_{\texttt{done}}, \Theta \rangle}$, the Transition Relation can be interpreted as follows. If the machine receives an input σ at time $\tau_{\texttt{arr}}$, and it has been long enough to have finished the process of entering state $q_{\texttt{curr}}$ (i.e. $\tau_{\texttt{arr}} \geq \tau_{\texttt{done}}$), it proceeds to a new configuration $\kappa_{\langle q_{\texttt{next}}, \tau'_{\texttt{done}}, \Theta' \rangle}$ by remembering (1) the next state $q_{\texttt{next}}$, (2) the time at which the new transition should be completed $\tau'_{\texttt{done}} = \tau_{\texttt{tran}} + \tau_{\texttt{arr}}$, and (3) the time it saw this current input, via $\Theta' = \Theta[\sigma \mapsto \tau_{\texttt{arr}}]$ (see Normal-κ). Otherwise, if it is not yet ready to receive inputs because $\tau_{\texttt{arr}} < \tau_{\texttt{done}}$ (see Error-κ Tran) or because any input σ' was received less than $\theta(q, \sigma') + \tau_{\texttt{dist}}$ ago (see Error-κ Cons), it proceeds to the special $q_{\texttt{err}}$ state, which is the target state of any transition whose timing conditions can't be satisfied.

Dispatch Relation The Dispatch Relation is used to retrieve the highest priority transition that leaves $q_{\texttt{curr}}$ for all the inputs σ in the set of simultaneous inputs $\vec{\sigma}$ arriving at $\tau_{\texttt{arr}}$ (choosing one non-deterministically if multiple candidate transitions have the same priority); this allows the machine to continue processing inputs.

Trace Relation The Trace Relation is used to determine the outputs that result from running the Dispatch Relation over the entirety of the inputs. This sequence of outputs produced by a PyLSE Machine is defined as follows:

Definition 4 (Output Trace for a PyLSE Machine)**.** *Given a set of simultaneous inputs* $\langle \vec{\sigma}, \tau \rangle$*, an output trace t is the sequence of sets of tagged outputs produced by the Transition Relation over these input sets, one by one via the Dispatch Relation. Given PyLSE Machine M and an initial configuration* $\kappa_{init} = \kappa_{\langle q_{init}, 0, \{\sigma \mapsto -\infty | \sigma \in M.\Sigma\} \rangle}$*,* $\kappa_{init} \downarrow_{trace} \langle \kappa, t \rangle$*. The PyLSE Machine M ends in the configuration* κ*.*

5.3.3 Formalizing a Network of PyLSE Machines

While each individual PyLSE Machine models a particular type of SCE cell, a network of communicating PyLSE Machines models a larger design:

Definition 5 (Network Domain of PyLSE Machines)**.** *A network of PyLSE Machines, which we call a circuit, is a tuple* $C = \langle \vec{M}, \vec{w}, \Sigma, \Lambda \rangle$ *composed of a set of PyLSE Machines* $\vec{M}$ (*accessed as C.machines*)*, a set of connective wires* $\vec{w}$ (*accessed as C.wires*) *and a set of circuit inputs* $C.\Sigma$ *and outputs* $C.\Lambda$*. A wire is a tuple* $w = \langle \alpha, \beta \rangle$ *such that* $\alpha \in M'.\Lambda \bigcup C.\Sigma$ *and* $\beta \in M''.\Sigma \bigcup C.\Lambda$ *for some* $M', M'' \in \vec{M}$*.*

Network Relation The Network Relation of Figure 5.7 shows the semantics of how a sequence of externally derived time-tagged pulses *ts* propagate through the network. We define an initial circuit configuration κ^C_{init}, composed of (1) all individual PyLSE Machine initial configurations $\vec{\kappa}$ and (2) a list of input pulses *ps* tagged with the wires where they are headed, i.e. $\kappa^C_{init} = \langle \vec{\kappa}, ps \rangle$, where $\vec{\kappa} = \{\kappa^M_{init} \big| M \in C.machines\}$ and $ps = \{\langle \sigma', \tau_{\texttt{arr}} \rangle \big| \langle \sigma, \tau_{\texttt{arr}} \rangle \in ts \wedge \langle \sigma, \sigma' \rangle \in C.wires\}$. The network proceeds until there is no more work to do (meaning all pending pulses in *ps* are directed toward the circuit output), i.e. $\langle \kappa^C_{init}, ps \rangle \twoheadrightarrow_{net} \langle \kappa^{C'}, ps' \rangle$. Non-determinism occurs when there are multiple simultaneous pending pulses on the heap *ps* going to different PyLSE Machines; the helper function `getSimPulses` chooses one to update before proceeding with the next.

5.3.4 Example: PyLSE Machine Definition of the Synchronous And Element

We conclude our discussion of PyLSE Machines by mathematically defining the Synchronous And Element PyLSE Machine as shown in Figure 5.6. Formally,

$$M_{And} = \langle Q_{And}, q_{0_{And}}, \Sigma_{And}, \Lambda_{And}, \delta_{And}, \mu_{And}, \theta_{And} \rangle$$

where

$$Q_{And} = \{q_{\text{idle}}, q_{\text{a arrived}}, q_{\text{b arrived}}, q_{\text{a and b arrived}}\}$$

$$q_{0_{And}} = q_{\text{idle}}$$

$$\Sigma_{And} = \{\sigma_A, \sigma_B, \sigma_{CLK}\}$$

$$\Lambda_{And} = \{\lambda_Q\}$$

$$\mu(q, \sigma) = \begin{cases} \{\langle \lambda_Q, \tau_{prop} \rangle\} & \text{if } q = q_{\text{a and b arrived}} \wedge \sigma = \sigma_{CLK} \\ \varnothing & \text{otherwise} \end{cases}$$

$$\theta(q, \sigma) = \begin{cases} \{\langle \sigma_A, \tau_{setup} \rangle, \langle \sigma_B, \tau_{setup} \rangle, \langle \sigma_{CLK}, \tau_{setup} \rangle\} & \text{if } q \in \{q_{\text{idle}}, q_{\text{a arrived}}, q_{\text{b arrived}}, q_{\text{a and b arrived}}\} \\ & \quad \wedge \sigma = \sigma_{CLK} \\ \varnothing & \text{otherwise} \end{cases}$$

δ	σ_A	σ_B	σ_{CLK}
q_{idle}	$\langle q_{\text{a arrived}}, 1, 0.0 \rangle$	$\langle q_{\text{b arrived}}, 1, 0.0 \rangle$	$\langle q_{\text{idle}}, 0, \tau_{hold} \rangle$
$q_{\text{a arrived}}$	$\langle q_{\text{a arrived}}, 1, 0.0 \rangle$	$\langle q_{\text{a and b arrived}}, 1, 0.0 \rangle$	$\langle q_{\text{idle}}, 0, \tau_{hold} \rangle$
$q_{\text{b arrived}}$	$\langle q_{\text{a and b arrived}}, 1, 0.0 \rangle$	$\langle q_{\text{b arrived}}, 1, 0.0 \rangle$	$\langle q_{\text{idle}}, 0, \tau_{hold} \rangle$
$q_{\text{a and b arrived}}$	$\langle q_{\text{a and b arrived}}, 1, 0.0 \rangle$	$\langle q_{\text{a and b arrived}}, 1, 0.0 \rangle$	$\langle q_{\text{idle}}, 0, \tau_{hold} \rangle$

5.4 PyLSE Language Design

We use the above PyLSE Machine formalisms to develop a *practical* embedded DSL that eases the description and analysis of SCE designs at multiple levels. By being embedded in Python, we lower the barrier of entry for new users and gain the enormous productivity benefits of using Python's libraries. The language will be open-sourced upon publication of this work.

5.4.1 Design Levels

Cell Definition Level: Given that there is still no dominant logic scheme for SCE designs, the ability to easily define new cells is crucial for the advancement of the field. We enable this by providing a `Transitional` Python abstract class; each SCE cell is modeled as an implementing class by defining the set of input and output names as well as a list of transitions. Each transition in this list is represented as a Python dictionary, storing key-value pairs exactly corresponding to the information found in Figure 5.5. PyLSE comes with a library containing all the basic SCE cells and provides templates for the creation of custom ones.

Let's take the example Synchronous And Element, as show in the state diagram in Figure 5.6 and the semantics given in end of Subsection 5.3.4. The PyLSE code for this cell is found in Figure 5.8. It implements an abstract class `SFQ`, which is a child of the `Transitional` class mentioned previously; its purpose is to require additional properties specific to SFQ cell design from its implementing classes. For our purposes, it only requires that the `jjs` (the number of Josephson junctions) and `firing_delay` properties exist on the class.

The priorities of transitions can be given explicitly, via the `priority` key in each transition dictionary, or implied by the order in which they are listed. For example,

```
class AND(SFQ):
    _setup_time = 2.8
    _hold_time = 3.0

    name = 'AND'
    inputs = ['a', 'b', 'clk']
    outputs = ['q']
    transitions = [
        {'id': '0',  'source': 'idle',           'trigger': 'clk',
         ↪  'dest': 'idle',
         'transition_time': _hold_time, 'past_constraints': {'*':
          ↪  _setup_time}},
        {'id': '1',  'source': 'idle',           'trigger': 'a',
         ↪  'dest': 'a_arrived'},
        {'id': '2',  'source': 'idle',           'trigger': 'b',
         ↪  'dest': 'b_arrived'},
        {'id': '3',  'source': 'a_arrived',      'trigger': 'b',
         ↪  'dest': 'a_and_b_arrived'},
        {'id': '4',  'source': 'a_arrived',      'trigger': 'a',
         ↪  'dest': 'a_arrived'},
        {'id': '5',  'source': 'a_arrived',      'trigger': 'clk',
         ↪  'dest': 'idle',
         'transition_time': _hold_time, 'past_constraints': {'*':
          ↪  _setup_time}},
        {'id': '6',  'source': 'b_arrived',      'trigger': 'a',
         ↪  'dest': 'a_and_b_arrived'},
        {'id': '7',  'source': 'b_arrived',      'trigger': 'clk',
         ↪  'dest': 'idle',
         'transition_time': _hold_time, 'past_constraints': {'*':
          ↪  _setup_time}},
        {'id': '8',  'source': 'b_arrived',      'trigger': 'b',
         ↪  'dest': 'b_arrived'},
        {'id': '9',  'source': 'a_and_b_arrived', 'trigger': 'clk',
         ↪  'dest': 'idle',
         'transition_time': _hold_time, 'past_constraints': {'*':
          ↪  _setup_time},
         'firing': 'q'},
        {'id': '10', 'source': 'a_and_b_arrived', 'trigger': ['a', 'b'],
         ↪  'dest': 'a_and_b_arrived'},
    ]
    jjs = 11
    firing_delay = 9.2
```

Figure 5.8: Synchronous And Element PyLSE code.

```
mem = defaultdict(lambda: 0)
raddr = waddr = wenable = data = 0

@pylse.hole(delay=5.0, inputs=['ra3', 'ra2', 'ra1', 'ra0', 'wa3', 'wa2', 'wa1',
 ↪  'wa0', 'd1', 'd0', 'we', 'clk'], outputs=['q1', 'q0'])
def memory(ra3, ra2, ra1, ra0,
           wa3, wa2, wa1, wa0,
           d1, d0, we, clk, time):
  nonlocal raddr, waddr, wenable, data
  raddr |= ra3*8 + ra2*4 + ra1*2 + ra0
  waddr |= wa3*8 + wa2*4 + wa1*2 + wa0
  data |= d1*2 + d0
  wenable |= we
  if clk:
    if wenable:
      mem[waddr] = data
    value = mem[raddr]
    raddr = waddr = wenable = data = 0
  else:
    value = 0
  return ((value >> 1) & 1), value & 1
```

Figure 5.9: An example `Functional` ("hole") element modeling a memory with 16 addresses, each storing 2 bits.

in the C Element, the transition leaving `idle` on `clk` is given before the transition leaving `idle` on `a`; thus, the former's trigger has priority over the latter's. This priority order is isolated to transitions with the same `source` state; for example, the first and fourth transitions have different `source` states (`idle` and `a_arrived`, respectively), and thus their relative order in this list of transitions is irrelevant. Finally, all transitions have a default `transition_time` of 0, and all firing delays for SFQ cells use the class's `firing_delay` for its value (for even greater flexibility, `Transitional` allows subclasses to set the firing delay individually per transition and output).

Hole Description Level: To facilitate the rapid prototyping and exploration of more complicated designs without the need to describe every single block via interacting tran-

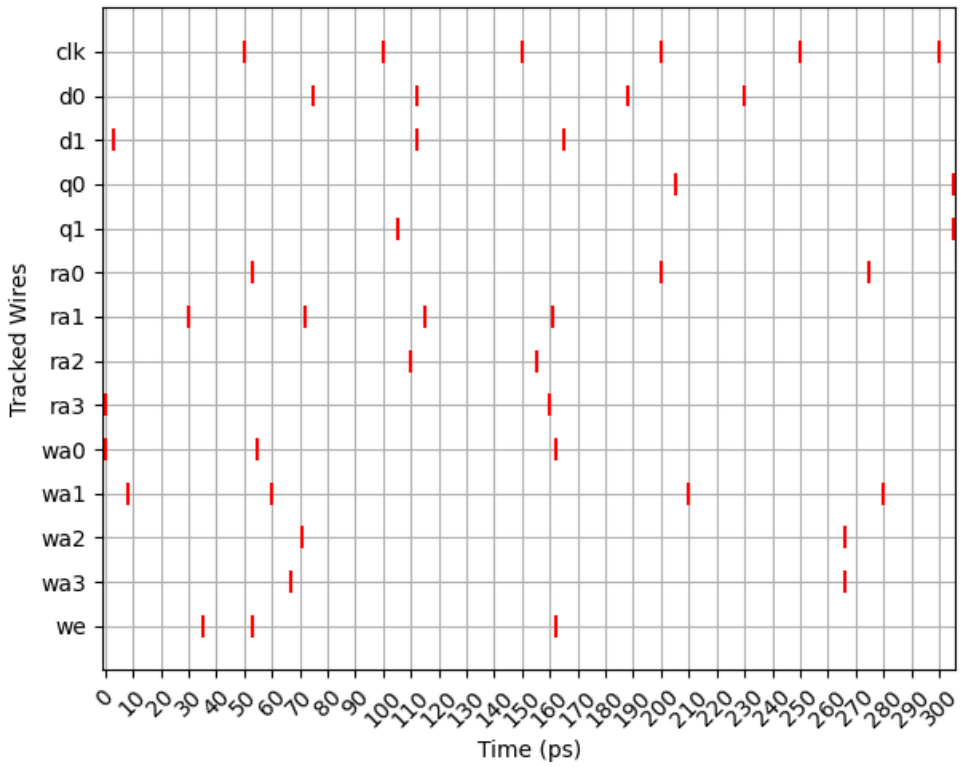


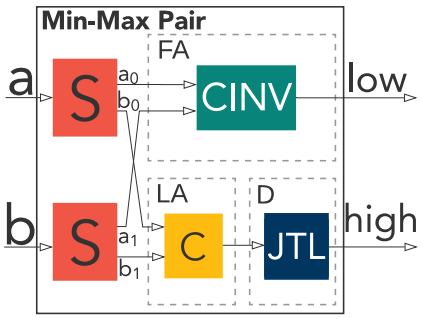
Figure 5.10: Graphical results of simulating the memory `Functional` class.

sition systems, PyLSE provides the Hole Description Level. At this level, pure Python code is wrapped behind a specialized interface (by implementing a `Functional` abstract class), allowing non-transition-based abstract "holes" to communicate via pulses with the rest of the system. The `Functional` class takes as initialization parameters (1) a `Callable` function mapping time-tagged input to output pulses, (2) the list of input and output names, and (3) the firing delay for each output. The user can also simply wrap a Python function (with the appropriate signature) with the `hole` decorator. Note that these holes do not abide by the formal semantics of Section 5.3.

An example functional element is found in Figure 5.9, which shows how to create a memory by wrapping a Python dictionary behind a function with a pulse-communicating interface. This function, `memory()`, takes in twelve boolean-valued arguments, and a thirteenth argument, `time`, which is implicitly passed as the last argument to all functional elements `time`. The read and write addresses, `ra*` and `wa*`, respectively, are split into four 1-bit inputs, and a pair of nonlocal variables `raddr` and `waddr` are used to accumulate which address bits have been seen since the last clock pulse. When a clock pulse arrives, the memory is updated if write is enabled, the newly read value is produced as tuple of 1-bit values, and the accumulators are reset, ready for the next period. The arguments are internally connected to PyLSE `Wire`s in the network, and the framework automatically converts the presence of a pulse on one or more of them at a particular instant as a call to `memory()`, passing a value of 1 for each of the corresponding arguments plus the current time in `time`.

Figure 5.10 shows the result of simulating the memory hole against a variety of inputs.

Full-Circuit Design Level: `Node`s of `Transitional` and `Functional` class instances are interconnected with `Wire`s and added to the circuit workspace at the Full-Circuit Design



(a) Block diagram.

```
from pylse import s, c, c_inv, jtl
def min_max(a, b):
  a1, a2 = s(a)
  b1, b2 = s(b)
  low = c_inv(a1, b1)
  high = c(a2, b2)
  high = jtl(high, firing_delay=2.0)
  return low, high
```

(b) PyLSE code.

Figure 5.11: Min-max pair. Inputs `a` and `b` are duplicated by the splitters. a_0 and b_0 enter the Inverted C Element, which propagates an output pulse on `low` some delay after the first of its inputs arrive. a_1 and b_1 are fed into the C Element, whose output is delayed via a Josephson transmission line (for path balancing) before being emitted as the `high` output. The `2.0` JTL delay has been calculated based on the difference in delays between the paths to `low` and to `high`, assuming a splitter delay of 11, C Element delay of 12, and Inverted C Element delay of 14. Thus, given `low, high = comp(a, b)`, the earlier input pulse propagates to `low` after 11 + 14 = 25 ps and the latter to `high` (likewise after 11 + 12 + 2 = 25 ps).

level. The code in Figure 5.11b provides an example of a Min-Max pair implemented with two splitters, a C Element, an Inverted C Element, and a JTL (all basic SFQ cells [171]) following recently introduced temporal conventions [150]. A Min-Max pair circuit is an attractive application for pulse-based computation because it sorts its input according to arrival times by using race logic primitives like First Arrival and Last Arrival (implemented by the Inverted C Element and C Element, respectively).

Calling the function `min_max(a,b)` causes its constituent cells and connective wires to be instantiated via the calls to the encapsulating functions `s`, `c`, `c_inv`, and `jtl`. These functions take in wire objects and return one or more output wire objects as result; by updating the circuit workspace automatically, this encapsulation enables basic cells to resemble Python operators and improve language usability. These functions can also take in optional arguments, making it easy to *override* properties like firing delay, transition time for arbitrary transitions, and the number of Josephson junctions used in a

particular element instance (the latter of which is an area metric based on the number of switching elements used by the design). At this level, full application implementations can be realized through the technique of elaboration-through-execution [119, 172, 173], although here, unlike traditional HDLs, the underlying primitives used by higher level generators are inherently stateful and pulse-based.

5.4.2 Syntactic and Semantic Checks

PyLSE provides several syntactic and semantic checks to alert the user if a design is ill-formed. At the Cell Definition level, PyLSE ensures the list of a cell's transitions constitute a well-formed transition system. These include simple checks such as the use of recognized field names, references to valid input triggers and output signal names, and the inclusion of an `idle` starting state. More advanced checks include the complete specification of transitions for every possible trigger, and that at least one transition has been defined that fires an output.

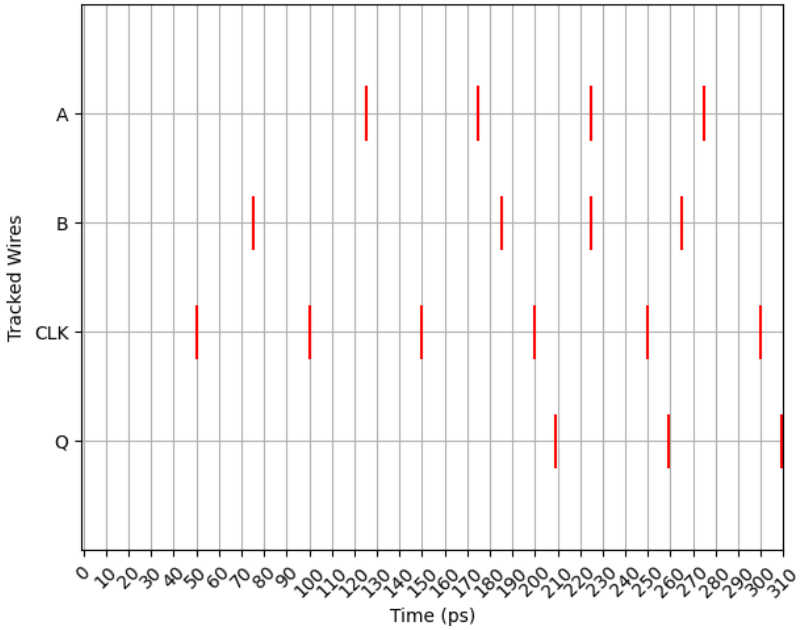
At the Circuit Design level, we currently check that all circuit outputs have a "fanout" of one. In SCE, the outputs of arbitrary cells cannot immediately be shared by multiple inputs; instead, a *splitter* cell must be used, which is specifically designed to forward an incoming pulse to two different outgoing wires. The example in Figure 5.11b includes two splitter cells (lines 3 and 4) to allow `a` and `b` to be used in two different places; PyLSE reports an error on instantiation if, for example, input `a` is used in both lines 5 and 6.

5.4.3 Simulation

PyLSE's built-in simulator can be used to validate designs against a given set of input signals. It follows the principles of other discrete-event simulation frameworks

```
from pylse import inp_at, inp, and_s, Simulation
a = inp_at(125, 175, 225, 275, name='A')
b = inp_at(75, 185, 225, 265, name='B')
clk = inp(start=50, period=50, n=6, name='CLK')
out = and_s(a, b, clk, name='Q')
sim = Simulation()
events = sim.simulate()
assert events['Q'] == [209.2, 259.2, 309.2]
sim.plot()
```

(a) Exhaustively simulating a Synchronous And Element.



(b) Simulation result in graphical form.

Figure 5.12: Simulation of the Synchronous And Element in PyLSE. Pulses occur on wires `A` at 125.0, 175.0, 225.0, and 275.0; `B` at 75.0, 185.0, 225.0, and 265.0; and `CLK` every 50.0 ps, six times starting at 50.0. The cell's default firing delay is 9.2, so we check for expected pulses on `Q` 9.2 ps after a clock period in which both `A` and `B` were received.

Table 5.1: Functions used in the code in Figure 5.12a. The first four return a `named` wire, while `simulate()` returns a mapping from each named wire to the ordered list of pulses that appeared on it. The last two are methods on the `Simulation` class.

Function	Description
`inp_at(*times, name=None)`	Produce pulses at each time in `*times`.
`inp(start=0, period=0, n=1, name=None)`	Produce pulses starting at `start`, occurring `n` more times every `period` picoseconds.
`split(wire, n=2, names=None, **overrides)`	Split a wire `n` ways, creating `n-1` splitter elements in a binary tree.
`inspect(wire, name)`	Give a `wire` a `name` for observation during simulation.
`simulate(self, until, variability=False)`	Simulate the circuit `until` a certain time or all pulses are processed.
`plot(self)`	Produce a graph plotting the pulses against time.

```
b = inp_at(99, 185, 225, 265, name='B')
...
events = sim.simulate()

pylse.pylse_exceptions.PylseError: Error while sending
input(s) 'clk' to the node with output wire '_0':
Prior input violation on FSM 'AND'. A constraint on
transition '7', triggered at time 100.0, given via the
'past_constraints' field says it is an error to trigger
this transition if input 'b' was seen as recently as
2.8 time units ago. It was last seen at 99.0, which is
1.7999999999999998 time units to soon.
```

Figure 5.13: Changing the first time at which a pulse is produced on `B` in the simulation of Figure 5.12a rightfully results in a past constraint error due to the setup time.

[174] and, according to the semantics provided in Figure 5.7, maintains a priority heap of pending pulses tagged with their destination cells. When their turn comes, these pulses get popped off the heap and propagate through the PyLSE circuit under test. Any newly generated pulses get pushed onto the heap, and the process continues iteratively until the heap is empty or a user-defined target time is reached (needed when there are loops in the system).

Figure 5.12a shows how a single instance of the Synchronous And Element gets instantiated and simulated, using many of the functions described in Table 5.1. In lines

2 and 3, we create two inputs named `A` and `B`, producing four pulses on each. Line 4 creates a periodic clock signal, while lines 6 and 7 create and start a simulation object. Line 8 verifies the correctness of pulses appearing on output `Q`; in this example, the first appears at 209.2 ps, exactly `firing_delay` after the input pulse on `CLK` that ended the first clock period in which both `A` and `B` appeared. Line 9 produces the graph in Figure 5.12b. Finally, Figure 5.13 shows the PyLSE simulator catching a past constraints violation (the setup time constraint); the first pulse produced on `B` arrives too soon before the next pulse that arrives on `CLK`.

5.4.4 Correspondence with Timed Automata

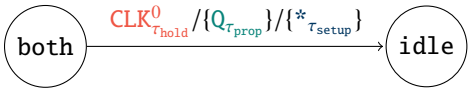
Timed Automata (TA) are a related formalism with a rich theoretical foundation, used extensively to model real-time systems with timing constraints. A Timed Automaton [155] is a finite state machine whose state transitions are guarded by conditions on a set of *resettable clocks*, defined formally as follows:

Definition 6 (Timed Automata)**.** *A Timed Automaton* $A = \langle L, l_0, \Sigma, C, E, I \rangle$ *is a tuple where* $(l \in)L$ *is a set of locations,* $l_{init} \in L$ *is the initial location,* $(\alpha \in)\Sigma$ *is the set of actions,* $(c \in)C$ *is a set of clocks,* $I : L \to \Phi(C)$ *are clock invariants at each location, and*

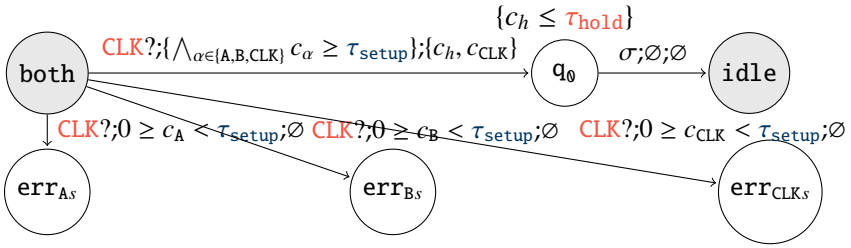
$$(e \in)E \subseteq L \times \Sigma \times \Phi(C) \times \mathcal{P}(C) \times L$$

is the set of transitions. $e = \langle l, \alpha, \varphi, \lambda, l' \rangle \in E$ *is a transition from location* l *to* l' *on action* α*,* φ *is the guard specifying conditions that must be true on the clocks, and* λ *is the set of clocks to be reset after the transition.*

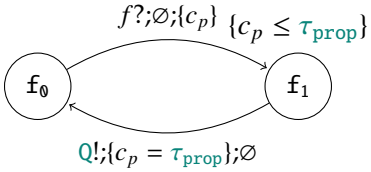
Semantics The operational semantics are a transition relation on snapshots of the timed automaton in time; we call these snapshots *configurations* and define them as



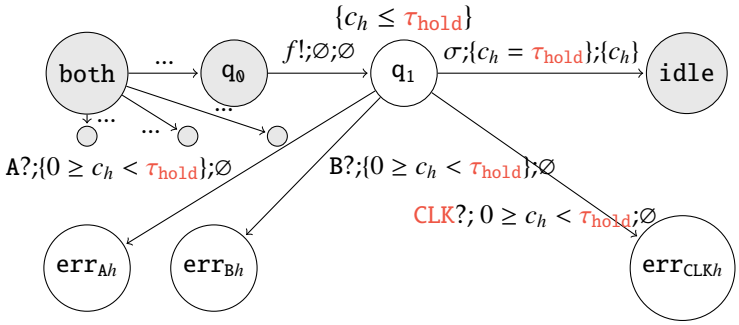
(a) Original PyLSE Machine transition, moving from `both` to `idle` on CLK only if τ_{hold} time has passed, producing output Q after τ_{prop} time. It is an error if any inputs (i.e. *) arrived in the last τ_{setup} time units prior to starting this transition. The priority 0 is ignored here in isolation.



(b) (Intermediary step) Expanding the original transition into two intermediate TA transitions that handle receiving a message on channel CLK (corresponding to original symbol CLK) (left edge), checking for the transition time to have passed (right edge), and erroring out (to $\texttt{err}_{As}$, $\texttt{err}_{Bs}$, or $\texttt{err}_{CLKs}$) if the τ_{setup} is violated.



(d) (Final step, part 2) Auxiliary TA created for modeling firing delay. The TA in Figure 5.14c sends a message on channel f, which is received here. After τ_{prop} time units, output finally appears on output channel Q. This firing TA, including a fresh clock c_p, is duplicated by a soaking factor $s = \lceil \tau_{prop}/\tau_{hold} \rceil$ to allow the network to fire again if needed during the transition.



(c) (Final step, part 1) Further expanding the transition to include transitions for firing output f and erroring out (to $\texttt{err}_{Ah}$, $\texttt{err}_{Bh}$, or $\texttt{err}_{CLKh}$) if unexpected inputs are received during a transitionary period.

Figure 5.14: Expanding a PyLSE Machine transition into its corresponding TA transitions, using an edge from the Synchronous And Element (for brevity, we've replaced the state named `a and b arrived` with `both`). We assume clocks c_h, c_s, and c_p and channels A, B, CLK, f and Q. Shaded states (or `...` edges) indicate old states (edges) repeated from the previous figure.

a location/clock valuation tuples, such that the invariant at that location is satisfied: $Conf(A) = \{\langle l, v\rangle |\ l \in L,\ v : C \to Time,\ v \models I(l)\}$. A configuration $\langle l, v\rangle$ moves to a configuration $\langle l', v'\rangle$ by one of two transitions: an *action* transition or a *delay* transition. In an *action* transition, $\langle l, v\rangle$ moves to $\langle l', v'\rangle$ by either sending or receiving a message on a channel; formally, $\langle l, v\rangle \xrightarrow{\lambda = a_{?!}} \langle l', v'\rangle \iff \exists \langle l, \alpha, \varphi, \lambda, l'\rangle \in E$ s.t. $v \models \varphi \wedge v' \models v[\lambda := 0] \wedge v' \models I(l')$. In a *delay* transition, $\langle l, v\rangle$ moves to $\langle l', v'\rangle$ by way of time passing; formally, $\langle l, v\rangle \xrightarrow{\lambda = t} \langle l', v + t\rangle \iff \forall t' \in [0, t] : v + t' \models I(l)$. The *silent* transition is represented by σ. Finally, the initial configuration is defined as the set:

$$Conf_{init} = \{\langle l_{init}, v_0\rangle\} \cap Conf(A) \text{ with } v_0(x) = 0,\ \forall c \in C$$

Using these definitions gives us:

Definition 7 (Timed Automata Semantics)**.** *The semantics S of a Timed Automaton are defined by*

$$S(A) = \langle Conf(A), Time \cup \Sigma_{?!}, \{\lambda \mid \lambda \in Time \cup \Sigma_{?!}\}, Conf_{init}\rangle$$

We now present two examples to better explain the different transition types:

Action Transition Example From a TA's initial configuration, assuming there exists another playing the role of an "environment" that sends a message on the a channel, the TA is able to make an action transition:

$$\langle 0,\ v(x) \mapsto 0\rangle \xrightarrow{\lambda = a?} \langle 1,\ v(x) \mapsto x\rangle$$

The above transition simply indicates that the TA can receive a message on channel a and move from location 0 to 1 instantaneously. In the first configuration, our clock valuation sets the local clock x to 0 and at the second configuration, we set the x to its

current value, indicating no time has passed.

Delay Transition Example By moving our imaginary token forward a few steps, the TA can additionally make a delay transition. Assuming the following configurations:

$$\langle 3,\ v(x) \mapsto t \rangle \xrightarrow{\lambda = 2.3} \langle 4,\ v(x) \mapsto t + 2.3 \rangle$$

This transition indicates that from location 3 the TA can't move to location 4 without moving time forward by 2.3 units (e.g. picoseconds). This concomitantly increases our local clock, which incidentally also increases the global clock.

Conversion to TA To directly obtain the benefits of TA, we can convert a PyLSE Machine to a network of Timed Automata running in parallel. Figure 5.14 graphically shows this conversion process for a single edge of the Synchronous And Element. This is the same edge highlighted in Figure 5.6 and dissected in Section 5.3.1, but we've replaced the state named `a and b arrived` with `both` for space constraints. At a high level, this process works by expanding the edges from the original PyLSE Machine into TA transition sequences. We first create TA clocks for each PyLSE Machine input ($c_{\mathtt{A}}$, $c_{\mathtt{B}}$, and $c_{\mathtt{CLK}}$), in addition to a clock measuring the time elapsed on transitions (c_h); these clocks are available to all edges of this TA. Given edge $\mathtt{CLK}^{0}_{\tau_{\mathtt{hold}}}/\{\mathtt{Q}_{\tau_{\mathtt{prop}}}\}/\{*_{\tau_{\mathtt{setup}}}\}$ emerging from state `both`, translation proceeds incrementally. The input symbol CLK of the PyLSE Machine becomes a TA channel CLK on which messages are only received by this automaton. The time it takes to complete the transition, $\tau_{\mathtt{hold}}$, becomes part of the inequality in both location $\mathtt{q_0}$'s invariant and in the guard involving clock c_h as part of final edge to `idle`. In addition, clocks $c_{\mathtt{A}}$, $c_{\mathtt{B}}$, and $c_{\mathtt{CLK}}$ are compared against the past constraint value, $\tau_{\mathtt{setup}}$, in the first edge's guard, going to an error state if violated and otherwise permitting the transition to $\mathtt{q_0}$ to be taken. Figure 5.14b is the result of this

first conversion, producing an intermediate TA.

To handle detecting inputs while in the transitional period, Figure 5.14c inserts three additional states, $\texttt{err}_\texttt{a}$, $\texttt{err}_\texttt{b}$ and $\texttt{err}_\texttt{clk}$, one per possible input message that can be received. Figure 5.14c also adds intermediate state $\texttt{q}_1$, with the dual purpose of sending a firing message f to an auxiliary TA created in Figure 5.14d, and for setting up the clock that is used for checking that the transitional timing period has been satisfied before going to state `idle`. The auxiliary TA in Figure 5.14d is created entirely *alongside* the previous TA; when it receives a message f to fire, it waits the designated firing delay time $\tau_{\texttt{prop}}$ before sending a message on channel `Q`. Here, producing output `Q` in the original PyLSE Machine corresponds to sending a message on the channel `Q` created solely for sending, allowing an output action and transition to occur in parallel. Finally, UPPAAL admits a notion of priority by allowing channels to be declared with the `priority` keyword along with an associated ordering or priorities. Given two channels with two different priorities, only the higher priority action will be enabled.

There is a significant increase in complexity as one moves down from PyLSE Machine to TA; the example in Figure 5.14 shows that at least 12 TA locations and 11 edges had to be created for a *single* PyLSE Machine transition. The entire resultant TA network for a single Synchronous And Element PyLSE Machine has 102 locations and 110 edges. PyLSE properly encapsulates this complexity, allowing this much larger TA network to instead be represented by the four states and twelve edges of the original PyLSE Machine, as shown in Figure 5.6. The user can succinctly write the transition system in our DSL, and it gets *automatically* converted to TA.

5.5 Evaluation

The goal of our evaluation is to prove the following claims:

Claim 1. *PyLSE can be used to accurately model the functional and timing behavior of basic SCE cells and larger designs.*

Claim 2. *PyLSE offers significant productivity gains over state-of-art HDLs for designing and simulating basic SCE cells and larger designs.*

Claim 3. *PyLSE can be used in conjunction with a state-of-the-art model checker to formally verify properties of basic SCE cells and larger designs.*

To evaluate these claims, we implemented 16 basic cells (constituting the PyLSE standard library) plus six larger designs as listed in Table 5.3. While these designs appear relatively small, we note that each basic cell takes 15-100+ lines to represent in Verilog (the most commonly used approach). In [175], for example, the OR cell autogenerated from their analog model takes 58 lines of Verilog. As far as we know, there are no open source cell libraries available containing all the basic cells we list in Table 5.3, making direct comparison difficult.

5.5.1 SPICE Simulation Comparison

Circuit designers perform simulations with low-level languages like SPICE [176] and WRSpice [177] to create analog gate models using fundamental electrical components. However, this process can be time-consuming and requires significant domain expertise. SPICE and PyLSE occupy different levels of abstraction, with the one complementing the other; the information on delays produced by highly accurate SPICE simulation can inform models of the gates via their respective PyLSE Machines. It is through this abstraction that PyLSE can improve productivity by making it easier to scale and simulate larger designs before physically implementing them. However, SPICE simulations also highlight other parasitic effects that can change delays when two or

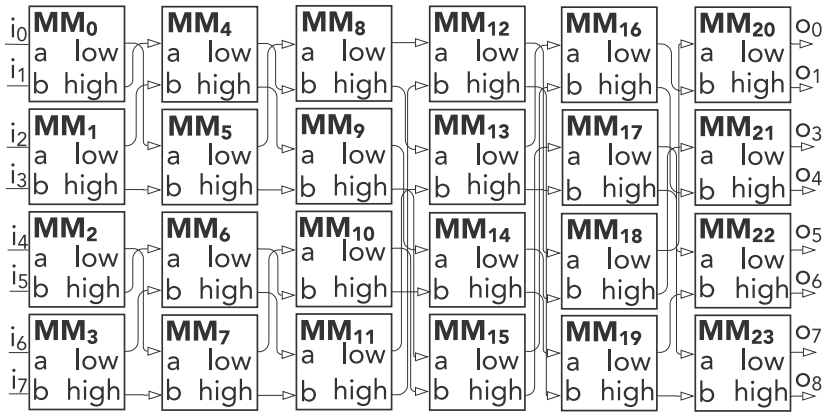


Figure 5.15: A eight-input bitonic sorter, composed of twenty-four comparators (see Figure 5.11a). It takes eight individual input wires (i_0 through i_7), which are produced in arrival time order, after some network delay, on o_0 through o_7.

more gates are connected together – thus a key to developing a successful simulator at a different level of abstraction is to verify that the two match in spite of design size. Small discrepancies in delays are expected to be found as a result of both loading effects and additional buffering stages used to improve signal fidelity.

For a more detailed discussion, we will focus on an 8-input bitonic sorter and the cells that compose it. A bitonic sorter [178] is a parallel sorting network made up of many Min-Max pair blocks (Figure 5.11), connected like in Figure 5.15. Figure 5.16 shows the PyLSE code for creating an arbitrary N input bitonic sorter (where N is a power of two). By setting it $N = 8$, we get the bitonic sorter presented in Figure 5.15. For line count comparison against SPICE code later, we use a version where N has been hard-coded to 8, meaning we manually connect the Min-Max pairs together, as shown in the PyLSE code in Figure 5.17; both Figures 5.16 and 5.17 are equivalent.

To validate the accuracy of PyLSE, we compare simulation results of running the four designs shown in Table 5.2 in both SPICE via Cadence [179] and PyLSE.

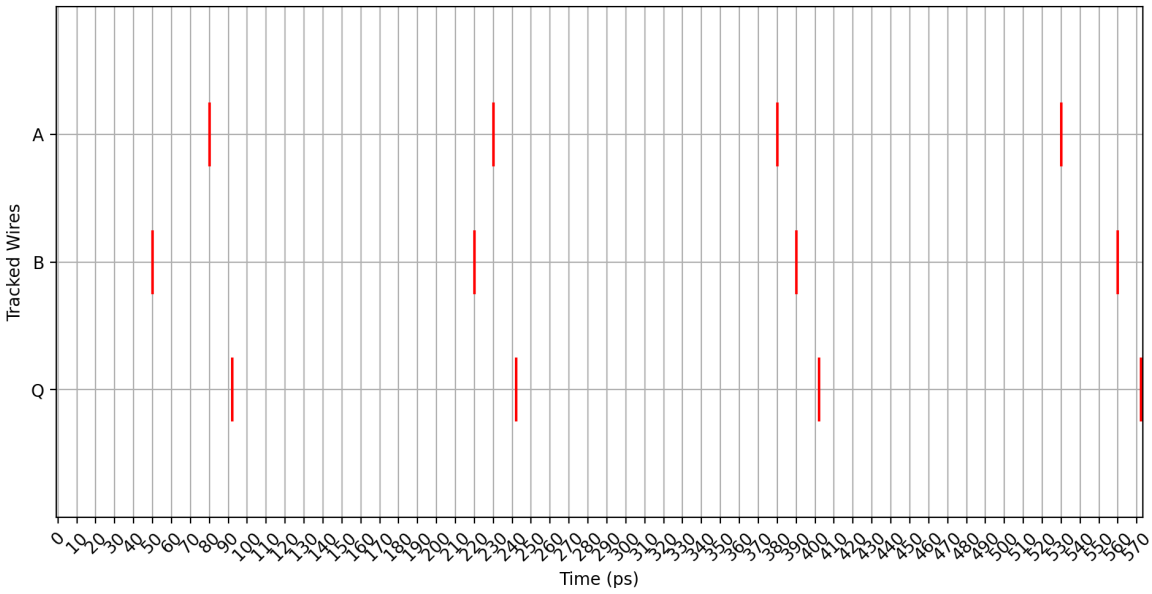
Figures 5.18, 5.19, 5.20, and 5.21 compare simulating three designs in PyLSE and SPICE. These SPICE simulations operate on the level of the analog design (for example,

```python
def split(*args):
  mid = len(args) // 2
  return args[:mid], args[mid:]

def cleaner(*args):
  upper, lower = split(*args)
  res = [min_max(*t) for t in zip(upper, lower)]
  new_upper = tuple(t[0] for t in res)
  new_lower = tuple(t[1] for t in res)
  return new_upper, new_lower

def crossover(*args):
  upper, lower = split(*args)
  res = [min_max(*t) for t in zip(upper, lower[::-1])]
  new_upper = tuple(t[0] for t in res)
  new_lower = tuple(t[1] for t in res[::-1])
  return new_upper, new_lower

def merge_network(*args):
  if len(args) == 1: return args
  upper, lower = cleaner(*args)
  return merge_network(*upper) + merge_network(*lower)

def block(*args):
  upper, lower = crossover(*args)
  if len(upper + lower) == 2: return upper + lower
  return merge_network(*upper) + merge_network(*lower)

def bitonic_helper(*args):
  if len(args) == 1: return args
  else:
    upper, lower = split(*args)
    new_upper = bitonic_helper(*upper)
    new_lower = bitonic_helper(*lower)
    return block(*new_upper + new_lower)

def bitonic_sort(*args):
  if len(args) == 0:
    raise pylse.PylseError("bitonic_sort requires at least one argument to sort")
  if len(args) & (len(args) - 1) != 0:
    raise pylse.PylseError("number of arguments to bitonic_sort must be a power of 2")
  return bitonic_helper(*args)
```

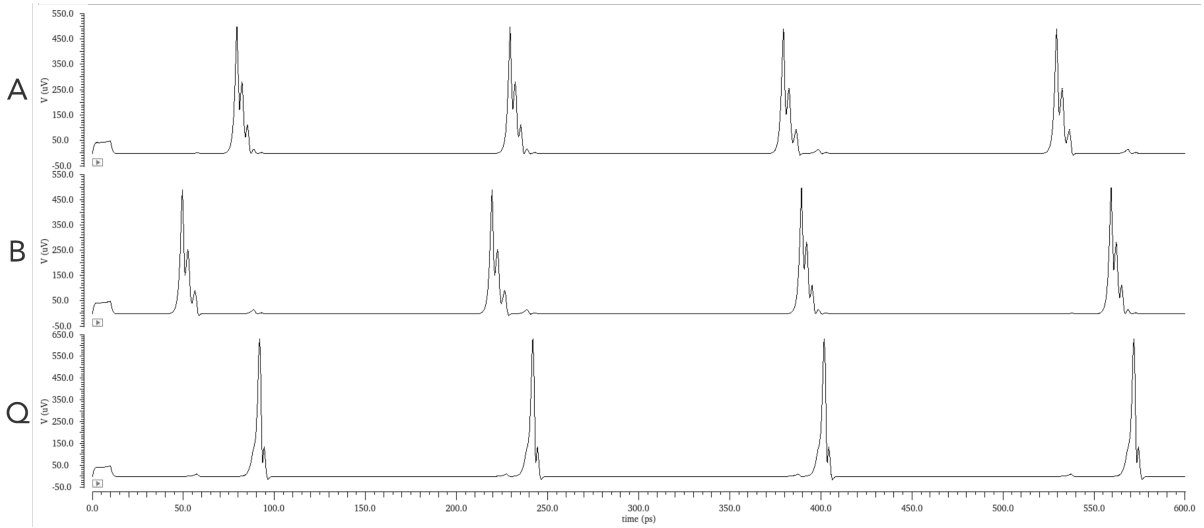
Figure 5.16: An N-input bitonic sorter implementation written in PyLSE. The only SFQ-specific cells are created in the calls within `cleaner()` and `crossover()` to `comp()`, whose definition is listed in Figure 5.11b. Given N unordered input wires, the sorter returns N ordered output wires. The "value" of a wire is determined by the time at which the pulse arrives, such that the comparison operation between two wires `a` and `b` is true if `a` arrives earlier than `b`. The associated block diagram is found in Figure 5.15.

```
def bitonic_sort_8(a1, a2, a3, a4, a5, a6, a7, a8):
  c1l, c1h = min_max(a1, a2)
  c2l, c2h = min_max(a3, a4)
  c3l, c3h = min_max(c1h, c2l)
  c4l, c4h = min_max(c1l, c2h)
  c5l, c5h = min_max(c4l, c3l)
  c6l, c6h = min_max(c3h, c4h)

  c7l, c7h = min_max(a5, a6)
  c8l, c8h = min_max(a7, a8)
  c9l, c9h = min_max(c7h, c8l)
  c10l, c10h = min_max(c7l, c8h)
  c11l, c11h = min_max(c10l, c9l)
  c12l, c12h = min_max(c9h, c10h)

  c13l, c13h = min_max(c5l, c12h)
  c14l, c14h = min_max(c5h, c12l)
  c15l, c15h = min_max(c6l, c11h)
  c16l, c16h = min_max(c6h, c11l)
  c17l, c17h = min_max(c14l, c16l)
  c18l, c18h = min_max(c16h, c14h)
  c19l, c19h = min_max(c13l, c15l)
  c20l, c20h = min_max(c15h, c13h)
  c21l, c21h = min_max(c19l, c17l)
  c22l, c22h = min_max(c19h, c17h)
  c23l, c23h = min_max(c18l, c20l)
  c24l, c24h = min_max(c18h, c20h)
  return c21l, c21h, c22l, c22h, c23l, c23h, c24l, c24h
```

Figure 5.17: An 8-input bitonic sorter implementation written in PyLSE. The SFQ-specific cells are created in the calls to `comp()`, whose definition is listed in Figure 5.11b. Simulation results are in Figure 5.21a. This function is an alternative way of writing the bitonic sorter produced by passing eight arguments to the function in Figure 5.16.

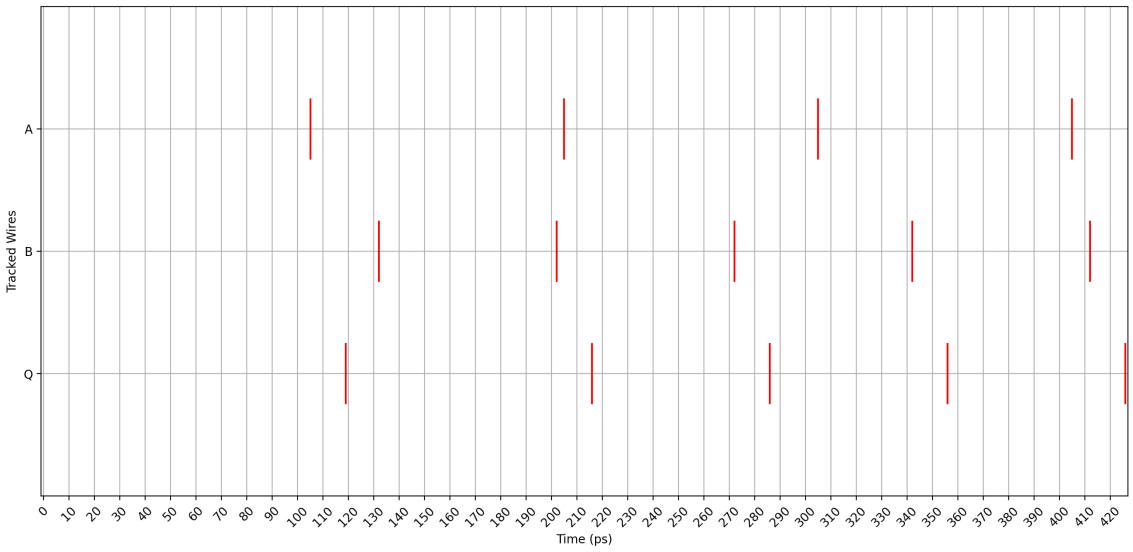


(a) PyLSE simulation (C Element).

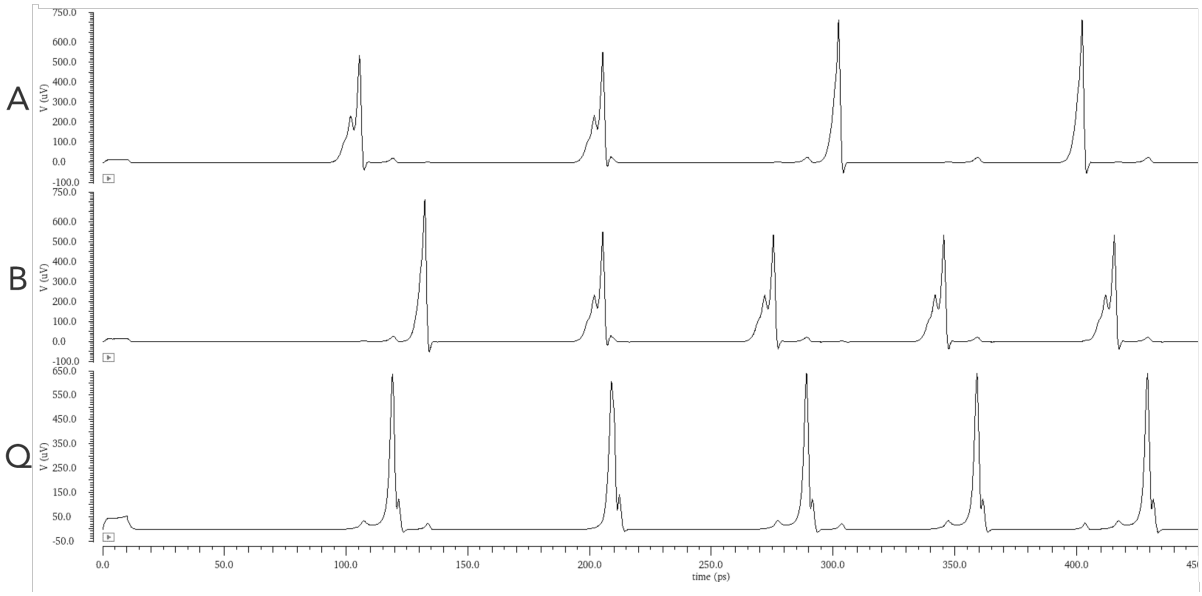


(b) SPICE simulation (C Element).

Figure 5.18: SPICE vs. PyLSE simulation results for the C Element.



(a) PyLSE simulation (Inverted C Element).

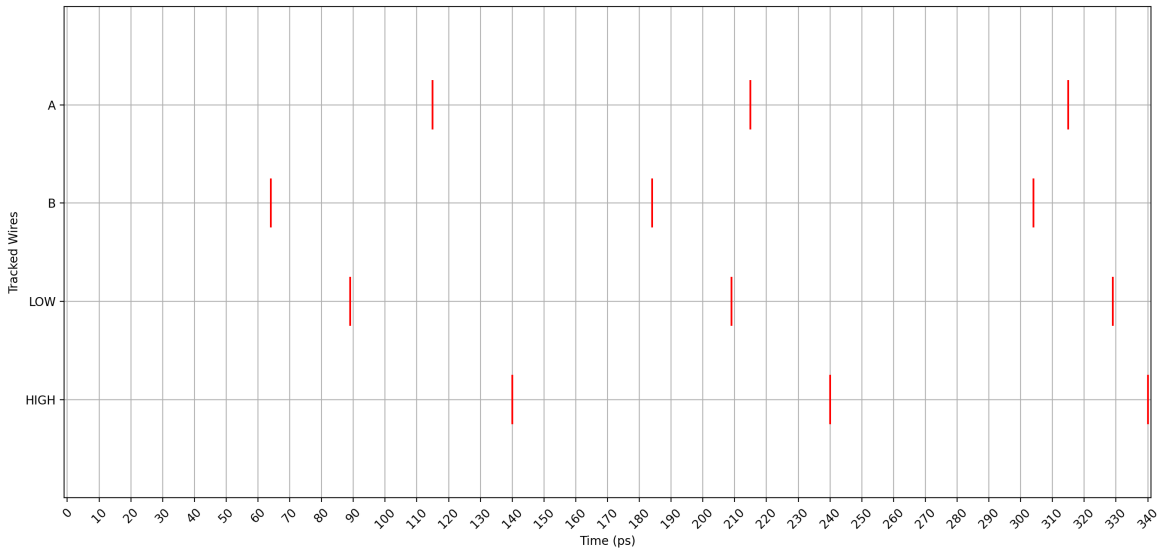


(b) SPICE simulation (Inverted C Element).

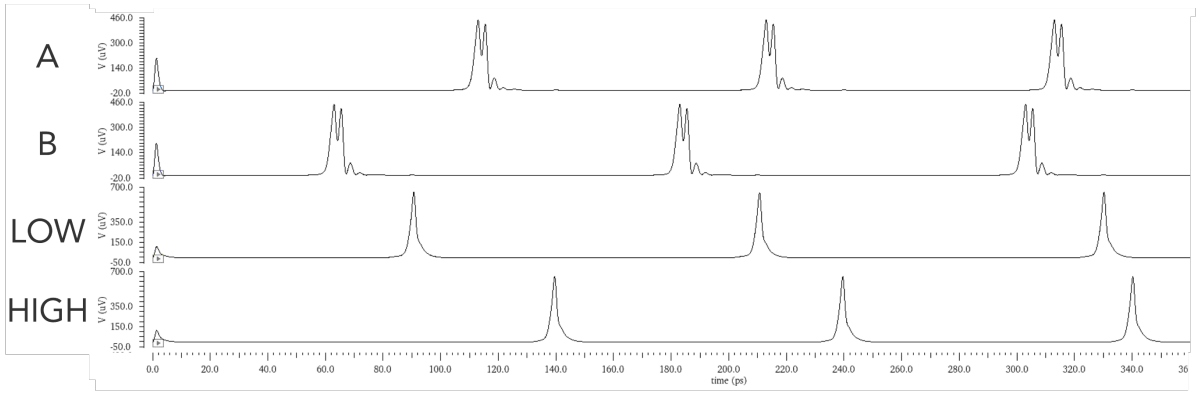
Figure 5.19: SPICE vs. PyLSE simulation results for the Inverted C Element.

Table 5.2: Simulation times of PyLSE vs. SPICE-level models. For the C and InvC elements, size refers to the number of transitions in the DSL (≈ the number of lines), and for the rest, the number of non-whitespace lines within the function. The number of SPICE lines reported is the size of the unflattened netlist.

Name	SPICE (Cadence)		PyLSE	
	Lines	Time (s)	Size	Time (s)
C	81	2.840	6	0.000298
InvC	87	2.987	6	0.000336
Min-Max Pair	140	4.608	5	0.000617
Bitonic Sort 8	250	52.565	24	0.003857

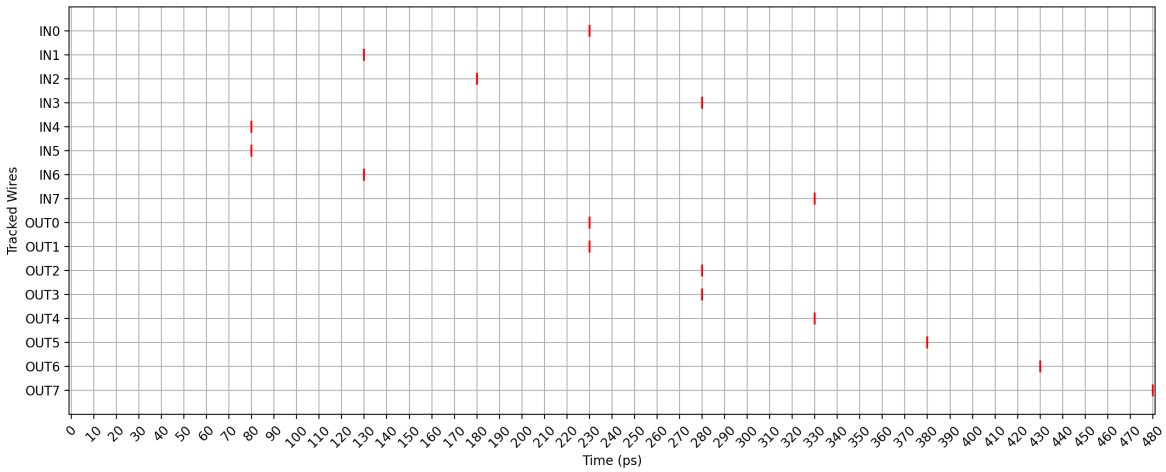


(a) PyLSE simulation (min-max).

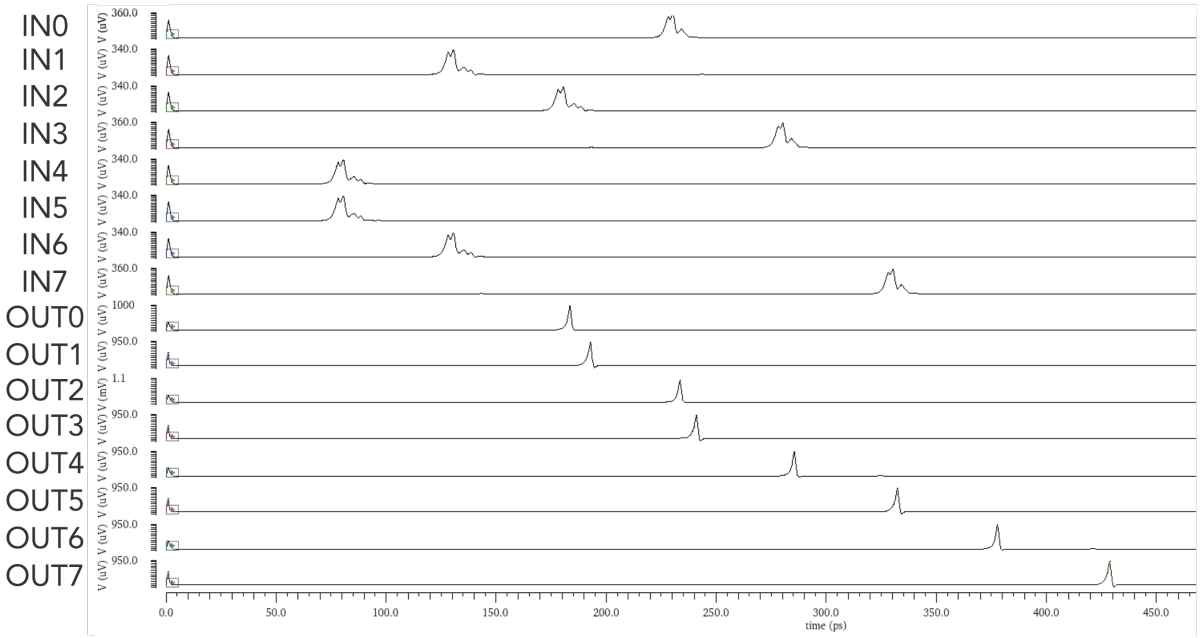


(b) SPICE simulation (min-max).

Figure 5.20: SPICE vs. PyLSE simulation results for the min-max pair.



(a) PyLSE simulation (8-input bitonic sort).



(b) SPICE simulation (8-input bitonic sort).

Figure 5.21: SPICE vs. PyLSE simulation results for the eight-input bitonic sorter.

Figure 5.3a) while the PyLSE simulations operate on the level of the PyLSE Machine (Figure 5.3b). Figures 5.18a (PyLSE) and 5.18b (SPICE) simulate the C Element; given identical inputs and C cell propagation delay (12 ps), the output times of both simulations match exactly: with inputs `A` arriving at 80.0, 230.0, 380.0, and 530.0 ps and `B` arriving at 50.0, 220.0, 390.0, and 560.0, respectively, and output `Q` arriving at 92.0, 242.0, 402.0, 572.0. Figures 5.20a (PyLSE) and 5.20b (SPICE) simulate the min-max pair, tested with three pairs of (`A`, `B`) inputs: (115, 64), (184, 215), and (304, 315) ps. The SPICE model is balanced, with a propagation delay along all paths of 22 ps; however, the PyLSE model's propagation delay is 25 ps. As a result, for the SPICE model `LOW` = min(`A`, `B`) + 22ps and `HIGH` = max(`A`, `B`) + 22ps, such that the first `LOW` pulse appears at 64 + 22 = 86 ps and first `HIGH` pulse appears at 115 + 22 = 137 ps. The PyLSE model's output pulses appear 3 ps later, e.g. for this first pair, at 89 ps and 140 ps.

The discrepancy comes because the given PyLSE design was created as a pure composition of the individual cells. When combined together in SPICE, however, the entire system exhibits a smaller total propagation delay than what would be assumed from the sum of its parts, due to the parasitic effects mentioned previously. Note that the delay of each individual cell in PyLSE can be tuned, or variability added (see Section 5.5.2), to match the SPICE behavior more closely if desired.

Figure 5.21a (PyLSE) and 5.21b (SPICE) show the waveforms for the 8-input bitonic sorter given inputs 80, 80, 130, 130, 180, 230, 280, 330 ps. The composability issue creeps up here as well: the SPICE model's entire propagation delay is between 100 and 110 ps, while a purely compositional delay would equal the min-max SPICE model's delay (22 ps) multiplied by the depth of the network (6 according to Figure 5.15), i.e. $22 * 6 = 132$ ps. Despite this discrepancy, we can see that the PyLSE design functions correctly according to its own total delay, i.e. $25 * 6 = 150$: here the pulse arriving on input `IN4` (the earliest input pulse) is produced 150 ps later on `OUT0`, and that generally,

the output pulses appear in rank order, behaving correctly. Table 5.2 compares sizes and simulation times of these designs; the PyLSE versions are an average of 16.6× *smaller* than their SPICE counterparts and take several orders of magnitude less time to simulate (average 9879× less). These example simulations demonstrate an important tradeoff: the extremely high accuracy of the analog design level (SPICE) versus the scalability and rapid prototyping of PyLSE.

5.5.2 Simulation and Dynamic Correctness Checks

Harnessing the rich features of Python, we can quickly validate our designs for basic correctness using the `events` dictionary returned from a simulation run; in this section we give some examples. We note that these and similar tests have been performed on all 22 designs from Table 5.3.

2x2 Join The 2x2 Join element is a dual-rail logic primitive that takes in two pairs of inputs, $\texttt{A}_T$ and $\texttt{A}_F$, and $\texttt{B}_T$ and $\texttt{B}_F$, and produces one of four outputs `Q00`, `Q01`, `Q10`, and `Q11` depending on the last pair of $\texttt{A}_*$, $\texttt{B}_*$ inputs seen. This element has been implemented in PyLSE and takes 12 transitions to fully specify. A precondition for this cell to function properly is that $\texttt{A}_T$ and $\texttt{A}_F$ never arrive consecutively without an interleaving $\texttt{B}_T$ or $\texttt{B}_F$ (and vice versa). This can be written succinctly as follows:

```
inputs = sorted(((w, p) for w, evs in events.items()
  for p in evs if w in ('A_T', 'A_F', 'B_T', 'B_F')),
  key=lambda x: x[1])
zipped = list(zip(inputs[0::2], inputs[1::2]))
assert all(x[0] != y[0] for x, y in zipped)
```

The cell behaves correctly if, given this precondition, `Q00` only pulses if $\texttt{A}_F$ and $\texttt{B}_F$ arrived, in any order (similarly for `Q01` being produced only when $\texttt{A}_F$ and $\texttt{B}_T$ arrived,

etc.); this condition is written in a similar fashion.

This can be written as follows:

```
outputs = sorted(((w, p) for w, evs in events.items()
  for p in evs if w in ('Q00', 'Q01', 'Q10', 'Q11')),
  key=lambda x: x[1])
def to_output(*names):
  x, y = sorted(names)
  return {
    ('A_F', 'B_F'): 'Q00', ('A_F', 'B_T'): 'Q01',
    ('A_T', 'B_F'): 'Q10', ('A_T', 'B_T'): 'Q11',
  }[(x, y)]
assert [w[0] for w in outputs] ==\
       [to_output(x[0], y[0]) for x, y in zipped]
```

Race Tree A race tree [180] is a decision tree that uses race logic to produce a label based on a set of internal decision branches. We implemented a race tree in PyLSE by composing 18 basic SFQ cells together in a total of 20 lines of code. A fundamental correctness property of these trees is that one and only one output label pulses for a given set of input pulses. The following assertion encodes this condition using the `events` dictionary of before:

```
assert sum(len(l) for o, l in events.items()
           if o in ('a', 'b', 'c', 'd')) == 1
```

8-input Bitonic Sorter A bitonic sorter is correct if, given a single pulse on each input at an arbitrary time (spaced far enough apart to satisfy transition time constraints), the outputs appear in rank order. This property can be expressed as follows, assuming the first output that should appear is named `00`, followed by `01`, etc. until the last output `0`N for some power-of-two N:

```
out_events = {e for e in events.items()
              if e[0].startswith('o')}
ordered_names = sorted(out_events.keys())
ranked = [es for _, es in sorted(out_events.items(),
          key=lambda x: ordered_names.index(x[0]))]
assert all(len(es) == 1 for es in ranked)
assert all(x[0] <= y[0] for x, y
           in zip(ranked, ranked[1:]))
```

Evaluating Robustness Given Timing Variability In the physical world, the propagation delays of the basic cells vary somewhat from their nominal values due to variability; we saw this in the bitonic sort example of Section 5.5.1, where the SPICE delay varied between 100 and 110 ps. Such variance can lead to pulses arriving at their destination cells too early or late, causing the design to fail unexpectedly. At a PyLSE Machine level, these failures are detected by violations of transition and past constraint times during simulation or by erroneous outputs seen after simulation, and might signify that the network needs to be redesigned to make it less sensitive to variability. PyLSE makes it easy to add variability to existing designs and evaluate their robustness in the presence of these variations; simply pass the flag `variability=True` to `simulate()`. Every individual propagation delay that occurs during the simulation will then have a small amount of delay, by default taken from a Gaussian distribution, added to or subtracted from it. The `variability` argument can be used to specify the cell types or the individual cell instances where the default variability should be added, or it can be set to a user-defined function for even greater fine-tuning.

The following demonstrates how to simulate the 8-input bitonic sorter with added variability. The variation from the original delay of each cell has been capped to +/- 20%.

5.5.3 Model Checking in UPPAAL

Model checking [181] is a formal verification technique used to check that a particular property, typically written in a temporal logic, holds for certain states on a given model of the system. Before it can be used, however, a *model* of the system must be created. Timed Automata is one such model, and as we have shown in Section 5.4, PyLSE can automatically transform PyLSE Machines into a network of communicating Timed Automata; in this way, designs written in PyLSE *are* the models themselves, and immediately amenable to formal verification.

We have chosen to integrate with UPPAAL, a state-of-the-art framework for modeling real-time systems based on TA [156]. The conversion process is straightforward: the PyLSE circuit is traversed, with every transition of every element being converted according to the steps in Figure 5.14 into a network UPPAAL-flavored TA. The result is saved to an XML file, which can then be simulated in UPPAAL or verified against certain properties on the command line via the `verifyta` program their distribution provides.

A Note on Converting Input Sequences Input sequences in PyLSE are lists of timestamps with an associated name; during simulation, a pulse is sent along the wire with the given name at the given time to its destination element. This pattern can be readily converted to a Timed Automaton. For each timestamped input pulse, we create a state and an invariant on the state indicating that time may not pass longer than the time indicated by the timestamp. For transitions, we place a guard indicating time may not pass less than that indicated by the timestamp, and then we issue an output pulse on the prescribed channel. Time may be indicated either in an absolute fashion or by

Table 5.3: Basic cells (first 16 rows) and larger designs (last six rows) implemented in PyLSE. Each has been validated via PyLSE simulation for functional correctness and timing constraint violation detection, and automatically converted into TA that have been simulated and verified in UPPAAL. The PyLSE columns display counts for size, cells, states, and transitions; for basic cells, these are numbers for an individual cell, while for the larger designs, it is the accumulation of every instantiated cell in the network. The size corresponds to the number of transitions in the DSL (roughly equal to the number of lines) for basic cells, and the number of lines for the larger designs. The first four UPPAAL columns are the number of TA, locations, transitions, and channels in the cell's generated TA network, while the latter two columns contain the time to verify the Queries 1 and 2 listed in Section 5.5.3 and the number of total states explored (only one number is listed in each column if the results for Queries 1 and 2 were the same). It took less than 1 second to simulate all of these designs in PyLSE.

Name	PyLSE				UPPAAL						Comparison		
	Size	Cells	States	Tran.	TA	Locs.	Tran.	Chan.	Time (s)	States	TA/Cells	Locs./States	Tran.(U)/Tran.(P)
C	6	1	3	6	2	39	42	3	<1	38	2	13	7
InvC	6	1	3	6	4	45	48	3	<1	69	4	15	8
M	2	1	1	2	2	17	18	3	<1	37	2	17	9
S	1	1	1	1	3	13	13	3	<1	56	3	13	13
JTL	1	1	1	1	2	9	9	2	<1	17	2	9	9
And	11	1	4	12	5	102	110	4	<1	69	5	25.5	9.17
Or	4	1	2	6	2	49	53	4	<1	48	5	24.5	8.83
Nand	12	1	4	12	2	95	103	4	<1	42	2	23.75	8.58
Nor	6	1	2	6	2	49	53	4	<1	36	2	24.5	8.83
Xor	9	1	3	9	3	75	81	4	<1	45	3	25	9
Xnor	12	1	4	12	2	94	102	4	<1	45	2	23.5	8.5
Inv	4	1	2	4	3	30	32	3	<1	14	3	15	8
DRO	4	1	2	4	2	27	29	3	<1	11	2	13.5	7.25
DRO SR	6	1	2	6	2	49	53	4	<1	23	2	24.5	8.83
DRO C	4	1	2	4	3	31	33	4	<1	14	3	15.5	8.25
2x2 Join	12	1	5	20	5	206	221	8	<1	58	5	41.2	11.05
Min-Max	5	5	9	15	24	149	155	14	<1	2471	4.8	16.56	10.33
Race Tree	16	18	32	56	50	440	464	54	127/84	262559	2.78	13.75	8.29
Adder (Sync)	13	19	33	71	57	627	665	62	669/515	1858153	3	19	9.37
Adder (xSFQ)	31	83	121	183	193	1449	1511	211	∞	N/A	2.33	11.98	8.26
Bitonic Sort 4	6	30	54	90	144	894	930	84	∞	N/A	4.8	16.56	10.33
Bitonic Sort 8	24	120	216	360	576	3576	3720	336	∞	N/A	4.8	16.56	10.33

duration since the last pulse, since clocks in UPPAAL are resettable. Typically, with well-designed circuits, this will permit only one possible enabled transition, which matches the determinism found in PyLSE.

Query 1: Correctness To verify that our translation process works, we automatically converted all 16 basic cells and six larger designs into UPPAAL, as shown in Table 5.3, where we note the resulting size of the TA network. Once in UPPAAL, we checked that their internal simulator agrees with ours from an input/output perspective. We also automatically generate a correctness formula in UPPAAL-flavored timed computation tree logic (TCTL) [182, 183] for each, based on a given PyLSE simulation's events, to formally verify that the given design generates the expected output. For example, here is a PyLSE-generated TCTL formula for the correctness of min-max pair, given pulses on `A` at 115, 215, and 315, on `B` at 64, 184, and 304, and a network delay of 25 ps:

```
A[] (((firingauto3.fta_end imply ((global == 890) ||
       (global == 2090) || (global == 3290))) &&
      (firingauto4.fta_end imply ((global == 890) ||
       (global == 2090) || (global == 3290))) &&
      (firingauto5.fta_end imply ((global == 890) ||
       (global == 2090) || (global == 3290)))) &&
      ((firingauto12.fta_end imply ((global == 1400) ||
       (global == 2400) || (global == 3400)))))
```

At the top of this formula, *A* is a path quantifier that expresses "for all subsequent time points", while [] is a branch quantifier meaning "for all possible branches." The `firingauto*` correspond to firing TA instances, and `fta_end` is the location in that instance that immediately follows sending a fire message to a particular network output sink. As many firing TA may be associated with each network output (see Figure 5.14d), there are multiple states to check for each time. This says that it is only possible to produce a pulse at the given output at the given time. These times have been upscaled

to integers to meet the requirements UPPAAL places on numbers involved in clock constraints; thus `global` == 2090 correspond to a simulated time of 209.0 ps in PyLSE.

As another example, here is PyLSE-generated TCTL formula for correctness for the 8-input bitonic sorter, given inputs occurring at times 230, 130, 180, 280, 80, 80, 130, and 330 ps on inputs `I0` through `I7`, respectively, and a network delay of 150 ps:

```
A[] (((firingauto185.fta_end imply ((global == 2300))) &&
      (firingauto186.fta_end imply ((global == 2300))) &&
      (firingauto187.fta_end imply ((global == 2300)))) &&
     ((firingauto333.fta_end imply ((global == 2300)))) &&
     ((firingauto107.fta_end imply ((global == 2800))) &&
     ...9 more lines...
      (firingauto263.fta_end imply ((global == 4300)))) &&
     ((firingauto321.fta_end imply ((global == 4800)))))
```

In Table 5.3, we also show the time it took to verify this property (customized to each cell). For the basic cells and the min-max pair, verification consistently took less than 1 second. The race tree, with 440 locations, took 127 seconds and explored 262559 states, while the synchronous full adder, with nearly 43% more locations, took 669 seconds (5.26×) and visited 7.077× more states. Model checking becomes infeasible due to the state explosion as we reach the bitonic sorters and xSFQ [151] full adder, which failed to finish in a day. Table 5.3 also shows how much larger the network of TA is compared to the original PyLSE Machines. On average, each cell (i.e. PyLSE Machine) requires 3.02 UPPAAL TA, each PyLSE Machine state requires 18.99 UPPAAL locations, and each PyLSE Machine transition requires 9.05 UPPAAL transitions.

Query 2: Unreachable Error States Our translation process inserts error states that are entered when transition time or past constraint violations occur (for example, $\texttt{err}_{Ah}$ and $\texttt{err}_{As}$, respectively, from Figure 5.14). Since these states have no outgoing edges, they cannot respond to additional input nor allow time to pass and so are *terminal*. Entering such a state would deadlock the TA, and verifying that no deadlock occurs (i.e.

`A[] not deadlock`) would normally be sufficient to show that the inputs to a design meet timing constraints. Unfortunately, this form of deadlock detection is not useful for our purposes, since "good" deadlock also occurs when the sequence of user-defined inputs has been exhausted and no more cells can progress. Instead, we automatically generate an UPPAAL verification query that checks that it is impossible to reach any error state in the network:

```
A[] not (c0.C_err_a_1 || c0.C_err_a_11 || c0.C_err_a_16 ||
         ...18 more lines...
         c_inv0.C_INV_err_b_8 || c_inv0.C_INV_err_b_9 ||
         s0.S_err_a_1 || s0.S_err_a_2 || jtl0.JTL_err_a_1 ||
         jtl0.JTL_err_a_2 || s1.S_err_a_1 || s1.S_err_a_2)
```

UPPAAL explores the same number of states as for Query 1 in under one second for all basic cells, with the larger designs similarly encountering exponential blowup difficulties. If the above property is not satisfied, UPPAAL will return a trace showing the path that led to the particular error state.

Likewise, here is the query we automatically generate for the 8-input bitonic sorter:

```
A[] not
  (jtl0.JTL_err_a_1 || jtl0.JTL_err_a_2 ||
   jtl1.JTL_err_a_1 || jtl1.JTL_err_a_2 ||
   c0.C_err_a_1 || c0.C_err_a_11 ||
   c0.C_err_a_16 || c0.C_err_a_21 ||
   c0.C_err_a_26 || c0.C_err_a_6 ||
   ...
   s47.S_err_a_1 || s47.S_err_a_2)
```

As of this writing, additional properties must be explicitly written out in UPPAAL's DSL for expressing TCTL formulas. As far as we know, we are the first to use *timed automata-based model checking* to check the correctness of SFQ circuits.

5.6 Related Work

Existing HDLs Existing HDLs like Verilog [106] model the timing constraints of SCE by coupling asynchronously-updated registers to latch incoming signals with complicated series of conditionals to track whether these constraints are satisfied [184, 166, 185, 186]. Designs using this approach have many downsides:

- They tend to be extremely verbose, spanning tens to hundreds of lines per cell module; for example, in [187], 90 lines of codes were needed to model a destructive readout (DRO) cell, while the PyLSE Machine equivalent takes four lines. Similarly, a model of the OR cell in [154] takes 18 lines of Verilog.
- A number of ambiguous internal signals must be generated for synchronization purposes; for example, for the implementation of said DRO cell, five edge-triggered `always` blocks and three artificial synchronization signals were required.
- There are no clear boundaries between functional and timing specification, leading to obfuscated code and an enlarged surface for programming bugs.
- They rely on the peculiar semantics of Verilog or the chosen simulator, instead of being based on a suitable formal foundation.

Recent approaches [188, 189] are more modular and compact, but the resemblance of their proposed coding scheme to multithreaded socket programming raises the barrier to entry and again makes them more prone to bugs. Finally, [190, 191, 192, 175] automatically extract state machine models and timing characteristics of SFQ cells from SPICE files, but in the end, still use them to generate Verilog HDL code that must be integrated with the rest of the user-coded design.

Verification There have been many attempts at formally checking the correctness of SCE designs at the HDL level. Recent work [170] uses a delay-based time frame model, which assumes that pulses arrive periodically according to a unique clock period. This assumption allows them to discretize the behavior of these pulse-based systems into a verifiable synchronous model. PyLSE instead makes no requirements about clock periodicity and is able to model systems that include asynchronous cells or no clock. VeriSFQ [193] is semi-formal verification framework that uses UVM [194] to check that their designs are properly path-balanced, have correct fanout, and that all synchronous SFQ gates have a clock signal. PyLSE, on the other hand, is an entirely new DSL for SCE design, statically preventing the creation of designs with these basic issues, and so a formal framework for checking them is unneeded. qMC [154] develops a framework that uses SMT-based model checkers to check the correct functionality of post-synthesis netlists via SystemVerilog assertions. However, their gate models do not include information on hold or setup times or propagation delay, such that they assume that pulses are delayed by one cycle. PyLSE instead represents and model checks against these timing constraints via a Timed Automata-based model checker like UPPAAL.

5.7 Conclusion

We present PyLSE, a language for the design and simulation of pulse-based systems like superconductor electronics (SCE). PyLSE simplifies the process of precisely defining the functional and timing behavior of SCE cells using a new transition-system based abstraction we call the PyLSE Machine. It facilitates a multi-level approach to design, making it easy to compose basic cells alongside abstract design models and create large, scalable SCE systems. As an alternative to Verilog, we argue that it is

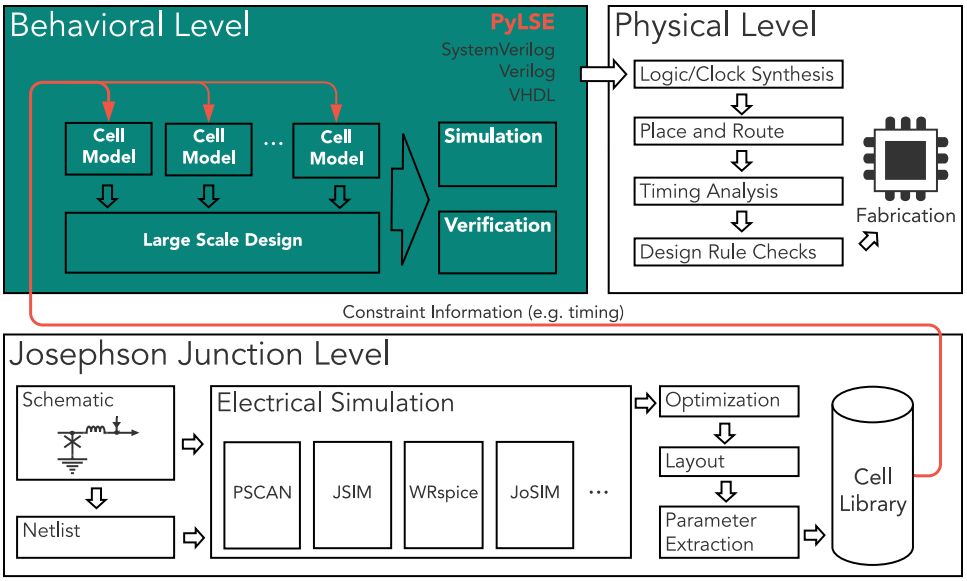


Figure 5.22: PyLSE's place in the superconductor electronics design flow. PyLSE is used to model, simulate, and verify SFQ cells at the *Behavioral Level*, using timing information from the *Josephson Junction Level*, similar to approaches using Verilog and System Verilog.

an effective behavioral-level modeling framework that can fit within SCE design flow (see Figure 5.22). We evaluate PyLSE by simulating and dynamically checking the correctness of 22 different designs, comparing these simulations against analog SPICE models, and verifying their timing constraints using the UPPAAL model checker. Compared to SPICE, PyLSE designs take 16.6× fewer lines code and take several orders of magnitude less time to simulate, all while maintaining the needed level of timing accuracy. Compared with specification directly as Timed Automata, PyLSE requires 18.9× fewer states and 9.0× fewer transitions. We believe, with the end of traditional transistor scaling, pulse-based logic systems will only continue to grow in importance. PyLSE, with its expressive timing, composable abstractions, and connection to well understood theory, has the potential to provide a new foundation for that growth for years to come.

Chapter 6

Conclusions and Future Work

The correctness of the programs we write ultimately depends on the correctness of the machines on which they run. In this thesis, I have shown that programming language principles can guide the design of more precise, verifiably correct, and secure ISAs and HDLs, resulting in hardware with better correctness guarantees. In summary:

1. Provable correctness and ease-of-reasoning deserve a place alongside performance and efficiency as *first-class goals* in the computer architecture design space.

2. Programming language principles that have traditionally benefited the software world can be applied toward and improve hardware design and IP integration.

3. Emerging technologies like SCE should be modeled and designed using new HDLs based on mathematical formalism like automata theory.

This work can be extended in the future in many ways. Practically, Zarf might find more widespread use if an operating system was developed for it; the process of making one may reveal fundamental aspects that need to change to make it POSIX-compliant, for example. In addition, I want to formally verify that the Zarf microarchitecture implements the ISA and apply additional programming language techniques like linear

types towards reducing Zarf's dependence on garbage collection. At the HDL level, I want to integrate Wire Sorts into other HDLs, like Verilog or Chisel, and experiment with applying more type-theoretic approaches to improving the composability of IP. My experience using and augmenting PyRTL has given me many ideas toward new and improved HDL design, in addition to a list of improvements I'd like to make in PyRTL itself. Finally, I want to further develop the PyLSE ecosystem so that it can integrate more fully with existing EDA workflows and become a transformative and impactful language in the SCE landscape.

Appendix A

Zarf and Bouncer

A.1 Small-Step Semantics

To create an accurate Coq interpreter that behaves similar to our ISA's implementation, we use an abstract machine small-step semantics description of Zarf's functional ISA, as described in Chapter 2 In the following sections we define its abstract syntax, domains, and transition rules.

$$n \in \mathbb{Z} \quad PC \in \mathbb{N}$$

$$\begin{aligned}
P \in \mathit{Program} &::= \overrightarrow{it}\ \overrightarrow{pw} \\
it \in \mathit{ITable} &::= \textbf{FunTable}\ n\ n\ PC \mid \textbf{ConTable}\ n \\
pw \in \mathit{ProgramWord} &::= \textbf{let}\ src\ n\ n \mid \textbf{case}\ src\ n \mid \textbf{result}\ src\ n \mid \\
&\quad\ \ \textbf{patlit}\ n\ n \mid \textbf{patcons}\ n\ n \mid \textbf{arg}\ src\ n \\
src \in \mathit{Source} &::= \textbf{LiteralSrc} \mid \textbf{ArgSrc} \mid \textbf{LocalSrc} \mid \textbf{FieldSrc} \mid \textbf{ITableSrc}
\end{aligned}$$

Figure A.1: Abstract syntax for the small-step semantics of Zarf's functional ISA.

$$i, \iota \in \mathbb{N} \quad \oplus \in \mathit{PrimOp} \quad \theta, \alpha, a \in \mathit{Address} = \mathbb{N}$$

$$\begin{aligned}
c \in \mathit{Constructor} &= \textbf{Con}\ \mathbb{N}\ \overrightarrow{\mathit{Wrapper}} \\
t \in \mathit{Thunk} &= \textbf{Thunk}\ \mathbb{N} \\
v \in \mathit{Value} &= \mathit{Constructor}\ +\ \mathit{Thunk}\ +\ \mathbb{Z} \\
\epsilon, \mathit{arg} \in \mathit{Wrapper} &= \mathit{Address}\ +\ \mathbb{Z} \\
\kappa \in \mathit{Kont} &= \textbf{caseK}\ \mathit{Address}\ \mathit{Environment}\ \mathbb{N}\ \mathit{PC}\ + \\
&\qquad \textbf{argsK}\ \overrightarrow{\mathit{Wrapper}}\ + \\
&\qquad \textbf{updateK}\ \mathit{Address}\ + \\
&\qquad \textbf{rightK}\ \mathbb{N}\ \mathit{Address}\ \mathcal{O}(\mathit{Wrapper})\ + \\
&\qquad \textbf{leftK}\ \mathbb{N}\ \mathit{Address}\ \mathit{Wrapper} \\
\rho \in \mathit{Environment} &= \mathbb{N} \to \mathit{Address} \\
\sigma \in \mathit{Store} &= \mathit{Address} \to \mathit{Value} \\
\Lambda \in \mathit{ITableMap} &= \mathbb{N} \to (\mathit{ITable} + \mathit{PrimOp}) \\
\Pi \in \mathit{ProgramMap} &= \mathbb{N} \to \mathit{ProgramWord}
\end{aligned}$$

Figure A.2: Semantic domains for the small-step semantics of Zarf's functional ISA.

A.1.1 Small-Step Abstract Syntax

Figure A.1 defines the abstract syntax for the small-step semantics of Zarf. A program is composed of a list of information tables (itables), which are either functions or constructors, and a list of 32-bit program words, which contain the instructions that define the bodies of the function itables. Like the big-step abstract syntax presented in Figure 2.2, there are similar constructs like the **let**, **case**, and **result** program words. **patlit** and **patcons** are program words corresponding to branches, and one of more **arg** words are used when the preceding instruction needs an argument (i.e. a function call).

A.1.2 Small-Step Semantics Domains

Figure A.2 defines the domains over which the small-step semantics operate. All functions and constructors (*ITable*) are uniquely numbered and put in an *ITableMap*. We define various continuations (*Kont*) needed for tracking the current execution context as well as the notion of an environment and store for use in the transition rules defined below.

A.1.3 Small-Step State Transition Rules

Execution begins by setting the initial program state to `initState`. The itable map (Λ) and program words map (Π) are also part of the program state but, since they do not change during the course of execution, are elided from the state transition rules as defined below for simplicity. Notably, the first itable in the program's $\overrightarrow{it}$ list is always assumed to be the program's entry point (the `main` function) and is assigned itable number 256 in the itable map (Λ); builtin functions have itable numbers 0 through 255, and user-defined functions are assigned unique numbers greater than 256. Upon termination, the program's final value will be found in the heap at address `0x0`.

The following table formally defines the transition function $\mathcal{F}$. Each rule describes a translation relation from one state.

A.1.4 Small-Step Semantics Helper Functions

The following describe several helper functions used in the state transition functions above. We use the notation $\overrightarrow{arg}[i]$ to represent getting the i^{th} element of the $\overrightarrow{arg}$ list. The notation $[\![\oplus, n]\!]$ and $[\![\oplus, n_1, n_2]\!]$ represents performing a unary or binary primitive operation $\oplus$ on one or two arguments, respectively. The function `builtins` is a map that associates ITable numbers (less than 256) with primitive operations.

Table A.1: Small-step state transition rules of Zarf's functional ISA.

Rule	Premises	PC_{new}	$O(\theta_{new})$	ϵ_{new}	ρ_{new}	ι_{new}	σ_{new}	α_{new}	κ
LetPrim	$\theta \neq \bullet, \Pi(PC) =$ **let LiteralSrc** n _	$PC+1$	θ	ϵ	$\rho[\iota \mapsto n]$	$\iota+1$	σ	α	$\vec{\kappa}$
StaticApp	$\theta \neq \bullet, \Pi(PC) =$ **let ITableSrc** $i\ n$, $\overrightarrow{arg} = \texttt{getArgs}(PC+1, n)$	$PC+n+1$	θ	ϵ	$\rho[\iota \mapsto \alpha]$	$\iota+1$	$\sigma[\alpha \mapsto$ **Thunk** $i\ \overrightarrow{arg}]$	$\alpha+1$	$\vec{\kappa}$
LetAssign	$\theta \neq \bullet, \Pi(PC) =$ **let** $src\ i\ 0$, $src \neq$ **LiteralSrc**, $src \neq$ **ITableSrc**, $arg = \texttt{getSrc}(src, i)$,	$PC+1$	θ	ϵ	$\rho[\iota \mapsto arg]$	$\iota+1$	σ	α	$\vec{\kappa}$
ThunkApp	$\theta \neq \bullet, \Pi(PC) =$ **let** $src\ i\ n$, $src \neq$ **LiteralSrc**, $src \neq$ **ITableSrc**, $n > 0$, $a = \texttt{getSrc}(src, i)$, **Thunk** $i'\ \overrightarrow{arg}' = \sigma(a), \overrightarrow{arg} = \texttt{getArgs}(PC+1, n)$	$PC+n+1$	θ	ϵ	$\rho[\iota \mapsto \alpha]$	$\iota+1$	$\sigma[\alpha \mapsto$ **Thunk** $i'\ (\overrightarrow{arg}' +\!\!+ \overrightarrow{arg})]$	$\alpha+1$	$\vec{\kappa}$
Case	$\theta \neq \bullet, \Pi(PC) =$ **case** $src\ i, \texttt{getSrc}(src, i) = \epsilon'$	PC	$\bullet$	ϵ'	$\{\}$	0	σ	α	**caseK** $\theta\ \rho\ \iota\ (PC+1) :: \vec{\kappa}$
Result	$\theta \neq \bullet, \Pi(PC) =$ **result** $src\ i, \texttt{getSrc}(src, i) = \epsilon'$	PC	$\bullet$	ϵ'	$\{\}$	0	σ	α	$\vec{\kappa}$
PatLit1	$\theta \neq \bullet, \Pi(PC) =$ **patlit** $i\ n, i = \eta(\epsilon)$	$PC+1$	θ	ϵ	ρ	ι	σ	α	$\vec{\kappa}$
PatLit2	$\theta \neq \bullet, \Pi(PC) =$ **patlit** $i\ n, i \neq \eta(\epsilon)$	$PC+n$	θ	ϵ	ρ	ι	σ	α	$\vec{\kappa}$
PatCons1	$\theta \neq \bullet, \Pi(PC) =$ **patcons** $i\ n$, **Con** $i\ \overrightarrow{arg} = \eta(\epsilon)$	$PC+1$	θ	ϵ	ρ	ι	σ	α	$\vec{\kappa}$
PatCons2	$\theta \neq \bullet, \Pi(PC) =$ **patcons** $i\ n$, **Con** $i\ \overrightarrow{arg} \neq \eta(\epsilon)$	$PC+n$	θ	ϵ	ρ	ι	σ	α	$\vec{\kappa}$
ThunkUnd1	$\theta = \bullet$, **Thunk** $i\ \overrightarrow{arg} = \eta(\epsilon)$, $\lvert\overrightarrow{arg}\rvert < \texttt{arity}(i)$, **argsK** $\overrightarrow{arg}' :: \vec{\kappa}' = \vec{\kappa}$	PC	θ	α	ρ	ι	$\sigma[\alpha \mapsto$ **Thunk** $i\ (\overrightarrow{arg} +\!\!+ \overrightarrow{arg}')]$	$\alpha+1$	$\vec{\kappa}'$
ThunkUnd2	$\theta = \bullet$, **Thunk** $i\ \overrightarrow{arg} = \eta(\epsilon)$, $\lvert\overrightarrow{arg}\rvert < \texttt{arity}(i)$, **updateK** $a :: \vec{\kappa}' = \vec{\kappa}$	PC	θ	α	ρ	ι	$\sigma[a \mapsto$ **Thunk** $i\ \overrightarrow{arg}]$	α	$\vec{\kappa}'$
ThunkOver	$\theta = \bullet, a = \epsilon$, **Thunk** $i\ \overrightarrow{arg} = \sigma(a)$, $\lvert\overrightarrow{arg}\rvert > \texttt{arity}(i)$, **FunTable** $n_1\ n_2\ PC' = \Lambda(i)$	PC'	a	ϵ	$\{\}$	0	σ	α	**argsK** $\texttt{drop}(n_1, \overrightarrow{arg}) ::$ **updateK** $a :: \vec{\kappa}$
ThunkPrim1	$\theta = \bullet, a = \epsilon$, **Thunk** $i\ (arg_1 :: []) = \sigma(a)$, $\lvert\overrightarrow{arg}\rvert = \texttt{arity}(i) = 1$	PC	θ	arg_1	ρ	ι	σ	α	**rightK** $i\ a \bullet :: \vec{\kappa}$
ThunkPrim2	$\theta = \bullet, a = \epsilon$, **Thunk** $i\ (arg_1 :: arg_2 :: []) = \sigma(a)$, $\lvert\overrightarrow{arg}\rvert = \texttt{arity}(i) = 2$	PC	θ	arg_1	ρ	ι	σ	α	**leftK** $i\ a\ arg_2 :: \vec{\kappa}$
ThunkCons	$\theta = \bullet$, **Thunk** $i\ \overrightarrow{arg} = \eta(\epsilon)$ $\lvert\overrightarrow{arg}\rvert = \texttt{arity}(i)$, **ConTable** $n = \Lambda(i)$	PC	θ	α	ρ	ι	$\sigma[\alpha \mapsto$ **Con** $i\ \overrightarrow{arg}]$	$\alpha+1$	$\vec{\kappa}$
ThunkExact	$\theta = \bullet, a = \epsilon$, **Thunk** $i\ \overrightarrow{arg} = \sigma(a)$ $\lvert\overrightarrow{arg}\rvert = \texttt{arity}(i)$, **FunTable** _ _ $PC' = \Lambda(i)$	PC'	a	ϵ	$\{\}$	0	σ	α	**updateK** $a :: \vec{\kappa}$
EndCaseK	$\theta = \bullet, \eta(\epsilon) = n \vee \eta(\epsilon) =$ **Con** _ _, **caseK** $a\ \rho'\ \iota'\ PC' :: \vec{\kappa}' = \vec{\kappa}$	PC'	a	ϵ	ρ'	ι'	σ	α	$\vec{\kappa}'$
LeftK	$\theta = \bullet, \eta(\epsilon) = n \vee \eta(\epsilon) =$ **Con** _ _, **leftK** $i\ a\ arg :: \vec{\kappa}' = \vec{\kappa}$	PC	θ	arg	ρ	ι	σ	α	**rightK** $i\ a\ \epsilon :: \vec{\kappa}'$
RightK1	$\theta = \bullet, \eta(\epsilon) = n$, **rightK** $i\ a \bullet :: \vec{\kappa}' = \vec{\kappa}$, $\oplus = \Lambda(i), \texttt{arity}(\oplus) = 1$	PC	θ	a	ρ	ι	$\sigma[a \mapsto [\![\oplus, n]\!]]$	α	$\vec{\kappa}'$
RightK2	$\theta = \bullet, \eta(\epsilon) = n_2$, **rightK** $i\ a\ arg :: \vec{\kappa}' = \vec{\kappa}$, $\oplus = \Lambda(i), \texttt{arity}(\oplus) = 2, \eta(arg) = n_1$	PC	θ	a	ρ	ι	$\sigma[a \mapsto [\![\oplus, n_1, n_2]\!]]$	α	$\vec{\kappa}'$
Update1	$\theta = \bullet, \epsilon = n$, **updateK** $a :: \vec{\kappa}' = \vec{\kappa}$	PC	θ	ϵ	ρ	ι	$\sigma[a \mapsto n]$	α	$\vec{\kappa}'$
Update2	$\theta = \bullet, \epsilon = a'$, **updateK** $a :: \vec{\kappa}' = \vec{\kappa}$	PC	θ	ϵ	ρ	ι	$\sigma[a \mapsto \sigma(a')]$	α	$\vec{\kappa}'$

`getArgs`

`getArgs` returns a list of the n argument words found starting at address PC.

$$\texttt{getArgs} \in PC \times \mathbb{N} \to \overrightarrow{Wrapper}$$

$$\texttt{getArgs}(PC, n) = \begin{cases} [] & \text{if } n = 0 \\ arg :: \texttt{getArgs}(PC, n+1) & \text{otherwise} \end{cases}$$

where

$$\textbf{arg}\ src\ n = \Pi(PC)$$

$$arg = \texttt{getSrc}(src, n)$$

`getSrc`

`getSrc` interprets the *Source* encoding to get a literal value stored in i, a local variable stored in the current environment, a thunk argument, or a constructor field.

$$\texttt{getSrc} \in Source \times \mathbb{N} \to \overrightarrow{Wrapper}$$

$$\texttt{getSrc}(src, i) = \begin{cases} i & \text{if } src = \textbf{LiteralSrc} \\ \overrightarrow{arg}[i] & \text{if } src = \textbf{ArgSrc}, \textbf{Thunk}\ \_\ \overrightarrow{arg} = \theta \\ \rho(i) & \text{if } src = \textbf{LocalSrc} \\ \overrightarrow{arg}[i] & \text{if } src = \textbf{FieldSrc}, \textbf{Con}\ \_\ \overrightarrow{arg} = \epsilon \end{cases}$$

`arity`

`arity` returns the arity of primitive operations, constructors, and user-defined functions, based on a given itable number.

$$\texttt{arity} \in \mathbb{N} \to \mathbb{N}$$

$$\texttt{arity}(i) = \begin{cases} i_1 & \text{if } i < 256 \\ i_2 & \text{otherwise} \end{cases}$$

where

$$i_1 = \begin{cases} 0 & \text{if } \texttt{builtins}(i) \in \{\textbf{error}\} \\ 1 & \text{if } \texttt{builtins}(i) \in \{\neg, \textbf{getint}\} \\ 2 & \text{if } \texttt{builtins}(i) \in \{+, -, \times, \div, =, \leq, \wedge, \vee, \bar{\wedge}, \bar{\vee}, \hat{}, \ll, \gg, \ggg, <, \textbf{putint}\} \end{cases}$$

$$i_2 = \begin{cases} n & \text{if } \textbf{ConTable } n = \Lambda(i) \\ n & \text{if } \textbf{FunTable } n \_ \_ = \Lambda(i) \end{cases}$$

η

η is a convenience function for getting the value associated with the current evaluation object. If it is a number, it simply returns the number; otherwise it looks up the address in the store.

$$\eta \in \mathit{Wrapper} \to \mathit{Value}$$

$$\eta(\epsilon) = \begin{cases} n & \text{if } n = \epsilon \\ \sigma(a) & \text{if } a = \epsilon \end{cases}$$

`initState`

`initState` creates the initial state used for beginning execution. The store is initialized with a mapping from address `0x0` to the thunk representing the main function, and the next usable store address is set to `0x1`.

$$c \in \mathit{Constructor} = \textbf{Con}\ \mathit{Name}\ \overrightarrow{\mathit{Wrapper}} \quad t \in \mathit{Thunk} = \textbf{Thunk}\ \mathit{Operator}\ \overrightarrow{\mathit{Wrapper}}$$
$$v \in \mathit{Value} = \mathbb{Z}\ +\ \mathit{Constructor}\ +\ \mathit{Thunk}$$
$$a \in \mathit{Address} = \mathbb{N} \quad w \in \mathit{Wrapper} = \mathit{Address}\ +\ \mathbb{Z}$$
$$\rho \in \mathit{Environment} = \mathit{Variable} \to \mathit{Wrapper} \quad \sigma \in \mathit{Store} = \mathit{Address} \to \mathit{Value}$$

Figure A.3: Semantic domains for the big-step semantics

$$\texttt{initState} \in \langle \mathbb{N}, \mathcal{O}(\mathit{Address}), \mathit{Wrapper}, \mathit{Environment}, \mathbb{N}, \mathit{Store}, \mathit{Address}, \overrightarrow{\mathit{Kont}} \rangle$$

$$\texttt{initState} = \langle 0, \bullet, \texttt{0x0}, \{\}, 0, \{\texttt{0x0} \mapsto \textbf{Thunk}\ 256\ []\}, \texttt{0x1}, [] \rangle$$

A.2 Big-Step Lazy Semantics for Typed Zarf

The following sections define the big-step lazy semantics for the type-extended Zarf functional ISA found in Chapter 3.

A.2.1 Big-Step Dynamic Semantics Domains

Figure A.3 shows the semantics domains for the big-step lazy semantics of Zarf. There are three possible values: an integer, a constructor (i.e. a data type instance), and a thunk. The primary way in which these domains differ from the eager domains presented in Figure 2.5 is via the use of the thunk, stores a function and its arguments, unapplied, until needed by a **case** statement.

A.2.2 Big-Step Dynamic Semantics Rules

Figure A.4 defines the big-step lazy semantics of the Zarf ISA. The primary difference between these semantics and those found in Figure 2.5 is that in **let** instructions,

$$\frac{\rho[x \mapsto n], \sigma \vdash e \Downarrow (v, \sigma')}{\rho, \sigma \vdash \textbf{let}\ x = n\ \textbf{in}\ e \Downarrow (v, \sigma')}\ \text{(LET-INT-1)}$$

$$\frac{\texttt{varLookup}(\rho, \sigma, x_2) = n \quad \rho[x_1 \mapsto n], \sigma \vdash e \Downarrow (v, \sigma')}{\rho, \sigma \vdash \textbf{let}\ x_1 = x_2\ []\ \textbf{in}\ e \Downarrow (v, \sigma')}\ \text{(LET-INT-2)}$$

$$\frac{\begin{array}{c}a \text{ is a fresh address} \quad \rho' = \rho[x \mapsto a] \quad (\vec{w}, \sigma') = \texttt{getArgs}(\overrightarrow{arg}, \rho', \sigma) \\ \sigma'' = \sigma'[a \mapsto \textbf{Thunk}\ op\ \vec{w}] \qquad \rho', \sigma'' \vdash e \Downarrow (v, \sigma''')\end{array}}{\rho, \sigma \vdash \textbf{let}\ x = op\ \overrightarrow{arg}\ \textbf{in}\ e \Downarrow (v, \sigma''')}\ \text{(LET-CON-FUN)}$$

$$\frac{\rho(x_2) = a \quad \sigma(a) = \textbf{Con}\ \_\ \_ \quad \rho[x_1 \mapsto a], \sigma \vdash e \Downarrow (v, \sigma')}{\rho, \sigma \vdash \textbf{let}\ x_1 = x_2\ []\ \textbf{in}\ e \Downarrow (v, \sigma')}\ \text{(LET-VAR-1)}$$

$$\frac{\begin{array}{ccc}a \text{ is a fresh address} & \texttt{varLookup}(\sigma, \rho, x_2) = \textbf{Thunk}\ op\ \vec{w}_1 & \rho' = \rho[x_1 \mapsto a] \\ (\vec{w}_2, \sigma') = \texttt{getArgs}(\overrightarrow{arg}, \rho', \sigma) & \sigma'' = \sigma'[a \mapsto \textbf{Thunk}\ op\ \vec{w}_1 +\!\!+ \vec{w}_2] & \rho', \sigma'' \vdash e \Downarrow (v, \sigma''')\end{array}}{\rho, \sigma \vdash \textbf{let}\ x_1 = x_2\ \overrightarrow{arg}\ \textbf{in}\ e \Downarrow (v, \sigma''')}\ \text{(LET-VAR-2)}$$

$$\frac{\begin{array}{c}a = \rho(x) \quad \texttt{valueOf}(\sigma(a), \sigma) = (\textbf{Con}\ cn\ \vec{w}, \sigma') \quad \sigma'' = \sigma'[a \mapsto \textbf{Con}\ cn\ \vec{w}] \\ (cn\ \vec{x} \Rightarrow e_1) \in \overrightarrow{br} \qquad \rho' = \rho[\vec{x} \mapsto \vec{w}] \qquad \rho', \sigma'' \vdash e_1 \Downarrow (v, \sigma''')\end{array}}{\rho, \sigma \vdash \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow (v, \sigma''')}\ \text{(CASE-CON)}$$

$$\frac{\begin{array}{c}a = \rho(x) \quad \texttt{valueOf}(\sigma(a), \sigma) = (\textbf{Con}\ cn\ \vec{w}, \sigma') \quad \sigma'' = \sigma'[a \mapsto \textbf{Con}\ cn\ \vec{w}] \\ (cn\ \vec{x} \Rightarrow e_1) \notin \overrightarrow{br} \qquad \rho, \sigma'' \vdash e_2 \Downarrow (v, \sigma''')\end{array}}{\rho, \sigma \vdash \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow (v, \sigma''')}\ \text{(CASE-CON-ELSE)}$$

$$\frac{\begin{array}{c}((n = \rho(x)) \quad \vee \quad (a = \rho(x) \quad \texttt{valueOf}(\sigma(a), \sigma) = (n, \sigma') \quad \sigma'' = \sigma'[a \mapsto n])) \\ (n \Rightarrow e_1) \notin \overrightarrow{br} \qquad \rho, \sigma'' \vdash e_2 \Downarrow (v, \sigma''')\end{array}}{\rho, \sigma \vdash \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow (v, \sigma''')}\ \text{(CASE-LIT)}$$

$$\frac{\begin{array}{c}((n = \rho(x)) \quad \vee \quad (a = \rho(x) \quad \texttt{valueOf}(\sigma(a), \sigma) = (n, \sigma') \quad \sigma'' = \sigma'[a \mapsto n])) \\ (n \Rightarrow e_1) \in \overrightarrow{br} \qquad \rho, \sigma'' \vdash e_1 \Downarrow (v, \sigma''')\end{array}}{\rho, \sigma \vdash \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow (v, \sigma''')}\ \text{(CASE-LIT-ELSE)}$$

$$\frac{v = \texttt{varLookup}(\rho, \sigma, x)}{\rho, \sigma \vdash \textbf{result}\ x \Downarrow (v, \sigma)}\ \text{(RESULT-1)} \qquad \frac{}{\rho, \sigma \vdash \textbf{result}\ n \Downarrow (n, \sigma)}\ \text{(RESULT-2)}$$

Figure A.4: Big-step lazy semantics of Zarf's functional ISA.

the function application is postponed by storing the function and argument identifiers in a thunk, which is evaluated as needed via the `valueOf` helper during **case** instructions. Note that when a **case** expression does not include an **else** expression (i.e. rules CASE-CON and CASE-LIT), the list of branches must contain a match (which is checked statically). Execution of these forms of **case** expressions proceeds like CASE-CON-ELSE and CASE-LIT-ELSE.

A.2.3 Big-Step Dynamic Semantics Helper Functions

The following section defines the helper functions used in Figure A.4. We omit formalization of those helpers with straightforward definitions.

`getArgs`

`getArgs` folds over the list of arguments to return a list of the addresses associated with those arguments; each literal argument is assigned an address that maps to the semantic value of that literal in the newly-returned store.

`interpret`

`interpret`(e, ρ, σ) is shorthand for $\rho, \sigma \vdash e$, which evaluates to a (v, σ') tuple per the big-step operational rules listed in Figure A.4.

`paramsOf`

`paramsOf` consults the list of function declarations to extract the parameter names of a function given its name.

`bodyOf`

`bodyOf` consults the list of function declarations to extract the body of a function given its name.

`arity`

`arity` consults the list of constructor and function declarations and builtin operators associated with a given name and returns the number of fields or parameters it accepts, respectively.

`eval`

`eval` applies a thunk's primitive operator (e.g. + or $*$) to the thunk's numeric arguments, returning a number and a new store (such that this new store remembers the values of any postponed thunks that had to be evaluated during the primitive application) as a result.

`varLookup`

`varLookup` looks up a variable in the environment, returning its entry if it maps to an integer. If it maps to an address, it looks up the address in the store.

$$\texttt{varLookup} \in \mathit{Environment} \times \mathit{Store} \times \mathit{Variable} \to \mathit{Value}$$

$$\texttt{varLookup}(\rho, \sigma, x) = \begin{cases} n & \text{if } \rho(x) = n \\ \sigma(a) & \text{if } \rho(x) = a \end{cases}$$

`valueOf`

`valueOf` reduces a fully- or over-saturated thunk to weak-head normal form; for any other value, it just returns that value.

$$\texttt{valueOf} \in \mathit{Value} \times \mathit{Store} \to \mathit{Value} \times \mathit{Store}$$

$$\texttt{valueOf}(v, \sigma) =$$

$$\begin{cases} (n, \sigma) & \text{if } v = n \\ (\textbf{Con}\ cn\ \vec{w}, \sigma) & \text{if } v = \textbf{Con}\ cn\ \vec{w} \\ (\textbf{Thunk}\ op\ \vec{w}, \sigma) & \text{if } v = \textbf{Thunk}\ op\ \vec{w}, |\vec{w}| < \texttt{arity}(op) \\ \texttt{overAppHelper}(op, \vec{w}, \sigma) & \text{if } v = \textbf{Thunk}\ op\ \vec{w}, |\vec{w}| \geq \texttt{arity}(op) \end{cases}$$

`overAppHelper`

`overAppHelper` reduces a thunk to a value by either evaluating a primitive operation to an integer, creating a saturated constructor, or evaluating the thunk's function body to a value.

$$\texttt{overAppHelper} \in \mathit{Operator} \times \overrightarrow{\mathit{Wrapper}} \times \mathit{Store} \to \mathit{Value} \times \mathit{Store}$$

$$\texttt{overAppHelper}(op, \vec{w}, \sigma) =$$

$$\begin{cases} \texttt{eval}(\oplus, \vec{w}, \sigma) & \text{if } op = \oplus, \texttt{arity}(\oplus) = |\vec{w}| \\ (\textbf{Con}\ cn\ \vec{w}, \sigma) & \text{if } op = cn, \texttt{arity}(cn) = |\vec{w}| \\ v' & \text{if } op = fn \end{cases}$$

where

$$(v, \sigma') = \texttt{interpret}(\texttt{bodyOf}(fn), \texttt{paramsOf}(fn)\ \texttt{zip}\ \vec{w}, \sigma)$$

$$v' = \begin{cases} \texttt{valueOf}(\textbf{Thunk}\ op\ \vec{w}' \mathbin{+\!\!+} (\vec{w}\ \texttt{drop}\ \texttt{arity}(fn))) & \text{if } v = \textbf{Thunk}\ op\ \vec{w}' \\ (v, \sigma') & \text{otherwise} \end{cases}$$

Bibliography

[1] J. McMahan, M. Christensen, L. Nichols, J. Roesch, S.-Y. Guo, B. Hardekopf, and T. Sherwood, *An architecture supporting formal and compositional binary analysis*, in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (New York, NY, USA), p. 177–191, Association for Computing Machinery, 2017.

[2] J. McMahan, M. Christensen, L. Nichols, J. Roesch, S.-Y. Guo, B. Hardekopf, and T. Sherwood, *An architecture for analysis*, *IEEE Micro* **38** (2018), no. 3 107–115.

[3] M. Christensen, J. McMahan, L. Nichols, J. Roesch, T. Sherwood, and B. Hardekopf, *Safe functional systems through integrity types and verified assembly*, *Theoretical Computer Science* **851** (2021) 39–61.

[4] J. E. McMahan, *The ZARF Architecture for Recursive Functions*. PhD thesis, UC Santa Barbara, Santa Barbara, CA, June, 2019.

[5] J. McMahan, M. Christensen, K. Dewey, B. Hardekopf, and T. Sherwood, *Bouncer: Static program analysis in hardware*, in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), p. 711–722, Association for Computing Machinery, 2019.

[6] M. Christensen, T. Sherwood, J. Balkind, and B. Hardekopf, *Wire sorts: A language abstraction for safe hardware composition*, in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, (New York, NY, USA), p. 175–189, Association for Computing Machinery, 2021.

[7] R. Mangharam, H. Abbas, M. Behl, K. Jang, M. Pajic, and Z. Jiang, *Three challenges in cyber-physical systems*, in *2016 8th International Conference on Communication Systems and Networks (COMSNETS)*, pp. 1–8, Jan, 2016.

[8] S. Shuja, S. K. Srinivasan, S. Jabeen, and D. Nawarathna, *A formal verification methodology for ddd mode pacemaker control programs*, *Journal of Electrical and Computer Engineering* (2015).

[9] J. M. Rushby, *Proof of separability: A verification technique for a class of a security kernels*, in *Proceedings of the 5th Colloquium on International Symposium on Programming*, (London, UK, UK), pp. 352–367, Springer-Verlag, 1982.

[10] N. Heintze and J. G. Riecke, *The slam calculus: Programming with secrecy and integrity*, in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, (New York, NY, USA), pp. 365–377, ACM, 1998.

[11] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, *A core calculus of dependency*, in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, (New York, NY, USA), pp. 147–160, ACM, 1999.

[12] D. Volpano, C. Irvine, and G. Smith, *A sound type system for secure flow analysis*, *J. Comput. Secur.* **4** (Jan., 1996) 167–187.

[13] D. E. Denning and P. J. Denning, *Certification of programs for secure information flow*, *Commun. ACM* **20** (July, 1977) 504–513.

[14] J. A. Goguen and J. Meseguer, *Security policies and security models*, in *Security and Privacy, 1982 IEEE Symposium on*, pp. 11–11, April, 1982.

[15] F. Pottier and V. Simonet, *Information flow inference for ml*, *ACM Trans. Program. Lang. Syst.* **25** (Jan., 2003) 117–158.

[16] A. Sabelfeld and A. C. Myers, *Language-based information-flow security*, *IEEE J.Sel. A. Commun.* **21** (Sept., 2006) 5–19.

[17] D. Terei, S. Marlow, S. Peyton Jones, and D. Mazières, *Safe haskell*, in *Proceedings of the 2012 Haskell Symposium*, Haskell '12, (New York, NY, USA), pp. 137–148, ACM, 2012.

[18] D. Yu, N. A. Hamid, and Z. Shao, *Building certified libraries for pcc: dynamic storage allocation*, in *Proceedings of the 12th European conference on Programming*, pp. 363–379, Springer-Verlag, 2003.

[19] A. Chlipala, *Mostly-automated verification of low-level programs in computational separation logic*, in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 234–245, ACM, 2011.

[20] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, *Kami: A platform for high-level parametric hardware specification and its modular verification*, *Proc. ACM Program. Lang.* **1** (Aug., 2017).

[21] R. S. Boyer and Y. Yu, *Automated correctness proofs of machine code programs for a commercial microprocessor*, in *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, (London, UK, UK), pp. 416–430, Springer-Verlag, 1992.

[22] N. G. Michael and A. W. Appel, *Machine instruction syntax and semantics in higher order logic*, in *Proceedings of the 17th International Conference on Automated Deduction*, CADE-17, (London, UK, UK), pp. 7–24, Springer-Verlag, 2000.

[23] A. Kennedy, N. Benton, J. B. Jensen, and P.-E. Dagand, *Coq: The world's best macro assembler?*, in *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, (New York, NY, USA), pp. 13–24, ACM, 2013.

[24] A. Fox and M. O. Myreen, *A trustworthy monadic formalization of the armv7 instruction set architecture*, in *Proceedings of the First International Conference on Interactive Theorem Proving*, ITP'10, (Berlin, Heidelberg), pp. 243–258, Springer-Verlag, 2010.

[25] J. S. Moore, *A mechanically verified language implementation*, *Journal of Automated Reasoning* **5** (1989), no. 4 461–492.

[26] W. A. Hunt Jr, *Microprocessor design verification*, *Journal of Automated Reasoning* **5** (1989), no. 4 429–460.

[27] *Journal of automated reasoning*, 2003.

[28] G. C. Necula, *Proof-carrying code. design and implementation*. Springer, 2002.

[29] A. W. Appel, *Foundational proof-carrying code*, in *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, LICS '01, (Washington, DC, USA), pp. 247–, IEEE Computer Society, 2001.

[30] J. Yang and C. Hawblitzel, *Safe to the last instruction: Automated verification of a type-safe operating system*, in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, (New York, NY, USA), pp. 99–110, ACM, 2010.

[31] T. Maeda and A. Yonezawa, *Typed assembly language for implementing os kernels in smp/multi-core environments with interrupts*, in *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2010.

[32] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, *Boogie: A modular reusable verifier for object-oriented programs*, in *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, (Berlin, Heidelberg), pp. 364–387, Springer-Verlag, 2006.

[33] H. Xi and R. Harper, *A dependently typed assembly language*, in *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, (New York, NY, USA), pp. 169–180, ACM, 2001.

[34] A. Chlipala, *A verified compiler for an impure functional language*, in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, (New York, NY, USA), pp. 93–106, ACM, 2010.

[35] G. C. Necula, *Translation validation for an optimizing compiler*, in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, (New York, NY, USA), pp. 83–94, ACM, 2000.

[36] P. Curzon and P. Curzon, *A verified compiler for a structured assembly language*, in *In proceedings of the 1991 international workshop on the HOL theorem Proving System and its applications. IEEE Computer*, pp. 253–262, Society Press, 1991.

[37] M. Strecker, *Formal verification of a java compiler in isabelle*, in *Automated Deduction—CADE-18*, pp. 63–77. Springer, 2002.

[38] X. Leroy, *A formally verified compiler back-end*, *Journal of Automated Reasoning* **43** (2009), no. 4 363–446.

[39] A. W. Appel, *Verified software toolchain*, in *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, (Berlin, Heidelberg), pp. 1–17, Springer-Verlag, 2011.

[40] G. Neis, C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis, *Pilsner: A compositionally verified compiler for a higher-order imperative language*, in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, (New York, NY, USA), pp. 166–178, ACM, 2015.

[41] T. Ramananandro, Z. Shao, S.-C. Weng, J. Koenig, and Y. Fu, *A compositional semantics for verified separate compilation and linking*, in *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, (New York, NY, USA), pp. 3–14, ACM, 2015.

[42] Z. Jiang, M. Pajic, and R. Mangharam, *Cyber-physical modeling of implantable cardiac medical devices*, *Proceedings of the IEEE* **100** (Jan, 2012) 122–137.

[43] A. O. Gomes and M. V. M. Oliveira, *Formal specification of a cardiac pacing system*, in *FM 2009: Formal Methods* (A. Cavalcanti and D. R. Dams, eds.), (Berlin, Heidelberg), pp. 692–707, Springer Berlin Heidelberg, 2009.

[44] T. Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre, *Quantitative verification of implantable cardiac pacemakers*, in *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pp. 263–272, IEEE, 2012.

[45] L. Cordeiro, B. Fischer, H. Chen, and J. Marques-Silva, *Semiformal verification of embedded software in medical devices considering stringent hardware constraints*, in *2009 International Conference on Embedded Software and Systems*, pp. 396–403, May, 2009.

[46] P. J. Landin, *The Mechanical Evaluation of Expressions*, *The Computer Journal* **6** (Jan., 1964) 308–320.

[47] B. Graham, *Secd: Design issues*, tech. rep., University of Calgary, 1989.

[48] T. J. Clarke, P. J. Gladstone, C. D. MacLean, and A. C. Norman, *Skim - the s, k, i reduction machine*, in *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, (New York, NY, USA), pp. 128–135, ACM, 1980.

[49] L. P. Deutsch, *A lisp machine with very compact programs*, in *Proceedings of the 3rd international joint conference on Artificial intelligence*, pp. 697–703, Morgan Kaufmann Publishers Inc., 1973.

[50] P. M. Kogge, *"The Architecture of Symbolic Computers"*. McGraw-Hill, Inc., New York, New York, 1991.

[51] T. F. Knight, *Implementation of a list processing machine*. PhD thesis, Massachusetts Institute of Technology, 1979.

[52] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, *Flicker: An execution infrastructure for tcb minimization*, *SIGOPS Oper. Syst. Rev.* **42** (Apr., 2008) 315–328.

[53] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, *Nohype: Virtualized cloud infrastructure without the virtualization*, *SIGARCH Comput. Archit. News* **38** (June, 2010) 350–361.

[54] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel, *Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses*, in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pp. 129–142, IEEE, 2008.

[55] S. Gollakota, H. Hassanieh, B. Ransford, D. Katabi, and K. Fu, *They can hear your heartbeats: non-invasive security for implantable medical devices*, in *Proc. ACM Conf. SIGCOMM*, pp. 2–13, 2011.

[56] T. Denning, K. Fu, and T. Kohno, *Absence makes the heart grow fonder: New directions for implantable medical device security*, in *Proceedings of the 3rd Conference on Hot Topics in Security*, HOTSEC'08, (Berkeley, CA, USA), pp. 5:1–5:7, USENIX Association, 2008.

[57] B. Hardekopf and C. Lin, *Flow-sensitive pointer analysis for millions of lines of code*, in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, (Washington, DC, USA), pp. 289–298, IEEE Computer Society, 2011.

[58] E. Moggi, *Notions of computation and monads*, *Information and computation* **93** (1991), no. 1 55–92.

[59] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, *A history of haskell: being lazy with class*, in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 12–1, ACM, 2007.

[60] R. Hindley, *The principal type-scheme of an object in combinatory logic*, *Transactions of the American Mathematical Society* **146** (1969) 29–60.

[61] R. Milner, *A theory of type polymorphism in programming*, *Journal of Computer and System Sciences* **17** (1978) 348–375.

[62] M. E. Conway, *Design of a separable transition-diagram compiler*, *Commun. ACM* **6** (July, 1963) 396–408.

[63] A. L. D. Moura and R. Ierusalimschy, *Revisiting coroutines*, *ACM Trans. Program. Lang. Syst.* **31** (Feb., 2009) 6:1–6:31.

[64] S. J. Connolly, M. Gent, R. S. Roberts, P. Dorian, D. Roy, R. S. Sheldon, L. B. Mitchell, M. S. Green, G. J. Klein, and B. O'Brien, *Canadian implantable defibrillator study (cids)*, *Circulation* **101** (2000), no. 11 1297–1302, [http://circ.ahajournals.org/content/101/11/1297.full.pdf].

[65] T. A. versus Implantable Defibrillators (AVID) Investigators, *A comparison of antiarrhythmic-drug therapy with implantable defibrillators in patients resuscitated from near-fatal ventricular arrhythmias*, *New England Journal of Medicine* **337** (1997), no. 22 1576–1584, [http://dx.doi.org/10.1056/NEJM199711273372202]. PMID: 9411221.

[66] J. Siebels, K.-H. Kuck, and C. Investigators, *Implantable cardioverter defibrillator compared with antiarrhythmic drug treatment in cardiac arrest survivors (the cardiac arrest study hamburg)*, *American Heart Journal* **127** (April, 1994) 1139–1144.

[67] "Living with your implantable cardioverter defibrillator (ICD)." http://www.heart.org/HEARTORG/Conditions/Arrhythmia/PreventionTreatmentofArrhythmia/Living-With-Your-Implantable-Cardioverter-Defibrillator-ICD_UCM_448462_Article.jsp, 09, 2016.

[68] "How many people have ICDs?." http://asktheicd.com/tile/106/english-implantable-cardioverter-defibrillator-icd/how-many-people-have-icds/, Accessed October 24, 2019.

[69] J. Pan and W. J. Tompkins, *A real-time qrs detection algorithm*, *IEEE Transactions on Biomedical Engineering* **BME-32** (March, 1985) 230–236.

[70] R. A. Álvarez, A. J. M. Penín, and X. A. V. Sobrino, *A comparison of three qrs detection algorithms over a public database*, *Procedia Technology* **9** (2013) 1159 – 1165. CENTERIS 2013 - Conference on ENTERprise Information Systems / ProjMAN 2013 - International Conference on Project MANagement/ HCIST 2013 - International Conference on Health and Social Care Information Systems and Technologies.

[71] "Open source ECG analysis software." http://www.eplimited.com/confirmation.htm, Accessed October 24, 2019.

[72] M. S. Wathen, P. J. DeGroot, M. O. Sweeney, A. J. Stark, M. F. Otterness, W. O. Adkisson, R. C. Canby, K. Khalighi, C. Machado, D. S. Rubenstein, and K. J. Volosin, *Prospective randomized multicenter trial of empirical antitachycardia pacing versus shocks for spontaneous rapid ventricular tachycardia in patients with implantable cardioverter-defibrillators*, *Circulation* **110** (2004), no. 17 2591–2596, [http://circ.ahajournals.org/content/110/17/2591.full.pdf].

[73] "The Coq proof assistant." https://coq.inria.fr, Accessed October 24, 2019.

[74] V. Kashyap, B. Wiedermann, and B. Hardekopf, *Timing- and termination-sensitive secure information flow: Exploring a new approach*, in *2011 IEEE Symposium on Security and Privacy*, pp. 413–428, May, 2011.

[75] V. Simonet, *Fine-grained information flow analysis for a λ calculus with sum types*, in *Proceedings of the 15th IEEE Workshop on Computer Security Foundations*, CSFW '02, (Washington, DC, USA), pp. 223–, IEEE Computer Society, 2002.

[76] D. Ricketts, G. Malecha, M. M. Alvarez, V. Gowda, and S. Lerner, *Towards verification of hybrid systems in a foundational proof assistant*, in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 248–257, 2015.

[77] Z. Cutlip, "Dlink dir-815 upnp command injection." http://shadow-file.blogspot.com/2013/02/dlink-dir-815-upnp-command-injection.html, February, 2013. [Online; accessed 01-November-2017].

[78] A. Cui, M. Costello, and S. J. Stolfo, *When firmware modification attack: A case study of embedded exploitation*, in *NDSS Symposium '13*, 2013.

[79] G. Hernandez, O. Arias, D. Buentello, and Y. Jin, *Smart nest thermostat: A smart spy in your home*, in *Black Hat Briefings*, 2014.

[80] G. Morrisett, D. Walker, K. Crary, and N. Glew, *From System F to Typed Assembly Language*, *ACM Trans. Program. Lang. Syst.* **21** (May, 1999) 527–568.

[81] K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic, *TALx86: A realistic typed assembly language*, in *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*, pp. 25–35, 1999.

[82] E. Buchanan, R. Roemer, and S. Savage, "Return-oriented programming: Exploits without code injection." https://hovav.net/ucsd/talks/blackhat08.html.

[83] K. Dewey, J. Roesch, and B. Hardekopf, *Fuzzing the rust typechecker using clp*, in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, (Washington, DC, USA), pp. 482–493, IEEE Computer Society, 2015.

[84] L. De Moura and N. Bjørner, *Z3: An efficient smt solver*, in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[85] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, *Mibench: A free, commercially representative embedded benchmark suite*, in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 2001.

[86] H. Xi and R. Harper, *A Dependently Typed Assembly Language*, in *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, (New York, NY, USA), pp. 169–180, ACM, 2001.

[87] G. Morrisett, K. Crary, N. Glew, and D. Walker, *Stack-based typed assembly language*, *Journal of Functional Programming* **12** (Jan., 2002).

[88] K. Crary, *Toward a Foundational Typed Assembly Language*, in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, (New York, NY, USA), pp. 198–212, ACM, 2003.

[89] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach, *A Verified Information-flow Architecture*, in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, (New York, NY, USA), pp. 165–178, ACM, 2014.

[90] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, *CodeSurfer/x86—A Platform for Analyzing x86 Executables*, in *Compiler Construction*, Lecture Notes in Computer Science, pp. 250–254, Springer, Berlin, Heidelberg, Apr., 2005.

[91] J. Lee, T. Avgerinos, and D. Brumley, *Tie: Principled reverse engineering of types in binary programs*, in *In Proceedings of the Network and Distributed System Security Symposium*, 2011.

[92] M. Noonan, A. Loginov, and D. Cok, *Polymorphic Type Inference for Machine Code*, *arXiv:1603.05495 [cs]* (Mar., 2016). arXiv: 1603.05495.

[93] J. Caballero and Z. Lin, *Type Inference on Executables*, *ACM Comput. Surv.* **48** (May, 2016) 65:1–65:35.

[94] Z. Chen, *Java card technology for smart cards: architecture and programmer's guide*. Addison-Wesley Professional, 2000.

[95] W. Coekaerts, "The java typesystem is broken." http://wouter.coekaerts.be/2018/java-type-system-broken.

[96] N. Amin and R. Tate, *Java and scala's type systems are unsound: The existential crisis of null pointers*, in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, (New York, NY, USA), pp. 838–848, ACM, 2016.

[97] R. Grigore, *Java generics are turing complete*, *CoRR* **abs/1605.05274** (2016) [arXiv:1605.0527].

[98] K. Zhai, R. Townsend, L. Lairmore, M. A. Kim, and S. A. Edwards, *Hardware synthesis from a recursive functional language*, in *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, CODES '15, (Piscataway, NJ, USA), pp. 83–93, IEEE Press, 2015.

[99] R. Townsend, M. A. Kim, and S. A. Edwards, *From functional programs to pipelined dataflow circuits*, in *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, (New York, NY, USA), pp. 76–86, ACM, 2017.

[100] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, *Softbound: Highly compatible and complete spatial memory safety for c*, in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, (New York, NY, USA), pp. 245–258, ACM, 2009.

[101] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, *Hardbound: Architectural support for spatial safety of the c programming language*, in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, (New York, NY, USA), pp. 103–114, ACM, 2008.

[102] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, *Watchdog: Hardware for safe and secure manual memory management and full memory safety*, in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, (Washington, DC, USA), pp. 189–200, IEEE Computer Society, 2012.

[103] R. Zhang, N. Stanley, C. Griggs, A. Chi, and C. Sturton, *Identifying security critical properties for the dynamic verification of a processor*, in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, 2017.

[104] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori, *Software-based gate-level information flow security for iot systems*, in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), pp. 328–340, ACM, 2017.

[105] R. S. Chakraborty and S. Bhunia, *Harpoon: An obfuscation-based soc design methodology for hardware protection*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **28** (2009), no. 10 1493–1502.

[106] *Ieee standard for verilog hardware description language*, *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006) 1–590.

[107] W. Snyder, P. Wasson, and D. Galbi, "Verilator-convert Verilog code to C++/SystemC." http://www.veripool.org/wiki/verilator, 2020.

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

[109] 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.

[110] 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)*, (Ghent, Belgium), pp. 1–7, IEEE, 2017.

[111] D. Dangwal, G. Tzimpragos, and T. Sherwood, *Agile hardware development and instrumentation with PyRTL*, *IEEE Micro* **40** (2020), no. 4 76–84.

[112] *IEEE standard for SystemVerilog–unified hardware design, specification, and verification language*, *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018) 1–1315.

[113] *IEEE standard for property specification language (PSL)*, *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)* (2010) 1–182.

[114] D. Geist, *The PSL/Sugar specification language a language for all seasons*, in *Correct Hardware Design and Verification Methods* (D. Geist and E. Tronci, eds.), (Berlin, Heidelberg), pp. 3–3, Springer Berlin Heidelberg, 2003.

[115] O. W. Group, *Accellera standard OVL v2 library reference manual*, tech. rep., Accellera Systems Initiative, 2014.

[116] M. Sheeran, *MuFP, a language for VLSI design*, in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, (New York, NY, USA), p. 104–112, Association for Computing Machinery, 1984.

[117] K. Claessen, *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers University of Technology and Göteborg University, 2001.

[118] A. Gill, T. Bull, A. Farmer, G. Kimmell, and E. Komp, *Types and type families for hardware simulation and synthesis*, in *Trends in Functional Programming* (R. Page, Z. Horváth, and V. Zsók, eds.), (Berlin, Heidelberg), pp. 118–133, Springer Berlin Heidelberg, 2011.

[119] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, *Lava: Hardware design in haskell*, in *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, (New York, NY, USA), p. 174–184, Association for Computing Machinery, 1998.

[120] 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.

[121] 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.

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

[123] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, *Protocol verification as a hardware design aid*, in *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, (Cambridge, MA, USA), pp. 522–525, IEEE, 1992.

[124] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang, *Predictable accelerator design with time-sensitive affine types*, in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, (New York, NY, USA), p. 393–407, Association for Computing Machinery, 2020.

[125] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, *High-level synthesis for FPGAs: From prototyping to deployment*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **30** (2011), no. 4 473–491.

[126] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, *LegUp: High-level synthesis for FPGA-based processor/accelerator systems*, in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, (New York, NY, USA), p. 33–36, Association for Computing Machinery, 2011.

[127] L. Truong and P. Hanrahan, *A golden age of hardware description languages: Applying programming language techniques to improve design productivity*, in *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA* (B. S. Lerner, R. Bodík, and S. Krishnamurthi, eds.), vol. 136 of *LIPIcs*, (Providence, RI, USA), pp. 7:1–7:21, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[128] D. Lockhart, G. Zibrat, and C. Batten, *PyMTL: A unified framework for vertically integrated computer architecture research*, in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, (Cambridge, United Kingdom), pp. 280–292, IEEE, 2014.

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

[130] J. P. P. Flor, W. Swierstra, and Y. Sijsling, *Pi-Ware: Hardware Description and Verification in Agda*, in *21st International Conference on Types for Proofs and Programs (TYPES 2015)* (T. Uustalu, ed.), vol. 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 9:1–9:27, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[131] J. Street, "Hardcaml: Register transfer level hardware design in OCaml." https://github.com/janestreet/hardcaml.

[132] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, *The essence of bluespec: A core language for rule-based hardware design*, in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, (New York, NY, USA), p. 243–257, Association for Computing Machinery, 2020.

[133] S. Sutherland and D. Mills, *Standard gotchas subtleties in the verilog and systemverilog standards that every engineer should know*, 2006.

[134] S. Sutherland, D. Mills, and C. Spear, *Gotcha again: More subtleties in the Verilog and SystemVerilog standards that every engineer should know*, 2007.

[135] 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.

[136] S. Xie and M. B. Taylor, *The basejump manycore accelerator network*, 2018.

[137] L. P. Carloni, *From latency-insensitive design to communication-based system-level design*, *Proceedings of the IEEE* **103** (2015), no. 11 2133–2151.

[138] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, *Theory of latency-insensitive design*, *Trans. Comp.-Aided Des. Integ. Cir. Sys.* **20** (Nov., 2006) 1059–1076.

[139] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, *Latency Insensitive Protocols*, in *Computer Aided Verification* (N. Halbwachs and D. Peled, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 123–133, Springer, 1999.

[140] B. Cao, K. A. Ross, M. A. Kim, and S. A. Edwards, *Implementing latency-insensitive dataflow blocks*, in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, (Austin, TX, USA), pp. 179–187, IEEE, 2015.

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

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

[143] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, *OpenPiton: An open source manycore research framework*, in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), p. 217–232, Association for Computing Machinery, 2016.

[144] P. University, "OpenPiton Design Benchmark." https://github.com/PrincetonUniversity/OPDB, 2020.

[145] A. Waterman, Y. Lee, D. Patterson, K. Asanović, and C. Division, *The RISC-V Instruction Set Manual Volume I: User-Level ISA*, 2016.

[146] J. Lowe-Power and C. Nitta, *The Davis In-Order (dino) CPU: A teaching-focused RISC-V CPU design*, in *Proceedings of the Workshop on Computer Architecture Education*, WCAE'19, (New York, NY, USA), Association for Computing Machinery, 2019.

[147] D. S. Holmes, A. L. Ripple, and M. A. Manheimer, *Energy-efficient superconducting computing—power budgets and requirements*, *IEEE Transactions on Applied Superconductivity* **23** (2013), no. 3 1701610–1701610.

[148] I. I. Soloviev, N. V. Klenov, S. V. Bakurskiy, M. Y. Kupriyanov, A. L. Gudkov, and A. S. Sidorenko, *Beyond moore's technologies: operation principles of a superconductor alternative*, *Beilstein journal of nanotechnology* **8** (2017), no. 1 2689–2710.

[149] R. Cai, A. Ren, O. Chen, N. Liu, C. Ding, X. Qian, J. Han, W. Luo, N. Yoshikawa, and Y. Wang, *A stochastic-computing based deep learning framework using adiabatic quantum-flux-parametron superconducting technology*, in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), p. 567–578, Association for Computing Machinery, 2019.

[150] G. Tzimpragos, D. Vasudevan, N. Tsiskaridze, G. Michelogiannakis, A. Madhavan, J. Volk, J. Shalf, and T. Sherwood, *A computational temporal logic for superconducting accelerators*, in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), p. 435–448, Association for Computing Machinery, 2020.

[151] G. Tzimpragos, J. Volk, A. Wynn, J. E. Smith, and T. Sherwood, *Superconducting computing with alternating logic elements*, in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, (Valencia, Spain), pp. 651–664, IEEE, 2021.

[152] T. E. Oliphant, *Python for scientific computing*, *Computing in Science Engineering* **9** (2007), no. 3 10–20.

[153] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to automata theory, languages, and computation*, *Acm Sigact News* **32** (2001), no. 1 60–65.

[154] M. Munir, A. Gopikanna, A. Fayyazi, M. Pedram, and S. Nazarian, *Qmc: A formal model checking verification framework for superconducting logic*, in *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, GLSVLSI '21, (New York, NY, USA), p. 259–264, Association for Computing Machinery, 2021.

[155] R. Alur and D. L. Dill, *A theory of timed automata*, *Theoretical Computer Science* **126** (Apr., 1994) 183–235.

[156] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, *Uppaal — a tool suite for automatic verification of real-time systems*, in *Hybrid Systems III* (R. Alur, T. A. Henzinger, and E. D. Sontag, eds.), (Berlin, Heidelberg), pp. 232–243, Springer Berlin Heidelberg, 1996.

[157] R. J. Baker, *CMOS: circuit design, layout, and simulation*. John Wiley & Sons, 2019.

[158] L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, *Electronic Design Automation*. Elsevier, 2009.

[159] *Ieee standard for vhdl language reference manual*, *IEEE Std 1076-2019* (2019) 1–673.

[160] L. N. Cooper, *Bound Electron Pairs in a Degenerate Fermi Gas*, *Physical Review* **104** (Nov., 1956) 1189–1190.

[161] B. Josephson, *Possible new effects in superconductive tunnelling*, *Physics Letters* **1** (1962), no. 7 251–253.

[162] K. Likharev and V. Semenov, *Rsfq logic/memory family: a new josephson-junction technology for sub-terahertz-clock-frequency digital systems*, *IEEE Transactions on Applied Superconductivity* **1** (1991), no. 1 3–28.

[163] J. Clarke, *Principles and applications of squids*, *Proceedings of the IEEE* **77** (1989), no. 8 1208–1223.

[164] G. H. Mealy, *A method for synthesizing sequential circuits*, *The Bell System Technical Journal* **34** (Sept., 1955) 1045–1079.

[165] M. Sipser, *Introduction to the theory of computation*, *ACM Sigact News* **27** (1996), no. 1 27–29.

[166] Q. Xu, C. L. Ayala, N. Takeuchi, Y. Yamanashi, and N. Yoshikawa, *Hdl-based modeling approach for digital simulation of adiabatic quantum flux parametron logic*, *IEEE Transactions on Applied Superconductivity* **26** (2016), no. 8 1–5.

[167] G. Tzimpragos, J. Volk, D. Vasudevan, N. Tsiskaridze, G. Michelogiannakis, A. Madhavan, J. Shalf, and T. Sherwood, *Temporal computing with superconductors*, *IEEE Micro* **41** (2021), no. 3 71–79.

[168] K. Gaj, E. G. Friedman, and M. J. Feldman, *Timing of Multi-Gigahertz Rapid Single Flux Quantum Digital Circuits*, in *High Performance Clock Distribution Networks* (E. G. Friedman, ed.), pp. 135–164. Springer US, Boston, MA, 1997.

[169] A. Krasniewski, *Logic simulation of rsfq circuits*, *IEEE Transactions on Applied Superconductivity* **3** (1993), no. 1 33–38.

[170] T. Kawaguchi, K. Takagi, and N. Takagi, *A verification method for single-flux-quantum circuits using delay-based time frame model*, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* **98-A** (2015) 2556–2564.

[171] R. S. Bakolo, *Design and implementation of a rsfq superconductive digital electronics cell library*, Master's thesis, University of Stellenbosch, 2011.

[172] 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.

[173] 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)*, (Ghent, Belgium), pp. 1–7, IEEE, 2017.

[174] N. Matloff, *Introduction to discrete-event simulation and the simpy language*, *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August* **2** (2008), no. 2009 1–33.

[175] L. Schindler, *The Development and Characterisation of a Parameterised RSFQ Cell Library for Layout Synthesis*. PhD thesis, Stellenbosch University, 2021.

[176] L. W. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, EECS Department, University of California, Berkeley, May, 1975.

[177] I. Whiteley Research, "Wrspice." http://wrcad.com. Accessed: 2021-10-22.

[178] K. E. Batcher, *Sorting networks and their applications*, in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), (New York, NY, USA), p. 307–314, Association for Computing Machinery, 1968.

[179] A. F. Kirichenko, I. V. Vernik, M. Y. Kamkar, J. Walter, M. Miller, L. R. Albu, and O. A. Mukhanov, *Ersfq 8-bit parallel arithmetic logic unit*, *IEEE Transactions on Applied Superconductivity* **29** (Aug, 2019) 1–7.

[180] G. Tzimpragos, A. Madhavan, D. Vasudevan, D. Strukov, and T. Sherwood, *Boosted race trees for low energy classification*, in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), p. 215–228, Association for Computing Machinery, 2019.

[181] E. M. Clarke and E. A. Emerson, *Design and synthesis of synchronization skeletons using branching time temporal logic*, in *Logics of Programs* (D. Kozen, ed.), (Berlin, Heidelberg), pp. 52–71, Springer Berlin Heidelberg, 1982.

[182] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, *Symbolic model checking for real-time systems*, in [*1992*] *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, (Santa Cruz, CA, USA), pp. 394–406, IEEE, 1992.

[183] G. Behrmann, A. David, and K. G. Larsen, *A tutorial on uppaal*, in *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures* (M. Bernardo and F. Corradini, eds.), pp. 200–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[184] V. Adler, Chin-Hong Cheah, K. Gaj, D. K. Brock, and E. G. Friedman, *A cadence-based design environment for single flux quantum circuits*, *IEEE Transactions on Applied Superconductivity* **7** (1997), no. 2 3294–3297.

[185] F. Matsuzaki, N. Yoshikawa, M. Tanaka, A. Fujimaki, and Y. Takai, *A behavioral-level hdl description of sfq logic circuits for quantitative performance analysis of large-scale sfq digital systems*, *Physica C: Superconductivity* **392-396** (2003) 1495 – 1500. Proceedings of the 15th International Symposium on Superconductivity (ISS 2002): Advances in Superconductivity XV. Part II.

[186] N. Katam, S. N. Shahsavani, T.-R. Lin, G. Pasandi, A. Shafaei, and M. Pedram, "Sport lab sfq logic circuit benchmark suite." https://ceng.usc.edu/techreports/2017/Pedram%20CENG-2017-1.pdf, 2017. Accessed: 2021-10-22.

[187] K. Gaj, C.-H. Cheah, E. Friedman, and M. Feldman, *Functional modeling of rsfq circuits using verilog hdl*, *IEEE Transactions on Applied Superconductivity* **7** (1997), no. 2 3151–3154.

[188] R. N. Tadros, A. Fayyazi, M. Pedram, and P. A. Beerel, *Systemverilog modeling of sfq and aqfp circuits*, *IEEE Transactions on Applied Superconductivity* **30** (2020), no. 2 1–13.

[189] R. N. Tadros, A. Fayyazi, M. Pedram, and P. A. Beerel, *Systemverilog modeling of sfq and aqfp circuits*, *IEEE Transactions on Applied Superconductivity* **30** (2020), no. 2 1–13.

[190] L. C. Müller and C. J. Fourie, *Automated state machine and timing characteristic extraction for rsfq circuits*, *IEEE Transactions on Applied Superconductivity* **24** (2014), no. 1 3–12.

[191] L. C. Müller and C. J. Fourie, *Automated state machine and timing characteristic extraction for rsfq circuits*, *IEEE Transactions on Applied Superconductivity* **24** (2014), no. 1 3–12.

[192] C. J. Fourie, *Extraction of dc-biased sfq circuit verilog models*, *IEEE Transactions on Applied Superconductivity* **28** (2018), no. 6 1–11.

[193] A. D. Wong, K. Su, H. Sun, A. Fayyazi, M. Pedram, and S. Nazarian, *Verisfq: A semi-formal verification framework and benchmark for single flux quantum technology*, in *20th International Symposium on Quality Electronic Design (ISQED)*, (Santa Clara, CA, USA), pp. 224–230, IEEE, 2019.

[194] *Ieee standard for universal verification methodology language reference manual*, *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)* (2020) 1–458.