My research sits at the intersection of **computer architecture** (designing chips) and **programming languages** (the tools we use to implement them). Modern chip design embodies enormous complexity, from general-purpose processors to specialized hardware accelerators. With the trend towards specialization, chip designers need techniques that let them quickly iterate over a design and fit into familiar languages and tools. However, designing a chip with *speed* and *robustness* remains a challenge, particularly for smaller teams who are increasingly entering the field. Compounding these challenges, hardware description languages (HDLs) are the core driver for chip development, which bear the burden of complexity. As a result, industry studies show roughly half of chip development time is spent on verification [1].

During my PhD, I have led research that improves design tools for agile chip design. My work *opens two entirely new areas* in the hardware design space and *enables two novel design processes* based around increasing developer productivity with **correctness guarantees**. The solutions I present in my research make **verifiability a first-class constraint** through the use of **solver-aided programming languages techniques** which provide formal guarantees through the integration of automated theorem provers and constraint solvers. Further, the techniques I developed are not mere prototypes; they integrate with existing open-source and industry-standard languages and tools, shortening the design process without increasing the verification burden. Given the extremely high cost of chip design, this research will reduce the cost and effort required to work in this area.

Next, I briefly summarize my two major contributions. As my research has opened two new areas in the hardware design space, there is much more to do in these areas as well as further areas to open up. The "golden age" of computer architecture [2] offers an opportunity to advance the design of open computing architectures and specialized hardware. However, we can only realize this if we improve the languages and tools that chip designers use. My research will move the field in this direction by integrating formal methods into open-source languages that reason about not only correctness, but energy, performance, and security.

Control Logic Synthesis

Chip design requires reasoning between different layers of abstraction: from an architectural specification (the instructions the chip executes), to the microarchitectural datapath (the functional units), down to the low-level control logic (which coordinates computation on the chip). Implementing control logic itself is tedious and error-prone, where changes at these levels propagate non-obvious changes to the control.

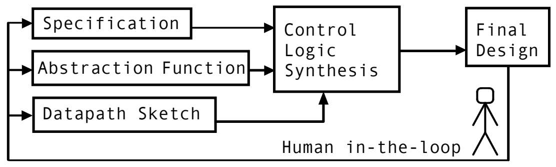


Figure 1: Overview of the technique.

This work, recently accepted for publication at ASPLOS [9], introduces a new technique, *control logic synthesis*, which automatically generates the control logic for a datapath according to an architectural specification (illustrated in Figure 1). To my knowledge, this work is the first to automatically generate control logic in a correct-by-construction way.

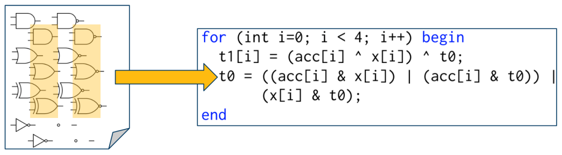
The insight is adapting program synthesis techniques to HDLs, bridging the gap between the datapath and the high-level specification—a key direction I previously identified [7]. This technique allows chip developers to freely modify and iterate over the designs of both the specification and the datapath without getting caught up in the abstruse details of control, as I show in case studies covering embedded-class RISC-V cores and accelerators geared for cryptographic applications.

I was awarded silver in the ACM Student Research Competition at PLDI for an initial presentation of this work. Further, I mentored six junior students in this work, who contributed to the published paper as well as future efforts to enable control logic synthesis on more complex architectures. These future efforts require advancing the state of the art in automated code generation for hardware and designing new languages enabled by control logic synthesis.

Hardware Decompilation

Where my first project applies program synthesis to hardware generation, my second project moves in the reverse direction, opening a new area called *hardware decompilation*. The idea is analogous to software decompilation—lifting a binary back to source code—except it analyzes a gate-level circuit represented as a graph called a *netlist*. Nobody prior had raised designs from the abstraction level of a gate-level netlist to high-level HDL code. This entirely new area opened up in my research helps developers by accelerating analyses such as simulation and enables unique design processes such as automated technology re-targeting. In my work published in PLDI 2023, I focus on one aspect of the problem: recognizing repeated logic in netlists and decompiling it into looping HDL code (as illustrated in Figure 2), termed *hardware loop rerolling* [8]. This work adapts programming languages techniques to the hardware domain to analyze netlists for repeated logic. Then, using a bespoke intermediate language, the hardware decompiler automatically generates looping HDL code that is semantically equivalent to the original netlist (proven so using an SMT solver).

Continuing this work I have mentored nine students, including three MS students as part of their theses, and one high school student through a summer research program. I currently have two papers under review addressing two more HDL abstractions beyond loops: standard module libraries [13], and memories [10]. Underpinning each new direction is a key programming language insight that enables hardware decompilation in a principled way. Recovering instances of modules from a standard library inside a gate-level netlist follows an inductive bottom-up merging technique guided by a type system describing generic library components. Memory decompilation follows a rewrite-driven approach where I developed an equational theory that encodes memory semantics into algebgraic rewrite rules. The benefit is that the term rewrites are bidirectional and so this approach enables not just decompilation but "re-compilation," porting a design targeted for one technology to another.



```
for (int i=0; i < 4; i++) begin
  t1[i] = (acc[i] ^ x[i]) ^ t0;
  t0 = ((acc[i] & x[i]) | (acc[i] & t0)) |
        (x[i] & t0);
end
```

Figure 2: Hardware loop rerolling.

Future Directions

I identify three areas of focus for my research program, outlined below. In each of these new thrusts I will develop new solver-aided programming techniques to address verifiability challenges in HDLs and open up new approaches to chip design.

Correct-by-construction chips from formal specifications. One of the insights of control logic synthesis is integrating architecture-level constraints into the generation of correct-by-construction HDL code. The long-term goal is to develop techniques that fully *derive* HDL code implementations for entire chips from formal architectural specifications, where compilation proceeds as a series of proof-carrying *refinements* moving down the layers of abstraction from the architecture-level to the microarchitecture, all the way to hardware. I have already outlined the theoretical foundations of this refinement process [6]. The insight is that effects at the architecture and microarchitecture levels can be viewed as monadic stream functions enabling correct-by-construction automated refinement of a circuit implementation from formal specifications.

This process works by abstracting microarchitectural optimizations and relating them to refinements as part of the compilation process. In support of this, there has been research into HDLs that encode microarchitectural information at the implementation level to create pipelines that obey timing constraints and avoid hazards [14, 5, 3]. However, these approaches place the burden on the programmer to write the appropriate types and have no integration with an architectural specification (foregoing formal verification). I envision that from the designer's perspective, these refinements are simply invoked as compiler passes. While at Galois, I did such research on correct-by-construction hardware generation focused on deriving automatically pipelined designs from formal specifications. Beyond automated pipelining, we need techniques that can derive many classes of microarchitecture-level optimizations to synthesize correct *and performant* chips.

Semantics-preserving EDA tools. Current HDL compilers and synthesis tools are decoupled, where synthesis passes have less semantic information to inform technology mapping and logic optimizations, and often need to recover lost information at the gate level. I am collaborating with researchers at the University of Washington to integrate state-of-the-art term rewriting systems to address challenges in EDA toolchains [11]. I am developing new languages for reasoning about hardware, powered by equational theories to rewrite designs according to cost functions—e.g., optimizing for power/area with compilation, or abstraction level for decompilation. I envision new open-source HDL compilers and synthesis tools which preserve design hierarchy through expressive type systems, exposing higher-level semantics (which are normally lost) to optimize downstream synthesis passes. My insights in developing hardware decompilers show how we can enrich synthesis tools by preserving higher-level abstractions during HDL compilation to increase developer productivity.

Enriching hardware/software interfaces. Besides general-purpose computing, there are other aspects of the hardware/software interface such as communication and data-movement protocols [4], and hardware accelerators that speed up specialized computation and system services [12]. The difficulty in using these interfaces lies in the development process; current languages and libraries do not provide the machinery to guide programmers to correctly use them. Further, these interfaces run through the whole hardware/software stack, encountering constraints through each layer. Characterizing these interfaces as abstractions found in programming language theory, I will develop new compilers that bridge the hardware/software interface at these new frontiers to ensure correctness of the use of—and even the automated synthesis of code using—these interfaces through symbolic reasoning and constraint solvers. My past work in control logic synthesis paves the way for exploring these new interfaces, assisting the programmer in reasoning about specialized hardware where they may have limited understanding.

Research funding. To deepen the work on hardware decompilation, I co-authored a grant proposal for an NSF Formal Methods in the Field (FMitF) grant. To fund my research program, I will pursue this and other NSF programs such as Secure and Trustworthy Cyberspace (SaTC), Principles and Practice of Scalable Systems (PPoSS), and Software and Hardware Foundations (SHF). I have also made connections with a US national laboratory and an industry research company who are interested in using the techniques developed in my research to solve problems in verified hardware design and indicated willingness to fund this work. Following my extensive experience mentoring undergraduates, I will also seek research funding to continue fostering research opportunities for undergraduates.

References

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

[2] J. L. Hennessy and D. A. Patterson. 2019. A new golden age for computer architecture. Commun. ACM 62, 2 (February 2019), 48–60. `doi:10.1145/3282307`.

[3] M. Jang, J. Rhee, W. Lee, S. Zhao, and J. Kang. Modular Hardware Design of Pipelined Circuits with Hazards. In *PLDI* 2024. `doi:10.1145/3656378`.

[4] H. Lu, Y. Xing, A. Gupta, and S. Malik. SoC Protocol Implementation Verification Using Instruction-Level Abstraction Specifications. ACM Trans. Des. Autom. Electron. Syst. 2023. `doi:10.1145/3610292`.

[5] R. Nigam, P. H. A. de Amorim, and A. Sampson. Modular Hardware Design with Timeline Types. In *PLDI* 2023. `doi:10.1145/3591234`.

[6] H. Kringen, **Z. D. Sisco**, J. Balkind, T. Sherwood, and B. Hardekopf. Semi-Automated Translation of a Formal ISA Specification to Hardware. Programming Languages for Architecture (PLARCH) 2023. `https://pldi23.sigplan.org/home/plarch-2023`.

[7] **Z. D. Sisco**, J. Balkind, T. Sherwood, and B. Hardekopf. A Position on Program Synthesis for Processor Development. Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE) 2022. `https://capra.cs.cornell.edu/latte22/paper/1.pdf`.

[8] **Z. D. Sisco**, J. Balkind, T. Sherwood, and B. Hardekopf. Loop Rerolling For Hardware Decompilation. In *PLDI* 2023. `doi:10.1145/3591237`.

[9] **Z. D. Sisco**, A. D. Alex, Z. Ma, Y. Aghamohammadi, B. Kong, B. Darnell, T. Sherwood, B. Hardekopf, and J. Balkind. Control Logic Synthesis: Drawing the Rest of the OWL. In *ASPLOS* 2024, Volume 4 (to appear). `doi:10.1145/3622781.3674170`.

[10] **Z. D. Sisco**, D. Petrisko, J. Xia, V. Rao, S. Wang, B. Hardekopf, and J. Balkind. A Memory Design Language for Automated Memory Technology Mapping. *Under review*, 2024.

[11] G. H. Smith, **Z. D. Sisco**, T. Techaumnuaiwit, J. Xia, V. Canumalla, A. Cheung, Z. Tatlock, C. Nandi, and J. Balkind. There and Back Again: A Netlist's Tale With Much Egraphin'. Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE) 2024. `https://capra.cs.cornell.edu/latte24/paper/8.pdf`.

[12] T. Wei, N. Turtayeva, M. Orenes-Vera, O. Lonkar, and J. Balkind. Cohort: Software-Oriented Acceleration for Heterogeneous SoCs. In *ASPLOS* 2023, Volume 3. `doi:10.1145/3582016.3582059`.

[13] J. Xia, **Z. D. Sisco**, S. Vasishta, J. Balkind, and B. Hardekopf. Mycelium: Module Finding with Functional Netlist Representation. *Under review*, 2024.

[14] D. Zagieboylo, C. Sherk, G. E. Suh, and A. C. Myers. PDL: a high-level hardware design language for pipelined processors. In *PLDI* 2022. `doi:10.1145/3519939.3523455`.