ADDRESSING THE VERIFICATION CHALLENGE OF AGILE HARDWARE METHODOLOGIES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by
Peitian Pan
May 2024

ADDRESSING THE VERIFICATION CHALLENGE
OF AGILE HARDWARE METHODOLOGIES

Peitian Pan, Ph.D.
Cornell University 2024

The slowdown of Moore's Law and the breakdown of Dennard scaling have driven computer architects towards more specialized hardware designs to meet the growing performance and energy efficiency demands. Such specialized hardware designs tend to have high non-recurring engineering (NRE) costs that hinder the research and development of promising hardware systems. The recent rise of agile hardware design methodologies addresses the high NRE costs by promoting the reuse of hardware designs and applying state-of-the-art software design and testing practices to hardware design. Unfortunately, agile hardware methodologies also face unique verification challenges in the hardware development process.

In this thesis, I identify and address key verification challenges in agile hardware methodologies. First, I address the verification challenges in dynamic HDLs. Modern HDLs embedded in dynamically typed programming languages facilitate the composition of statically typed hardware instances and dynamically typed test harnesses. However, existing dynamic HDLs suffer from the lack of early-in-design-cycle and complete safety guarantees for mixed-typed compositions and low simulation performance. I propose GT-HDL, an embedded HDL that leverage a combination of optional type checkers, guarded generator parameters, and type-based simulation optimizations to improve simulation performance of dynamic HDLs without compromising safety. Second, I address the verification challenges in generator creation. Recent HDLs heavily rely on parametrized and sophisticated hardware generators to achieve and maximize design reuse. However, it is challenging to verify the correctness of the hardware generators across the entire parameter space. To systematically and statically verify generator properties over a large parameter space, I propose symbolic elaboration, a static analysis technique based on satisfiability modulo theory (SMT) solving. Symbolic elaboration targets a synthesizable subset of HDL syntax and translates generator properties into integer constraints that can be solved by SMT solvers. Third, I overcome the verification challenges in instance composition. Latency-insensitive (LI) interfaces are critical to instance composition in agile hardware because they enable modular and composable hardware designs. However, it is challenging to verify the correctness of the LI interface handshake logic using dynamic verification techniques. I propose a formal verification solution to automatically verify hardware designs with LI interfaces that also generates counter example waveforms to debug LI handshake issues. Finally, I address the verification challenges in co-simulation. Modern HDLs used in agile hardware are typically embedded in a general-purpose host programming language. To maximize verification productivity, designers ideally should iterate on the target design using native simulations before generating RTL in a prototyping language such as Verilog. However, it is challenging to co-simulate the generated Verilog RTL and the test bench in the host language due to semantic gap between the host and the prototyping languages. To overcome this limitation, I propose seamless co-simulation based on a translation-import mechanism. I implement the translation-import mechanism in the PyMTL3 framework and demonstrate how it can improve verification productivity by reusing one test bench for both native simulation and co-simulation. In addition to the proposed solutions to verification challenges, I also present a coarse-grain reconfigurable array (CGRA) chip tape-out case study in GlobalFoundries 14nm technology. This case study demonstrates how the proposed solutions in this thesis can be integrated into an ASIC prototyping workflow and contribute to productive design and testing of the target hardware.

BIOGRAPHICAL SKETCH

Peitian Pan was born on March 13th, 1996 to Hu Pan and Hui Wang in Taian, Shandong, China. He is the son of a piano teacher mother and a civil servant father. In his early childhood, he enjoyed reading the illustrated encyclopedia and listening to his mother playing piano. Peitian's family got a computer when he was ten, and he started learning programming when he was in Taian No.6 High School. His computer class teacher Mr. Zhengqing Bu taught him the basics of C programming, simple algorithms and data structures and prepared him for participating the National Olympiad in Informatics in Province. It was also at this time that he picked up the much-needed vim editing skills. He continued his exploration of algorithms and data structures in Taian No.1 Senior High School and decided to pursue the study of computer science after high school.

Peitian was accepted to Shanghai Jiao Tong University as an undergraduate student majoring in computer science and engineering. He spent the next four years happily learning about operating systems, computer networks, programming languages, and more. As an undergraduate student, his favorite course was the introductory computer architecture course. For the first time he peeked through all layers of software/hardware abstractions and witnessed the clever designs of out-of-order CPUs and pipelined caches. With the curiosity to learn about the architectures of all kinds of hardware, Peitian applied to PhD programs in the United States at the end of his undergrad.

In 2018, Peitian was admitted to Cornell University as a PhD student in Electrical and Computer Engineering. He met Professor Christopher Batten at Cornell, who he was fortunate to work with in the next six years. At Batten Research Group, Peitian had the opportunities to work on projects on computer architecture, agile hardware methodologies, parallel programming, and ASIC prototyping, which helped him gain invaluable knowledge, experiences, and skills in various fields of computer engineering. During his PhD, Peitian interned at the Heterogeneous Platforms Lab of Intel Lab (mentored by Dr. Tanay Karnik) and at the Google TPU compiler team (mentored by Dr. Berkin Ilbeyi and Dr. Yunming Zhang). He is grateful for these industrial experiences which broadened his horizon to computer engineering challenges outside the academia.

Peitian is grateful for his PhD experience, which he found very worthwhile and rewarding. Despite the difficult work-from-home times during the pandemic, he was fortunate to have worked with brilliant colleagues, advisors, and mentors. It was truly a wonderful experience of invaluable personal growth and bonding with so many amazing people.

This document is dedicated to my parents and Jian, for their love and support.

ACKNOWLEDGEMENTS

I had both delightful and challenging times in my graduate career. I would like to thank my friends, colleagues, and mentors for making my PhD experience much more worthwhile and enjoyable.

First of all, I would like to thank my advisor Professor Christopher Batten. Chris is the best advisor I could have imagined having, and he provided me help and guidance on things more than just doing solid research. When I started my graduate career, I was not very good at communicating my ideas. Chris brought this up to me and offered advices that help me become a clear and efficient communicator. When I became a mid-career student and decided to pursue a position in the industry, Chris reminded me to develop my personal networks. He encouraged me to talk to more people at conferences and make more connections. I learned from Chris about the importance of constantly breaking out of my comfort zone. I think without Chris, I would not have gained the invaluable personal growth in entire PhD career.

I am also deeply thankful to my friends and colleagues at Batten Research Group for their help and support. Most of my PhD friends were more senior and they have profound impacts on me. I would like to thank Christopher Torng, who was a great collaborator and an inspiring model for me to take ownership and develop visions for my research projects. I entered the same time Berkin Ilbeyi was about to graduate and he was the JIT wizard who always managed to speed up Python. I am very thankful to Berkin for hosting me during my internship at Google's TPU compiler team and introducing me to the industrial ML compiler. Similar thanks go out to Ji Kim, who helped me onboard the TPU sparse core compiler team and was always fun to talk to. I have been fortunate to have the support from my PhD friends at BRG during the pandemic. Moyang Wang, you were the senior PhD student model and I will always remember your humorous comments on quantum computing. Shunning Jiang, you were a great mentor and helped me so much when I first joined the group; I learned about pursuing impactful research from you and will never forget the time we work together on PyMTL3. Khalid Al-Hawaj, you are a great friend and I will never forget how you never hesitate to help anyone in need. Tuan Ta, you were a generous researcher and were always willing to share all the implementation details of gem5; I enjoyed working with you on the vector graphic analytics project and having cheerful daily conversations. Lin Cheng, I will never forget rushing B with you after a long week and thank you for those get-togethers during the pandemic. Yanghui Ou, you are always the all-around Boss Ou to me and thank you for carrying me through the digital design course when I was an RTL newbie. Nick Cebry, it is amazing how quickly you pick things up and started leading your research project shortly into the group. Derin Ozturk, we did not overlap for long but it was always a pleasure to learn about robotics from you. I have also been fortunate to work with brilliant postdoc researchers at BRG. Dr. Cheng Tan, thank you for your invaluable input when I was struggling with CGRAs and their compilers. Dr. Shady Agwa, you were always so willing to help and thank you for teaching me about the PDKs and helping me on VLSI results of the CGRA. Dr. Austin Rovinski, it was always a pleasure to hear about VLSI and open-source EDA from you and I wish you a successful career as a professor. I want to thank the younger researchers at BRG for being great fun to work with: Kaishuo Cheng, Cameron Haire, Krithik Ranjan, Eric Hall, Courtney Golden, Yixiao Zhang, Raymond Yang, Dhruv Sharma, Megha Shyam, and Anya Prabowo.

The Computer System Laboratory (CSL) has been a supportive and caring community and I am thankful for my CSL friends, colleagues, and mentors. I want to thank Professor Zhiru Zhang and Professor Adrian Sampson for agreeing to be on my committee and providing invaluable insights and guidance on my research. I also want to thank Zhiru and Adrian for teaching me high-level synthesis and programming languages and inspiring me to pursue the hardware design methodology research. I would also like to thank my fellow PhD friends at CSL. Philip Bedoukian, I totally enjoyed the cheerful conversations and I look forward to more Counter-Strike sessions with you. Neil Adit and Rachit Nigam, I am glad you are around to chat about compilers and programming languages. Helena Caminal, Kailin Yang, Cecilio C. Tamarit, and Socrates Wong, I took inspirations from your computer architecture research to develop better methodologies to realize your ideas. Ritchie Zhao and Jordan Dotzel, thank you for your hard work leading the CSL student steering committee to make CSL a better community for everyone. It was always great fun working with the younger generation of CSL researchers: Lisa Li, Yuzong Chen, and Preslav Ivanov. I am also indebted to many friends for their help in both life and work: Yu Gan, Shuang Chen, Yanqi Zhang, Mulong Luo, Yichi Zhang, Chenhui Deng, Hanchen Jin, Mingyu Liang, Zhuangzhuang Zhou, Jie Liu, Zichao Yue, and Niansong Zhang.

From the broader architecture community, I am thankful to Professor Michael B. Taylor and his research group at University of Washington, who were amazing collaborators during the SDH project and the subsequent Bigblade tapeout. There are many people I would like to thank here: Dustin Richmond, Dai Cheol Jung, Max Ruttenberg, and Daniel Petrisko. I owe a special thank-you to Daniel Petrisko, who generously agreed to help the Cornell team and integrated the CGRA module into the Bigblade tapeout. From the formal verification community, I would like to thank Professor Cunxi Yu at University of Maryland, College Park and Marcelo Orenes-Vera at Princeton University. Their insights into formal verification were invaluable while I was trying to step into the formal verification realm. I am also thankful to the mentors of my internships, who answered my naive questions and opened my horizon to challenges in the industry. Berkin Ilbeyi and Phitchaya Mangpo Phothilimthana, it was great fun learning about ML compilers and how ML itself can make its compilers better. Yunming Zhang, Josh Varty, and Ji Kim, thank you for teaching me the sparse side of things of ML and I look forward to working with you in the future. Tanay Karnik, I was amazed by state-of-the-art heterogeneous architectures and am thankful for your mentoring throughout my time at Intel Lab.

Finally, I would like to thank my parents for raising me up and supporting me through my PhD career. It was really rough during the pandemic time and I look forward to coming back home to see them. I also want to thank Jian Zhou for his support – this thesis would not have been possible without him.

This thesis was supported in part by DARPA POSH Award #FA8650-18-2-7852, DARPA SDH Award #FA8650-18-2-7863, the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, a research gift from Xilinx, Inc., and equipment, tool, and/or physical IP donations from Intel, Synopsys, Cadence, and ARM. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

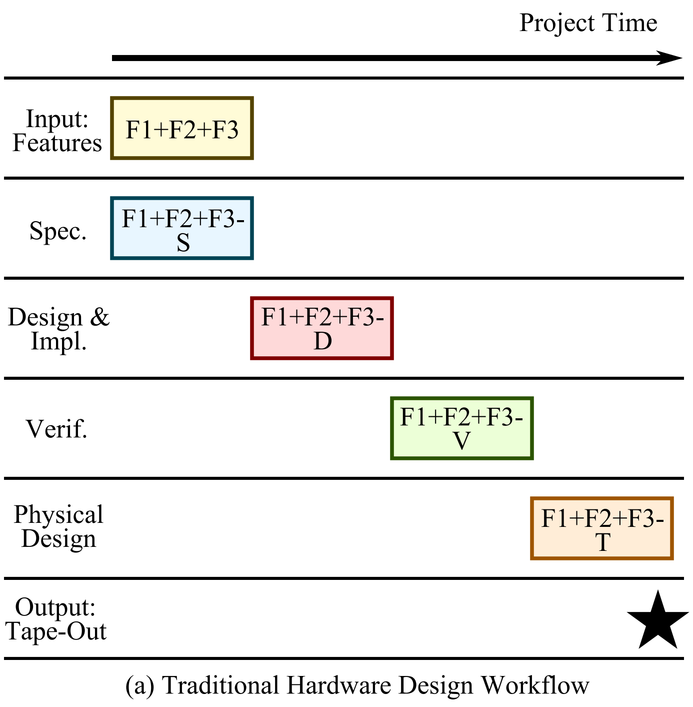
LIST OF ABBREVIATIONS

NRE	non-recurring engineering
CAD	computer aided design
VLSI	very-large-scale integration
ASIC	application-specific integrated circuit
FPGA	field-programmable gate array
HDL	hardware description language
RTL	register-transfer level
LI	latency-insensitive
IP	intellectual property
UVM	universal verification methodology
DSL	domain-specific language
AST	abstract syntax tree
RTLIR	register-transfer level intermediate representation
IR	intermediate representation
DUT	design under test
DUV	design under verification
TB	test bench
HLS	high-level synthesis
AoT	ahead-of-time
SMT	satisfiability modulo theories
CFFI	C Foreign Function Interface
DVFS	dynamic voltage and frequency scaling
DMA	direct memory access
SoC	system-on-chip
NoC	network-on-chip
CGRA	coarse-grain reconfigurable array
PE	processing element
ISA	instruction set architecture
SRAM	static random access memory
DRAM	dynamic random access memory
I$	instruction cache
D$	data cache

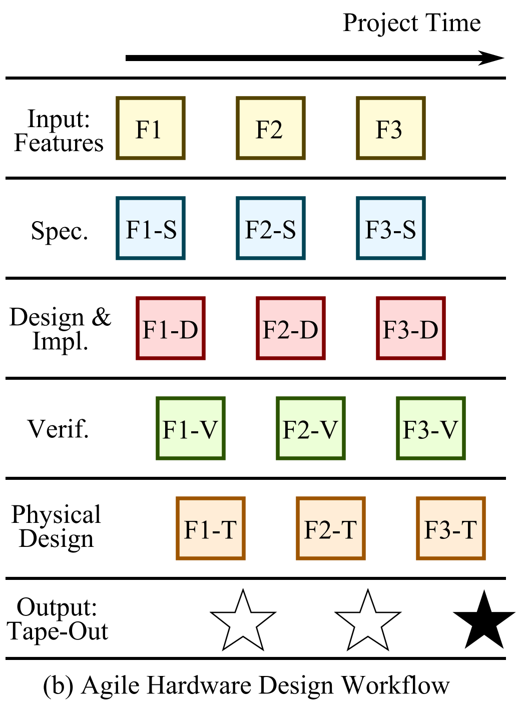
CHAPTER 1
INTRODUCTION

With Moore's Law and Dennard scaling coming to an end [Dub05], computer architects have turned to specialized hardware to satisfy the energy efficiency and performance needs of emerging applications [WLP+14,CKES17,AHY+15,CLX+16,HLM+16]. In response to the high NRE costs of specialized hardware [KZVT17], researchers propose agile hardware methodologies, which promises productive hardware design and testing. Contrary to the existing waterfall model of hardware development which consists of sequential steps, agile hardware design prioritizes creating a small yet working hardware prototype and iterates on this prototype to implement more features [LWC+16]. Unfortunately, agile hardware methodologies often lead to unique challenges in the verification step of hardware development. First, the increasing adoption of dynamically typed test harnesses creates challenges for safe and performant mixed-typed compositions in modern HDLs. Second, the promotion of parametrized and sophisticated hardware generators makes it challenging to statically verify generator properties ahead of time. Third, the ubiquitous latency-insensitive (LI) interfaces in agile hardware design facilitates modularity and composition of components but complicates the verification of its handshake logic. Finally, the use of a high-level domain-specific language (DSL) in recent HDLs poses a challenge to productively verify the functionality of the translated Verilog RTL against the DSL hardware model.

In this thesis, I take a vertically-integrated approach to address the key verification challenges in agile hardware methodologies. I identify the verification challenges in a representative agile hardware design workflow: dynamic HDLs, generator development, instance composition, and co-simulation. I argue that innovations across all steps in the workflow are necessary to fulfill the promise of productive hardware design and verification: to enable safe and performant mixed-typed composition, I propose GT-HDL to safeguard compositions and selectively remove runtime type checks for better simulation performance. To enable verification of generator properties, I leverage SMT solving to statically analyze parametrized and sophisticated generators. To enable verification of LI interfaces in RTL instance composition, I leverage formal verification to determine the correctness of LI handshake logic. To enable verification of the generated RTL, I propose a translation-import mechanism to achieve seamless co-simulation of the translated RTL in the high-level DSL test bench.



(a) Traditional Hardware Design Workflow



(b) Agile Hardware Design Workflow

Figure 1.1: Traditional vs. Agile Hardware Design Workflow – Spec.: specification. Design & Impl.: design and implementation. Verif.: verification. In both figures, F1 through F3 represent features to be included in the hardware. Specification, design and implementation, verification, and physical design represent the typical steps in hardware design. The star symbols indicate the tape-out of the target hardware. ☆: partial-featured yet manufacturable prototype; ★: full-featured and manufacturable prototype. (a) a waterfall workflow adopted by traditional hardware methodologies; this workflow revolves around a fully featured prototype (F1+F2+F3) and components and tools used in the workflow are specific to this instance; (b) an iterative workflow adopted by agile hardware methodologies; this workflow emphasizes on reusable components and tools to enable smoother, iterative inclusion of features (F1, F2, F3). Figure inspired by [LWC$^+$16].

1.1 Agile Hardware Methodologies

Hardware prototyping is a sophisticated activity that generally spans across several sequentially dependent steps: specification, where the behaviors of the target hardware system are determined and specified unambiguously; design and implementation, where the target hardware is divided into subcomponents and implemented at the register-transfer level (RTL); verification, where the implemented RTL is tested against the specification to identify violations; and physical design, where the verified RTL is synthesized, placed, and routed to produce ASIC layout or an FPGA bitstream.

Figure 1.1 contrasts the workflow of a traditional hardware design versus that of an agile hardware design. Traditional hardware methodologies adopt a waterfall workflow (Figure 1.1(a)) with a strict sequence of prototyping steps. For example, the specification of features F1 through F3 has to be finished before the design and implementation step. The hardware including all desired features has to be completed before the verification steps can start. And the physical design has to wait until the verification of all features are done. The waterfall workflow enables collaboration between hardware design engineers of different expertise. However, it has high cost of communicating the design intent from one step to another and the risk of subtle misunderstandings between different teams [LWC^{+}16]. Verification with traditional hardware methodologies can also be challenging due to the large number of features in the target hardware. And since each step in the waterfall workflow is centered around a monolithic hardware design of a large number of features, traditional hardware methodologies encourage use of instances and specialized tools and scripts over generic and reusable components.

Unlike the traditional hardware methodologies, agile hardware methodologies typically adopt an iterative workflow. The agile hardware design workflow generally revolves around one manufacturable hardware prototype which can be a minimal, feature-incomplete prototype at an early stage of the development. Hardware designers iteratively add new features to the prototype and push the prototype through all steps after each feature is added. Figure 1.1(b) shows that with agile hardware methodologies, the desired features F1 through F3 are integrated into the target hardware via an iterative workflow. Hardware designers work on a small subset of features (potentially one) at a time and each feature is executed deep into the physical design step. Compared to the traditional hardware methodologies, agile hardware methodologies reduce the risk of misunderstanding in design intents due to fewer number of features. Agile hardware methodologies can potentially reduce the verification step time because with a successful previous iteration, each iteration exposes a much smaller surface of change than with a complete feature set. With an iterative workflow, agile hardware methodologies encourage use of parametrized generators and reusable tools.

Through the above comparison between agile and traditional hardware methodologies, I highlight two principles of agile hardware design described by [LWC^{+}16].

- Incomplete, fabricatable prototypes over fully featured models.
- Improvement of tools and generators over improvement of the instance.

The first principle states that agile hardware methodologies generally favor incomplete yet working prototypes. As shown in Figure 1.1(b), an iterative workflow centers around models with incomplete features, and features are added to this prototype at a much finer granularity than in the traditional workflow. The second principle highlights the importance of creating and maintaining reusable components throughout the prototyping steps to boost productivity. The traditional workflow in Figure 1.1(a) concerns only the fully featured model at each step, which tends to encourage components and tools that are specialized for the said instance. On the other hand, agile hardware methodologies promote reusable components through iterative feature additions and incrementally increase the development velocity. Putting together, these two principles promise a productive iterative workflow where feature additions become smoother by reusing quality components and tools.

1.2 Implementing Agile Hardware Methodologies

To encourage the wider adoption of agile hardware methodologies, the research and the engineering communities have developed tools and frameworks that facilitate agile hardware. Table 1.1 provides an overview of existing work to facilitate the specification, design and implementation, verification, and physical design of agile hardware. In this thesis, I specifically focus on research that identifies and tackles agile hardware verification challenges. The remaining section provides a more thorough survey of representative tools and frameworks that implement agile hardware methodologies.

High-level synthesis (HLS) [CLN+11,CCA+11] is a productive hardware development methodology where the HLS tools accept a high-level description of hardware behaviors (usually in C or its variants) and synthesize the description into timed RTL implementations. HLS raises the level of abstractions so that agile hardware designers can focus on specifying critical hardware behaviors and leave most of the RTL implementation to HLS tools.

HeteroCL [LCH+19] is a programming framework that facilitates the programming of FPGAs. HeteroCL implements a novel Python-based domain-specific language (DSL) that decouples the actual algorithm to be implemented from the customizations of computation, data types, and memory architectures. With HeteroCL, programmers can efficiently explore various tradeoffs between performance and accuracy and generate FPGA implementations.

Some existing works tackle the specification step from the programming language perspective and propose DSLs to raise the level of abstraction. Dahlia [NAT+20] is a language that targets predictable architecture generation. To avoid generating hardware of significant resource require-

		Specification	**Design & Implementation**	**Verification**	**Physical Design**
High-Level Synthesis	[CLN+11]	✓	✓		
LegUp	[CCA+11]	✓	✓		
HeteroCL	[LCH+19]	✓	✓		
Dahlia	[NAT+20]	✓	✓		
Calyx	[NTLS21]	✓	✓		
Filament	[NdAS23]	✓	✓		
BaseJump STL	[Tay18a]		✓		
OpenPiton	[BCJ+20]		✓		
Rocket Chip	[AAB+16]		✓		
OpenCGRA	[TAZ+21]		✓		
OpenFPGA	[TGC+20]		✓		
PRGA	[LW21]		✓		
PyOCN	[TOJ+19]		✓		
Constellation	[ZANA22]		✓		
Chipyard	[ABG+20]		✓	✓	
MyHDL	[Dec04]		✓	✓	
Migen	[Mig23]		✓	✓	
SpinalHDL	[Spi23]		✓	✓	
Magma	[Han24]		✓	✓	
Cλash	[BKK+10]		✓	✓	
Chisel	[BVR+12]		✓	✓	
FIRRTL	[IKL+17]		✓	✓	
PyRTL	[DTS20]		✓	✓	
PyMTL3	[JPOB20]		✓	✓	
fault	[THS+20]			✓	
cocotb	[Git24]			✓	
PyVSC	[Bal24]			✓	
BlueCheck	[NM15]			✓	
CoSA	[MMB+18]			✓	
MinJie	[XYT+22]			✓	
FireSim	[KMK+18]			✓	
mflowgen	[CTN+22]				✓
Hammer	[LGW+22]				✓
Silicon Compiler	[ORM22]				✓

Table 1.1: Existing Works that Implement Agile Hardware Methodologies – ✓: this work facilitates the corresponding step in an agile hardware workflow.

ments, Dahlia adopts an affine type system to systematically rules out programs of unclear or expensive hardware implementations. Dahlia can generate C++ code suitable for HLS tools which can then be used to drive an ASIC or FPGA flow. Calyx [NTLS21] is an intermediate representation for hardware designs. It allows structural description of the hardware with control flow primitives for productive specification of hardware accelerators. Unlike Dahlia which generates HLS C++ output, Calyx directly outputs RTL code for the target hardware. Filament [NdAS23] is a language for specifying modular hardware designs with enforced timing and structural constraints. Filament uses timeline types to describe the latency of static pipelines and ensures that statically scheduled modules always produce and consume messages at the right cycle.

The BaseJump standard template library [Tay18a] is a collection of highly parametrized hardware generators described in the SystemVerilog HDL. Hardware designers reuse the IPs in the BaseJump template library by instantiating the generators with the desired parameters in their SystemVerilog source code.

OpenPiton is a framework for simulating, emulating, and building manycore processors [BCJ+20]. As a library of IPs, OpenPiton provides a parametrized and extensible manycore design that includes processors, caches, interconnects, and peripherals. In addition to facilitating IP reuse, OpenPiton also distributes scripts of CAD tools to help reuse scripts in the emulation and prototyping of manycore hardware.

The Rocket chip generator [AAB+16] is a library of parametrized RISC-V processors, caches, and interconnects in the Chisel hardware construction language [BVR+12]. Compared to BaseJump and OpenPiton, the Rocket chip generator leverages Scala, the general-purpose programming language in which Chisel is embedded, to implement sophisticated hardware generators.

Generators for reconfigurable architectures have also witnessed a surge of interests in recent years. OpenCGRA [TXL+20, TAZ+21] is an open-source framework for modeling, testing, and evaluating coarse-grain reconfigurable arrays (CGRAs). CGRAs consist of loosely interconnected functional units and are a promising hardware architecture of more flexibility than ASIC accelerators and higher energy efficiency than FPGAs. OpenCGRA supports a full design flow from functional-level modeling all the way to area, power, and timing characterization. OpenFPGA [TGC+20] is an open-source framework for agile FPGA prototyping. Comparing to CGRAs, FPGAs achieve higher flexibility with finer-grain bit-level reconfigurability. OpenFPGA allows customization of FPGAs through a high-level architecture description language and supports generation of Verilog netlists that can drive an ASIC or FPGA flow. Princeton Reconfigurable Gate Array (PRGA) [LW21] is a customizable open-source framework for building custom FPGAs. It is capable of generating synthesizable Verilog RTL from user specifications of FPGA architectures and open-source CAD tool scripts. PRGA facilitates exploring FPGA architectures for emerging application domains and the prototyping of such architectures.

Network-on-chip (NoC) IPs also attract agile hardware designers because they provide flexible interconnection of cores, accelerators, memory systems, and more. PyOCN [TOJ+19] is an open-source framework that supports modeling, testing, and evaluation of NoC IPs. PyOCN offers a wide selection of NoC IPs from functional-level descriptions to detailed register-transfer level implementations. It also provides a comprehensive testing suite of unit tests, integration tests, and property-based random tests. Furthermore, PyOCN enables productive evaluations of NoC IPs with user-friendly NoC generation, simulation, and characterization. Constellation [ZANA22] is an open-source NoC IP generator specialized for heterogeneous SoCs. In addition to supporting NoC RTL generation, testing, and characterization, Constellation also leverages a parameter system, which allows specification of logical, physical, and routing behaviors, to customize NoC IPs.

FIRRTL is a hardware compiler framework that serves as an IR between Chisel [BVR+12] and synthesizable Verilog [IKL+17]. Chisel is a hardware construction language embedded in Scala, which enables sophisticated and parametrized hardware generators and acts as one frontend of FIRRTL. FIRRTL automatically performs analysis, transformations, and optimizations on the IR of the target hardware design and supports lowering into HDLs such as Verilog for prototyping.

PyRTL is a toolkit of Python-based hardware development utilities that bridges the designing, testing, analysis, and evaluation aspects of agile hardware [DTS20]. PyRTL allows describing the design, testing, analysis, and evaluation tasks of prototyping using modern Python programming features, which enables fast design iterations on a wide range of hardware generators.

Also based on Python, PyMTL3 is a framework for hardware modeling, generation, simulation, and verification [JPOB20]. Similar to PyRTL, PyMTL3 also draws on the expressiveness of Python to facilitate reusing some prototyping tasks as Python code. In addition, PyMTL3 also supports direct co-simulation between PyMTL3 models and Verilog models.

In addition to the above works, MyHDL [Dec04], Migen [Mig23], SpinalHDL [Spi23], Magma [Han24], and Cλash [BKK+10] are all HDLs embedded in a high-level programming language. These works enable highly parametrized generators by constructing a hardware module using the productive features of the host programming language, which facilitates the design and implementation step of an agile workflow. They also allow using the target hardware as a simulatable object to improve verification productivity.

fault [THS+20] is a verification language embedded in Python. Thanks to its close integration with the Magma hardware description language, fault allows detailed inspection of the target hardware and the reuse of verification components across numerous commercial and open-source verification tools. fault provides a unified interface to constrained random testing and formal verification, making these two forms of verification more productive in an agile workflow.

cocotb [Git24] is a library for digital circuit verification in Python. Unlike traditional verification approaches which requires complex testbenches in SystemVerilog, cocotb allows using coroutines in Python as testbenches and interfaces arbitrary Python code with the DUT running in a simulator. cocotb can significantly improve the verification productivity in an agile workflow thanks to the productive features of Python.

PyVSC [Bal24] is a verification library embedded in Python with a focus on providing constrained random stimulus generation and coverage data collection. Inspired by the constrained random and functional coverage tools for SystemVerilog, PyVSC provides familiar syntax for agile hardware designers and improves the verification productivity for any tests using constrained randomization and coverage metrics.

BlueCheck is a generic synthesizable test bench capable of automatic test sequence generation and counter-example shrinking [NM15] in BlueSpec HDL. BlueCheck enables reuse of a significant number of test bench utilities by parametrizing the generic test bench with a correctness specification of the target hardware. The correctness specification, which is implemented as an executable in BlueSpec HDL, determines whether an input-output combination of the hardware is valid. Therefore, the verification process can be automated and reused for a vast number of hardware designs.

CoSA is a symbolic model checker that enables close integration of formal verification into agile hardware [MMB+18]. CoSA targets hardware designs in the CoreIR intermediate representation, which supports commonly used hardware description formats including Verilog and preserves the design intents in these formats to assist formal verification.

MinJie is an open-source platform for agile processor development including tools for functional verification [XYT+22]. Similar to BlueCheck, MinJie adopts an agile verification methodology where the test bench is parametrized by the correctness specification known as diff-rules.

The use of parametrized test benches eliminates the needs of creating test benches for each processor design and significantly improves verification productivity. MinJie also includes functional verification tools that facilitate bug reproducing and visualization.

FireSim is a verification technique for cycle-exact simulation on cloud FPGAs [KMK$^+$18, ABG$^+$20]. Compared to prior FPGA-accelerated simulation tools, FireSim has significantly better scalability and is capable of simulating thousands of processors. FireSim improves verification productivity of large-scale hardware systems at affordable cost and acceptable simulation speed.

mflowgen is a modular flow generator with a focus of improving physical design productivity [CTN$^+$22]. To enable parametrization of the physical design flow, mflowgen implements an instrumentation layer in Python over Tcl scripts, which helps preserve reusable physical design code in the face of needs for specializations.

Hammer is a modular physical design flow tool that is compatible with numerous commercial and open-source CAD tools and technology nodes [LGW$^+$22]. Similar to mflowgen, Hammer is implemented in Python and manages the entire physical design flow through customizable hooks and plugins. In addition, Hammer adopts a human-readable intermediate representation that specifies parameters and operations for certain steps in the physical design process.

Silicon compiler is a modular hardware build system also implemented in Python [ORM22]. Similar to mflowgen and Hammer, silicon compiler adopts a tool abstraction layer over vastly different CAD tools. It also supports a readable representation of ASIC/FPGA design intents which can be parsed by modular plugins of the build system.

1.3 Verification Challenges in Agile Hardware Methodologies

Despite their success at managing the NRE costs of hardware designs, agile hardware methodologies also face unique challenges in verification. Section 1.2 identified numerous existing works that facilitate verification. However, the verification challenges in agile hardware methodologies remain and this thesis focuses on identifying and addressing these challenges. This section introduces four verification challenges that I specifically focus on in this dissertation. The list of verification challenges is not exhaustive and future research may identify and address more verification challenges to make agile hardware methodologies even more productive.

Verification Challenges in Dynamic HDLs – Dynamic HDLs, which are embedded in dynamically typed programming languages such as Python, facilitate the composition of statically typed hardware instances and dynamically typed test harnesses. However, existing dynamic HDLs often lack early static type checking to type check statically typed hardware instances, generally fail to safeguard mixed-typed compositions, and suffer from low simulation performance. Without safe mixed-typed compositions, agile hardware designers may encounter subtle, hard-to-debug errors from the statically typed hardware instances which are caused by ill-typed messages or parameters from dynamic components. An ideal solution to this challenge requires both static type checking or analysis techniques that target commonly used hardware constructs in dynamic HDLs *and* carefully inserted type guards to safeguard mixed-typed compositions without sacrificing simulation performance.

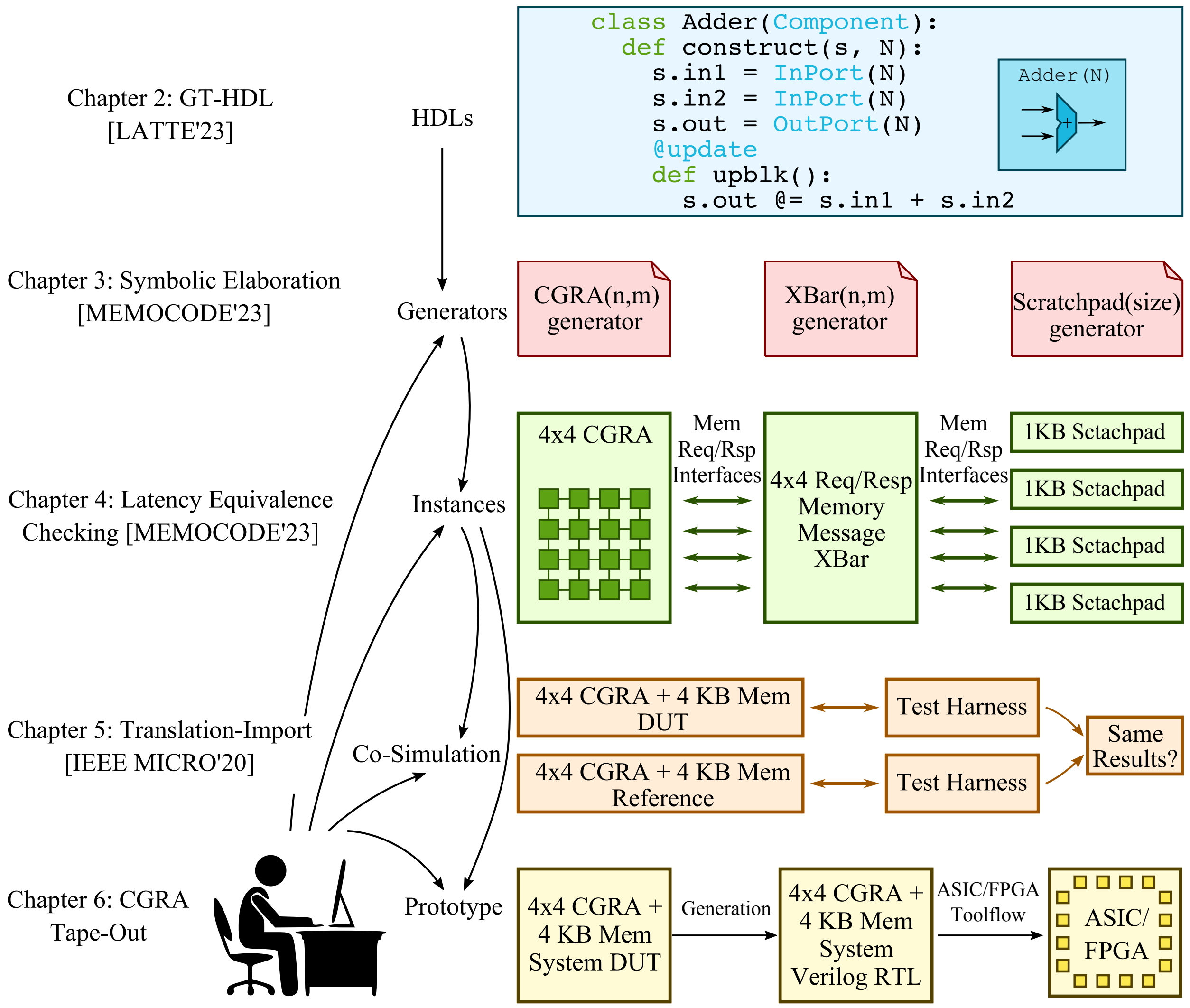
Verification Challenges in Generator Development – Hardware generators are a central concept to IP reuse in agile hardware design. Recent HDLs for agile hardware methodologies typically define a set of core modeling primitives and are often embedded in a host general-purpose programming language. Generators are often expressed as a function in the host language which takes generators parameters and constructs the desired hardware instance. Certain generator properties have to hold before it can be reused. For example, a generator should have matching bitwidth for all connections under all possible combinations of parameters. A synthesizable generator should not include cross-hierarchy accesses under all possible combinations of parameters. And a generator should not include any out-of-bound array accesses under all possible parameter combinations. The verification challenges in generator development arise because the potentially enormous space of generator parameters renders most traditional verification approaches that exhaustively search through the parameter space intractable. Static analysis approaches are a promising approach to verifying generator properties. However, existing static analysis tools generally do not target the HDL syntax commonly used in agile hardware methodologies and fail to extract the hardware-specific semantics from the syntactic constructs in the host programming languages. A solution to these verification challenge has to be tractable even for generators with a large parameter space and is capable of reasoning about generator parameters.

Verification Challenges in Instance Composition – Latency-insensitive (LI) interfaces [CMSV99, CMSSV99] are a critical enabler of modular and reusable hardware designs. By decoupling the output of a hardware design from its internal timings, the LI interfaces are pivotal to the seamless composition of hardware instances. However, the complex LI handshake protocols are often a source of design bugs in agile hardware methodologies. For example, in a processing element that consumes input from multiple LI interfaces, the hardware designer has to carefully orchestrate the LI handshake logic to make sure the input messages are consumed at exactly the same cycle. Furthermore, certain subtle LI handshake bugs can only be triggered by stalling the LI interface for a specific number of cycles. The verification challenges in instance composition arise due to the difficulty of creating test vectors that trigger the potential LI handshake design bugs. It takes tremendous efforts to construct a comprehensive test suite that covers almost all stall events of a LI interface to discover LI handshake bugs. Even worse, such comprehensive test suites are highly unlikely to be reusable due to the unique functionalities and internal timings of hardware modules. An ideal solution to these verification challenge can be automatically reused across different LI RTL modules and produce concise counter-examples to facilitate debugging.

Verification Challenges in Co-Simulation – Existing HDLs for agile hardware methodologies are typically embedded in general-purpose programming languages and have their own primitives for hardware modeling. To improve interoperability with open-source and commercial computer-aided design (CAD) tools, most modern HDLs support translation into Verilog or SystemVerilog [BVR+12, IKL+17, DTS20, LZB14, JPOB20, Dec04, Mig23], the best supported HDL for FPGA or ASIC prototyping. The verification challenges in co-simulation arise because the test benches for the translated RTL are typically written in the host programming language of the HDL and therefore cannot be productively reused to test the translated RTL. Some agile hardware methodology tools choose to implement a read-write compatibility layer between the translated Verilog RTL and the test bench in the host language using the Verilog programming interface [BVR+12, JB99]. However, this design choice hinders verification productivity because the target hardware has to be translated into Verilog RTL before verification. Ideally, the solution to co-simulation verification challenges should support both native simulations and co-simulations using the same test bench in the host language.

1.4 Thesis Overview

To identify and address the key verification challenges in agile hardware methodologies, I adopt a vertically integrated research approach to identify verification challenges in typical agile hard-


Chapter 2: GT-HDL [LATTE'23]
HDLs
Chapter 3: Symbolic Elaboration [MEMOCODE'23]
Generators
Chapter 4: Latency Equivalence Checking [MEMOCODE'23]
Instances
Chapter 5: Translation-Import [IEEE MICRO'20]
Co-Simulation
Chapter 6: CGRA Tape-Out
Prototype
Adder(N)
CGRA(n,m) generator
XBar(n,m) generator
Scratchpad(size) generator
4x4 CGRA
Mem Req/Rsp Interfaces
4x4 Req/Resp Memory Message XBar
Mem Req/Rsp Interfaces
1KB Sctachpad
1KB Sctachpad
1KB Sctachpad
1KB Sctachpad
4x4 CGRA + 4 KB Mem DUT
Test Harness
4x4 CGRA + 4 KB Mem Reference
Test Harness
Same Results?
4x4 CGRA + 4 KB Mem System DUT
Generation
4x4 CGRA + 4 KB Mem System Verilog RTL
ASIC/FPGA Toolflow
ASIC/ FPGA

```
class Adder(Component):
  def construct(s, N):
    s.in1 = InPort(N)
    s.in2 = InPort(N)
    s.out = OutPort(N)
    @update
    def upblk():
      s.out @= s.in1 + s.in2
```

Figure 1.2: Thesis Overview – This figure shows the key components in a typical agile hardware design workflow starting from HDLs to an FPGA or ASIC prototype. Chapter 2, Chapter 3, Chapter 4, and Chapter 5 identify and address verification challenges in the discussed workflow. Chapter 6 presents a case study of a CGRA tape-out in GlobalFoundries 14nm technology to demonstrate the effectiveness of the proposed techniques in this thesis.

ware workflows: the dynamic HDL verification challenge, the generator property verification challenge, the composition interface verification challenge, and the seamless co-simulation verification challenge. I also demonstrate the effectiveness of the proposed solutions with a CGRA tape-out case study.

Chapter 2 focuses on the dynamic HDL verification challenge and proposes gradually typed HDL (GT-HDL), an embedded HDL with safe and efficient mixed-typed compositions. To provide safety guarantees for mixed-typed compositions, GT-HDL incorporates guarded generator parameters and simulation type checks. These type guards are inserted to prevent ill-typed parameters or values from propagating from dynamically typed components into statically typed hardware instances, which reduces debugging efforts involved in mixing dynamic test harnesses with DUTs. In addition, GT-HDL also employs typed-based simulation optimizations to improve simulation performance without sacrificing safety guarantees. I present an evaluation of GT-HDL on eight RTL modules and demonstrate up to 48.1% better simulation performance over a state-of-the-art dynamic HDL. This chapter is based on my paper published at the Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE) held in conjunction with the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) in 2023 [POJB23].

Chapter 3 focuses on the generator property verification challenge, which aims to prove or falsify common generator properties (such as matching bitwidths, in-bound array indices, etc.) ahead of time. This chapter proposes symbolic elaboration, a constraint-solving technique that targets the PyMTL3 hardware modeling syntax to statically prove or falsify generator properties. Symbolic elaboration leverages the observation that many critical generator properties can be encoded as integer constraints over the generator parameters. Furthermore, these integer constraints can be effectively solved by an SMT solver. This chapter presents a detailed evaluation of symbolic elaboration and shows that on average, symbolic elaboration can detect 90.6% of randomly injected bugs to a benchmark of five IP generators and three design generators. This chapter is based on my paper published at the International Conference on Formal Methods and Models for Co-Design (MEMOCODE) in 2023 [PJOB23].

Chapter 4 focuses on the composition interface verification challenge. Specifically, this chapter describes the challenge of verifying the handshake logic of latency-insensitive (LI) interfaces which are ubiquitous in agile hardware designs. This chapter formalizes the behavior of LI interfaces through the concept of stall invariant, which abstracts the messages communicated over an LI interface as an untimed sequence of events. This chapter also proposes latency equivalence checking, a formal verification technique to automatically verify or falsify the stall invariant property of a given LI RTL module. The main contribution of latency equivalence checking includes an algorithm that constructs a verification harness targeting the LI interfaces of a given RTL module. To demonstrate the effectiveness of the proposed technique, this chapter also includes case studies that apply latency equivalence checking to real-world hardware designs including a RISC-V processor. In all cases, latency equivalence checking successfully identified the bug in the studied design and generates counter-examples for reproducibility. This chapter is based on my paper published at the International Conference on Formal Methods and Models for Co-Design (MEMOCODE) in 2023 [PB23].

Chapter 5 focuses on the seamless co-simulation verification challenge. Specifically, this chapter describes the challenge of productively testing the functionality of the generated Verilog RTL using a test bench in the host programming language. This chapter discusses translation and import mechanisms that enable reuse of the same test bench on both the DSL and the Verilog hardware model. The translation mechanism relies on a robust IR called register-transfer level intermediate representation (RTLIR), which provides a canonical representation of all synthesizable hardware instances in the PyMTL3 DSL. To enable co-simulation of the translated RTL with Python test benches, this chapter introduces the import mechanism, which is built on the shared interface between the PyMTL3 and the compiled Verilog hardware models. This chapter also presents a case study on the ultra-elastic CGRA (UE-CGRA) to demonstrate how translation-import significantly improves the productivity of evaluating the ultra-elastic CGRA (UE-CGRA), a complex digital design with multiple clock networks. The UE-CGRA case study is based on the paper I co-authored and published at the International Symposium on High-Performance Computer Architecture (HPCA) in 2021 [TPO+21].

Chapter 6 presents a case study of a CGRA tape-out in GlobalFoundries 14nm technology using the techniques proposed in the previous chapters. This chapter describes the architecture of the target CGRA, the challenges of verifying the target CGRA, and the solutions employed in the tape-out. The CGRA tape-out leverages symbolic elaboration to verify the properties of the CGRA generator. It also leverages the latency equivalence checking to verify the correctness of LI handshake in the CGRA's processing element (PE). Finally, the translation-import mechanism allows for a sophisticated direct, random, and property-based testing suite for the CGRA to achieve high confidence of design correctness. This case study shows that the techniques proposed in the thesis significantly improve the verification productivity of the CGRA design.

The primary contributions of this thesis are:

- I propose GT-HDL, an embedded HDL that supports safe and performant mixed-typed compositions. I apply GT-HDL to three standalone hardware designs and demonstrate that it improves simulation performance without compromising safety.

- I propose symbolic elaboration, an SMT solving-based static analysis approach to validating hardware generator properties. I apply symbolic elaboration on a suite of IP and design generators and demonstrate its effectiveness on the generator property verification challenge.
- I propose the stall invariant property and latency equivalence checking, a formal verification technique that automatically proves the correctness of or finds falsifying inputs to the stall invariant property. I apply latency equivalence checking to three LI designs and demonstrate that the proposed technique correctly identify and provides easy-to-debug LI handshake waveforms.
- I propose and implement a translation-import mechanism in the PyMTL3 HDL to enable productive verification of the generated RTL against the high-level DSL hardware model. I apply the translation-import mechanism to co-simulate an ultra-elastic CGRA (UE-CGRA) and demonstrate it significantly improves verification productivity.
- I present a case study of a CGRA tape-out in the GlobalFoundries 14nm technology and demonstrate that the techniques proposed in this thesis help overcome the verification challenges of the CGRA module.

1.5 Collaboration, Previous Publications, and Funding

This thesis would not have been possible without support from all of the members of the Batten Research Group and others. My advisor Professor Christopher Batten was the key source of guidance and inspiration. I cannot understate how many times my ideas became much more interesting through our brainstorming sessions.

I am a core contributor to the PyMTL3 project. I collaborated closely with Dr. Shunning Jiang and Yanghui Ou to design and implement the PyMTL3 framework from scratch. Dr. Shunning Jiang led the PyMTL3 project and he designed and implemented the core DSL, the pass mechanism, the scheduling and simulation passes, the first PyMTL3 standard library, and various other aspects of the framework. Yanghui Ou is also a core contributor to the PyMTL3 project and he contributed to the method-based interfaces, the parameter mechanism in the DSL, and numerous IPs using PyMTL3. I designed and implemented the translation mechanism in PyMTL3 to enable extensible translation from the PyMTL3 DSL to numerous backends. I implemented RTLIR, an

intermediate representation of synthesizable PyMTL3 hardware components, which also serves as the canonical input of the translation passes. I implemented a significant portion of the unit test suite in PyMTL3 to ensure timely feedbacks from PyMTL3 CI/CD pipeline. I also contributed to PyMTL3's import mechanism with numerous crash bug fixes and improved integration with the Verilator simulator. Chapter 5 covers a majority of my work on the PyMTL3 framework in details. The PyMTL3 framework was published in IEEE Micro in 2020 [JPOB20].

I was a core contributor to the ultra-elastic CGRA (UE-CGRA) project led by Dr. Christopher Torng. Yanghui Ou contributed to the first implementation of the CGRA in PyMTL3. Dr. Cheng Tan contributed to the CGRA compiler and the efforts to map microbenchmarks to actual CGRA placements and routings. Dr. Christopher Torng explored the opportunity of fine-grain DVFS in CGRAs and implemented the energy and performance analysis model of UE-CGRA and the architecture and VLSI designs of UE-CGRA. I implemented the UE-CGRA RTL in PyMTL3 based on Yanghui Ou's initial efforts, led the ASIC energy and area analysis of the UE-CGRA, and the performance results of UE-CGRA based on RTL simulations. The UE-CGRA project was published and presented at the International Symposium on High-Performance Computer Architecture (HPCA) in 2021 [TPO$^+$21].

I was a core contributor to the Software-Defined Hardware (SDH) project between the collaboration of Cornell University and University of Washington (UW). I led the implementation of the ASIC energy analysis flow which was used by Cornell and UW teams to carry out accurate gate-level energy analysis in the SDH evaluation. As part of the efforts to evaluate the performance and energy efficiency of the manycore architecture in SDH by UW, I led the efforts to design and implement a high-performance parallel fast-fourier transform (FFT) algorithm on the manycore. I was also a graduate student lead of the Cornell team in the SDH tape-out. I led a group of three Cornell graduate students to implement and test a CGRA block at RTL, which was then handed off to Professor Michael Taylor's group at UW for integration and physical design. Dr. Lin Cheng contributed to the random testing of the CGRA block both at RTL and gate-level. Nick Cebry contributed to the directed testing of the CGRA block and the continuous integration of each CGRA RTL release candidate. Professor Michael Taylor's group at UW led the design, implementation, and testing of the chip which includes the Hammerblade manycore and the Blackparrot RISC-V processors [PGW$^+$20]. I wish to thank Dan Petrisko from Professor Michael Taylor's group, who generously agreed to help the Cornell team by doing the physical design for the CGRA block and testing the connections between the CGRA block and the manycore block.

I led the gradually typed hardware description language (GT-HDL) project. I explored applying optional static type checkers to verifying hardware generator properties and proposed to augment the dynamically typed PyMTL3 HDL with static type checking capabilities. I implemented the symbolic elaborator in PyMTL3, which statically proves or falsifies generator properties leveraging an SMT solver. I further leveraged the static type information and implemented elaboration-time guards and simulation-time optimizations. I evaluated the symbolic elaboration technique on a suite of 5 IP generators and 3 design generators and demonstrated its effectiveness at detecting violations of generator properties. Professor Adrian Sampson (Cornell Univeristy) and Rachit Nigam shared their valuable insights about gradual type systems, numeric types, and many more inspirational thoughts and examples in the programming language landscape. Dr. Shunning Jiang and Yanghui Ou contributed to this project by helping with the iterative divider, the processor, and the CGRA hardware models in PyMTL3 that are used in the evaluation. The symbolic elaboration aspect of the GT-HDL project was published and presented at the International Conference on Formal Methods and Models for Co-Design (MEMOCODE) in 2023 [PJOB23]. The vision of the GT-HDL project was published and presented by me at the Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE) held in conjunction with the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) in 2023 [POJB23].

I led the stall invariant project. I explored applying formal verification techniques to RTL modules with latency-insensitive (LI) interfaces. Professor Cunxi Yu (University of Maryland, College Park) and Marcelo Orenes-Vera (Princeton University) shared their valuable insights about this formal verification question at an early stage of this project. I proposed stall invariant, a property of LI interfaces which indicates the absence of LI handshake errors. I also proposed latency equivalence checking to prove or find counter-examples to the stall invariant property for a given RTL module. I evaluated the proposed latency equivalence checking technique on an iterative divider, a processing element (PE), and a RISC-V processor and demonstrated its effectiveness at identifying LI handshake bugs. Dhruv Sharma and Megha Shyam implemented the LI PE in Verilog and identified the bug used in the PE case study. This project was published and presented at the International Conference on Formal Methods and Models for Co-Design (MEMOCODE) in 2023 [PB23].

This thesis was supported in part by DARPA POSH Award #FA8650-18-2-7852, DARPA SDH Award #FA8650-18-2-7863, the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, a research gift from Xilinx, Inc., and equipment, tool, and/or physical IP donations from Intel, Synopsys, Cadence, and ARM. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

CHAPTER 2
GT-HDL:
ADDRESSING VERIFICATION CHALLENGES IN DYNAMIC HDLS

Recent research in agile hardware methodologies has argued for using dynamically typed components in dynamic HDLs to improve the productivity of verification. However, mixing dynamically typed components with statically typed hardware models leads to verification challenges for dynamic HDLs. Existing dynamic HDLs generally have to make a trade-off between safety guarantees for mixed-typed compositions and simulation performance. In this chapter, I propose GT-HDL, a new dynamic HDL with optional typing to facilitate safe and performant composition of dynamically typed components with statically typed hardware models. GT-HDL ensures safe mixed-typed compositions through guarded generator parameters and simulation-time type checks. To improve simulation performance, GT-HDL selectively removes redundant simulation-time type checks without sacrificing safety. I evaluate GT-HDL on eight hardware generators. My evaluation shows that, on average, GT-HDL has 48.1% better simulation performance than a state-of-the-art embedded dynamically typed HDL.

2.1 Introduction

The slowdown of Moore's law and the breakdown of Dennard scaling have driven computer architects towards more specialized hardware designs to meet applications' growing performance and energy efficiency demands. However, such specialized hardware designs tend to have high non-recurring engineering (NRE) costs that hinder the research and development of promising hardware systems. Recent research in hardware development methodologies addresses verification aspect of the NRE costs by dynamically typed high-level components to facilitate the creation of highly parametrized and polymorphic test harnesses, golden reference models, and cycle-approximate hardware models [LKK$^+$18, FZ03, NZP12, LZB14, JPOB20, JOP$^+$21]. Figure 2.1 illustrates a typical three-step hardware design cycle that involves parametrized hardware generators and dynamically typed components. The early step of the design cycle is to create or reuse hardware generators for the target design and its test harness. The middle step performs elaboration, which takes the top-level generator (typically a test harness generator) and a set of parameters

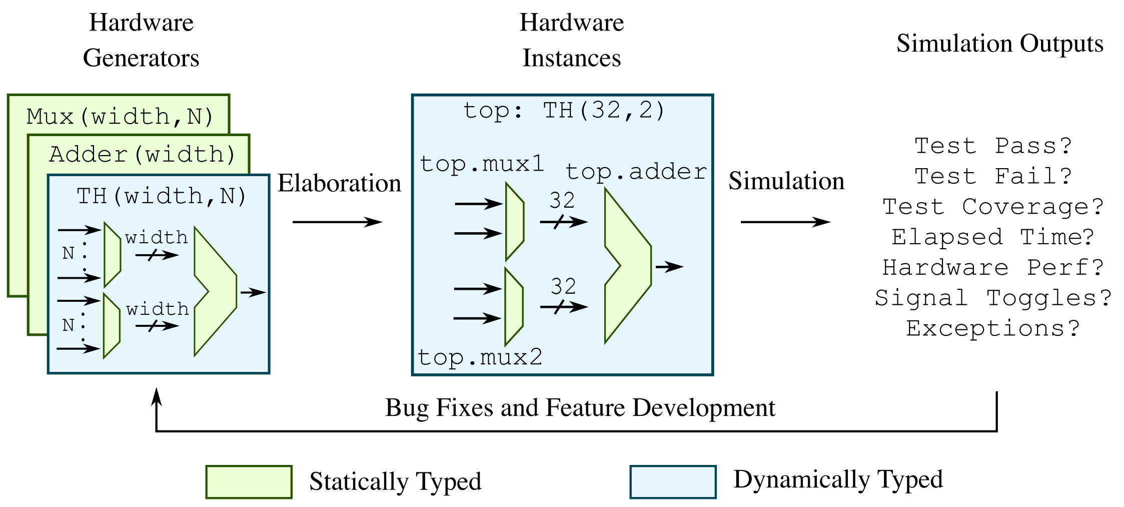


Figure 2.1: A Typical Hardware Design Cycle with a Dynamic HDL – This figure shows how hardware designers leverage hardware generators and dynamically typed components in their typical workflow.

to construct a hierarchy of hardware instances. The final step of the design cycle performs simulation on the elaborated hierarchy of instances and generates simulation outputs for debugging and feature development purposes. Dynamically typed high-level components promise a significant improvement in productivity for verification of new hardware designs.

Unfortunately, it is challenging to realize the benefits of dynamically typed components, even in state-of-the-art dynamic HDLs. More specifically, it is challenging to provide both safety guarantees for mixed-typed compositions and high simulation performance. In dynamic HDLs, statically typed components[1] are guaranteed to drive values of expected types on all output ports, as long as all input port values have the expected types. In contrast, dynamically typed components can drive potentially ill-typed values on output ports because the input values can be ill-typed, or there might be a type error in the component. Almost all existing dynamic HDLs do not provide static type checking nor elaboration- or simulation-time type checks to safeguard mixed-typed compositions. The naive approach to enable safety guarantees is to insert simulation-time type checks for every signal assignment at each simulation cycle. However, this approach incurs simulation performance overhead and fails to leverage the static type information for performance optimizations.

In this chapter, I make the case for gradually typed HDLs to address the above challenge and unlock the productivity benefits of recent hardware development methodology research. Inspired by gradually typed programming languages [ST06, SVCB15], I propose GT-HDL, an HDL that

[1] I refer to components that are instances of hardware generators that have been type checked before elaboration as statically typed components. All other components are considered to be dynamically typed.

supports the safe and performant composition of mix-typed components. GT-HDL is based on PyMTL3, a modern dynamically typed HDL embedded in Python [JPOB20]. GT-HDL ensures the safe composition of mix-typed components with guarded generator parameters and simulation type checks (Section 2.4.1). GT-HDL also leverages optional static type information to improve its simulation performance (Section 2.4.2).

This chapter makes the following contributions:

- I apply optional type checkers to embedded dynamic HDLs and demonstrate its success at type checking hardware generators (Section 2.3).
- I propose *guarded generator parameters* and *simulation-time type checks* to safeguard mixed-typed compositions (Section 2.4.1).
- I propose *simulation type check pruning* and *signal coalescing* to improve simulation performance (Section 2.4.2).
- I evaluate the simulation performance of a prototype of GT-HDL on three standalone hardware designs (Section 2.5).

This chapter is organized as follows: Section 2.2 provides the background for existing HDLs and their features. Section 2.3 discuss using an off-the-shelf option type checker to validate generator properties. Off-the-shelf optional type checkers require low HDL implementation and testing efforts and can statically type check simple hardware generators. Section 2.4.1 describes two systematic approaches to securing mixed-typed elaboration and simulation in GT-HDL: guarded generator parameters and simulation type checks. Section 2.4.2 describes simulation optimizations that leverage static type information to improve the simulation performance of GT-HDL. Section 2.5 presents the evaluation of a GT-HDL prototype using three standalone hardware designs.

2.2 Background

I begin by providing some background on hardware description languages. Section 2.2.1 introduces some of the existing HDLs and their characteristics. Section 2.2.2 introduces the standard functionalities of HDLs and demonstrate the safety and performance challenges of dynamic HDLs using a registered adder example.

HDL Type System	**Examples**	**Safety**	**Performance**
Simple Static Typing	Verilog/SystemVerilog [iee17], VHDL [iee09]	◒	●
Advanced Static Typing	Bluespec SystemVerilog [Nik04], Cλash [BKK+10]	●	●
Embedded Static Typing	Chisel [BVR+12], SpinalHDL [Spi23], Lava [BCSS98]	●	●
Embedded Dynamic Typing	PyRTL [DTS20], Migen [Mig23], MyHDL [myh14], PyMTL [LZB14], PyMTL3 [JPOB20]	◒	○
Embedded Gradual Typing	GT-HDL*	●	◒

Table 2.1: Existing HDLs and Their Characteristics – ●: almost no type safety guarantees/very low simulation performance; ◒: incomplete or late in the design cycle type safety guarantees/medium simulation performance; ○: complete and early in the design cycle type safety/high simulation performance. *: our proposal.

2.2.1 Existing HDLs

Table 2.1 summarizes different HDLs based on the following characteristics: generator support, dynamically typed high-level component support, type invariant enforcement, and when type check happens in the hardware design cycle (earlier is better). Table 2.1 shows that existing HDLs either (1) do not support both sophisticated hardware generators and dynamically typed high-level components (e.g., traditional HDLs, high-level statically typed HDLs, embedded statically typed HDLs) or (2) perform type checks late in the hardware design cycle (e.g., embedded dynamically typed HDLs). Compared to existing HDLs, GT-HDL realizes the benefits of *both* quality hardware generators and dynamically typed high-level components and performs most type checks on generators to identify design issues early in the design cycle.

Simple Statically Typed HDLs – Verilog/SystemVerilog [iee17] and VHDL [iee09] are traditional statically typed HDLs and are widely used in industry hardware design, implementation, and verification. It is challenging to programmatically construct hardware instances in those HDLs because they are not designed to be general-purpose programming languages. Their rigid static type systems also impose significant challenges on creating and using dynamically typed components. Simple statically typed HDLs enforce type invariants by thoroughly type checking the elaborated hierarchy of hardware instances, which happens in the middle of a typical hardware design cycle. Despite their awkwardness, many modern HDLs choose to emit hardware models in these HDLs (typically Verilog/SystemVerilog) to improve interoperability with commercial EDA tools.

Advanced Statically Typed HDLs – Bluespec SystemVerilog [Nik04] is an HDL with a powerful static type system. It supports sophisticated hardware generators and can type check the

```
function Bit#(2) FullAdder(
  Bit#(1) a, Bit#(1) b, Bit#(1) cin
);
  Bit#(1) sum  = a ^ b ^ cin; ❸
  Bit#(1) cout = ((a ^ b) & cin) | (a & b);
  return {cout, sum};
endfunction
```

(a) Bluespec SystemVerilog Full Adder

```
class FullAdder() extends Module {
  val io = IO(new Bundle{
    val a    = Input(UInt(1.W))
    val b    = Input(UInt(1.W))
    val cin  = Input(UInt(1.W))
    val sum  = Output(UInt(1.W))
    val cout = Output(UInt(1.W))
  })
  io.sum  := io.a ^ io.b ^ io.cin ❸
  io.cout := ((io.a ^ io.b) & io.cin) | (io.a & io.b)
}
```

(b) Chisel Full Adder

```
class FullAdder(Component):
  def construct(s):
    s.a    = InPort(Bits1)
    s.b    = InPort(Bits1)
    s.cin  = InPort(Bits1)
    s.sum  = OutPort(Bits1)
    s.cout = OutPort(Bits1)

    @update
    def upblk():
      s.sum  @= s.cin ^ s.a ^ s.b ❸
      s.cout @= ((s.a ^ s.b) & s.cin) | (s.a & s.b)
```

(c) PyMTL3 Full Adder

Figure 2.2: Full Adder Examples in Different HDLs – (a)-(c): full adder generators in Bluespec SystemVerilog/Chisel/PyMTL3. ❸ behavioral modeling; (see 2.2.2 for details)

generators to discover some potential design issues early in the design cycle. One design issue category that is not caught statically in Bluespec is bitwidth mismatches during vector slicing since the bounds for slicing operations have to be values. Bluespec SystemVerilog checks the bitwidths of slicing operations on hardware instances during elaboration. Bluespec SystemVerilog does not support dynamically typed components for improved hardware modeling and verification productivity. Cλash [BKK+10] is a Haskell dialect for hardware development. It benefits from Haskell's static type system and is able to type check the generators before elaboration. Similar to Bluespec SystemVerilog, Cλash does not support dynamically typed components.

Embedded Statically Typed HDLs – Chisel [BVR+12] is an HDL embedded in Scala, a statically typed programming language. A hardware generator in Chisel is a Scala class that programmatically constructs class instances with members that model the desired hardware (e.g., members representing ports, wires, and circuits). SpinalHDL [Spi23] is another HDL embedded in Scala and also supports sophisticated generators. Lava [BCSS98] is an HDL embedded in Haskell capable of sophisticated generators. Unlike Cλash, which is a Haskell dialect, Lava does not leverage Haskell's type system to type check hardware generators. Although it is possible to partially type check hardware generators (e.g., using features like Scala traits), embedded statically typed HDLs

```
function Bit#(TAdd#(n,1)) Adder(
  Bit#(n) a, Bit#(n) b    ❶
);
  Bit#(n) sum
  Bit#(TAdd#(n,1)) carry = 0;

  for (Integer i = 0; i<valueOf(n); i=i+1)
  begin
    Bit#(2) t = FullAdder(a[i], b[i], carry[i]); ❷
    carry[i+1] = t[1];
    sum[i] = t[0];
  end

  return {carry[valueOf(n)], sum};  ❹
endfunction
```

(a) Bluespec SystemVerilog Adder

```
class Adder(n: Int) extends Module {  ❶
  val n_inc = n + 1

  val io = IO(new Bundle{
    val a   = Input(UInt(n.W))
    val b   = Input(UInt(n.W))
    val out = Output(UInt(n_inc.W))
  })

  val fa = Array.fill(n)(Module(new FullAdder()).io)  ❷
  val carry = Wire(Vec(n_inc, UInt(1.W)))
  val sum = Wire(Vec(n, Bool()))

  carry(0) := 0.U
  for(i <- 0 until n) {
    fa(i).a    := io.a(i)
    fa(i).b    := io.b(i)
    fa(i).cin  := carry(i)
    carry(i+1) := fa(i).cout
    sum(i)     := fa(i).sum
  }

  io.out := Cat(carry(n), sum.asUInt)
}
```

(b) Chisel Adder

```
class Adder(Component):
  def construct(s, Width):  ❶
    n = get_nbits(Width)
    s.a     = InPort(Width)
    s.b     = InPort(Width)
    s.out   = OutPort(mk_bits(n+1))

    s.fa    = [FullAdder() for _ in range(n)]  ❷
    s.carry = Wire(mk_bits(n+1))
    s.sum   = Wire(Width)

    s.carry[0] //= 0
    for i in range(n):
      s.fa[i].a     //= s.a[i]
      s.fa[i].b     //= s.b[i]
      s.fa[i].cin   //= s.carry[i]
      s.carry[i+1] //= s.fa[i].cout
      s.sum[i]      //= s.fa[i].sum

    s.out //= lambda: concat(s.carry[n], s.sum)
```

(c) PyMTL3 Adder

Figure 2.3: Adder Examples in Different HDLs – (a)-(c): adder generators in Bluespec SystemVerilog/Chisel/PyMTL3. ❶ generators; ❷ structural modeling; ❹ type checks on generators; (see 2.2.2 for details)

mainly enforce HDL type invariants by type checking the elaborated hardware instances. More specifically, for Chisel, type checks on hardware instances happen through detailed analysis on FIRRTL, a post-elaboration intermediate representation of hardware [IKL+17]. Similar to advanced statically typed HDLs, embedded statically typed HDLs do not support dynamically typed components.

Embedded Dynamically Typed HDLs – PyRTL [CTD+17], Migen [Mig23], MyHDL [Dec04], PyMTL [LZB14], and PyMTL3 [JPOB20] are all HDLs embedded in Python, a dynamically typed

```
interface RegAdder;
  method Action write(Bit#(32) a, Bit#(32) b);
  method Bit#(33) read();
endinterface

module mkRegAdder( RegAdder );
  Reg#(Bit#(33)) reg_out <- mkReg(0);  ❷

  method Action write(Bit#(32) a, Bit#(32) b);
    reg_out <= Adder(a, b);
  endmethod

  method Bit#(33) read();
    return reg_out;
  endmethod
endmodule
```

(a) Bluespec SystemVerilog Registerd Adder

```
class RegAdder() extends Module {
  val io = IO(new Bundle{
    val a   = Input(UInt(32.W))
    val b   = Input(UInt(32.W))
    val out = Output(UInt(33.W))
  })

  val adder   = Module(new Adder(32)).io
  val reg_out = Reg(UInt(33.W))  ❷

  adder.a := io.a
  adder.b := io.b
  reg_out := adder.out
  io.out  := reg_out
}
```

(b) Chisel Registerd Adder

```
class RegAdder(Component):
  def construct(s):
    s.a       = InPort(Bits32)
    s.out     = OutPort(Bits33)
    s.b       = InPort(Bits32)

    s.reg_out = Wire(Bits33)
    s.adder   = Adder(Bits32);  ❷

    s.adder.a //= s.a
    s.adder.b //= s.b
    s.out     //= s.reg_out

    @update_ff
    def up_reg():
      s.reg_out <<= s.adder.out
```

(c) PyMTL3 Registered Adder

```
class PolyTestHarness:
  def __init__(s, m, test_vectors, ifunc, ofunc):
    m.apply(DefaultPassGroup())
    m.sim_reset()
    for t in test_vectors:   ❺
      ifunc(m, t)
      m.sim_eval_combinational()
      ofunc(m, t)
      m.sim_tick()
    print('Test Passed!')

th = PolyTestHarness(
  RegAdder(),    [
    (  3,     5,  '?'),
    ( 10,     8,   43),
    (  0,     0,   18),
  ],
  lambda m, t: (assign(m.a, t[0]), assign(m.b, t[1])),
  lambda m, t: assert_eq(m.out, t[2]) if t[2] != '?' else None
)
```

(d) Polymorphic Test Harness

Figure 2.4: Registered Adder Examples in Different HDLs – (a)-(c): registered adder generators in Bluespec SystemVerilog/Chisel/PyMTL3. (d): a polymorphic test harness for RegAdder in PyMTL3. ❷ structural modeling; ❺ dynamically typed high-level components. (see 2.2.2 for details)

programming language. All these HDLs support sophisticated hardware generators, and many of them support dynamically typed components thanks to Python's flexibility, which allows the mixing of almost any components. Similar to simple statically typed HDLs, embedded dynamically typed HDLs (or dynamic HDLs for short) generally do not check the generators and focus on type checking the elaborated hierarchy of hardware instances. Due to the lack of pre-elaboration type checks in existing embedded dynamically typed HDLs, components in such HDLs can always be treated as dynamically typed components. Most embedded dynamically typed HDLs do not perform elaboration- nor simulation-time type checks, and type errors generally fail in places where an ill-typed object is *used*, which causes difficulties for debugging.

2.2.2 HDL Features and Challenges

This section describes the standard features of HDLs and the safety and performance challenges of mixed-typed compositions in dynamic HDLs. I use examples in Figure 2.2, Figure 2.3, and Figure 2.4, which include three hardware generators (full adder, adder, and registered adder) and a dynamically typed high-level component (a polymorphic test harness).

Hardware Generators – All three HDLs support the creation and use of sophisticated hardware generators. In Figure 2.3, ❶ marks lines that correspond to the definition of hardware generators. In Chisel and PyMTL3, hardware generators are classes that programmatically construct instances of the desired attributes. This is a common practice in embedded HDLs, where an HDL provides different types of hardware modeling primitives (including input and output ports, wires, modules, and bundles) that are instantiated during elaboration. Generators in Bluespec SystemVerilog are represented using functions and modules. Note that a function in Bluespec generally maps to some combinational logic instead of a program.

Structural Modeling – Structural modeling describes the composition of a hardware instance. It includes interface instantiation, subcomponent instantiation, and connections between different components. In Figure 2.3 and Figure 2.4, ❷ marks lines that perform structural modeling. Chisel and PyMTL3 instantiate interfaces and subcomponents using the object instantiation syntax in their host language. Both languages also leverage overloaded operators (`:=` in Chisel and `//=` in Python) to connect different signals. Bluespec uses similar object instantiation syntax to instantiate components and interfaces. In the registered adder example (Figure 2.4(a)), connections in Bluespec are inferred from the arguments used to call functions `FullAdder` and `Adder`.

Behavioral Modeling – Behavioral modeling describes the behavior of circuits using high-level languages (e.g., arithmetic operators). In Figure 2.2, ❸ marks lines that perform behavioral modeling. In all three HDLs, designers describe circuit behaviors using arithmetic expressions, as shown in Figure 2.2(a,b,c). PyMTL3 further supports two types of *update blocks* to model the combinational and sequential behaviors of hardware: `@update` and `@update_ff`.

Tension between Safety and Performance of Mixed-Typed Adder Composition – ❺ in Figure 2.4 marks a polymorphic test harness `PolyTestHarness`. This test harness achieves polymorphism by applying the test vector input and verifying the module output using customizable input and output functions (`ifunc` and `ofunc`). `PolyTestHarness` increases hardware verification productivity because the test harness details (e.g., line 3, 4, 7, 9 of Figure 2.4(d)) are abstracted away and can be reused across any design under test (DUT). Embedded dynamically typed HDLs naturally supports dynamically typed high-level components. However, composing dynamically and statically typed components may cause subtle type errors. For example, the input function `ifunc` of the polymorphic test harness can be any dynamically typed code and may not respect the bitwidth requirements of input ports. If the input function produces ill-typed messages, these messages can propagate deep into the statically typed DUT and eventually emerge as abstruse simulation errors that do not directly point to the source of the issue.

One possible solution to overcome the potential composition issues during the simulation of mixed-typed components is to insert simulation-time type checks to signal assignments, significantly increasing simulation performance overhead [JPOB20]. Many of the inserted simulation-time type checks are unnecessary. For example, line 14-18 in Figure 2.3(c) do not require simulation-time type checks because a static type checker can prove that they are all well-typed.

2.3 Checking Generators using Optional Type Checkers

In this section, I discuss applying off-the-shelf optional type checkers to hardware generators. Even in dynamically typed programming languages like Python, programmers generally find it helpful to document their code with types and have those types checked automatically by static type checkers [THFF$^+$17, ST06]. *Optionally typed* programming languages address this issue by adding an optional type system and *type annotation* syntax to an existing dynamically typed language. A defining characteristic of optionally typed languages is that type annotations do not change the runtime semantics of the language – the compiler only leverages type annotations for static type checking purposes and erases the annotations while emitting code [MMI14, BSDTH16, BAT14]. Optional type checkers can be useful for a growing number of embedded dynamically typed HDLs. Compared to designing and implementing static analysis tools that target one specific HDL, using off-the-shelf software significantly reduces the efforts required by HDL design, implementation, and testing.

I focus on repurposing Mypy, an optional static type checker for Python [Leh23], to statically type check PyMTL3 hardware generators. I present a set of type annotations for the PyMTL3 hardware modeling DSL, which Mypy requires to analyze hardware generators correctly. With an

```
 1 class Bits: ...  ❶
 2 class Bits1(Bits): ...
 3
 4 # T_Sig is invariant
 5 T_Sig = TypeVar("T_Sig", bound=Bits)
 6 class Signal(Generic[T_Sig]):  ❷
 7   def __init__(s, Width: Type[T_Sig]) -> None:
 8     ...
 9   def __xor__(s, o: Signal[T_Sig]) -> Signal[T_Sig]:
10     ...
11   def __and__(s, o: Signal[T_Sig]) -> Signal[T_Sig]:
12     ...
13   def __or__(s, o: Signal[T_Sig]) -> Signal[T_Sig]:
14     ...
15
16 # Ignore directionality of ports
17 InPort = OutPort = Wire = Signal
18
19 T_Con = TypeVar("T_Con", bound=Bits)  ❸
20 @overload
21 def connect(l: Signal[T_Con], r: Signal[T_Con]) -> None:
22   ...
23 @overload
24 def connect(l: Signal[T_Con], r: int) -> None:
25   ...
26
27 def mk_bits(nbits: int) -> Type[Bits]:  ❹
28   ...
29 def get_nbits(Width: Type[Bits]) -> int:
30   ...
31
32 def concat(*args: List[Any]) -> Signal[Bits]:  ❺
33   ...
```

Figure 2.5: Type-Annotated Core PyMTL3 DSL – ❶: annotations of PyMTL3 data types. ❷: annotation of signals. ❸: annotations of structural `connect` method. ❹: annotations of numeric type utilities. ❺: annotation of structural `concat` method.

adder generator example, I show that Mypy can verify simple bitwidth matching cases in hardware generators and discuss its limitations.

Mypy leverages Python's type annotation syntax to type check programs statically. Type annotations for both the PyMTL3 hardware modeling DSL (Figure 2.5) and the generator itself (Figure 2.6) are necessary when applying Mypy to PyMTL3 generators.

Type Annotated PyMTL3 Hardware Modeling DSL – Figure 2.5 shows the type-annotated core PyMTL3 hardware modeling DSL. The remaining of this subsection explains the modeling primitives and their type signatures in ❶ to ❺.

❶ marks the annotated hardware data types in the PyMTL3 DSL. Hardware data types are Python classes that represent a specific bitwidth and are used to specify the bitwidth of signals. PyMTL3 dynamically generates and caches such class objects during import, which is a dynamic behavior that cannot be annotated precisely. Instead, line 1-2 in Figure 2.5 lists *all* data types used in the generators (only `Bits1` is used in this example). When the type checker cannot determine the exact bitwidth of a signal, the base class object `Bits` is used.

```
class FullAdder(Component):
  def __init__(s) -> None:
    ...
  def construct(s) -> None:
    # All ports have type Signal[Bits1]
    s.a    = InPort(Bits1)
    s.b    =  InPort(Bits1)
    s.cin  = InPort(Bits1)
    s.sum  = OutPort(Bits1)
    s.cout = OutPort(Bits1)

    @update
    def upblk() -> None:
      # Type check because all operands are Signal[Bits1]
      s.sum  @= s.cin ^ s.a ^ s.b
      s.cout @= ((s.a ^ s.b) & s.cin) | (s.a & s.b)
```

(a) Full Adder Generator Checked by Mypy

```
T_Adder = TypeVar("T_Adder", bound=Bits)
class Adder(Component, Generic[T_Adder]):
  def __init__(s, Width: Type[T_Adder]) -> None:
    ...
  def construct(s, Width: Type[T_Adder]) -> None:
    n = get_nbits(Width)

    s.a   = InPort(Width)                   # Signal[T_Adder]
    s.b   = InPort(Width)                   # Signal[T_Adder]
    s.out = OutPort(mk_bits(n+1))           # Signal[Bits]
    s.fa = [FullAdder() for _ in range(n)] # List[FullAdder]
    s.carry = Wire(mk_bits(n+1))            # Signal[Bits]
    s.sum   = Wire(Width)                   # Signal[T_Adder]

    for i in range(n):
      if i >= 0:
        # Both sides of connect are of Signal[Bits1]
        connect(s.fa[i].a   , s.a[i]      )
        connect(s.fa[i].b   , s.b[i]      )
        connect(s.fa[i].cin , s.carry[i]  )
        connect(s.carry[i+1], s.fa[i].cout)
        connect(s.sum[i]    , s.fa[i].sum )
      if i == 0:
        connect(s.carry[i], 0)

    @update
    def upblk() -> None:
      # concat arg1:               Signal[Bits1]
      # concat arg2:               Signal[T_Adder]
      # concat(s.carry[n], s.sum): Signal[Bits]
      s.out @= concat(s.carry[n], s.sum)
```

(b) Adder Generator Checked by Mypy

Figure 2.6: Type Check PyMTL3 Generators using Mypy – (a) A full adder generator type checked using Mypy. (b) An adder generator type checked using Mypy. Both the full adder and the adder generator have the same functionality as Figure 2.2(c) and Figure 2.3(c).

❷ points to the annotation of the signal type in generators. I declare `Signal` as a generic type over type variable `T_Sig`, which represents a hardware data type. Line 7 in Figure 2.5 declares that only class objects are allowed to be passed into signal constructors. Line 9-14 specify the expected type of both operands in common bitwise operations: both sides of the operation have to be a signal of the same bitwidth, and the operation returns a signal of the same bitwidth. This allows Mypy to detect simple bitwidth mismatch errors in generators. It is also worth noting that encoding the direction of signals into types is challenging. Therefore, I disregard the direction of signals while annotating the PyMTL3 DSL (line 17 in Figure 2.5).

❸ shows the type annotations of the `connect` method. The signature of `connect` is overloaded to support connecting signals to other signals and integers. The method signature ensures that if two signals are connected, they must have the same bitwidth to satisfy the matching bitwidth property. If a signal is connected to an integer, no checks are necessary because the PyMTL3 semantics ensures that the integer will be cast to fit the bitwidth of the other signal.

❹ marks the type annotations of the `mk_bits` and `get_nbits` methods. These two methods are used to convert hardware data types from and to integers. This allows hardware designers to derive new hardware data types from existing ones and enables more complicated hardware generators. Since the exact value of the integer `nbits` is not known until elaboration time, I use the `Bits` class object in the annotations because it does not contain explicit bitwidth information.

❺ points to the annotation of the `concat` method. This method takes a variable number of signal arguments and returns a concatenated signal whose bitwidth is the sum of all input signal bitwidths. Since it is not possible to know the exact bitwidth of the resulting signal before elaboration, `Signal[Bits]` is the best annotation I can do for `concat`'s return type. It might be tempting to annotate its input argument as having type `List[Signal]`, but this will only accept an array of the same bitwidth signals and reject all other potentially reasonable cases (e.g., an array of signals that have different bitwidth). This is the place where I intentionally make the type system unsound by giving the input argument a `List[Any]` signature, and this design decision allows reasonable uses of the `concat` method to be accepted as well.

Type Checking the Adder Generator – Figure 2.6(b) shows an example adder generator that uses a full adder subcomponent (Figure 2.6(a)) and can be type checked by Mypy. Mypy is able to verify the matching bitwidths for signal assignments on line 15-16 in Figure 2.6(a) because all operands involved in the bitwise and signal assignment (`@=`) operations are explicitly marked to be single-bit wide. Mypy is also able to verify the matching bitwidths for structural connections on line 18-22 in Figure 2.6(b).

2.4 Safe and Performant Mixed-Typed Compositions

In the previous section, I demonstrated how Mypy, an optional static type checker, can statically type check simple generators in dynamic HDLs. Optional static type checkers provide some static correctness guarantees about the statically typed hardware modules but does not eliminate type errors at elaboration and simulation.

In this section, I describe how GT-HDL achieves safe and performant mixed-typed compositions during elaboration and simulation. My approach to safe and performant composition has two components: (1) guarded generator parameters and simulation-time type checks, which are systematic and comprehensive mechanism to insert elaboration- and simulation-time checks to help agile designers pin-point type errors; (2) simulation type checking pruning and signal coalescing, which selectively remove redundant simulation-time type checks to improve simulation performance.

2.4.1 Safe Mixed-Typed Composition

The use of dynamically typed high-level components in dynamically typed HDLs can incur composition issues during elaboration *and* simulation. In this section, I motivate the need for safe mixed-typed composition with concrete examples in Figure 2.7 and propose two techniques: (1) a novel *guarded generator parameter* technique to secure mixed-typed elaboration (Section 2.4.1) and (2) simulation type checks to secure mixed-typed simulation (Section 2.4.1). These two techniques address the safety composition issue (discussed in Section 2.2.2). GT-HDL employs both techniques to ensure the disciplined composition of dynamically and statically typed components.

Securing Mixed-Typed Elaboration with Guarded Generator Parameters

Dynamically typed components can cause subtle composition issues during elaboration, especially when the statically typed generators have other generators as their parameters.

Problem – Figure 2.7(a) illustrates the problem of composing a statically typed registered adder with a dynamically typed test harness (polymorphic test harness in Figure 2.4(d)). A registered adder (line 23-35) instantiates a concrete adder (`s.adder`) and registers the adder's output. The adder generator on line 1-7 defines the common interface of adders. Two concrete types of adders inherit from `Adder` and are defined on line 9-21: a ripple carry adder and a behavioral adder. The

```
 1 T_Adder = TypeVar("T_Adder", bound=Bits)
 2 class Adder(Component):
 3   def construct(s, Width: Type[T_Adder]):
 4     n = get_nbits(Width)
 5     s.a = InPort(Width)
 6     s.b = InPort(Width)
 7     s.out = OutPort(mk_bits(n+1))
 8
 9 T_RippleCarry = TypeVar("T_RippleCarry", bound=Bits)
10 class RippleCarryAdder(Adder):
11   def construct(s, Width: Type[T_RippleCarryAdder]):
12     super().construct(Width)
13     # model ripple carry adder ...
14
15 T_Behavioral = TypeVar("T_Behavioral", bound=Bits)
16 class BehavioralAdder(Adder):
17   def construct(s, Width: Type[T_BehavioralAdder]):
18     super().construct(Width)
19     @update
20     def up_adder():
21       s.out @= s.a + s.b  ❷
22
23 class RegAdder(Component):
24   def construct(s, AdderType: Type[Adder]):
25     s.a = InPort(Bits32)
26     s.b = InPort(Bits32)
27     s.out = OutPort(Bits33)
28     s.adder = AdderType(Bits32)  ❶
29
30     connect(s.a, s.adder.a)
31     connect(s.b, s.adder.b)
32
33     @update_ff
34     def up_reg():
35       s.out <<= s.adder.out
36
37 ripple_carry_th = PolyTestHarness(
38   RegAdder(FooBar), # Elaboration Error!  ❶
39   [ (  3,     5,  '?') ],
40   lambda m, t: (assign(m.a, t[0]), assign(m.b, t[1])),
41   lambda m, t: assert_eq(m.out, t[2]) if t[2] != '?' else None
42 )
```

(a) Elaboration-Time Composition Issue

```
behavioral_th = PolyTestHarness(
  RegAdder(BehavioralAdder), [
    (  '3',   '5',  '?'), # Simulation Error!   ❷
  ],
  lambda m, t: (assign(m.a, t[0]), assign(m.b, t[1])),
  lambda m, t: assert_eq(m.out, t[2]) if t[2] != '?' else None
)
```

(b) Simulation-Time Composition Issue

Figure 2.7: Composition Issue during Elaboration and Simulation using a Registered Adder Example – ❶ Elaboration error: expecting an `Adder` but gets `FooBar`. ❷ Simulation error: expecting integers but gets strings.

type annotation of the `AdderType` generator parameter on line 24 implies that the connections and signal assignments on line 30, 31, and 35 can all be statically verified.

However, dynamically typed high-level components may not respect this type annotation and may pass in ill-typed objects as generator parameters. ❶ in Figure 2.7(a) shows a dynamically typed test harness that passes a non-adder `FooBar` object where an adder generator is expected.

In the best case, the elaboration process will halt at line 28 because `FooBar` is not a component generator or does not take one data type parameter. The elaboration process may also halt at line 30, 31, or 35 because the `FooBar` generator does not have ports `a`, `b`, `out` of the desired bitwidth. In the worst case, the `FooBar` generator provides the same interface as `Adder` but implements a different behavior. These problems are hard to debug without proper handling of mixed-typed components during elaboration because the symptoms do not point to the root cause. For example, the problem in the worst case can only be detected during simulation by comparing the simulation behavior of `FooBar`, which is buried deep in the component hierarchy, against an adder.

Solution – I propose *guarded generator parameters*, an elaboration-time technique to help secure the mixed-typed composition in GT-HDL. I observe that the subtle composition issues discussed above can be detected effectively if the generator parameters are checked against their type annotations during elaboration. The implementation of guarded generator parameters leverages Python's run-time inspection feature to extract and enforce the type annotations of parameters. With guarded generator parameters, the problem ❶ in Figure 2.7(a) can be reported during elaboration because `FooBar` is not a subclass of `Adder`.

Securing Mixed-Typed Simulation with Simulation Type Checks

Dynamically typed components can also cause issues during simulation. A dynamically typed component may drive arbitrary data into the input ports of a statically typed component.

Problem – ❷ in Figure 2.7(b) shows a dynamically typed test harness (the polymorphic test harness in Figure 2.4(d)) causing simulation issues. Instead of driving integer test vectors into the behavioral adder, the dynamically typed test harness drives *strings* (line 3 in Figure 2.7(b)) into the design-under-test (DUT). This typically leads to simulation-time value errors because most arithmetic operations between signals do not apply to non-integer types. However, ❷ represents a subtle simulation problem because the overloaded + operator also concatenates strings (line 16 on the right). This problem leads to confusing simulation errors and requires extensive inspection of internal component states to debug.

Solution – I reuse the simulation type check technique from PyMTL3 [JPOB20] to address the above composition issues. After elaboration, the GT-HDL simulator performs two steps to convert the elaborated DUT hierarchy into a simulatable function: (1) *block generation*, which dynamically compiles the structural and behavioral models of the DUT hierarchy into Python functions called *blocks*; (2) *scheduling*, which determines the execution order of all blocks in one simulated cycle to model the desired hardware behavior. *Signal assignments* are performed extensively in blocks to propagate the values of signals from the top-level input ports to output ports. To implement simulation type checks, I instrument the signal assignment operator to check if the right-hand side (RHS) of the assignment has the same type (including bitwidth) as the left-hand side (LHS) before performing the actual assignment. In the case of ❷ in Figure 2.7(b), simulation type checks allow early detections of the type error when the `assign` function (line 5 in Figure 2.7) attempts to drive strings into the input ports `a` and `b` of `RegAdder`.

2.4.2 Performant Mixed-Typed Composition

Section 2.4.1 shows that simulation type checks are necessary to ensure a safe composition of mixed-typed components in GT-HDL. However, the inserted simulation type checks also add performance overhead. In this section, I propose two type-based simulation optimizations to improve the simulation performance of GT-HDL: *simulation type check pruning* and *signal coalescing*. Simulation type check pruning removes unnecessary type checks during simulation. Signal coalescing analyzes the connectivities between statically typed signals and coalesces a group of connected signals into one signal proxy. These two techniques address the performance issue (described in Section 2.2.2) by eliminating redundant computations based on type information. GT-HDL adopts both techniques to improve simulation performance.

Simulation Type Check Pruning

As discussed in Section 2.4.1, GT-HDL inserts simulation type checks as part of signal assignments to prevent ill-typed values from propagating and to reduce debugging difficulties. However, the inserted type checks also incur simulation performance overhead. Simulation type check pruning is a systematic approach to reducing the number of simulation type checks without undermining the safe mixed-typed composition by leveraging the static type information.

Motivation – I use a mixed-typed composition in 2.8 to motivate the simulation type check pruning technique. The composition in 2.8 includes two dynamically typed components (one test source and one test sink) in blue and one statically typed DUT (an iterative divider, with all subcomponents) in green. Section 2.4.1 proposes to insert simulation type checks to *all* signal assign-

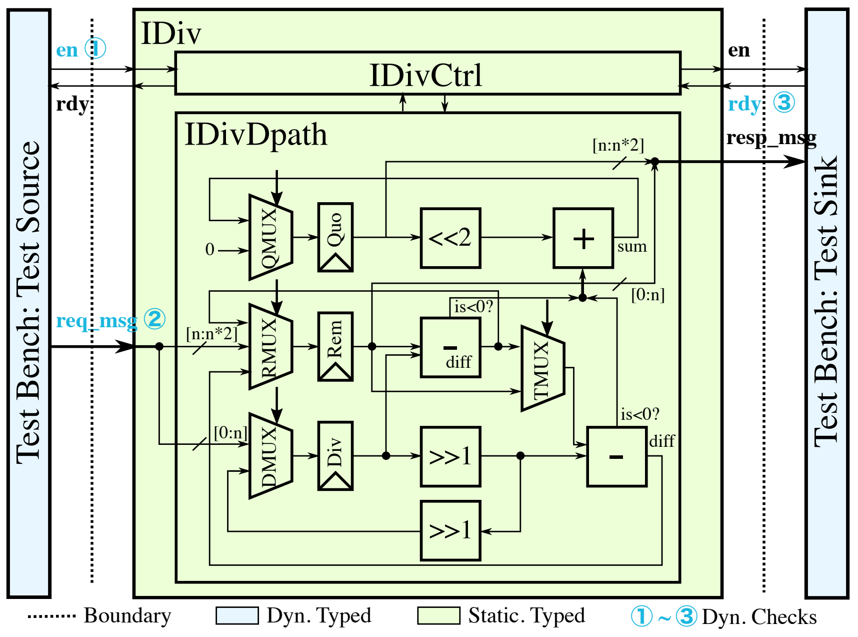


Figure 2.8: A Composition of Dynamically Typed Test Bench and Statically Typed Divider in GT-HDL – Signals ① through ③ form the minimal set of signals that require simulation-time signal assignment type checks.

ments, which include (1) assignments inside the dynamic domain (blue), (2) assignments inside the static domain (green), and (3) assignments between these two domains (the boundary between blue and green). I make the observation that *adding simulation type checks to signal assignments inside the dynamic and static domains and from the static domain to the dynamic domain are not necessary*. On the one hand, no type errors will be generated in the static domain as long as all components in the static domain have been type checked and the input to the static domain is well-typed. On the other hand, propagating ill-typed values in the dynamic domain (i.e., "garbage-in, garbage-out") is acceptable because the domain only includes intentionally made untyped components. Therefore, simulation type checks are only necessary at the static and dynamic domain boundary when values propagate from the dynamic domain to the static one.

Implementation – GT-HDL's implementation of simulation type check pruning is based on the type check scheme described in Section 2.4.1. I implement the pruning scheme by removing type checks in the signal assignment operator and selectively inserting type checks into the dynamically compiled blocks. During assembling of signal assignment code in the block generation step, we check if this assignment propagates data from the dynamic domain to the static domain. I only insert a simulation type check before the corresponding assignment if that is true.

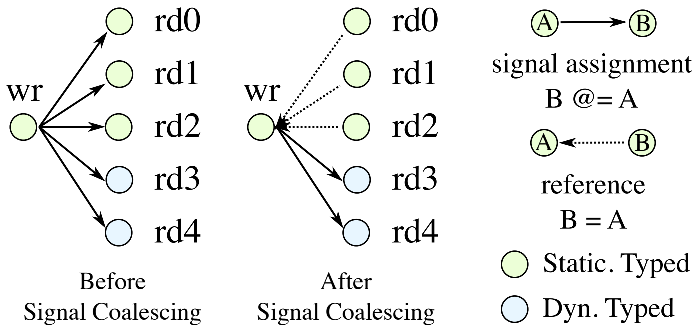


Figure 2.9: Signal Coalescing on a Net of Five Readers

Signal Coalescing

In this section, I describe *signal coalescing*, an optimization technique that leverages the static type information to reduce the number of signal assignments.

Motivation – A critical aspect of simulation in HDLs is to update the values of interconnected signals correctly. GT-HDL uses the *net* data structure to represent a group of connected signals, where precisely one *writer* signal continuously drives values to zero or more *readers*. In PyMTL3, this continuous update is implemented as signal assignments from the writer to all readers, which happen every simulated clock cycle. This assignment-based implementation prevents a dynamically typed writer[2] from driving ill-typed signals to every reader in the net. However, this implementation also introduces performance overhead due to unnecessary signal assignments.

Signal coalescing leverages the fact that as long as a pair of writer and reader is statically typed, it is safe to make the reader a *reference* of the writer instead of performing a signal assignment. Whenever a statically typed writer is updated, all statically typed readers in the same net will see the same value via reference. This significantly reduces the number of signal assignments performed during simulation. I use a net of five readers in Figure 2.9 to illustrate the idea of signal coalescing. Before signal coalescing, five signal assignments are needed to propagate the writer's value to all five readers, regardless of whether the writer or the readers are statically typed or not. Signal coalescing reduces the required signal assignments from five to two.

Implementation – I implement signal coalescing by modifying how signal assignments are generated in the block generation step of the simulation step in GT-HDL (related discussions in Section 2.4.1). I enumerate reader signals in each net and generate code to set up a reference between the reader and the writer if both signals are from statically typed components. Signal

[2]More precisely, a writer from a dynamically typed component

coalescing requires relatively low implementation efforts because even without coalescing, the simulation infrastructure needs to keep track of the signal values through a map from attribute names to the actual value objects. The signal coalescing implementation can be easily built on the existing attribute-value map.

2.5 Evaluation

GT-HDL draws together guarded generator parameters (Section 2.4.1), simulation type checks (Section 2.4.1), simulation type check pruning (Section 2.4.2), and signal coalescing (Section 2.4.2) to achieve safe mixed-typed compositions and high simulation performance. In this section, I describe my methodology of evaluating the GT-HDL prototype and the evaluation results. In order to quantitatively evaluate GT-HDL's static type checking capabilities, I implement a mutation-based abstract syntax tree (AST) fuzzer that can randomly inject six common categories of bugs that I learned from actual hardware generator development. I compare the simulation performance of GT-HDL against PyMTL3 using a composition of dynamically typed test harnesses and statically typed designs.

This section is organized as follows: Section 2.5.1 introduces the eight RTL design generators used in my evaluation and why they were chosen. Section 2.5.2 discusses the AST fuzzer and the categories of bugs it is able to inject. Section 2.5.3 evaluates the bug detection accuracy of GT-HDL (using Mypy) against PyMTL3 and provides a detailed breakdown of when the bugs are identified. Section 2.5.4 compares the simulation performance of GT-HDL against PyMTL3.

2.5.1 Evaluation Designs

My evaluation of GT-HDL uses eight hardware design generators shown in Table 2.2. LFSR, Gray encoder/decoder, priority encoder, round-robin arbiter, and FIFO queue are commonly used hardware IPs that can be found in most hardware IP libraries. Divider, processor, and CGRA represent three standalone hardware designs: an integer divider, a RISC-V processor, and an elastic coarse-grain reconfigurable array [HIT+13]. The divider generator generates a radix-4 iterative divider for a given data path width. The processor generator generates one or more RV32-IM five-stage cores. The CGRA generator generates a reconfigurable processing element array with elastic

	Generator LoC (Python)	Instance LoC (Verilog)	Instance Configuration
LFSR	31	23	32-bit register
Gray Encoder/Decoder	42	76	32-bit input
Priority Encoder	45	79	32-bit input
Round-Robin Arbiter	49	88	8 requesters
FIFO Queue	125	174	32-bit 2-element queue
Divider	172	535	32-bit datapath
Processor	903	2135	single-core RV32-IM
CGRA	1170	4004	8×8 32-bit PE array

Table 2.2: Evaluated Hardware Generators – LoC (lines of code) reported by `cloc`; lines exclude comments and blanks. The upper segment includes standard hardware IPs and the lower segment includes standalone hardware designs.

flow control for the given dimension. I choose these eight designs because they form a representative suite of hardware generators that include both commonly used IPs and standalone hardware designs. Standard hardware IPs are generally harder to statically type check because such generators are heavily parameterized and sometimes customized for specific parameter combinations.

2.5.2 Mutation-Based Abstract Syntax Tree Fuzzer

To evaluate the effectiveness of GT-HDL's static type checking capability, I implement a mutation-based AST fuzzer to inject design bugs to designs in Section 2.5.1. The AST fuzzer is capable of injecting six kinds of bugs that I categorized from 249 git commits in the development and testing of the three standalone design generators. The AST fuzzer parses the syntax tree of the target hardware generator and searches for syntactic constructs that are suitable for bug injection. The AST fuzzer recognizes the following commonly used syntactic constructs: constant integers, explicitly sized constants, signal slicing operations, accessing attributes of a component or interface, referencing an object by identifier, and all arithmetic and boolean operations. The AST fuzzer injects a bug by mutating part of the target syntactic construct to produce a syntactically correct but semantically flawed hardware generator.

Bitwidth mutation – Based on my experiences reviewing hardware development git commits, bitwidth mismatching is a common category of design bugs in generators. To perform a bitwidth mutation, the AST fuzzer searches for signal bitwidth declaration statements, slicing operations, and constants and mutates the bitwidth to force a bitwidth mismatch. This category of bugs stresses GT-HDL's ability to verify the *bitwidth matching* generator property.

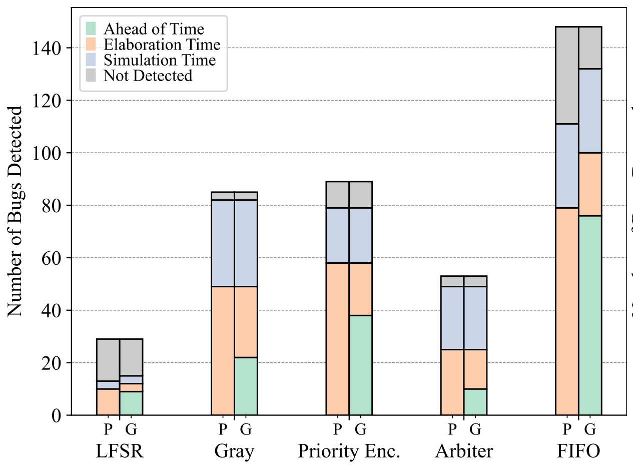
Component attribute mutation – Embedded dynamically typed HDLs rely heavily on accessing component attributes to construct hardware programmatically. It is often easy for designers to mix the name of one attribute with the other, and *component attribute mutation* aims to inject this kind of bug. More specifically, the AST fuzzer looks for an attribute name in attribute access (get or set) and replaces it with the name of another attribute from the same object. This mutation stresses GT-HDL's ability to verify the *valid hierarchical reference* property because the mutated attribute may not be a valid hierarchical reference.

Port direction mutation – Incorrect port direction is another common category of design issues. The AST fuzzer introduces port direction bugs by flipping the direction of one port in the given design. This mutation stresses GT-HDL's ability to verify the *local port direction* property.

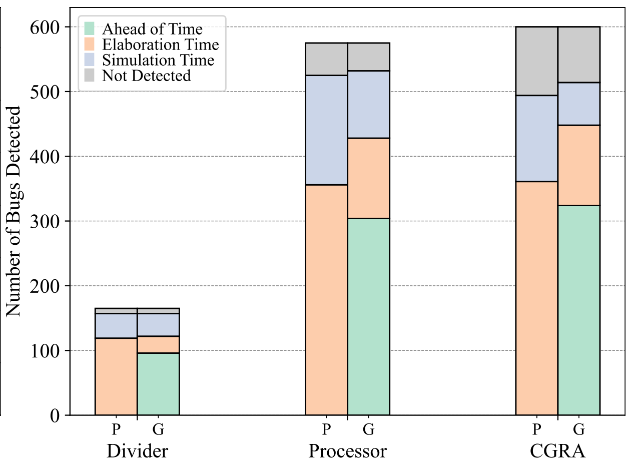
Name expression mutation – This mutation focuses on changing the variable identifiers to model various typos that are common in the git commits I reviewed. However, simply mutating a variable name will almost certainly generate references to nonexistent variables, which is trivial. To avoid referencing nonexistent variables (which is a trivial bug), the AST fuzzer keeps track of all available variable names in the current scope and only replaces the target name with a name in the current scope. This mutation stresses the general robustness of GT-HDL's symbolic elaborator.

Attribute base mutation – This is a surprisingly simple category of bugs that I found in actual hardware development commits where hardware designers remove the base object from attribute access expressions (e.g., `s.out` becomes `out`). Most of these bugs lead to accessing nonexistent variables. However, when this bug appears on the left-hand side of an assignment expression, the mutated syntax will create a temporary variable, an incorrect behavior that can only be detected at simulation time. This mutation stresses the general robustness of GT-HDL's symbolic elaborator.

Functionality mutation – The AST fuzzer implements this mutation by flipping constant values and arithmetic operators in the code of hardware generators. Unlike other mutations, most functional bugs cannot be detected through type checking (e.g., `a-b` type checks if `a+b` already type checks). Therefore, a test suite of high code coverage is required to achieve a high detection rate in this category. Some cases of functionality mutation stress GT-HDL's ability to verify the *bounded array indexing* property because the constants in the indexing expression can be mutated.



(a) Bug Detection Results on Common IPs



(b) Bug Detection Results on Standalone Designs

Figure 2.10: Bug Detection Results – (a) and (b) show the number of bugs detected by PyMTL3 (P) and GT-HDL using Mypy (G) at different stages. The more bugs detected ahead of time the better.

2.5.3 Bug Detection

In this section, I evaluate the bug detection effectiveness of PyMTL3 and GT-HDL (using Mypy as an optional type checker) on eight bug-injected hardware modules. I randomly inject bugs into a generator using the AST fuzzer (one bug at a time) and then evaluate if PyMTL3 and GT-HDL are able to detect the injected bug. This evaluation compares the effectiveness of Mypy against state-of-the-art elaboration-time checks in PyMTL3.

Figure 2.10 shows the number of bugs detected by PyMTL3 and GT-HDL at each stage: ahead of time (statically), during elaboration, during simulation, or not detected. A syntax mutation may not be detected because the mutation does not lead to a type error, does not change the generator's functionality, or the simulation test vectors do not reach 100% coverage. The figure indicates that in general, GT-HDL with Mypy can statically detect a significant number of design bugs that would have been caught with elaboration time checks in PyMTL3. In the cases of a standalone design (divider, processor, and CGRA), GT-HDL with Mypy can even detect some design bugs that are only captured with simulation time checks in PyMTL3. These results show that GT-HDL with Mypy can detect more design bugs than PyMTL3 alone and Mypy discovers these bugs ahead of time.

	Divider		**Processor**		**CGRA**	
Sim. Cycles	360K		130K		10K	
Performance	cycles/s	speedup	cycles/s	speedup	cycles/s	speedup
PyMTL3	5.91K	-	1.04K	-	9.42	-
GT-HDL	10.01K	70.57%	1.42K	37.35%	12.85	36.35%

Table 2.3: Simulation Performance

2.5.4 Simulation Performance

I evaluate the simulation performance implications of type-based simulation optimizations by comparing the simulation performance of GT-HDL against PyMTL3. I run long simulations on a mixed-typed composition (including a dynamically typed test harness and a statically typed design instance) and record simulation performance (simulated cycles per second). For the divider instance, I send 20K integer division requests to the 32-bit divider. For the processor instance, I use a program with a 10K-iteration loop that performs memory loads, arithmetics, and branch instructions. For the 32-bit 8-row by 8-column CGRA instance, I simulate a dot product kernel whose input is two 10K-element vectors. I use the CPython interpreter for the simulation performance experiments and take the average execution time of five simulation runs. Table 2.3 shows the simulation performance and speedups.

We can see from the table that, on average, the GT-HDL simulation performance is 48.1% higher than the simulation performance of PyMTL3. This considerable performance improvement of GT-HDL supports my claim (Section 2.4.2) that type-based simulation optimizations can significantly improve simulation performance (simulated cycles per second) by eliminating unnecessary simulation type checks *and* signal assignments inside the statically typed components.

2.6 Related Work

Scaling dynamically typed scripting languages to large applications is challenging partly due to the lack of type annotation syntax or a static type system. Optionally typed programming languages generally add an optional static type system to a scripting language to support static type warnings while still preserving the common programming idioms in the scripting languages. Existing optionally typed programming languages include Typed Lua [MMI14], Typed Clojure [BS-

DTH16], TypeScript [BAT14], and Dart [FNT15]. The optional static type system allows for a relatively smooth transition from an existing scripting language codebase. However, most optionally typed programming languages implement type erasure in their compilers, which means they do not leverage static type information for performance optimization.

Siek and Taha proposed gradual typing [ST06, ST07], a promising solution to the scalability issue of dynamically typed languages. Researchers have since designed numerous gradually typed languages. Typed Racket [THF08, TFG$^+$16] implements part of the support for sound gradual typing using contracts [FF02]. When a higher-order value crosses the boundary between statically and dynamically typed code, a contract is created to monitor the value's runtime behavior. If the value's behavior is incompatible with its type, the contract signals this violation to the programmer. Reticulated Python [VKSB14] is a framework of several gradually typed Python dialects. Vitousek et al. studied the performance implications of different dynamic semantics using Reticulated Python. Similar to the transient dynamic semantics in Reticulated Python, guarded generator arguments in GT-HDL enforce the type annotations of generator arguments by systematically inserting elaboration-time type checks. It is worth noting that combining Reticulated Python and PyMTL3 should produce a PyMTL3 implementation with the same dynamic semantics as the PyMTL3 reference implementation. However, Reticulated Python+PyMTL3 will almost certainly not be as capable and performant as GT-HDL.

More recently, researchers have come to define a more rigorous criterion that examines the program's behaviors when type annotations are added or removed [SVCB15]. This criterion, known as the "gradual guarantee", helps determine if a new language qualifies as a gradually typed one. Since GT-HDL assumes that type annotations have to be added at a per-generator granularity, adding or removing type annotations in GT-HDL simply changes the number of statically and dynamically typed components in the component hierarchy. Removing type annotations from a well-typed design does not affect any other typed components or change the simulation outputs. Adding type annotations to a well-typed design does not affect the simulation outputs either unless the typed generator is rejected by GT-HDL due to an incorrect type annotation. Therefore, we hypothesize that GT-HDL does provide the gradual guarantee.

2.7 Conclusion

In this chapter, I address the verification challenge in dynamic HDLs with GT-HDL, a new HDL with optional type checking to ensure safe and efficient mixed-typed composition. GT-HDL statically checks generators using an off-the-shelf optional type checker, which can type check simple hardware modules in dynamic HDLs. To enable safe and performant mixed-typed compositions, GT-HDL first insert elaboration- and simulation-time type checks and then selectively removes redundant simulation-time checks based on static type information to improve simulation performance. My evaluation of a GT-HDL prototype demonstrates up to 70.1% improvement of simulation performance across three standalone hardware designs.

CHAPTER 3
SYMBOLIC ELABORATION: ADDRESSING VERIFICATION CHALLENGES IN GENERATOR DEVELOPMENT

Recent research in agile hardware methodologies has argued for using sophisticated and highly parametrized hardware generators to significantly improve the productivity of hardware design and implementation. However, challenges for verifying critical generator properties across all possible parameter combinations have made it challenging for hardware description languages (HDLs) to realize the productivity benefits of these advances. In this chapter, I propose *symbolic elaboration*, an SMT-based static analysis technique that validates generator properties for all parameter combinations. Symbolic elaboration validates common generator properties (e.g., matching bitwidths, bounded array indexing) to improve the verification productivity during generator development. I evaluate symbolic elaboration on a suite of hardware generators. My evaluation shows that, on average, the symbolic elaborator is able to statically detect 90.6% of randomly injected bugs.

3.1 Introduction

Dynamic HDLs play a pivotal role in addressing the verification challenges associated with generator development within the context of agile hardware methodologies. As mentioned in the previous chapter, the end of Moore's Law and Dennard scaling is driving computer architects to specialized hardware systems to meet the performance and efficiency demands of emerging applications. Dynamic HDLs promote the creation of highly-parameterized generators and enable rapid prototyping [SWD+12, AAB+16, IKL+17, BVR+12, TH19, JPOB20, NM15], addressing both the increasing complexity of specialized hardware designs and the need to mitigate substantial non-recurring engineering (NRE) costs. Additionally, dynamic HDLs facilitate generator verification by enabling the integration of generated hardware instances with flexible behavioral models [LKK+18, FZ03, NZP12, LZB14, JPOB20, JOP+21], further streamlining the development cycle.

Unfortunately, dynamic HDLs generally do not provide any static correctness guarantees for hardware generators, which creates a prolonged design cycle where generator properties have to be validated during elaboration or simulation. Figure 3.1 shows an adder generator example and

```python
class FullAdder(Component):
  def construct(s):
    s.a    = InPort(Bits1)
    s.b    = InPort(Bits1)
    s.cin  = InPort(Bits1)
    s.sum  = OutPort(Bits1)
    s.cout = OutPort(Bits1)

    @update
    def upblk():
      s.sum  @= s.cin ^ s.a ^ s.b
      s.cout @= ((s.a ^ s.b) & s.cin) | (s.a & s.b)

class Adder(Component):
  def construct(s, Width):
    n = get_nbits(Width)
    s.a   = InPort(Width);
    s.b = InPort(Width)
    s.out = OutPort(mk_bits(n+1))

    s.fa  = [FullAdder() for _ in range(n)]
    s.carry = Wire(mk_bits(n+1));
    s.sum = Wire(Width)

    s.carry[0] //= 0
    for i in range(n):
      s.fa[i].a    //= s.a[i]
      s.fa[i].b    //= s.b[i]
      s.fa[i].cin  //= s.carry[i]
      s.carry[i+1] //= s.fa[i].cout
      s.sum[i]     //= s.fa[i].sum

    s.out //= lambda: concat(s.carry[n], s.sum)
```

(a) PyMTL3 Adder

```python
 1 class PolyTestHarness:
 2   def __init__(s, m, test_vectors, ifunc, ofunc):
 3     m.apply(DefaultPassGroup())
 4     m.sim_reset()
 5     for t in test_vectors:
 6       ifunc(m, t)
 7       m.sim_eval_combinational()
 8       ofunc(m, t)
 9       m.sim_tick()
10     print("Test Passed!")
11
12 th = PolyTestHarness(
13   Adder(Bits32),    [
14     (  3,     5,    8),
15     (  1,    42,   32),
16     ( 10,     8,   18),
17     (  0,     0,    0),
18   ],
19   lambda m,t: (Assign(m.a,t[0]), Assign(m.b,t[1])),
20   lambda m,t: Eq(m.out,t[2])
21 )
```

(b) Polymorphic Test Harness

Figure 3.1: Adder and Polymorphic Test Harness in PyMTL3 – (a) `Adder` is parametrized by its data path width (`Width`); it uses the `FullAdder` module as its sub-components; ports and wires are constructed with their bitwidth in parentheses; `//=` is the connection operator that connects two ports or wires together. (b) Line 12-21 show how to specialize the test harness for the adder, which includes test vectors and input setting/output checking lambda functions; the harness is dynamically typed to allow passing in customized input and output functions of any valid Python code.

its test harness in PyMTL3, a state-of-the-art dynamic HDL. Similar to other dynamic HDLs, PyMTL3 performs no static checks on the `FullAdder` nor the `Adder` generator because the concrete generator parameters are not available before elaboration. Given a set of concrete generator parameters (e.g., 32-bit data path width for `Adder`), PyMTL3 elaborates the generator into a hierarchy of hardware *instances* and checks for structural connection errors during elaboration among those instances (e.g., dynamic HDLs can verify the signals connected to the inputs of the adder have the same bitwidth 32, as indicated in line 27 31 in Figure 3.1(a)). And further given a set of concrete test vectors, PyMTL3 simulates the target design with the test harness and check for behavioral errors using a polymorphic test harness (Figure 3.1(b)). The lack of capable static checking abilities creates a long design-debug cycle where design issues can only be identified and fixed after elaboration or simulation, which hinders design productivity.

In this chapter, I propose *symbolic elaboration* (SE) to provide static correctness guarantees for hardware generators in dynamic HDLs and shorten the design-debug cycle. I first make the observation that optional type checkers, which can type check some hardware generators, are too specific to software programs and cannot effectively verify critical hardware generator properties. To overcome this limitation, I observe that most hardware generators can be expressed as an abstract model whose structural and behavioral models rely on a *symbolic* generator parameter rather than concrete values. I propose symbolic elaboration, a novel technique that translates these abstract structural and behavioral models into integer constraints that can be solved by SMT solvers. The proof of the correctness of hardware generators is obtained if the solver finds such constraints unsatisfiable, and a counterexample of the violation of generator properties is generated if the solver finds a satisfiable assignment of variables. The prototype implementation of symbolic elaboration can verify critical generator properties including matching bitwidths, correct local port directions, bounded array indexing, and valid hierarchical references. I evaluate the symbolic elaboration prototype on a suite of IP and design generators in PyMTL3 and demonstrate significant improvement in bug detection rate over an optional type checker. It is worth noting that even though I evaluate symbolic elaboration in a dynamic HDL, the concept of symbolic elaboration can be applied to virtually all HDLs that support generators.

This chapter makes the following contributions:

- I describe the limitations of using off-the-shelf optional type checkers to type check hardware generators (Section 3.3).

HDLs	Prop. Checked			Target of Prop. Enforcement		
	AoT	ET	ST	AoT	ET	ST
Traditional Static HDLs Verilog/SystemVerilog [iee17], VHDL [iee09]	○	●	○		Single Instance	
High-Level Static HDLs Bluespec SystemVerilog [Nik04], Cλash [BKK$^+$10]	●	◒	○	All Instances	Single Instance	
Constructional Static HDLs Chisel [BVR$^+$12], SpinalHDL [Spi23], Lava [BCSS98]	◒	●	○	All Instances	Single Instance	
Dynamic HDLs PyRTL [DTS20], Migen [Mig23], MyHDL [myh14], PyMTL3 [JPOB20]	○	●	●		Single Instance	Single Input
Dynamic HDLs with Symbolic Elaboration*	●	◒	◒	All Instances	Single Instance	Single Input

Table 3.1: Existing HDLs and Their Characteristics – Prop.: properties. Hardware properties can be enforced at three times: ahead of time (AoT) by checking hardware generators (static); elaboration time (ET) by checking hardware instances; simulation time (ST) by checking signal assignments in simulation. We focus on hardware properties discussed in Section 3.2.2. All Instances: the given hardware property is guaranteed to hold on all instances of the target generator under all input; Single Instance: the given hardware property is guaranteed to hold on the target instance under all input; Single Input: the given hardware property is guaranteed to hold on the target instance under the given input. ●/◒/○ : almost all/some/no properties enforced; Dyn. typed: dynamically typed. *: our proposal statically provides strong generator correctness guarantees (All Instances) for dynamic HDLs.

- I propose symbolic elaboration to overcome the limitations of off-the-shelf static type checkers and statically verify critical hardware generator properties (Section 3.4).
- I evaluate a prototype SE implementation on eight RTL design generators in a state-of-the-art dynamic HDL using a mutation-based abstract syntax tree (AST) fuzzer (Section 3.5).

3.2 Background

I begin by providing some background on hardware description languages and the target hardware generator properties I focus on in this chapter. Section 3.2.1 introduces existing HDLs. Section 3.2.2 introduces the hardware properties I try to verify in this chapter.

3.2.1 Existing HDLs

Table 3.1 summarizes existing HDLs based on the following characteristics: generator support, dynamically typed behavioral model support, when hardware properties are enforced in the hardware design cycle, and the target on which the given properties are enforced. Table 3.1 shows that (1) traditional HDLs like SystemVerilog and VHDL fail to support programmatic generators and dynamically typed behavioral modules; (2) high-level HDLs and constructional HDLs are able to enforce hardware properties on all instances of some generators but generally do not support dynamically typed behavioral models; (3) dynamic HDLs support both programmatic generators and dynamically typed behavioral modules but do not enforce hardware properties ahead of time.

Traditional Static HDLs – Verilog/SystemVerilog [iee17] and VHDL [iee09] belong to traditional statically typed HDLs. Verilog, SystemVerilog, and VHDL all support limited forms of hardware generation through generate statements (if- and for-statements). However, hardware generators in these HDLs cannot leverage more advanced programmatic constructs such as recursive functions, object-oriented abstractions, and advanced data structures beyond lists. This makes it challenging to programmatically construct hardware instances with these HDLs. Their static type systems also impose challenges on creating dynamically typed components. Traditional HDLs enforce hardware properties by type checking the elaborated hierarchy of hardware instances, which happens in the middle of a hardware design cycle.

High-Level Static HDLs – Bluespec SystemVerilog [Nik04] has a powerful static type system. It supports programmatic hardware generators and can type check the generators to discover potential design issues early in the design cycle. However, Bluespec cannot verify critical properties such as bitwidth mismatches in vector slicing operations and defers this check to elaboration. Cλash [BKK+10] is a Haskell dialect for hardware development. It benefits from Haskell's static type system and is able to type check the generators before elaboration. Both Bluespec SystemVerilog and Cλash adopt a different level of abstraction from the conventional register-transfer level abstraction and do not support dynamically typed components.

Constructional Static HDLs – Chisel [BVR+12] and SpinalHDL [Spi23] are HDLs embedded in Scala. Both Chisel and SpinalHDL support programmatic hardware generators. Lava [BCSS98] is an HDL embedded in Haskell capable of programmatic generators. Unlike Cλash, Lava does not leverage Haskell's type system to type check hardware generators. Constructional static HDLs mainly enforce HDL hardware invariants by type checking the elaborated hardware instances.

More specifically, for Chisel, type checks on hardware instances happen through a detailed analysis on FIRRTL, a post-elaboration intermediate representation of hardware [IKL$^+$17]. Constructional static HDLs focus on hardware construction using various modeling primitives embedded in the statically typed host language (such as Scala) and do not support dynamically typed components.

Dynamic HDLs – Dynamic HDLs support both programmatic hardware generators and dynamically typed behavioral models thanks to the flexibility of its host language. PyRTL [CTD$^+$17], Migen [Mig23], MyHDL [Dec04], PyMTL [LZB14], and PyMTL3 [JPOB20] are all HDLs embedded in Python, a dynamically typed programming language. Figure 3.1 shows a parametrized adder generator and a dynamically typed polymorphic test harness in PyMTL3, a state-of-the-art dynamic HDL. The polymorphic test harness highlights how dynamic HDLs help improve verification productivity by promoting the reuse of parametrizable verification modules. In the polymorphic test harness example, the input setter and output checker of the test harness can be abstracted as functions taking variable number of arguments. Therefore, it is possible to reuse the harness and simulation setup across different designs by passing in different functions. Almost all existing dynamic HDLs do not check generators due to the lack of static checking capabilities. Instead, most dynamic HDLs rely on elaboration- and simulation-time checks on a flattened module hierarchy to enforce critical hardware properties.

Scope of This Chapter – I focus on symbolic elaboration in the context of dynamic HDLs in this chapter. Symbolic elaboration is particularly well suited to dynamic HDLs because it helps fill the static checking gap commonly found in these languages. It is also worth noting that the core idea of symbolic elaboration can be applied to other HDLs to enable more powerful static checking on generators than currently available in static HDLs.

3.2.2 Target Hardware Generator Properties

In this section, I introduce four key hardware generator properties on design correctness and synthesizability, which are targeted by symbolic elaboration.

Matching Bitwidths – One important aspect of structural modeling is modeling the circuit using a set of interconnected signals. Therefore, it is crucial for HDLs to verify that structurally connected signals have the same bitwidth. The matching bitwidth property also applies to behavioral modeling, where the bitwidths of operands in arithmetic operations should be equal. As an example, the left-hand side (LHS) and right-hand side (RHS) of connections on line 27-31 in Figure 3.1(a) should have the same bitwidth.

Correct Local Port Directions – Interconnected signals in circuits are generally further elaborated into *nets* of signals where at most one active signal drives all other signals in the same net. A complete net representation requires extensive cross-module analysis on the elaborated component [IKL+17,JPOB20] and may not be possible using static analysis. *Local* port direction analysis focuses on the connections within a component and can find common direction issues such as attempting to drive input-only ports from inside a component. As an example, only an output port can be used on the LHS of a signal assignment (line 33 in Figure 3.1(a)).

Bounded Array Indexing – Hardware generators often leverage arrays to model signals and sub-components. Design issues can therefore arise from out-of-bound array indexing. Unlike array index calculation in programs which can have arbitrary computation, array index generation in hardware generators generally consists only of simple arithmetics over constants, generator parameters, and loop induction variables. Therefore, static analysis on hardware generators should be able to detect almost all out-of-bound array indexing. As an example, array indices have to be smaller than the array length (line 25, 27-31 in Figure 3.1(a)).

Valid Hierarchical References – Traditional HDLs like Verilog allow accessing any attribute in the elaborated module hierarchy using a globally unique *hierarchical name* to facilitate hardware testing [iee17]. However, arbitrary hierarchical references do not model actual hardware behaviors. Synthesizable hardware generators must communicate through input and output data ports between two immediate levels in the module hierarchy. As a concrete example, the valid hierarchical reference property requires that only the input and output ports of the `FullAdder` sub-components can be accessed (line 27-31 in Figure 3.1(a)).

3.3 Limitations of Optional Type Checkers

In Section 2.3 of the previous chapter, I demonstrated that Mypy, an optional type checker for Python, can provide static correctness guarantees for simple hardware generators in PyMTL3. However, more sophisticated generators typically have complex properties that are challenging to encode in optional type checker's simple type system. Therefore, optional type checkers like Mypy often fails to capture some of the trial generator properties violations. In this section, I discuss the concrete limitations of applying off-the-shelf optional type checkers on hardware generators. These limitations motivate the symbolic elaboration technique and Section 3.4 presents solutions to these limitations.

Mypy cannot statically verify all matching bitwidths – In the adder generator in Figure 3.1(a), Mypy cannot reason about the exact bitwidth of `s.out` and the result of `concat()`. Therefore, it fails to verify that the LHS and RHS of line 33 in Figure 3.1(a) have the same bitwidth. I make the observation that *Mypy is only able to verify matching bitwidths that are not parametrized.* Unfortunately, many hardware generators rely on parametrized signal bitwidths (line 16 in Figure 3.1(a)) and deriving new hardware data types (line 19, 22 in Figure 3.1(a)), which undermine Mypy's effectiveness of type checking generators.

Verifying other generator properties with Mypy is challenging – Apart from the matching bitwidth property, it is challenging to encode other important generator properties discussed in Section 3.2.2 into Mypy. The bounded array indexing property requires analysis of the possible values of array indices, which Mypy does not perform. Both the correct local port direction and valid hierarchical reference properties involve complicated analysis of signals and components, which is challenging to encode in Mypy's simple type system.

Mypy does not account for path conditions during analysis – Hardware generators can also include if-statements to conditionally generate different hardware instances based on the given generator parameters. I refer to the set of if-conditions required to model a structural connection or behavior as *path conditions*. Path conditions can significantly affect the analysis of hardware generators because connections and child components may be instantiated conditionally. If-conditions can affect the result of the array index analysis because the possible values of an array index might depend on the if-condition. Mypy does not account for path conditions by design and therefore cannot perform the analysis described above.

3.4 Checking Generators using Symbolic Elaboration

In the previous section, I showed that existing off-the-shelf optional type checkers can verify certain generator properties but still have severe limitations in their capabilities. To overcome the limitations of existing optional type checkers, In this section, I propose a novel technique, *symbolic elaboration*, to address the limitations of off-the-shelf static type checkers. I observe that important generator properties can be encoded into integer constraints and that such constraints can be solved by a satisfiability modulo theory (SMT) solver. I design and implement a *symbolic elaborator* to perform symbolic elaboration, which statically analyzes the given hardware generator, builds an abstract generator model, constructs the integer constraints corresponding to the properties, and solves them using the Z3 SMT solver [MB08]. Compared to Mypy, the symbolic elaborator is able to reason about path conditions and statically verify all four properties in Section 3.2.2.

3.4.1 Building Abstract Generator Models

In this section I discuss how to build the abstract generator model based on a given target generator. I use Figure 3.2(a) and (b) as an example and demonstrate how to build an abstract adder model based on an example adder generator.

Reasoning about Generator Argument Arithmetics – The key feature that distinguishes symbolic elaboration from static type checkers like Mypy is the ability to precisely reason about generator argument arithmetics. Figure 3.2(b) shows the symbol table (which keeps track of active symbols the elaborator has encountered) and the abstract generator model (which records all active attributes of the current generator). The abstract generator model demonstrates that generator attributes can be potentially generic over a symbol or any arithmetics on symbols and concrete numbers. For example, the `s.out` port has a bitwidth derived from the arithmetics between the argument `Width` and an integer one. This arithmetic operation is preserved and will be translated into SMT-solvable constraints when checking properties of the abstract model.

Accounting for Path Conditions – In contrary to static type checkers like Mypy, path conditions are first-class citizens in symbolic elaboration: every entry in the symbol table or the abstract generator model has a *definition condition* (or defcond) which is the path condition up to the entry's point of definition. The symbolic elaborator maintains path conditions by pushing the if-condition or its negation to the *PathConds* stack before visiting the statements in the if- or else-branch. It pops the condition from the *PathConds* stack after the if-statement visit has finished. When the elaborator registers an entry with the symbol table or the abstract generator model, it constructs a boolean expression of the conjunction of all conditions in *PathConds* and use it as the definition condition of the entry. For example, in Figure 3.2(b), the dark red entry for loop induction variable i has a definition condition $i \geq 0$ because it is under the if-branch of the if-condition on line 17.

```
1  T = TypeVar("T", bound=Bits)
2  class Adder(Component, Generic[T]):
3    def __init__(s, Width:Type[T]) -> None:
4      ...
5    def construct(s, Width:Type[T]) -> None:
6      n       = get_nbits(Width)
7      s.a     = InPort(Width)
8      s.b     = InPort(Width)
9      s.out   = OutPort(mk_bits(n+1))
10     s.fa    = [FullAdder() for _ in range(n)]
11     s.carry = Wire(mk_bits(n+1))
12     s.sum   = Wire(Width)
13
14     for i in range(n):
15       if i >= 0:
16         ...
17         connect(s.carry[i+1],s.fa[i].cout)
18       if i == 0:
19         connect(s.carry[i],0)
20
21     @update
22     def upblk() -> None:
23       s.out @= concat(s.carry[n], s.sum)
```

(a) Target adder generator

`Adder` Symbol Table

Name	Type	DefCond
Width	Bits; generator arg	true
n	int; to_value(Width)	true
i	**int; 0 ≤ i < n**	**i ≥ 0**

`Adder` Abstract Generator Model

Name	Type	DefCond
a	InPort[Width]	true
b	InPort[Width]	true
out	OutPort[n+1]	true
fa	List[FullAdder] of n	true
carry	Wire[n+1]	true
sum	Wire[Width]	true

(b) Symbolic elaboration results

Property: Bounded Array Index

`s.carry[i+1]`

Array Length:	n+1
Index Expression:	i+1
Use Condition:	i ≥ 0

$$\neg\,(0 \le i+1 < n+1)\ \wedge\ (i \ge 0)$$
$$\wedge\ (0 \le i < n)\ \wedge\ (n = Width)$$

Property: Matching Bitwidth

`out@=cat(carry[n],sum)`

LHS Width:	n+1
RHS Width:	1+to_value(Width)
Use Condition:	true

$$\neg\,(n+1 = 1+Width)$$
$$\wedge\ (n = Width)$$

(c) Properties to Constraints

Figure 3.2: Symbolic Elaboration of an Adder – (a) target adder generator under elaboration (identical to `Adder` in Figure 3.1(a)); (b) symbolic elaboration results: dark red line indicates the state of the symbol table when the elaborator is processing line 16; (c) translation of bounded array index and bitwidth matching properties into integer constraints.

```
module Generator {
  -- Types
  data_type = GeneratorArgDataType(string name)
            | DataTypeFromNum(num_expr bitwidth)

  type = ComponentType(string comp)
       | DataType(data_type t)

  -- Statements
  stmt = Construct(arg* args, stmt* body)
       | If(bool_expr cond, stmt* body, stmt* orelse)
       | TmpVarAssign(string target, num_expr value)
       | AttrAssign(string target, inst value)
       | SignalAssign(sig_expr target, sig_expr value)
       | Connect(sig_expr u, sig_expr v)
       | For(string var, int start, int end, stmt* body)

  -- Expressions
  num_expr = GeneratorArgNum(string name)
           | InductionVarNum(string name, int start, int end)
           | UnsizedNum(int num)
           | UniOpNum(num_uni_op op, num_expr value)
           | BinOpNum(num_expr left, num_bin_op op, num_expr right)
           | NumFromDataType(data_type t)

  num_uni_op = NumClog2
  num_bin_op = NumAdd | NumSub | NumMult

  sig_expr = CurrentGenerator()
           | AttributeAccess(sig_expr value, string attr)
           | ArrayIndex(sig_expr value, num_expr index)
           | VectorIndex(sig_expr value, num_expr index)
           | BinOpSig(sig_expr left, sig_bin_op op, sig_expr right)
           | Concat(sig_expr* args)

  sig_bin_op = Add | Sub | Mult | LShift

  bool_expr = NumCompare(num_expr left, bool_cmp_op op, num_expr right)
            | BinOpBool(bool_expr left, bool_bin_op op, bool_expr right)
            | UniOpBool(bool_uni_op op, bool_expr value)

  bool_cmp_op = BoolCmpEq | BoolCmpLt
  bool_bin_op = BoolAnd | BoolOr | BoolXor
  bool_uni_op = BoolNot

  -- Component/Signal Instantiation
  inst = CompInst(string comp, arg* args, int* dims)
       | SignalInst(data_type bitwidth, int* dims)

  -- Construction Arguments
  arg = Argument(string name, type t)
}
```

Figure 3.3: Core Generator Modeling Syntax Targeted by the Symbolic Elaborator – *: zero or more objects.

3.4.2 Checking Properties of Abstract Generator Models

In this section I discuss how to encode hardware generator properties and path conditions into integer constraints that the Z3 SMT solver can verify. I use Figure 3.2(c) as an example and demonstrate how to check the bounded array index and matching bitwidth properties on an example adder generator.

Translating If-Conditions and Numeric Expressions – I use the term numeric expressions to refer to the arithmetics between symbols (i.e., generator arguments and loop induction variables) and concrete integers. Translating numeric and boolean expressions into Z3 expressions is straightforward because (1) generator arguments have a one to one correspondence to integer variables in Z3; (2) for-loop induction variables correspond to integer variables with two constraints on its lower and upper bound (e.g., induction variable `i` in `for i in range(n)` has constraints `i >= 0` and `i < n`); (3) conversion between bitwidth types and integers using `get_nbits` and `mk_bits` functions can be handled by adding one constraint that asserts the two symbols involved in the conversion are equal; (4) the Python binding of Z3 also offers integer constants and common binary arithmetic/comparison operations over integers.

Encoding the clog2 Function – One difficulty in encoding numeric expressions is that the `clog2` operation (`clog2(n)` $= \lceil log_2 n \rceil$) does not have a straightforward encoding in SMT solvers like Z3. I address this issue in two ways. I first add constant folding support so that `clog2` operations on constant values can be evaluated and replaced with its result. For non-constant `clog2` operands, I encode the `clog2` operation as an *uninterpreted function* from an integer to an integer in Z3. Z3 makes no assumptions about an uninterpreted function F except that $x = y \Rightarrow F(x) = F(y)$. This encoding scheme makes sure that any verified generator properties related to `clog2` must be true (i.e., no false-negatives).

Verifying Generator Properties – After translating numeric and boolean expressions to Z3, it is straightforward to verify the generator properties. To verify the bounded array index property, I construct an integer constraint that asserts the index has values out of the array bound. More specifically, for index expression i and array length len with definition condition def and under use condition *use*, integer constraint Equation 3.2 will be checked by Z3 for counter examples (as shown in Figure 3.2(c)). To verify if bitwidth u is the same as v under definition condition def and use condition *use*, I solve Equation (3.1) (as shown in Figure 3.2(c)). To verify the correct port direction or the valid hierarchical access property, I solve the conjunction of definition condition def and use condition *use* Equation 3.3 and consult the symbolic elaboration results to check if the port direction or hierarchical name is valid. It is worth noting that if Z3 finds $def \wedge use$ unsatisfiable, checking the target property is unnecessary because the definitions of symbols and signals in the target property are not available at the point of use.

$$(u \neq v) \land def \land use \tag{3.1}$$

$$\neg(0 \leq i \land i < len) \land def \land use \tag{3.2}$$

$$def \land use \tag{3.3}$$

If Z3 finds Equation 3.1 or Equation 3.2 unsatisfiable, we have obtained a proof that $u = v$ or i does not cause out-of-bound accesses for all possible values of integer variables. Otherwise we have obtained a counter example which corresponds to a set of symbol values that lead to design issues to be fixed.

3.4.3 Symbolic Elaboration Implementation

I implement a symbolic elaborator prototype that targets a subset of PyMTL3 hardware modeling DSL. My implementation requires type annotations for generator parameters. The supported modeling syntax allows efficient static analysis and is also expressive enough to model most hardware generators. Figure 3.3 shows the core components of the supported modeling syntax in the Zephyr abstract syntax description language [WaK97]. The core syntax is designed to be similar to the Python 3 syntax [py23] and serve as a straightforward target for PyMTL3 hardware generators.

Statements – The root of the generator abstract syntax tree is a `Construct` node, which corresponds to the `construct` method of each hardware generator. I explain the following three syntax nodes because they interact with the elaborator data structure in important ways: (1) `If` nodes corresponds to an if-else statement in the generator (e.g., line 17,19 of Figure 3.2(a)); this node has an if-condition of boolean expressions which are used to derive path conditions; (2) `For` nodes corresponds to a for-loop in the generator (e.g., line 14 of Figure 3.2(a)); this node introduces a new name constrained by the star and end of the for-loop under the current path condition; (3) `AttrAssign` nodes indicate the addition of signal and sub-component attributes to the current generator (e.g., line 7-13 of Figure 3.2(a)); this node adds an attribute to the current generator under the current path condition.

Signal Expressions – Signal expressions reference signals in the current generator and are common operands of arithmetic operations and structural connections (e.g., `concat(s.carry[n], s.sum)` in Figure 3.1(a) are both signal expressions). Common signal expressions include attribute

```
Require: $T$: AST node of the generator under elaboration.
Require: $P$: Path conditions (set of bool_expr).
Require: $N$: Names encountered (set of num_expr).
Require: $G$: Generator (set of pairs (attribute name, inst)).
Ensure: Return type of AST node $T$; none if $T$ is a statement.
 1: function SYMBELAB($T, P, N, G$) ▷ Args passed by reference.
 2:     if $T$ is Construct then
 3:         for all $stmt \in T.body$ do
 4:             SYMBELAB($T, P, N \vee T.args, G$)
 5:     if $T$ is If then
 6:         for all $t \in T.body$ do
 7:             SYMBELAB($t, P \vee T.cond, N, G$)
 8:         for all $t \in T.orelse$ do
 9:             SYMBELAB($t, P \vee \neg T.cond, N, G$)
10:     if $T$ is For then
11:         for all $t \in T.body$ do
12:             SYMBELAB($t, P, N \vee (T.var, T.start, T.end), G$)
13:     if $T$ is AttrAssign then ▷ $v|P$: $v$ only valid if $P$ holds
14:         $v \leftarrow$ SYMBELAB($T.value, P, N, G$)
15:         $G \leftarrow G \vee (T.target, v)|P$
16:     if $T$ is GeneratorArgNum then
17:         $N \leftarrow N \vee T.name|P$
18:         return GeneratorArgNum(T.name)
19:     if $T$ is ArrayIndex then
20:         $I \leftarrow$ bitwidth of SYMBELAB($T.index, P, N, G$)
21:         $L \leftarrow$ length of SYMBELAB($T.value, P, N, G$)
22:         $C \leftarrow$ defcond of $I$ and $L$ via $N \vee G$
23:         if $\neg(0 <= I < L)$ solvable w.r.t $C \wedge P$ then
24:             Report out-of-bound array index at $T$
25:     if $T$ is SignalAssign,Connect,BinOpSig then
26:         ▷ Assume two operands are $T.left$ and $T.right$. ◁
27:         $L \leftarrow$ bitwidth of SYMBELAB($T.left, P, N, G$)
28:         $R \leftarrow$ bitwidth of SYMBELAB($T.right, P, N, G$)
29:         $C \leftarrow$ defcond of $L$ and $R$ via $N \vee G$
30:         if $\neg(L = R)$ solvable w.r.t $C \wedge P$ then
31:             Report bitwidth mismatch at $T$
32:         if $T$ is SignalAssign then
33:             $t \leftarrow$ SYMBELAB($T.left, P, N, G$)
34:             if $\exists H \in G_s : t \in G_i \vee H_o$ w.r.t $C \wedge P$ then
35:                 Report incorrect port direction at $T$
36:     if $T$ is AttributeAccess then
37:         $V \leftarrow$ SYMBELAB($T.value, P, N, G$)
38:         $C \leftarrow$ defcond of $V$ via $N \vee G$
39:         if $V \in G_s \wedge T.attr \notin V_p$ w.r.t $C \wedge P$ then
40:             Report invalid hierarchical reference at $T$
```

Figure 3.4: Core Symbolic Elaboration Algorithm. defcond=definition condition; G_i=set of input ports of generator G; G_o=set of output ports of generator G; G_s=set of sub-generators of generator G (i.e., generators instantiated inside G); G_p=$G_i \vee G_o$ (set of all ports of generator G).

accesses (`AttributeAccess`), indexing into a signal array (`ArrayIndex`), indexing into a signal (`VectorIndex`), binary arithmetic operations between signals (`BinOpSig`), and signal concatenation (`Concat`).

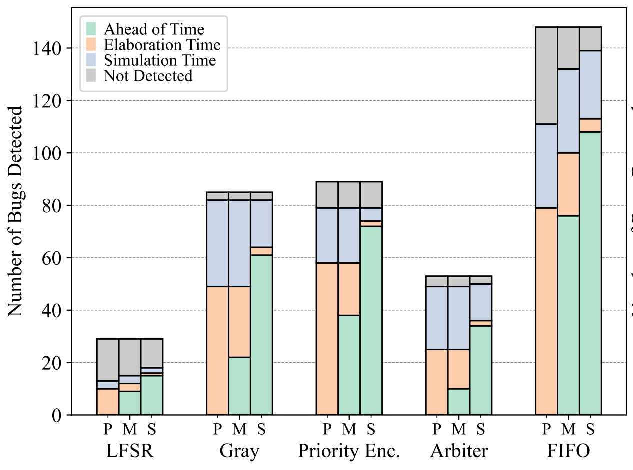
Numeric Expressions – Numeric Expressions represent integers used in hardware generators. Common numeric expressions include generator arguments that have an integer type (`GeneratorArgNum`), for-loop induction variables (`InductionVarNum`), and integer literals (`UnsizedNum`). The numeric expressions also keep track of the binary (`BinOpNum`) or unary (`UniOpNum`) operations between numeric expressions. This enables precise reasoning about numeric values, which is critical in matching bitwidth analysis.

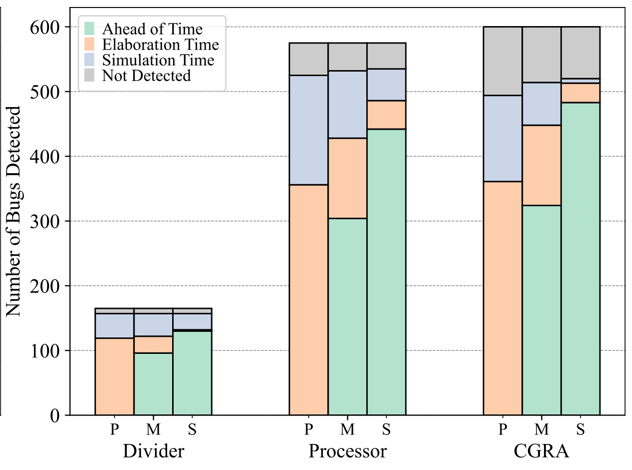
Boolean Expressions – Boolean expressions are used in if-conditions. The syntax in Figure 3.3 assumes that only comparisons between numeric expressions can be used in the if-conditions. However, it is straightforward to extend the syntax to support if-conditions with signal values (often used in behavioral modeling). Boolean expressions enable the precise reasoning of array index values, which is critical to detecting out-of-bound array indices.

Core Symbolic Elaboration Algorithm – Figure 3.4 shows the core algorithm of symbolic elaboration which is based on the traversal of the target generator's abstract syntax tree (AST; syntax defined in Figure 3.3). The symbolic elaborator holds three bookkeeping data structures during elaboration; as the elaborator walks through the generator AST, it keeps tracks of all symbols derived from the generator's arguments and for-loop induction variables (N), maintains the current path condition based on the if-conditions encountered and the if-else branches (P), and records all attributes added to the generator (G). While verifying the properties discussed in Section 3.2.2, the algorithm generates a *path constraint* which is the conjunction between the current use condition and the definition conditions of names and expressions under verification ($C \wedge P$). The algorithm checks for out-of-bound array indices by searching for index values less than zero or larger than or equal to the length under the path constraint with the SMT solver; similarly it checks for bitwidth mismatches by searching for mismatched left and right bitwidths under the path constraint; checks for correct port directions and valid hierarchical names only invoke the SMT solver to verify the path constraint is satisfiable and verify the properties by looking up attributes in the bookkeeping data structures.

3.5 Evaluation

In this section, I describe my methodology of evaluating the symbolic elaborator prototype and the evaluation results. In order to quantitatively evaluate symbolic elaborator's static type checking capabilities, I reuse the mutation-based abstract syntax tree (AST) fuzzer that can randomly inject six common categories of bugs from the evaluation of GT-HDL in Section 2.5.2. I apply the prototype symbolic elaboration implementation to the eight RTL design generators in Section 2.5.1.





(a) Bug Detection Results on Common IPs (b) Bug Detection Results on Standalone Designs

Figure 3.5: Bug Detection Results – (a) and (b) show the number of bugs detected by PyMTL3 (P), Mypy+PyMTL3 (M), and Symbolic Elaboration+PyMTL3 (S) at different stages. The more bugs detected ahead of time the better.

3.5.1 Bug Detection

In this section, I evaluate the bug detection effectiveness of PyMTL3, Mypy+PyMTL3 (using Mypy as an optional type checker), and symbolic elaboration+PyMTL3 on eight bug-injected design generators. I randomly inject bugs into a generator using the AST fuzzer (one bug at a time) and then evaluate if PyMTL3, PyMTL3+Mypy, and symbolic elaboration+PyMTL3 are able to detect the injected bug. This evaluation compares the effectiveness of symbolic elaboration's symbolic elaborator against state-of-the-art elaboration-time checks (PyMTL3) and an optional type checker (Mypy).

Figure 3.5 shows the number of bugs detected by PyMTL3, Mypy+PyMTL3, and symbolic elaboration+PyMTL3 at each stage: ahead of time (statically), during elaboration, during simulation, or not detected. A syntax mutation may not be detected because the mutation does not lead to a type error, does not change the generator's functionality, or the simulation test vectors do not reach 100% coverage. We can see that *symbolic elaboration detects the same number of more bugs than PyMTL3 and Mypy+PyMTL3*. On average, symbolic elaboration+PyMTL3 detects 90.6% of the injected syntax mutations, which is slightly higher than PyMTL3 (86.6%) and Mypy+PyMTL3 (89.4%). This demonstrates the effectiveness of symbolic elaboration's symbolic elaborator, which is able to detect bugs missed by simulation (simulation test vector may not achieve 100% coverage). Figure 3.5 also shows that *symbolic elaboration is able to detect significantly more bugs*

		Bug	BitWid	CpAttr	PortDir	NameExp	AtBase	Funct	Total	%
Divider		Num. Bugs	26	30 *	20	30 *	30 *	29	165	
Divider	PyMTL3	ET	20	20	20	29	30	0	119	72.1 %
		ST	3	10	0	1	0	24	38	23.0 %
		Total	23	30	20	30	30	24	157	
		%	88.5 %	100.0 %	100.0 %	100.0 %	100.0 %	82.8 %		95.2 %
	Mypy+ PyMTL3	AoT	18	19	0	29	30	0	96	58.2 %
		ET	2	4	20	0	0	0	26	15.8 %
		ST	3	7	0	1	0	24	35	21.2 %
		Total	23	30	20	30	30	24	157	
		%	88.5 %	100.0 %	100.0 %	100.0 %	100.0 %	82.8 %		95.2 %
	SE	AoT	22	23	20	29	30	6	130	78.8 %
		ET	0	2	0	0	0	0	2	1.2%
		ST	1	5	0	1	0	18	25	15.2 %
		Total	23	30	20	30	30	24	157	
		%	88.5 %	100.0 %	100.0 %	100.0 %	100.0 %	82.8 %		95.2 %
Processor		Num. Bugs	100 *	100 *	75	100 *	100 *	100 *	575	
Processor	PyMTL3	ET	42	79	75	39	96	25	356	61.9 %
		ST	54	14	0	49	0	52	169	29.4 %
		Total	96	93	75	88	96	77	525	
		%	96.0 %	93.0 %	100.0 %	88.0 %	96.0 %	77.0 %		91.3 %
	Mypy+ PyMTL3	AoT	77	70	0	59	98	0	304	52.9 %
		ET	2	19	75	3	0	25	124	21.6 %
		ST	17	5	0	30	0	52	104	18.1 %
		Total	96	94	75	92	98	77	532	
		%	96.0 %	94.0 %	100.0 %	92.0 %	98.0 %	77.0 %		92.5 %
	SE	AoT	92	76	73	80	98	23	442	71.8 %
		ET	1	14	2	3	0	24	44	9.0%
		ST	3	4	0	12	0	30	49	13.0 %
		Total	96	94	75	95	98	77	535	
		%	96.0 %	94.0 %	100.0 %	95.0 %	98.0 %	77.0 %		93.0 %
CGRA		Num. Bugs	100 *	100 *	100 *	100 *	100 *	100 *	600	
CGRA	PyMTL3	ET	52	77	100	35	93	4	361	60.2 %
		ST	43	2	0	57	0	31	133	22.2 %
		Total	95	79	100	92	93	35	494	
		%	95.0 %	79.0 %	100.0 %	92.0 %	93.0 %	35.0 %		82.3 %
	Mypy+ PyMTL3	AoT	55	69	0	100	100	0	324	54.0 %
		ET	7	13	100	0	0	4	124	20.7 %
		ST	33	2	0	0	0	31	66	11.0 %
		Total	95	84	100	100	100	35	514	
		%	95.0 %	84.0 %	100.0 %	100.0 %	100.0 %	35.0 %		85.7 %
	SE	AoT	98	77	83	100	100	25	483	80.5 %
		ET	0	9	17	0	0	4	30	5.0%
		ST	0	1	0	0	0	6	7	1.2%
		Total	98	87	100	100	100	35	520	
		%	98.0 %	87.0 %	100.0 %	100.0 %	100.0 %	35.0 %		86.7 %

Table 3.2: Number of Injected Bugs Detected at Different Stages – AoT/ET/ST: bug detected ahead of time/during elaboration/during simulation. SE: symbolic elaboration. Numbers indicate the number of bugs detected. *I randomly sample the same number of mutations to make sure bug injection is not biased.

ahead of time than Mypy. On average, symbolic elaboration detects 77.1% of syntax mutations ahead of time, whereas Mypy only detects 50.4%. This confirms that the symbolic elaborator is a better approach to statically type check generators than existing optional type checkers.

RCA Generator	2-bit RCA	4-bit RCA	8-bit RCA	16-bit RCA
0.087s	0.055s	0.082s	0.131s	0.241s

Table 3.3: Symbolic Elaboration Run Time – RCA: ripple-carry adder. Ripple-carry adders with a specific bitwidth indicates an adder instance with explicitly described structural connections. Run time is averaged across five runs of symbolic elaboration.

To better understand the results on standalone designs, I break down the number of bugs detected in each syntax mutation category (Section 2.5.2) in Table 3.2. We can see from the table that symbolic elaboration is particularly effective at detecting bitwidth mismatches and incorrect port directions ahead of time. On average, symbolic elaboration detects 91.5% of the injected bitwidth mutations and 93.4% of the port direction mutations, while Mypy detects only 67.1% and 0.0% of them. This demonstrates the significant benefits of the symbolic elaborator's constraint solving approach capabilities, which is superior at handling bitwidths and directions than Mypy.

3.5.2 Scalability

In this section, I discuss the scalability of symbolic elaboration with increasingly complex hardware generators. Since the symbolic elaborator detects generator property violations at the source code level, the run time of symbolic elaboration is, to the first order, linear to the number of lines of generator code. To validate this claim, I apply the symbolic elaborator to a ripple-carry adder generator and ripple-carry adder instances of bitwidth 2, 4, 8, and 16, which are described with structural connections between full adders. Table 3.3 shows the averaged run time of symbolic elaboration on these designs. We can see from the table that symbolic elaboration spends more time on the adder generator than on adder instances of bitwidth one and two. This is mostly because the adder generator contains if-statement and for-loops that lead to more definition and use conditions for the SMT solver. However, as the bitwidths of adder instances grow larger, the instances have more lines of code which translate to longer symbolic elaboration run time. The averaged run times show that symbolic elaboration scales roughly linearly with respect to the increasing bitwidth of adders. For sophisticated hardware generators in agile workflows, this scalability analysis implies that agile designers may experience longer symbolic elaboration time on complex generators with a significant focus of structural modeling, but not on individual hardware instances because their parent generator has been symbolically elaborated.

3.6 Related Work

Symbolic execution generally applies constraint solving techniques to reason about path conditions and is commonly used in program analysis activities including software testing, software vulnerability detection, and security analysis [CGK+11, SGS+16, SAB10]. Traditional concrete execution of programs requires concrete test inputs, and each execution is limited to one control flow. Symbolic execution is able to explore multiple conditional paths in parallel and keep track of the constraints applied to the input symbols along the path as a series of boolean expressions. To solve the constraints, symbolic execution often leverages a satisfiability modulo theories (SMT) solver to verify whether certain program properties have been violated and whether some paths are feasible [BCD+18]. If the SMT solver proves that some program properties are violated, or a path is feasible, it can generate a feasible solution to the boolean expressions, which can be mapped back to a concrete test input that triggers the offending violations or exercises certain branches.

The symbolic elaboration technique of GT-HDL is similar to symbolic execution techniques in two ways: (1) symbolic elaboration is a purely static analysis technique and does not support arbitrary code execution; (2) symbolic elaboration also heavily relies on constraint solving and SMT solvers. However, symbolic elaboration is different from symbolic execution in that (1) symbolic elaboration leverages constraint solving to reason about path conditions, matching bitwidths, and bounded array indexing, whereas symbolic execution typically uses constraint solving to trigger rarely explored program control paths that might include program bugs; (2) symbolic elaboration maintains much fewer symbolic states (only generator arguments in numeric expressions are symbolic) than symbolic execution, which generally maintains symbolic states for memory locations to generate accurate analysis results [Kin76].

Salama et al. propose to use constraint solving to detect consistency issues in Verilog generator interconnects [SMT+11]. They define Featherlight Verilog, a hardware modeling language inspired by Verilog syntax, as the target of their consistency analysis. Compared to symbolic elaboration, both Featherlight Verilog and symbolic elaboration encodes signal bitwidths and array indices as integer constraints and use an SMT solver to detect inconsistencies among bitwidths and indices. However, the symbolic elaborator of GT-HDL targets more hardware generator properties and more aspects of hardware modeling. Featherlight Verilog specifically targets bitwidth and array index inconsistencies in structural modeling, and symbolic elaboration targets bitwidths, array indexing, port directions, and hierarchical references in both structural and behavioral modeling.

Rondon et al. propose liquid types, a type system that infers dependent types to aide static detections of out-of-bound array indexing errors [RKJ08]. Similar to symbolic elaboration, liquid types leverage a constraint solver to verify the safety of array accesses. Both SE and liquid type try to reduce programmer type annotation efforts by inferring precise types whenever possible. However, symbolic elaboration focuses on inferring types for dynamic HDLs and can statically verify generator properties discussed in Section 3.2.2. In contrast, liquid types focuses on protecting array indexing accesses in traditional functional programming languages.

3.7 Conclusion

In this chapter, I address the verification challenge in generator development through symbolic elaboration, an SMT-based static analysis technique that validates generator properties ahead of time. symbolic elaboration supports powerful static checks on hardware generators. Symbolic elaboration statically checks generators using the symbolic elaborator that enforces critical generator properties such as matching bitwidths, correct local port directions, bounded array indexing, and valid hierarchical references. My evaluation of a symbolic elaboration prototype demonstrates a significant improvement in its ability to identify design bugs early in the design cycle.

CHAPTER 4
LATENCY EQUIVALENCE CHECKING: ADDRESSING VERIFICATION CHALLENGES IN INSTANCE COMPOSITION

Latency-insensitive protocols are widely used in hardware standard libraries and network-on-chip IPs because they enable modular hardware design and efficient circuit implementation of communication channels. However, RTL modules with latency-insensitive protocols at their interfaces (or *latency-insensitive RTL modules*) create a verification challenge because subtle design bugs in these RTL modules may only be triggered after a specific number of stall cycles. Verifying latency-insensitive RTL modules with simulation-based techniques requires a comprehensive test suite that covers all possible stall cycles up to a sufficiently large number, which needs significant verification efforts to build and maintain.

In this chapter, I address the verification challenge in the composition of latency-insensitive instances. I propose a formal verification methodology to detect bugs in latency-insensitive RTL modules by verifying the stall invariant property of these modules. I introduce bounded latency equivalence checking (BLEC) to detect violations of the stall invariant property under finite buffering. BLEC includes a systematic approach to construct a verification harness which applies ingress and egress stalls and checks if the DUV egress results are the same under varying stall conditions. I implement the proposed method with state-of-the-art commercial formal verification tools and demonstrate its effectiveness with case studies on a latency-insensitive processing element, a greatest common divisor unit, and a pipelined RISC-V processor. In all three case studies, the proposed method can detect subtle design bugs inserted in the design. With some manual simplifications to the target RTL modules, existing formal verification tools can provide a bounded proof of the stall invariant property to many RTL modules.

4.1 Introduction

Latency-insensitive (LI) protocols [CMSV99, Car15, CMSV01] are an effective hardware design methodology that significantly improves design productivity with minimal performance, power, and area overhead [CMSSV99, LSC10]. By decoupling the communication and computation aspects of hardware design, RTL modules with latency-insensitive interfaces (or simply latency-

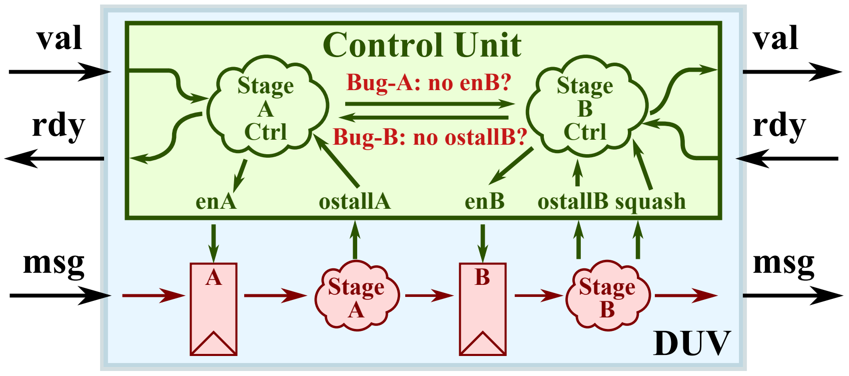


Figure 4.1: Examples of Bugs in a Two-Stage Pipelined Latency-Insensitive RTL Module – the design under verification (DUV) has complex control logic to adapt to possible delays on the ingress (left) or the egress (right) interface. Both pipeline registers (`A` and `B`) are enabled when there is no stall originating from their respective stages; stage B can also *squash* stage A due to a hazard that is only visible in a later stage. Bug-A is a design bug where the `ostallA` signal is not accounted for in the enable signal of stage B (`enB`); Bug-B shows a bug where the `ostallB` signal is not propagated from stage B to stage A. `ostallA/B`: signal is asserted if stage A/B originates a stall; the `ostall` signals may be propagated to earlier stages; `enA/B`: enable signal for pipeline register of stage A/B.

insensitive modules) offer two major benefits over the traditional synchronous design paradigm [BCE+03, HS07]. First, hardware designers can safely compose modules with latency-insensitive interfaces without worrying about the potentially *variable* latencies of upstream and downstream modules. In the case where the upstream or downstream modules are not generating valid messages or not ready to accept messages, a *stalling* event occurs on the latency-insensitive interface and sequential states are preserved until an *informative* event containing the real message eventually happens [CMSV01]. Second, latency-insensitive protocols enable more efficient circuit implementation of communication channels than with the synchronous design paradigm. Inter-module communication channels designed under a synchronous system assumption often synthesize into long global wires that limit the system clock frequency. On the other hand, the communication between latency-insensitive modules can be pipelined by inserting *relay stations* [CMSV01] between the modules to achieve higher clock frequency. Because of these benefits, latency-insensitive modules are virtually ubiquitous across hardware standard libraries, hardware compositions, and network-on-chip IPs [Tay18b, FAP+12, TOJ+19, DT01, BM02, Ltd11].

However, implementing latency-insensitive RTL modules presents a unique verification challenge. Figure 4.1 shows a two-stage pipelined latency-insensitive RTL module. To handle the potential backpressure from the egress interface or the input delays on the ingress interface, the design under verification (DUV) has complex control logic which includes pipeline register enable signals, per-stage originating stall signals, and a squash signal. Two examples of control logic bugs

No Stalls					
Cycle	1	2	3	4	5
Ingress	a	b	c		
Egress	−	a	b	c	

Ingress Stall					
Cycle	1	2	3	4	5
Ingress	a	$\ominus$	b	c	
Egress	−	a	−	b	c

Egress Stall					
Cycle	1	2	3	4	5
Ingress	a	b	#	c	
Egress	−	a	$\textcircled{\#}$	b	c

Egress Informative Events			
No Stalls	a	b	c
Ingress Stall	a	b	c
Egress Stall	a	b	c

(a) Behaviors of the Correct Design

No Stalls					
Cycle	1	2	3	4	5
Ingress	a	b	c		
Egress	−	a	b	c	

Ingress Stall					
Cycle	1	2	3	4	5
Ingress	a	$\ominus$	b	c	
Egress	−	a	×	b	c

Egress Stall					
Cycle	1	2	3	4	5
Ingress	a	b	#	c	
Egress	−	a	$\textcircled{\#}$	b	c

Egress Informative Events				
No Stalls	a	b	c	
Ingress Stall	a	×	b	c
Egress Stall	a	b	c	

(b) Behaviors of the Design with Bug-A

No Stalls					
Cycle	1	2	3	4	5
Ingress	a	b	c		
Egress	−	a	b	c	

Ingress Stall					
Cycle	1	2	3	4	5
Ingress	a	$\ominus$	b	c	
Egress	−	a	−	b	c

Egress Stall					
Cycle	1	2	3	4	5
Ingress	a	b	c		
Egress	−	a	$\textcircled{\#}$	b	

Egress Informative Events			
No Stalls	a	b	c
Ingress Stall	a	b	c
Egress Stall	a	b	

(c) Behaviors of the Design with Bug-B

Figure 4.2: Design Behaviors under Different Stall Conditions – Designs are the same as in Figure 4.1. −: not-val stalling event; #: not-ready stalling event. Three designs are used in this figure: the correct design as shown in Figure 4.1, the design with Bug-A (wrong `enB` signal), and the design with Bug-B (`ostallB` signal not forwarded to stage A). Three stall conditions are used in this figure: no stalls: the ingress LI interface is always valid to produce a message and the egress LI interface is always ready to accept an output message; ingress stall: the ingress LI interface is not valid at cycle 2 (marked with black circle) which leads to a bubble (− at cycle 3) in the correct design's pipeline; egress stall: the egress LI interface is not ready at cycle 3 (marked with black circle) which causes the pipeline in the correct design to stall (# at cycle 3). Only the correct design is stall-invariant because the other two designs have a different sequence of egress informative events (×: incorrect value registered; c: message c accepted when pipeline stalls) either under ingress stalls (Bug-A) or egress stalls (Bug-B).

are also in this figure. For Bug-A, the `ostallA` signal is not propagated to the enable signal of stage B (`enB`), which means pipeline register `B` can be enabled while stage A is originating a stall and may register incorrect data. For Bug-B, the `ostallB` signal is not propagated to the control logic of stage A, which can lead to data loss because the content of pipeline register `B` can be over-written by outputs from stage A even when stage B is originating a stall. It is worth noting that these two bugs only manifest when there is backpressure on the egress interface, and that similar and subtler bugs might only get triggered with a specific number of cycles of stalls on the ingress and/or the egress interface. It is challenging to discover these bugs via simulation-based dynamic verification techniques. Detecting these bugs in simulation needs a comprehensive test suite that covers all possible stall cycles (up to a sufficiently large number) on the DUV's latency-insensitive interfaces, which requires significant testing and verification efforts to build and maintain.

In this chapter, I propose a formal verification methodology to address the verification challenges of latency-insensitive RTL modules. I make the observation that most correct latency-insensitive RTL modules have the same behavior even under different number of stall cycles, which I call the *stall invariant* property. I propose bounded latency equivalence checking (BLEC), a technique that detects violations of the design under verification (DUV)'s stall invariant property under finite buffering. BLEC constructs a verification harness that contains two duplicated DUVs with different stall conditions and verifies the latency equivalence [CMSV99] between the DUVs using formal verification. A BLEC verification process generates one of two possible outcomes: (1) BLEC finds a violation to the stall invariant property of the DUV and provides a waveform to help identify origin of issues or (2) BLEC proves that the stall invariant property holds true for the DUV up to a certain number of stall cycles.

This chapter makes the following contributions:

- I introduce the stall invariant property and make the observation that many bugs in latency-insensitive RTL modules violate the stall invariant property;
- I propose bounded latency equivalence checking, a formal verification technique to detect violations of the DUV's stall invariant property under finite buffering; I implement bounded latency equivalence checking using state-of-the-art commercial formal verification tools;
- I demonstrate the effectiveness of the proposed method by evaluating bounded latency equivalence checking on three latency-insensitive RTL designs: a latency-insensitive processing element, a greatest common divisor (GCD) unit, and a RISC-V pipelined processor.

4.2 The Stall Invariant of Latency-Insensitive RTL Modules

In this section, I introduce the stall invariant property with the motivating DUV in Figure 4.1. Figure 4.2 (a)-(c) refer to the behaviors of the DUV without bugs, with Bug-A, with Bug-B, respectively. I examine the behaviors of the DUV both with and without bugs and compare the behaviors under different stall conditions. I make the observation that bugs in RTL modules generally lead to inconsistent behaviors on the egress LI interface under different ingress and/or egress stalls.

Events in Latency-Insensitive RTL Modules – The behaviors of each design in Figure 4.2 are characterized by the sequence of *events* that occurs on the ingress and egress LI interfaces of the DUV. Using the terminology from the original latency-insensitive design theory paper [CMSV01], I call events where a message is successfully transferred over the LI interface an *informative event* (cycles that are marked a, b, c, or × in Figure 4.2); I call any other events where a message is not transferred *stalling events*. I further classify stalling events into two categories: (1) not-valid (indicated by symbol − in Figure 4.2), where the sender of the LI interface is not valid to send a message at the cycle of the event; (2) not-ready (indicated by symbol # in Figure 4.2), where the sender of the LI interface has valid message to send but the receiver is not ready to accept that message at the cycle of the event. In an RTL module that implements a val-rdy LI interface (e.g., DUV in Figure 4.1), not-valid stalling events correspond to cycles where `val` is low and not-ready stalling events are cycles where `val` is high but `rdy` is low.

Stall Conditions of a Latency-Insensitive DUV – The ingress and egress interfaces of a LI DUV need to be connected to upstream and downstream modules for the DUV to function properly. An upstream module can apply *input stalls* to the DUV by de-asserting the `val` signal at cycles it does not have valid messages to send, which creates a not-valid stalling event. Similarly a downstream module can apply *output stalls* to the DUV by de-asserting the `rdy` signal at cycles it is not ready to accept messages from DUV, which can create a not-ready stalling event. For the same sequence of informative events, I call the cycles where input and output stalls are applied the *stall condition* of the DUV. To make my explanations more clear, I examine three simple stall conditions for each design in Figure 4.2: (1) No Stalls, where the ingress interface is always valid to send a message to the DUV and the egress interface is always ready to accept a message from the DUV; (2) Ingress Stall, where the ingress interface is not valid at cycle 2 for illustration purposes; (3) Egress Stall, where the egress interface is not ready at cycle 3.

Behaviors of the DUV – Figure 4.2 (a) shows the behaviors of the correct DUV under the three stall conditions described above. It is straightforward that the correct design exhibits pipeline behaviors between its input stage A and output stage B: it always takes two cycles for a message to traverse from ingress to egress when no stalls are applied; applying input stalls creates bubbles in the pipeline, as shown by the not-valid stalling events in cycle 2 and 3 (Ingress Stall); and applying output stalls stalls the pipeline as shown in cycle 3 (Egress Stall). Figure 4.2 (b) shows the behaviors of the DUV with Bug-A, where the pipeline registers of stage B can still be enabled even when stage A is stalling. Bug-A has the same behaviors as the correct design when no stalls or only output stalls are applied because stage A is not stalled in these two cases; however, when ingress stall is applied on cycle 2, the pipeline registers of stage B will register invalid data from the stalled stage A, which leads to an erroneous output message on cycle 3 (marked by red $\times$). Figure 4.2 (c) shows the behaviors of the DUV with Bug-B, where stage A is not stalled when stall B is stalled. Bug-B has the same behaviors as the correct design when no stalls or only input stalls are applied because stage B is not stalled in these two cases; however, when egress stall is applied on cycle 3, message C is lost at this cycle because stage A does not stall.

The Stall Invariant Property – The example in Figure 4.2 shows that some designs have inconsistent behaviors (as determined by the sequence of informative events on their egress interfaces) under different stall conditions. I call a latency-insensitive RTL module *stall invariant* if the module has the same sequence of informative events on its egress interfaces under all possible stall conditions. The stall invariant property is useful for catching bugs that lead to a different sequence of informative events on the LI interface of the DUV, which include numerous subtle bugs especially in a pipelined DUV module. It is worth noting that the stall invariant property only requires the equivalence of the sequence of egress informative events and does not imply functional correctness of the DUV.

4.3 Bounded Latency Equivalence Checking

In this section, I introduce bounded latency equivalence checking (BLEC), a formal verification technique that detects violations of the DUV's stall invariant property under finite buffering. For a given latency-insensitive RTL module (the DUV), BLEC constructs a verification harness with formal assertions that can be verified by hardware formal property verification (FPV) tools. The FPV tools can either find a violation of the stall invariant property (which generally indicates the existence of a design bug) or provide a potentially bounded proof that the target DUV is stall invariant. I first introduce the necessary verification modules that are used in the BLEC verification harness (Section 4.3.1). I then propose a systematic method that constructs the BLEC verification harness for any given latency-insensitive RTL module (Section 4.3.2).

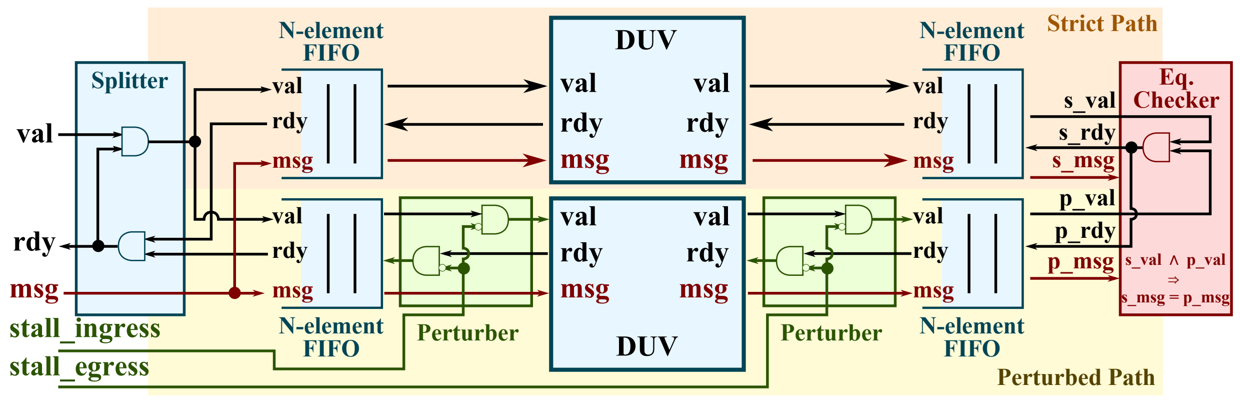


Figure 4.3: Verification Harness for a DUV with One Ingress and One Egress LI Interface in Bounded Latency Equivalence Checking – `stall_ingress/egress`: stall variables for the ingress/egress LI interface. `N`: a parameter which determines the depth of FIFOs in the verification harness. Strict path: a path in the harness where no ingress or egress stalls are applied on the DUV. Perturbed path: a path in the harness where the *perturbers* apply a random number of stalls on the ingress and egress interface. `Eq. checker`: equivalence checker; a module that checks if the result messages from the two paths are the same.

4.3.1 Verification Modules

Figure 4.3 shows the verification harness of a DUV with one ingress and one egress latency-insensitive interface. The verification harness in Figure 4.3 exposes five input and output ports:

- `val`, `rdy`, and `msg`: these three ports form the LI interface that generates input messages to the ingress LI interface of the DUV.
- `stall_ingress` and `stall_egress`: these two ports are *stall variables* whose value decides if an ingress stall or an egress stall is applied on the DUV's LI interface (1 for stall and 0 for not stall).

As is shown in the figure, the verification harness contains two duplicated instances of the target DUV with different stall conditions: the DUV in the *strict path* (i.e., the *strict DUV*) has no ingress or egress stalls under with FIFOs of N elements; the DUV in the *perturbed path* (i.e., the *perturbed DUV*) has random ingress and egress stalls injected by *perturbers*. At the end of both path, a *equivalence checker* compares the result messages in the output FIFOs and reports a violation of the stall invariant property if the two messages are different.

N-Element FIFOs – The verification harness includes four N-element FIFOs to decouple the LI interfaces of the two DUV instances, where N is a constant determined ahead of the construction of verification harness. Two FIFOs are inserted between the ingress LI interfaces of the two DUVs and the top-level ingress LI interface (`val`, `rdy`, `msg`). These FIFOs decouple the strict DUV from the ingress stalls of the perturbed DUV, which achieves almost zero ingress stalls for the strict DUV. Similarly, the two FIFOs between the egress LI interfaces of the DUVs and the message checker decouple the strict DUV from the egress stalls of the perturbed DUV, which achieves almost zero egress stalls for the strict DUV.

Assuming no ingress stalls nor egress stalls are applied on the DUV, FPV tools can generate a proof that the DUV is indeed stall invariant. This can be shown by comparing the behaviors of the perturbed DUV against the strict DUV: the equivalence checker ensures that the sequence of egress informative events of the DUV under all stall conditions (output of the perturbed path) is the same as if no stalls are applied (output of the strict path); therefore, the DUV is stall invariant by definition (Section 4.2).

It is worth noting that even with deep FIFOs (large N's), the strict DUV may still experience ingress or egress stalls. The FPV tools can still prove that the perturbed and strict DUV have the same sequence of egress informative events. I call this proof a *bounded* stall invariant proof because the strict DUV experiences ingress and/or egress stalls due to finite buffering. The finite buffering also defines the bounded nature of the proposed BLEC technique: BLEC is only able to provide bounded stall invariant proofs because FIFO sizes are finite. The finite size of FIFOs does not affect the technique's effectiveness in finding violations of the stall invariant property because stall variant DUVs mostly generate different sequences of egress informative events under non-zero stalls, not necessarily zero stalls. Large depths of FIFOs may also have negative impacts on the performance of the formal property verification tools. Therefore, I choose a small FIFO depth of two (2) in this chapter to decouple the strict and perturbed DUVs without causing too much tool performance overhead.

Perturbers – Perturbers are a verification module inserted between the DUV and FIFOs to inject random stalls to the ingress or egress LI interface of the DUV (i.e., to perturb the DUV with random ingress or egress stalls). A perturber takes as input a stall variable (`stall_ingress` and `stall_egress` input ports in Figure 4.3), which decides if stall is applied on the LI interface. As the green-shaded components in Figure 4.3 show, the perturbers connect the `val` and `rdy` LI handshake signals and the corresponding negated stall variable with an AND gate. This logic suppresses the LI handshake (and thus stalls the LI interface) when the stall variable is high.

Equivalence Checker – The equivalence checker is a module that checks if the results of the egress latency-insensitive interface from the two paths are the same. As shown in Figure 4.3, the

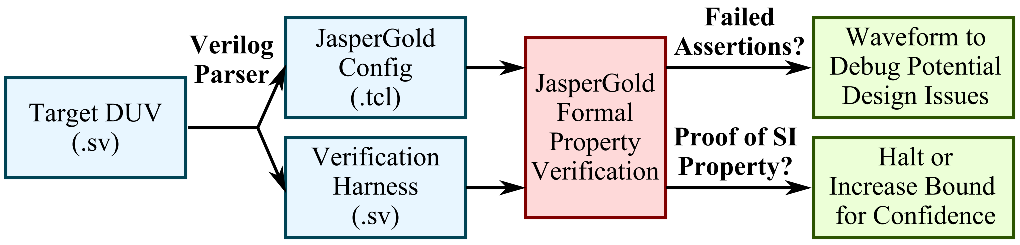


Figure 4.4: Workflow with BLEC Implementation – SI: stall invariant. The Verilog parser implements Figure 4.5 to generate the verification harness and necessary JasperGold configuration scripts. The generated verification harness contains assertions that JasperGold FPV proves or finds counterexamples to.

```
Require: $D$: The target design under verification.
Require: $N$: The depth of FIFOs in the verification harness.
Ensure: Verification harness $H$ with ports $H_p$ and connections $H_c$.
 1: function CONSTRUCTHARNESS($D, N$)
 2:     $H \leftarrow D_s \cup D_p$
 3:     $H_p, H_c \leftarrow \varnothing$
 4:     for all $i \in$ IngressLatencyInsensitiveInterface$(D)$ do
 5:         $H \leftarrow H \cup \text{N-FIFO}_{s,i} \cup \text{N-FIFO}_{p,i} \cup \text{Perturber}_i$
 6:         $H_p \leftarrow H_p \cup TopLI_i \cup \text{StallVariable}_i$
 7:         $H_c \leftarrow H_c \cup (TopLI_i, \text{N-FIFO}_{s,i}, D_s) \cup (TopLI_i, \text{N-FIFO}_{p,i}, \text{Perturber}_i, D_p)$
            $\cup (\text{StallVariable}_i, \text{Perturber}_i, )$
 8:     for all $e \in$ EgressLatencyInsensitiveInterface$(D)$ do
 9:         $H \leftarrow H \cup \text{N-FIFO}_{s,e} \cup \text{N-FIFO}_{p,e} \cup \text{Perturber}_e \cup \text{EqChecker}_e$
10:         $H_p \leftarrow H_p \cup \text{StallVariable}_e$
11:         $H_c \leftarrow H_c \cup (D_s, \text{N-FIFO}_{s,e}, \text{EqChecker}_e) \cup (D_p, \text{Perturber}_e, \text{N-FIFO}_{p,e}, \text{EqChecker}_e)$
            $\cup (\text{StallVariable}_e, \text{Perturber}_e, )$
```

Figure 4.5: Construction of the BLEC Verification Harness – s and p in subscripts indicate the module belongs to the strict/perturbed path; i and e in subscripts indicate the signal or module is associated with the ingress interface i or the egress interface e. $TopLI_i$: toplevel latency-insensitive interface that generates messages to the LI interface i. N-FIFO, EqChecker: N-element FIFOs, equivalence checkers as introduced in Section 4.3.1. H is a set of modules; H_p is a set of interfaces and ports; H_c is a set of tuples where neighboring tuple elements are connected and data flow through elements in ascending index order.

checker (in red) is interfaced to the two egress FIFOs. The equivalence checker only dequeues from the FIFOs and performs the equivalence check if both FIFOs are non-empty (i.e., `val` is asserted). The behaviors of the equivalence checker can be expressed as a property of an RTL module, which is boolean expressions between its signals. In Figure 4.3, I use the implication operator ($\implies$) to indicate that the equivalence check between `s_msg` and `p_msg` only happens when both `s_val` and `p_val` are true.

4.3.2 Construction of Verification Harness

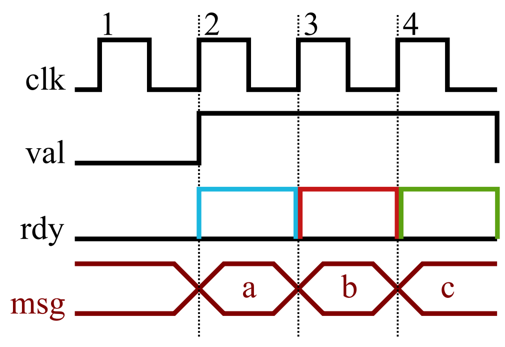
I demonstrate the verification harness of a DUV with one ingress and one egress LI interface in the previous section. In this section, I describe a systematic method to construct a verification harness for any latency-insensitive RTL modules.

Figure 4.5 shows the steps to construct a BLEC verification harness for any given latency-insensitive RTL DUV *D* with *N*-element FIFOs. The algorithm proceeds by enumerating all ingress and egress LI interfaces of *D* and adds modules, ports, and connections to the verification harness H. For each ingress LI interface *i* of *D*, the algorithm adds one toplevel LI interface to generate messages to *i*, one perturber to apply random stalls on *i*, and two N-element FIFOs; for each egress LI interface *e* of *D*, the algorithm similarly adds two FIFOs, one perturber, and one equivalence checker to compare the results of the strict and perturbed paths. The generated verification harness *H* may have multiple equivalence checkers and a violation of the stall invariant property is found if the FPV tool finds a failed assertion in any of these checkers.

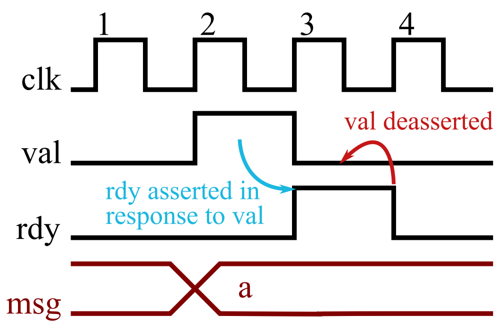
4.4 Implementation

In this section, I describe the implementation of the bounded latency equivalence checking technique in JasperGold, a state-of-the-art commercial formal property verification tool. Section 4.4.1 describes the specifications of the key properties in the verification harness in the SystemVerilog Assertion language. Section 4.4.2 discusses how I improve JasperGold's performance by incorporating proof acceleration modules into the verification harness.

To provide an overview of my BLEC implementation, Figure 4.4 shows an example workflow with our implementation of BLEC: I implement a Verilog parser that assumes the naming of ports in a latency-insensitive interface, which generates the verification harness and JasperGold configuration scripts using a templated approach; the JasperGold FPV tool either finds a counterexample to the stall invariant property (in which case the designers can debug the potential design issues with a waveform from the counterexample) or proves the stall invariant property with respect to the bounded FIFO size (in which case the verification engineer can stop or increase the FIFO size for higher confidence of the proof).



(a) Wrong Message



(b) Phantom Message

Figure 4.6: Bugs in Unconstrained Latency-Insensitive Interface – (a) `msg` may change while `val` is asserted; the downstream module may sample a wrong message depending on when `rdy` is asserted. (b) `val` may get de-asserted before a previously asserted `val` is acknowledged by `rdy`; if `rdy` is asserted in response to `val`, the downstream module may end up acknowledging an non-existent transaction.

4.4.1 Property Specification in SystemVerilog Assertion

In this section, I discuss how to specify some of the critical assumptions and properties in the SystemVerilog Assertion (SVA) language [iee17]. These are assumptions and properties are embedded in the verification harness and are generated by the Verilog parser in a templated fashion. While solving the formal property verification problem, JasperGold will assume the constraints to be true and try to find counterexamples to the asserted properties.

Constraints on Toplevel LI Interface – As discussed in Section 4.3.2, each ingress latency-insensitive interface in the target DUV will add a toplevel LI interface which streams messages to the ingress interface in the strict and perturbed DUV. However, an unconstrained LI interface of three ports (`val`, `rdy`, and `msg`) may not implement the correct LI handshake behaviors. Figure 4.6 shows two possible bugs when each of the three ports are allowed to change independently from each other. Figure 4.6 (a) shows a bug where the downstream module may accept a wrong message because `msg` is allowed to change while `val` is asserted. Figure 4.6 (b) demonstrates a bug where the downstream module tries to acknowledge a non-existent transaction because `val` gets deasserted before a previous `val` is acknowledged.

To ensure correct LI handshakes, I add the following assumption to the toplevel LI interface to constrain its behavior.

```
li_ifc_asms: assume property (
  @(posedge clk) disable iff (reset) (
    (val |-> rdy) or
    (val |=> ($stable(msg) & $stable(val))
                  s_until_with (val & rdy))
  )
);
```

In the above assumption, `|->` and `|=>` are implication operators in the SVA language that indicates the *consequent* (right hand side of the operator) is true if the *antecedent* (left hand side of the operator) is true [CDHK15]. The difference between `|->` and `|=>` is that `|->` requires the consequent to be true *at the same cycle* when the antecedent is true; `|=>` requires the consequent to be true *at the next cycle* after the antecedent becomes true. This assumption uses the `s_until_with` operator, which indicates that `msg` and `val` have to remain stable *at the same cycle* `val & rdy` becomes true. This assumption states that at any non-reset cycle, if `val` is asserted, then either `val` and `rdy` are asserted at the same cycle or `val` and `msg` remain stable until the transaction is acknowledged (`val & rdy`).

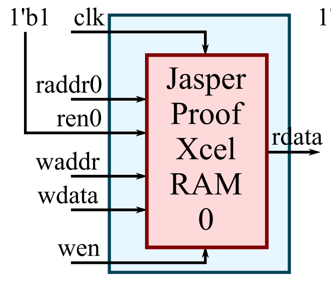
Properties of Equivalence Checkers – As mentioned in Section 4.3.1, the equivalence checker checks if the results from the strict and perturbed paths are the same when both egress FIFOs are not empty. I formalize this equivalence check into the following SVA assertion, which guards the check with an antecedent of both `val` signals asserted.

```
same_msg_ast: assert property (
  @(posedge clk) disable iff (reset) (
    (s_val & p_val) |-> (s_msg == p_msg)
  )
);
```

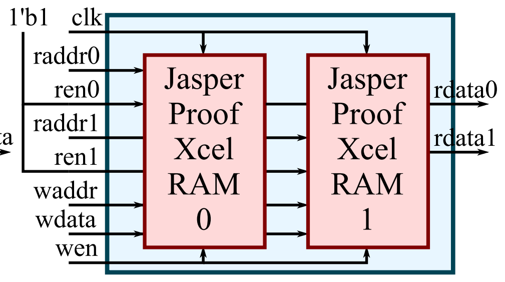
However, the `same_msg_ast` assertion alone is not sufficient to capture all violations to the stall invariant property. Consider one category of violations where the perturbed DUV fails to assert the `val` signal on the egress interface at all. In this case, the formal property verification tool considers this property to be *vacuously true* because the antecedent of the `same_msg_ast` assertion is false [CDHK15]. To detect this category of design bugs, I add the following SVA assertion.

```
same_vals_ast: assert property (
  @(posedge clk) disable iff (reset) (
    (s_val & ~p_val) |->
      s_eventually (s_val & p_val)
    and
    (~s_val & p_val) |->
      s_eventually (s_val & p_val)
  )
);
```

The assertion `same_vals_ast` has the same consequent among its two clauses which indicates that `s_val & p_val` will become true in some future cycle. The `s_eventually` operator provides a way to express that some event will happen after a finite but uncertain number of cycles. This assertion indicates that no matter which DUV (strict or perturbed) asserts the egress `val` signal, the other DUV will eventually assert its `val` as well.



(a) Register File 1r1w



(b) Register File 2r1w

Figure 4.7: Register Files with Integrated Proof Acceleration RAM – 1r1w: one read port and one write port; 2r1w: two read ports and one write port.

4.4.2 Proof Acceleration

To reduce the run time of the verification tool, the BLEC implementation incorporates JasperGold's proof acceleration modules into commonly used RTL modules. Proof acceleration modules are behavioral modules that have built-in behaviors in JasperGold and can be verified more efficiently than their manually implemented RTL counterparts. I specifically target the RTL RAM modules because (1) they generally contain a large number of states and the increasing number of states often strongly correlates with longer tool run time [CDHK15]; (2) the RTL RAM modules are widely reused across IPs including FIFOs, register files, caches, and behavioral memories.

Figure 4.7 shows how I integrate the RAM proof acceleration module into two kinds of register files. For the register file with one read port and one write port (1r1w), I wrap the proof acceleration RAM within the regular register file module and connect all ports accordingly. The read enable port on the proof acceleration RAM is driven by high voltage because the register file is read every cycle. For the register file with two read ports and one write port (2r1w), I duplicate the proof acceleration RAM within the register file module to support simultaneous reads. The write address and data are applied on both proof acceleration RAM. Since I do not modify the interface of the register files, our integration of proof acceleration modules reduces the run time of verification without changing the RTL code of the DUV.

4.5 Case Studies

In this section, I perform case studies on the following three RTL modules with my implementation of BLEC to demonstrate its effectiveness in detecting numerous design bugs: a latency-

Design	Flip-Flops	Gates	RTL Lines	FIFO Depth
PE	113	729	143	2
GCD	66	655	490	2
Proc.	5983	86830	4898	2

Table 4.1: RTL Modules and BLEC Parameters Used in Case Studies – PE: the latency-insensitive processing element; GCD: the greatest common divisor unit; Proc.: the RISC-V processor

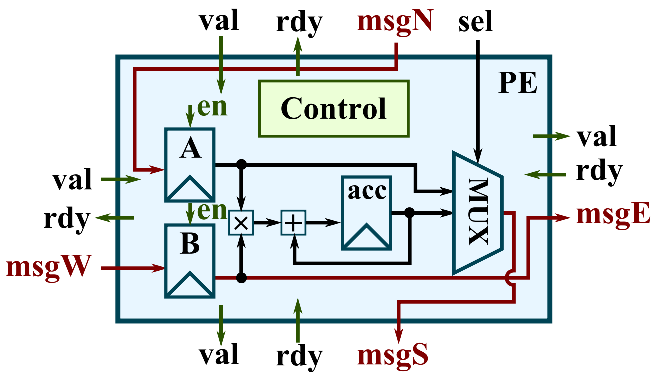


Figure 4.8: Latency-Insensitive PE – N, W: ingress interface on the north and west side of PE; E, S: egress interface on the east and south side of PE. acc: accumulation register.

insensitive processing element (PE), a greatest common divisor (GCD) unit, and a RISC-V processor. I use JasperGold FPV 2023.03 as the formal verification tool and run the case studies on a commodity server with 72 cores of Intel Xeon E7-8867 v4 CPU and 256 GiB of main memory. Table 4.1 shows the number of flip-flops, gates, the lines of RTL code, and the BLEC parameters used in our case studies.

4.5.1 The Latency-Insensitive Processing Element

The first case study is on a latency-insensitive processing element (PE) RTL module which is intended to be used as sub-modules of a latency-insensitive systolic array. Figure 4.8 shows the architecture of the PE module. The PE takes input from two LI interfaces at the north and west side and produces output to the east and south LI interfaces. The PE also performs multiply-accumulation and stores the sum into its internal accumulation register. The PE also forwards the west message to the east side. Depending on the selection input signal, the PE either forwards the north message or the accumulation result to the south side.

Bug: Incorrect Ingress Ready Condition – I examine a PE bug discovered from the commit history of an in-house systolic array (performing matrix multiplication) git repository. According to the commit history, the designers created wrong control logic for the ingress `rdy` signals: `rdy` from the east egress interface was simply bypassed to the west ingress interface and `rdy` from the south egress interface was bypassed to the north ingress interface. This bug created an incorrect ingress ready condition (ingress ready should be true only if both `rdy` from the east and the south side interface are true) which escaped the designer's unit test because the behavioral downstream module of PE always applies egress stalls at the same cycle.

My implementation of BLEC detects this bug in under ten seconds. JasperGold finds a 5-cycle counterexample to the `same_msg_ast` assertion in the equivalence checker: the strict DUV in the counterexample registers `msgN` and `msgW` at the same cycle; the perturbed DUV has one cycle of egress stall on the east interface, which causes `msgW` to be registered one cycle later than `msgN`. This difference in the timing of registering ingress messages eventually leads to different results from the strict path and the perturbed path.

The PE designer initially identified this bug with a manually crafted test case which captures the exact timing of egress stalls required to trigger this bug. With the waveform of this counterexample derived from BLEC, the PE designer is able to identify and fix the root cause of the failed assertion much faster without manipulating the timings of egress stalls.

Bounded Proof: PE is Stall Invariant – After fixing the ingress ready condition bug, I also leverage BLEC to generate a bounded proof that the PE module is stall invariant. I observe that JasperGold is not able to converge on the PE design because the single-cycle multiplier (two 32-bit inputs, one 32-bit output) in the PE datapath significantly increases the complexity of verification. To help the FPV tool converge, I leverage the fact that the precise multiplier functionality is not required in BLEC. Therefore, we can replace the complex multiplier logic with a much simpler bit-wise XOR operation to improve converge time. Since the LI handshake logic does not depend on the multiply-accumulate result, performing this replacement does not affect the equivalence properties BLEC tries to prove. After replacing the single-cycle multiplier with bit-wise XOR gates, JasperGold is able to prove both the `same_msg_ast` assertion and the `same_vals_ast` assertion within 1.5 hours.

4.5.2 The Greatest Common Divisor Unit

The second case study design is a greatest common divisor (GCD) unit which computes the GCD of two input 32-bit integers using a subtraction-based Euclidean algorithm. Figure 4.9 shows

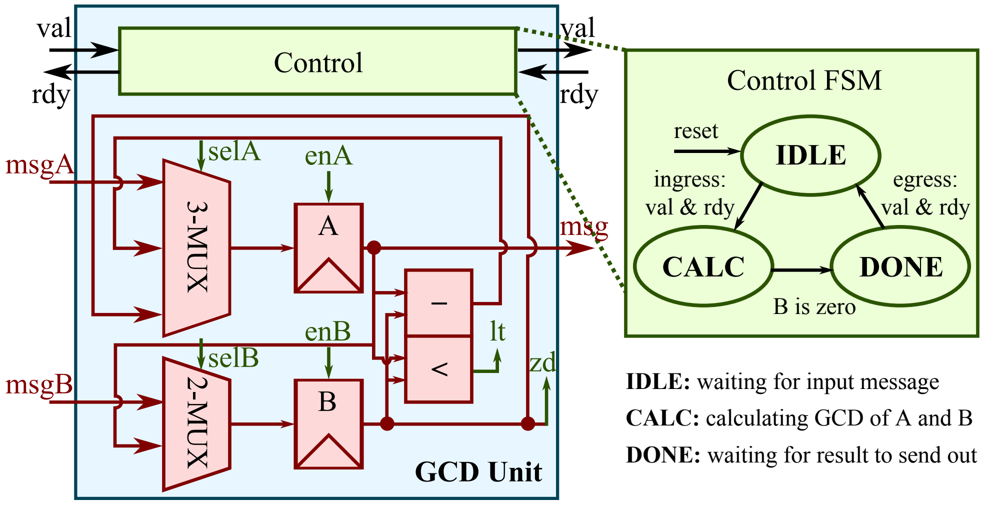


Figure 4.9: GCD Unit – lt: if A is less than B; zd: if B is zero.

the RTL GCD unit and the finite state machine (FSM) in its control unit. The GCD unit has one ingress LI interface to stream in the two input integers within a single bundle and one egress LI interface to stream out the result. In this case study, I examine and detect two bugs with my BLEC implementation and also prove that the correct GCD unit is stall invariant.

Bug: Unconditional Transition from CALC to DONE – The first bug I investigate is when the control FSM transits unconditionally from the DONE state to the IDLE state. With this bug, the GCD unit may not send out the result correctly if the downstream module is not ready in the cycle GCD unit is in the DONE state. However, this bug is only observed if there is more than one cycle of stalls on the egress interface, which helps the bug escape some simulation-based testing that assumes no egress stalls on the DUV.

My implementation of BLEC detects this bug in under one minute. JasperGold finds a 7-cycle counterexample to the `same_msg_ast` assertion in the equivalence checker: the toplevel LI interface generates two messages into the two instances of DUV; the egress perturber applies one cycle of stall on the egress interface, which causes the first result of the perturbed DUV to drop; the equivalence checker therefore finds the first result from the strict DUV and the second result from the perturbed DUV to be different, triggering a failed assertion. Verification engineers can deduce from the counterexample waveform that the DUV has different behaviors under different stall conditions, which helps debugging.

Bug: Wrong Transition Condition from CALC to DONE – The second bug creates a wrong transition condition where the FSM only transits to DONE if the egress interface is ready and transits to IDLE otherwise. With this bug, the GCD unit will function correctly if there is no egress stalls; but the DUV will not generate valid output messages if there is egress stalls. Similar to the unconditional transition bug, this bug can escape simple simulation tests that assume no egress stalls.

BLEC detects this bug in under one minute. JasperGold identifies that the `same_msgs_ast` assertion vacuously passes (i.e., the antecedent condition is unreachable) because under this bug the strict and the perturbed DUV cannot generate a valid output message at the same cycle (perturbed DUV has at least one cycle egress stall). But JasperGold does find a counterexample of infinite length to the `same_vals_ast` assertion: the toplevel LI interface generates two input messages and the strict DUV produces two output messages before becoming idle; the perturbed DUV does not generate any output and remains idle for the rest of the trace. Similar to the unconditional transition bug, verification engineers can leverage the counterexample to debug the design issue.

Bounded Proof: GCD Unit is Stall Invariant – I also leverage BLEC to generate a bounded proof that the GCD unit without bugs is stall invariant. I make two minor changes to the GCD unit design to help the FPV tool converge without undermining the stall invariant proof.

First, I make the observation that for large 32-bit inputs, the GCD unit may spend a significant number of cycles in the `CALC` state to compute the greatest common divisor using the subtraction-based Euclidean algorithm. Therefore, formally verifying the complete 32-bit GCD unit design is intractable because the FPV tool has to examine all 32-bit input pairs and step through the Euclidean algorithm calculation to find potential violations of the stall invariant property. To help the FPV tool converge on the GCD unit design, I modify the state transition condition from state `CALC` to `DONE` to expedite the GCD computation process. As shown in Figure 4.9, the control FSM in the GCD unit transits from `CALC` to `DONE` when the registered B value is zero. I remove this condition and make the transition to the `DONE` state unconditional. This change effectively reduces the number of cycles required to compute the greatest common divisor.

Second, I apply a similar change to the bitwise-XOR operation in the latency-insensitive PE to avoid reasoning about complex computations in the GCD unit datapath. As shown in Figure 4.9, the datapath of the GCD unit includes a subtraction operation between the registered A and B values. I replace the subtraction operation with a bitwise-AND operation so that the FPV tool can reason about simpler bitwise-AND operations instead of a 32-bit subtraction.

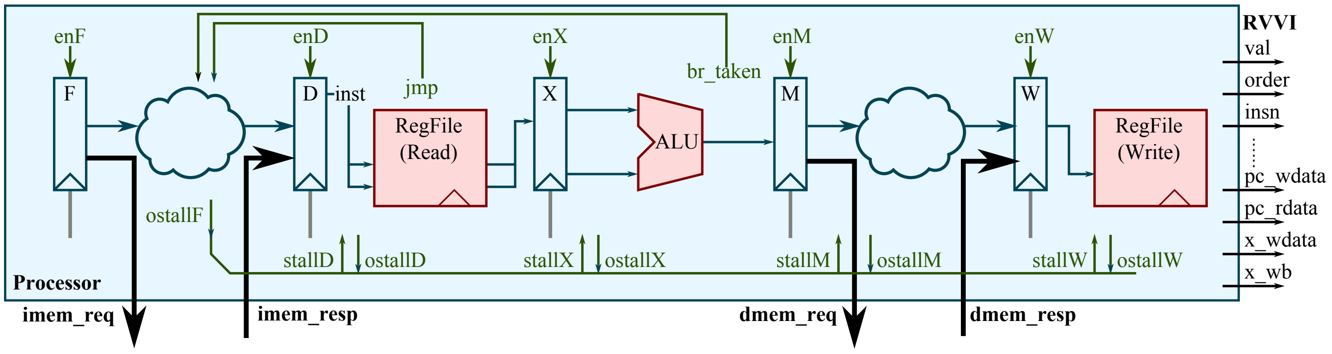


Figure 4.10: Pipelined Processor – RVVI: RISC-V verification interface. `ostall`: if this pipeline stage is originating an event that stalls (`stall`) this stage and all stages after; `jmp`: squash stage F if the decoded instruction is a jump instruction; `br_taken`: squash stage F and D if the current instruction is a branch and the branch is taken. Each thick black arrow represents one latency-insensitive val-rdy interface.

Both of the above changes do not undermine the stall invariant proof because the changes only affect logic outside of the GCD unit's handshake control logic. After applying the above two changes, JasperGold is able to prove both the `same_msg_ast` assertion and the `same_vals_ast` assertion in the GCD verification harness within 20 minutes.

4.5.3 The Pipelined RISC-V Processor

The final case study design is a five-stage pipelined RISC-V processor. Figure 4.10 shows the simplified datapath and control diagram of the pipelined processor used in this case study. The target processor RTL module communicates to the instruction memory and data memory through four memory interfaces: memory requests are transferred through the `imem_req` and `dmem_req` interfaces, and memory responses come back through the `imem_resp` and `dmem_resp` interfaces. Internally, the processor has five pipeline stages: fetch (F), decode (D), execution (X), memory (M), and write-back (W). The processor reads the register file at stage D and writes back to the register file at stage W. The processor has a simple branch predictor that always predicts not taken. In the event of a branch mis-prediction (`jmp` or `br_taken`), the processor squashes stage F (if a jump instruction) or stage F and D (if a branch instruction) to discard invalid states. Each pipeline stage may also *originate a stall* (`ostall` signals) in the event of hazards or when memory responses have not arrived, which stalls all stages after the originating stage.

The BLEC verification harness of the processor is different from that of the previous case studies. I make the observation that the memory request latency-insensitive interfaces of the processor are inherently stall variant: branch instructions may squash earlier memory requests and therefore

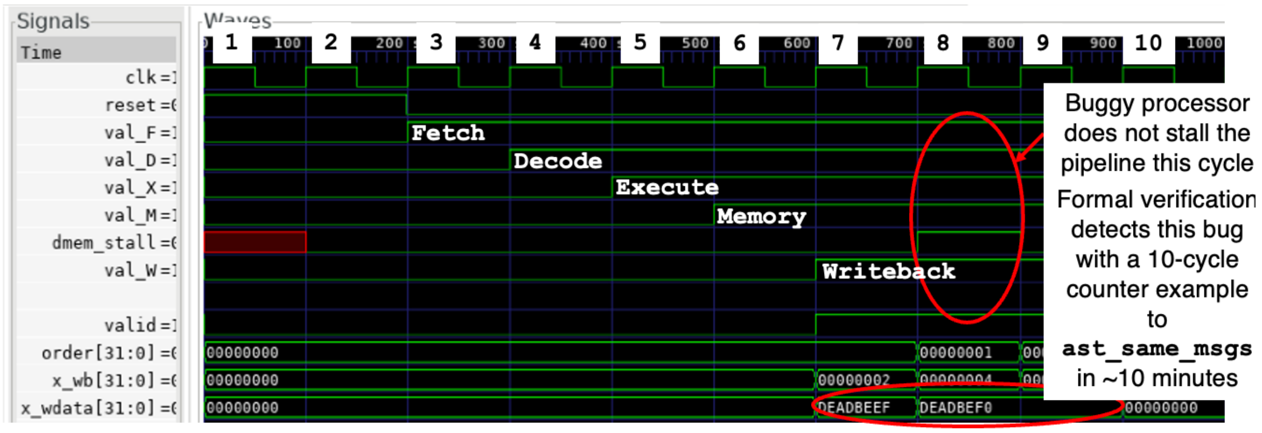


Figure 4.11: Pipelined Processor Bug – This figure shows the buggy behavior of the pipelined processor where the M stage is not correctly stalled when a memory response is pending. The data memory response is pending (`dmem_stall` is high) on cycle 8 but stage W does not stall, leading to incorrect data written back to the register file.

memory response stalls can lead to different informative memory requests. I choose to implement equivalence checking on the RISC-V verification interface [rvv23], which exposes the states of the processor in instruction commit order and is guaranteed to be stall invariant regardless of instruction and data memory stalls. The right side of Figure 4.10 shows some of the exposed processor states used in the case study. `val` is the latency-insensitive valid signal which indicates if the output signals are valid at a cycle; `order` is a counter that keeps track of the number of committed instructions; `insn` is the 32-bit instruction; `pc_rdata` is the PC register value for the current instruction and `pc_wdata` is the PC register value for the immediate next instruction; `x_wb` is a bit vector that tracks which architectural register is written; `x_wdata` is the content of all architectural registers at instruction commit.

To provide instruction and data memory responses, I also include a behavioral memory backed by the JasperGold proof accelerator described in Section 4.4.2. To retain generality of my method, I do not fill the behavioral memory and instead allow the FPV tool to the memory response message as free variables. To reduce the FPV tool time, I also add the assumption to the processor decode stage that all instructions at the decode stage is a valid RISC-V instruction.

Bug: Not Stalling Pipeline When Memory Response is Pending – I examine a bug where the processor does not correctly stall its pipeline when a data memory response is pending (that is, stage W proceed to write back invalid data from stage M). For the processor in Figure 4.10, this bug is equivalent to clamping the `ostall_M` signal in stage M to low.

JasperGold finds a 10-cycle counterexample to the `same_msg_ast` property on the RISC-V Verification Interface (RVVI) in about 10 minutes. This counterexample includes 3 valid RISC-V load word instructions that fetch words from the data memory and write them back to the register file. Figure 4.11 shows the waveform of key pipeline control signals of the bugged processor. On cycle 8, the response side of the data memory is stalled (indicated by `dmem_stall` being high). The correct processor would stall stage M and perform no write-backs at stage W on cycle 9. The strict processor without stalls on the data memory response interface also writes back the correct messages because the bug only manifests with stalls in the M stage. However, the buggy processor would still perform a write-back at stage W on cycle 9 with *an incorrect data word*, leading to inconsistent RVVI informative events when compared to the strict processor.

Attempted Bounded Proof: Processor is Stall Invariant – Despite being able to find violations of the processor's stall invariant property within a relatively short period of time, in the case study JasperGold cannot establish a proof of the equivalence properties in BLEC verification harness within a reasonable amount of time (48 hours wall time). The main reason for the extended time to converge is the processor register file and the instruction and data memory. The target RISC-V processor includes a register file of 32 32-bit entries, and the instruction and data memory both have 64 32-bit entires (I choose a small number of memory entries to reduce converge time). These RTL memory modules represent an enormous state space, which the FPV tool has to exhaustively search through to eventually generate a bounded proof of the stall invariant property.

I have attempted several methods to reduce the processor complexity by introducing extra constraints. For example, I add assumptions that certain RISC-V instructions will not appear to reduce the decoder complexity; I remove the support for several arithmetic operations in the ALU; I also reduce the bitwidth of the long data bus (`x_wdata`) in the RVVI to shrink the state space the FPV tool needs to search through. Future research may need to further reduce the state space of the verification harness to eventually establish a bounded proof of the stall invariant property.

4.5.4 Discussions

Based on my experiences performing the above case studies, I observe that BLEC is effective at detecting bugs in the given latency-insensitive RTL modules. The FPV tool (JasperGold) usually takes a reasonably short period of time to discover a counterexample to the stall invariant property in the original RTL module. As a concrete example, in the RISC-V processor case study, Jasper-

Gold discovers a counterexample of 27 cycles in the processor RTL with uninitialized behavioral instruction and data memory in 20 minutes.

However, it usually takes the FPV tools significantly longer time to achieve a bounded proof of stall invariant on the given RTL module, and some manual changes are necessary to help the FPV tool converge faster. Fortunately, BLEC is compatible with many design changes that can significantly reduce tool converge time. Most of these changes reduce the complexity of the target DUV's datapath by replacing complex computations (typically with a large number of gates) with simpler computations. Since the latency-insensitive handshake logic of most DUVs do not depend on the exact values of these computations, those changes generally do not undermine the stall invariant proof. Concrete examples of those changes include replacing the multiplication logic with bitwise-XOR gates (PE case study) and replacing the subtraction logic with bitwise-AND gates (GCD unit case study).

For agile hardware designers, BLEC can be smoothly integrated into their agile workflow because the algorithm shown in Figure 4.5 automatically constructs the verification harness for most latency-insensitive DUVs. As part of the continuous integration pipeline, BLEC can be kept running in the background to find potential violations of the stall invariant property. However, hardware generators that instantiate or interact with memory modules may face significant increase in run time because FPV tools converge much slower on memory modules. Agile designers may need to focus on applying BLEC to the unit tests of small latency-insensitive RTL modules to avoid prolonged FPV tool time.

4.6 Related Work

Bounded model checking [CBRZ01] is a formal verification technique which verifies if a given transition system obeys the specification of its intended behaviors. The industry has adopted bounded model checking based formal verification techniques to verify the functional correctness of large RTL designs [BLM01, CFF$^+$01, BCRZ99, AK95]. Both these existing works and my work leverage bounded model checking based formal verification methods to prove or find counterexamples to the intended behaviors of RTL modules. However, there are two major differences between the above existing works and my work. First, existing works mainly focus on verifying the functional correctness of the RTL modules and my work focuses on finding violations of the stall invariant property. Second, to achieve a detailed and unambiguous specification, the above existing works mainly reply on manual specifications of intended behaviors of an RTL module. This requires intimate knowledge of both the design's functionalities and the specification language, which limits formal methods' accessibility to a relatively small audience. In contrast, my proposal democratizes the formal verification techniques by encapsulating details of the specification into verification modules.

Carloni et al. propose a correct-by-construction methodology to develop latency-insensitive designs using a helper modules including channels, relay stations, and shells [CMSSV99]. *Shells* are wrapper modules around the target DUV to enable correct-by-construction latency-insensitive communications with other LI channels. The authors claim that a shell can be automatically synthesized from a given DUV, which reduces the time required to implement a correct latency-insensitive RTL module. In the face of stalling events, the shell stalls the wrapped DUV instance through clock gating to preserve its internal states and only allows state changes when all input messages have become valid. Comparing to my work, Carloni et al.'s proposal represents an orthogonal correct-by-construction solution to the verification challenge of latency-insensitive designs.

Researchers have also explored properties similar to the stall invariant property and applied it in other contexts. Dai et al. propose to leverage formal verification techniques to validate high-level synthesis (HLS) results based on the latency-equivalence of the design under different inputs [DKR$^+$21]. Piccolboni et al. propose to formally verify the latency equivalence of different high-level synthesis results to achieve high confidence in HLS results. Piccolboni's proposal, KAIROS, assumes an incremental modification workflow and verifies if the result of each synthesis step produces results that are latency equivalent to the reference module. Similar to my proposal, Dai et al. and Piccolboni et al. also construct a verification harness with latency-insensitive input manipulation logic. However, both my work and their proposals have different focuses and represent orthogonal efforts on tackling HLS verification issues and a more traditional ASIC/FPGA prototyping verification challenges. Suhaib et al. propose to validate LI components by verifying the latency-equivalence between a LI component and its synchronous counterpart, both of which are described using a verification modeling language [SMBS06]. My work focuses on verifying the stall invariant property of LI components modeled at RTL, which includes most of the hardware modules used in ASIC and FPGA prototyping. Wijayasekara investigates a similar property to the stall invariant property in the context of asynchronous circuits and tackles the verification challenges in the asynchronous context [Wij16], where as my work focuses on the correctness of digital LI components.

4.7 Conclusions

Despite its success in enabling hardware standard libraries and numerous network-on-chip IPs, latency-insensitive protocols have imposed a unique verification challenges in instance composition where existing simulation-based dynamic verification techniques require significant efforts to build test suites that cover a large number of stall conditions. In this chapter, I propose a formal verification methodology to address the verification challenge of latency-insensitive RTL modules. I introduce the stall invariant property of latency-insensitive RTL modules and make the observation that most bugs in LI modules are violations of the stall invariant property. I propose bounded latency equivalence checking, which constructs a verification harness accepted by a formal property verification tool to find inconsistent latency-insensitive behaviors under different stall conditions. I implement the proposed BLEC technique with a state-of-the-art commercial formal verification tool and perform three case studies to evaluate its effectiveness. The case studies demonstrate that BLEC can find all injected bugs within relatively short period of time. The case studies also find existing commercial formal verification tools can provide a bounded proof of the stall invariant property on many manually simplified RTL modules.

CHAPTER 5
TRANSLATION-IMPORT: ADDRESSING VERIFICATION CHALLENGES IN CO-SIMULATION

To better support sophisticated and highly-parametrized hardware generators, modern hardware description languages (HDLs) are often embedded in a general-purpose host programming language. To maximize development velocity, agile hardware designers generally model the target hardware and create test benches in the host language before generating Verilog RTL for ASIC or FPGA prototyping. Ideally, agile hardware designers should be able to run a comprehensive test suite on both the host language hardware model (the native model) *and* the generated Verilog RTL model (the external model) by reusing the same test bench in the host language. Unfortunately, most of the existing agile hardware frameworks lack the support for this seamless co-simulation between the host language and the prototyping language (such as Verilog).

In this chapter, I present translation-import, a systematic approach that enables seamless co-simulation between the PyMTL3 HDL and multiple prototyping languages. Translation-import consists of two highly extensible components: the translation framework and the import pass. The translation framework is built on top of a register-transfer level intermediate representation (RTLIR), which is a canonical in-memory representation of a PyMTL3 native model. RTLIR facilitates translation by providing a unified view of numerous hardware constructs and performing comprehensive type checking on the IR. Based on RTLIR, the translation framework provides user-friendly APIs that support the one-to-one mapping from a PyMTL3 construct to a prototyping language construct, which can be easily extended to new prototyping languages. On the other hand, the import pass achieves seamless co-simulation by exposing an external model in PyMTL3 as an interactive model that can be incorporated into native PyMTL3 test benches. To make external models interactive, the import pass creates a wrapper, a series of C functions that manage the status and ports of the model. The import pass compiles the target external model plus the wrapper into a shared library with a standard C compiler and a simulator of the target prototyping language. The shared library can then be dynamically linked into the host PyMTL3 process for simulation. I also present a case study of an ultra-elastic coarse-grain reconfigurable array (UE-

CGRA) to demonstrate how translation-import improves functional verification productivity in an agile workflow.

5.1 Introduction

Hardware description languages have profound impacts on the IP and verification reuse in hardware developments. Traditional HDLs generally focus on the structural and behavioral modeling of hardware and lack the expressiveness of most general-purpose programming languages that enables sophisticated generators or generic test benches. To overcome this limitation, modern HDLs for agile hardware are typically embedded in a programming language known as *the host language*. These *embedded HDLs* often provide a collection of modeling primitives and utilities implemented in the host language to support structural and behavioral modeling, parametrized hardware generators, and reusable test benches. Embedded HDLs are ubiquitous in agile hardware and significantly improve the IP and verification reuse because of the improved expressiveness of their host languages over traditional HDLs.

Unfortunately, most of the existing embedded HDLs face a productivity challenge when agile hardware designers try to verify the target hardware using simulations. To maintain a high development velocity, designer want to implement both the target hardware and the test benches in the host language (known as the *native design* and the *native test benches*) and iterate until the design passes a reasonable number of unit and integration tests. After the designer has high confidence about the design's correctness, a *prototypable design* can be generated from the native design which can be used for ASIC/FPGA prototyping purposes. Ideally, designers should be able to reuse the native test benches to verify the correctness of the prototypable design. However, existing HDLs either do not support the above productive verification workflow or have limited support of co-simulation between the native test bench and the generated design. I refer to this dilemma of agile hardware designers as a *seamless co-simulation verification challenge*.

5.1.1 A Taxonomy of Simulation-Based Verification in HDLs

To better understand the seamless co-simulation verification challenge, I present a taxonomy of simulation-based verification strategies. Figure 5.1 shows four possible simulation strategies with embedded HDLs, where host language indicates the host programming language of the HDL

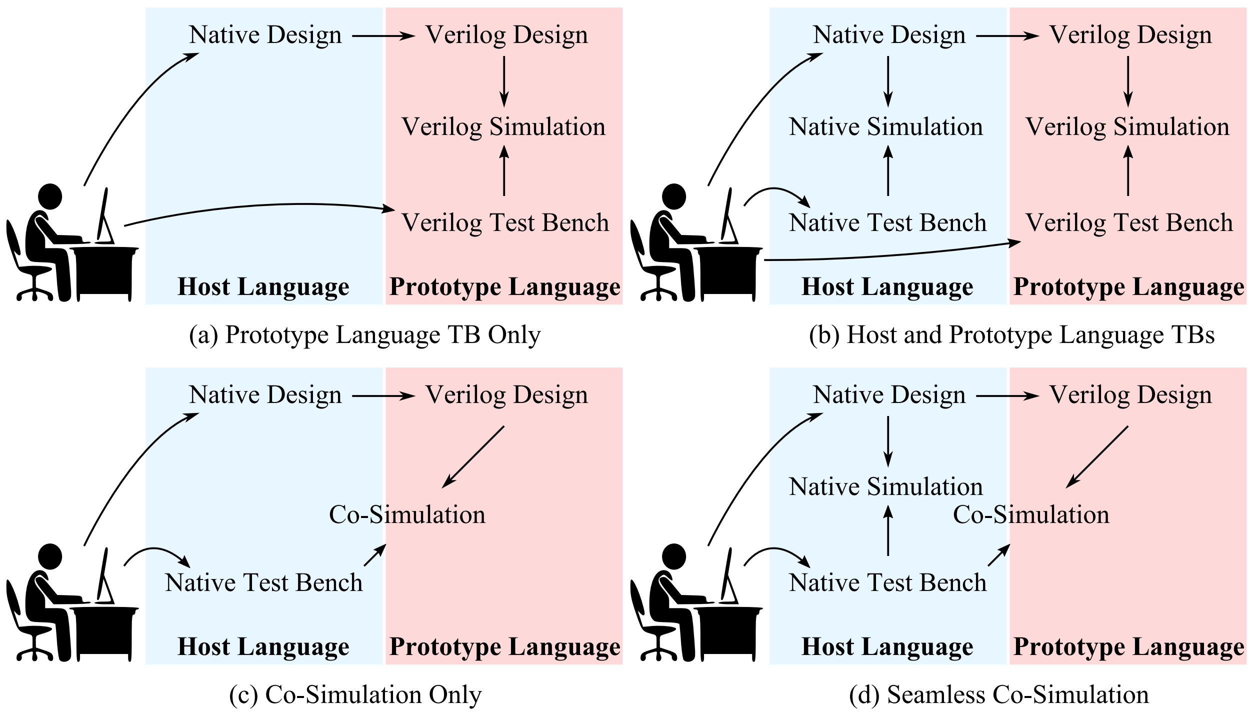
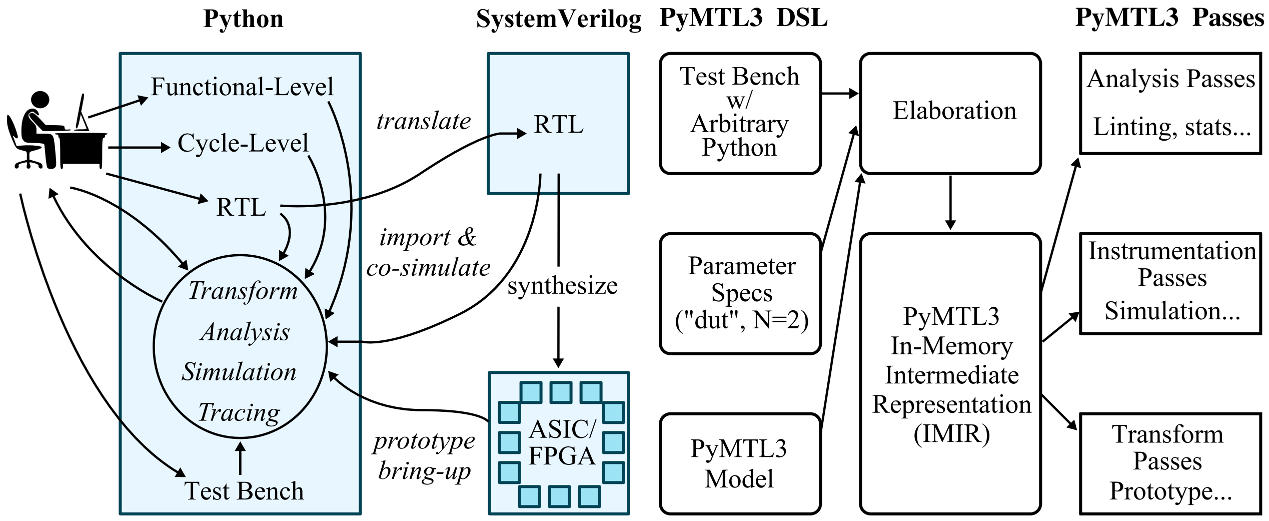


Figure 5.1: A Taxonomy of Simulation-Based Verification – (a) only creates a test bench in the prototype language which fails to leverage the host language for productive testing; (b) creates two test benches, which takes too much verification time; (c) creates a native test bench but fails to iterate natively: only co-simulates the generated Verilog (d) seamless co-simulation: iterate natively before co-simulating the generated Verilog.

and prototype language refers to the HDL used for ASIC/FPGA prototyping (typically Verilog). Figure 5.1(a) corresponds to a Verilog-only simulation strategy. Under this strategy, agile hardware designers only use the embedded HDL to create parametrized generators and fail to leverage the productive host language for test benches, which is not ideal for productive verification. Figure 5.1(b) performs both native and Verilog simulations. However, this strategy requires the implementation of at least one native test bench and one Verilog test bench, which significantly increases the workload of verification. Figure 5.1(c) represents a co-simulation only strategy. Under this strategy, agile hardware designers have to generate a Verilog design before co-simulating it with the native test bench. This strategy does not support native simulation, which is critical to the iterative improvement of the native design in the host language environment. Figure 5.1(d) presents the seamless co-simulation strategy, which supports both native simulation and co-simulation with Verilog. Comparing to the co-simulation only strategy, seamless co-simulation enables designers to productively iterate on the native design using native simulation until a reasonably large number of test cases have passed. Seamless co-simulation is the ideal simulation strategy in my taxonomy because it allows rapid iterations of the native design without sacrificing co-simulation capabilities.



(a) An Agile Hardware Workflow in PyMTL3 (b) An Overview of the PyMTL3 Framework

Figure 5.2: PyMTL3 Workflow – Hardware designers implement functional-level, cycle-level, or RTL models using PyMTL3 modeling primitives. Designers can also use arbitrary Python language constructs for test benches. PyMTL3 passes (in *italic*) targets hardware models and test benches to enable features such as translation to SystemVerilog, co-simulation, and prototype bring-up. Figure adapted from [JPOB20].

5.1.2 PyMTL3 Background

As a simulation-based verification strategy, seamless co-simulation is orthogonal to the choice of HDLs and can be implemented in almost any embedded HDL. However, in this chapter, I focus on translation-import, a seamless co-simulation solution implemented in the PyMTL3 embedded HDL. This section provides the relevant background about PyMTL3, an open-source Python framework for agile hardware, which lays the foundation of the remaining sections in this chapter.

PyMTL3 is an open-source Python framework for hardware modeling, generation, simulation, and verification [JPOB20]. It supports multi-level modeling including functional-level, cycle-level, and RTL. Figure 5.2(a) shows an agile hardware workflow using PyMTL3. Hardware designers implement the target model using a set of modeling primitives. Designers are also allowed to use arbitrary Python code in their test benches. PyMTL3 supports translation from the RTL PyMTL3 model into SystemVerilog, which can then used to drive an ASIC/FPGA flow or co-simulate with the native PyMTL3 test benches. To facilitate prototype bring-up, PyMTL3 also provides passes that bridges the native test benches and the ASIC/FPGA prototype.

Figure 5.2(b) provides an overview of the PyMTL3 framework. For hardware designers, PyMTL3 provides the PyMTL3 domain-specific language (DSL), which includes modeling primitives to support hardware models, parameter specifications, and test benches. *Elaboration* is a process which converts the target hardware models, the designer-specified parameters, and the given test bench into a hierarchy of model instances. The representation of the elaborated hierarchy is known as the in-memory intermediate representation (IMIR), which provides query and modification APIs to allow post-elaboration operations on the instance hierarchy.

PyMTL3 *passes* provide a systematic approach to organizing and applying post-elaboration utilities, including translation-import. As shown in Figure 5.2(b), there are three broad categories of PyMTL3 passes: (1) analysis passes, which only collect information and statistics from the elaborated hierarchy. Utilities such as linters and hierarchy statistics collectors can be conveniently implemented as analysis passes. (2) instrumentation passes, which attach additional meta-data to the target hierarchy of instances. Instrumentation passes do not modify the given instances, and important utilities including simulation, tracing, and translation belong to the instrumentation pass category. (3) transform passes, which may modify the given elaborated hierarchy such as modifying connections and adding new components to the hierarchy. An example of transform passes is the ASIC/FPGA prototype passes, which wrap the target hierarchy within a component that communicates to the PyMTL3 test benches. The translation-import mechanism is implemented as a translation pass generator and an import pass in the PyMTL3 framework.

5.2 Translation-Import Mechanism in PyMTL3

In this section, I represent the design and implementation details of the translation-import mechanism in PyMTL3. To provide an overview of the translation-import implementation, Figure 5.3 shows how translation and import passes converts a pure PyMTL3 model into a SystemVerilog model that can be co-simulated with PyMTL3 test benches. As is shown in the figure, both the translation pass and the import pass depend on register-transfer level intermediate representation (RTLIR), which serves as a canonical representation of synthesizable PyMTL3 model instances. The translation pass takes the RTLIR representation of a model as input and generates the source code of a backend representation (typically SystemVerilog but can be other HDLs). The import pass takes the RTLIR model and the generated backend source code and creates both a C wrapper and a PyMTL3 wrapper for the target; the C wrapper is then compiled into a shared library together with the Verilated RTL and then dynamically linked through CFFI (C Foreign Function Interface) into the PyMTL3 wrapper.

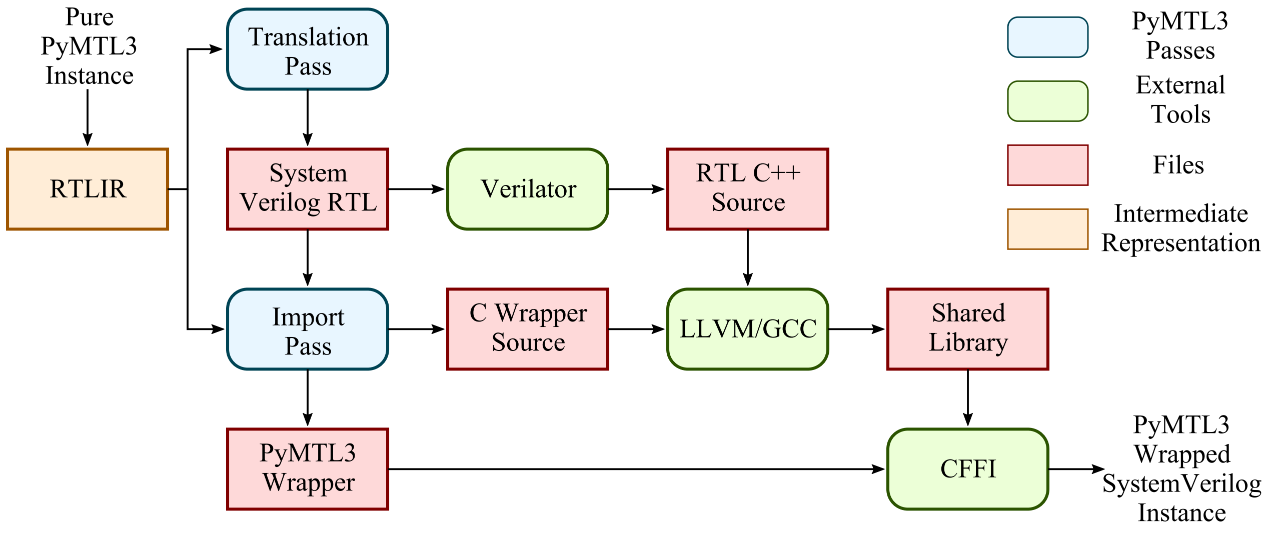


Figure 5.3: PyMTL3 Translation and Import Passes – RTLIR: register-transfer level intermediate representation. Verilator is an open-source simulator which takes Verilog/SystemVerilog RTL and compiles it into C++ source code. LLVM/GCC represents standard C/C++ compilers that can compile the RTL C++ source code and C wrapper into a shared library. CFFI, which stands for C Foreign Function Interface, is a Python package that facilitates loading dynamic shared libraries.

The remaining section is organized as follows: Section 5.2.1 presents the design and implementation of RTLIR. Section 5.2.2 describes the translation framework, which is an extensible translation pass generator based on RTLIR. Section 5.2.3 shows the design and implementation of the import pass.

5.2.1 The Register-Transfer Level Intermediate Representation

RTLIR is an intermediate representation for synthesizable PyMTL3 RTL models. Comparing to PyMTL3's in-memory representation (IMIR), RTLIR provides a canonical representation to a focused subset of the PyMTL3 DSL and enables self-contained sanity checks through type checks. RTLIR consists of three main components: the RTLIR types, which represent the types used in synthesizable RTL modeling; the structural RTLIR, which represents the structural modeling aspect of RTL models; and the behavioral RTLIR, which represents the behavioral modeling in the form of an update block in PyMTL3.

RTLIR Types – RTLIR types carry important information about the target RTL model and are a critical component of RTLIR. Figure 5.4(a) shows the core RTLIR types definitions, which include three main kinds of types: data types, interface types, and structural modeling types.

```
module RTLIR_Types
{
  -- Data Types
  dtype = Vector(int nbits)
        | Struct(string* fields, dtype* types)
        | PackedArray(dtype* types)

  -- Interface Types
  itype = Port(direction dir, dtype type)
        | Interface(string* names, itype* types)
        | IfcArray(itype* types)

  -- Structural Modeling Types
  type = Signal(dtype type)
       | Component(string name)
       | Array(type *types)
}
```

(a) Core RTLIR Types

```
1  @bitstruct
2  class Foo:
3    x : Bits8
4    y : Bar
5
6  @bitstruct
7  class Bar:
8    x : [Bits4 for _ in range(2)]
9    y : Bits4
10
11 class Woo(Component):
12   def construct(s):
13     s.in_ = InPort(Foo)
14     s.out = OutPort(Bits8)
15
16     s.out[0:4] //= s.in_.y.x[0]
17     s.out[4:8] //= s.in_.y.x[1]
```

(b) Example

Figure 5.4: Core RTLIR Types and an Example Model – (a) RTLIR types include hardware data types, which are commonly used to specify the data type of a signal, interface types, which are the types of potentially nested RTL interfaces, and structural modeling types, which are the types of primitives such as signals and components. (b) Woo has an input port of a nested struct data type.

Data types specify the kind of data that may pass through a port or a wire. They are recursively defined over three cases: vector types, which are fixed width vectors; struct types, which may have fields of arbitrarily nested data types; packed array types, which is a list of any data types. For example, in Figure 5.4(b), the input port s.in_ has a struct data type with a nested field y. Line 16-17 show how to create connections between the output port and different entries in a packed array data type. Interface types are the types of interfaces of an RTL model. They account for the names of ports and nested interfaces and their corresponding interfaces types. As shown in Figure 5.4(b), both s.in_ and s.out have interface types of struct and vector data type. Finally, the structural modeling types are used for non-interface attributes of an RTL model. For example, a model may instantiate a sub-component, and it has Component type with the name of component class. Another commonly used structural modeling type is Signal, which describes the types of wires. It is also possible to create a new structural modeling type by organizing some types into a list, as is indicated by the Array case shown in Figure 5.4.

Structural RTLIR – Structural RTLIR provides a representation to structural modeling in an RTL model. Figure 5.5(a) shows the definition of the *signal expression* (or sexpr), the core structural RTLIR. Signal expressions describe a signal used in structural modeling, which typically involves signals that are connected. Signal expressions always have a root operand of type CurrentComponent, which represents the model that owns the signal. The other cases in the definition of sexpr represent operations that may yield a new signal. For example, Attribute

```
module Structural_RTLIR
{
  sexpr = CurrentComponent(string name)
        | Attribute(sexpr base, string attr)
        | Index(sexpr base, sexpr index)
        | Slice(sexpr base, sexpr lower, sexpr upper)
        | Const(int n)
}
```

(a) Core Structural RTLIR

```
1 class Foo(Component):
2   def construct(s):
3     s.in_ = InPort(Bits8)
4     s.out = OutPort(Bits4)
5     s.adder = Adder(Bits2)
6
7     s.adder.in_ //= s.in_[0:2]
8     s.out[0:2]  //= s.adder.out
9     s.out[2:4]  //= 0
```

(b) Example

Figure 5.5: Core Structural RTLIR and an Example Model – (a) The definition of signal expressions, the main component of structural RTLIR. (b) Foo demonstrates how to instantiate sub-components (Adder) and create connections, which are the main ways to perform structural modeling in RTLIR.

```
module Behavioral_RTLIR
{
  -- Statements
  stmt = Assign(expr target, expr value)
       | If(expr cond, stmt* body, stmt* orelse)
       | For(loopvar var, expr lo, expr hi, expr inc)

  -- Expressions
  expr = Concat(expr* values)
       | ZeroExt(int nbits, expr value)
       | SignExt(int nbits, expr value)
       | BinOp(expr l, operator op, expr r)

  loopvar = LoopVar(string name)
  operator = Add | Sub | Mult
}
```

(a) Core Behavioral RTLIR

```
class Foo(Component):
  def construct(s):
    s.a = InPort(Bits4)
    s.b = InPort(Bits4)
    s.sel = InPort(Bits1)
    s.out = OutPort(Bits8)

    @update
    def upblk():
      if s.sel:
        s.out[0:4] @= concat(s.a[0:2], s.b[2:4])
        s.out[4:8] @= s.a + s.b
      else:
        s.out[0:4] @= zext(s.a[0:2], 4)
        s.out[4:8] @= sext(s.b[0:2], 4)
```

(b) Example

Figure 5.6: Core Behavioral RTLIR and an Example Model – (a) the definition of the core behavioral RTLIR, which is used in the behavioral modeling update blocks of PyMTL3 RTL models. (b) Foo includes an update block upblk that demonstrates how behavioral RTLIR represents common behavioral modeling primitives.

represents the getting attribute operation, which yields a new signal by accessing a signal of the current component or a field in a port of nested struct data type; Index represents the indexing operation, which produces the signal at the given index or the bit signal at the given bit of a vector.

Figure 5.5(b) provides a concrete example of signal expressions. On line 7, s.adder.in_ is a signal expression formed by two Attribute operations on on CurrentComponent; s.in_[0:2] is a signal expression formed by a Slice operation on top of the in_ attribute of the current component. Specially, literal values such as integers are also signal expressions (e.g., literal 0 on line 9). It it worth noting that signal expressions are typically used with RTLIR types to facilitate reasoning about operations between signals. For example, signal expression s.adder.in_ has type Port(Vector(2)) and s.in_[0:2] has type Signal(Vector(2)). These information can help designers deduce that this connection is well-formed because both sides have the same bitwidth for their data types.

Behavioral RTLIR – Contrary to structural modeling which can leverage arbitrary Python code to instantiate sub-components and create connections, the PyMTL3 behavioral modeling is done inside *update blocks* and is often limited to a subset of Python for synthesizable RTL models. Figure 5.6(a) shows the definition of the core behavioral RTLIR. Each statement (e.g., assignment, if-else statement, for-statement) in an update block corresponds to a statement clause in the behavioral RTLIR. Expressions in these statements are captured by the `expr` clause of behavioral RTLIR's definitions. This includes commonly used operations such as binary and unary operations as well as hardware-specific modeling primitives such as concatenation (`Concat`) and extensions (`ZeroExt` and `SignExt`).

As a concrete example, Figure 5.6(b) presents an example PyMTL3 model that includes behavioral RTL modeling. The update block `upblk` includes an if-else statement at the top level, and the signal `s.sel` is used as its condition. There are four `Assign` statements in the if-else statement, and each assignment is formed by combining signal assignments and hardware modeling primitives such as concatenation and extension.

Type Checking Behavioral RTLIR – Behavioral modeling in PyMTL3 RTL models differs significantly from structural modeling. All structural modeling primitives are executed, collected, and converted to the PyMTL3 IMIR at elaboration time. Behavioral modeling primitives, however, are only executed at simulation time. Since PyMTL3 passes are applied post-elaboration and before simulation, this creates the need of a robust and systematic approach to verify if a given behavioral modeling update block is well-formed. Behavioral RTLIR achieves this through type checking.

The behavioral RTLIR type checker targets statements and signal expressions in update blocks and takes a two-pass approach to perform type checking. In the first pass, the type checker recursively walks down the behavioral RTLIR representation and computes the RTLIR type of each expression. In the second pass, the type checker inspects the RTLIR type of each expression and determines if a type error occurs. For `Assign`, the type checker recursively collects the RTLIR type of both sides of the assignment and checks if they are the same and if the left hand side can be assigned. For `If`, the type checker verifies if the condition expression has signal type. As a concrete example, the update block `upblk` in Figure 5.6(b) contains one condition and four assignments. For slicing operations (e.g., `s.out[0:4]`), the behavioral RTLIR type checker computes the bitwidth of the vector by subtracting the lower bound from the upper bound. For concatenation

```python
class Translator:
  def visit(s, rtype):
    # Dispatch based on the type of rtype:
    # Vector, Struct, or PackedArray
    ...

  def visit_Vector(s, nbits):
    raise NotImplementedError()

  def visit_Struct(s, fields, types):
    raise NotImplementedError()

  def visit_PackedArray(s, types):
    raise NotImplementedError()

  ...
```

(a) Base Translator with Virtual APIs

```python
class ExampleTranslator(Translator):
  def visit_Vector(s, nbits):
    return f"[{nbits-1}:0]"

  def visit_Struct(s, fields, types):
    results = []
    for name, rtype in zip(fields, types):
      results.append(s.visit(rtype))

    return f"struct packed {{
        {'\n'.join(results)}
    }}"

  def visit_PackedArray(s):
    return f"[{s.ndims-1}:0]"
```

(b) Example Data Type Translator

Figure 5.7: Translation APIs and an Example Translator – (a) Base translator that defines the APIs of the translation framework. Each backend should inherit the base translator class. (b) An example translator that converts RTLIR data types to strings.

and extensions, the type checker uses the second argument to determine the bitwidth of the vector. The behavioral RTLIR type checker requires that the bitwidth to be constant at elaboration time. However, to support the fixed-width slicing operations that are commonly used in Verilog, the type checker allows slicing operations where the lower and upper bounds differ by a elaboration-time constant.

5.2.2 The Translation Framework

To facilitate ASIC and FPGA prototyping, modern embedded HDLs almost always support translation, which generates RTL hardware models in prototype languages such as SystemVerilog from the native RTL models. In this section, I describe the design and implementation of PyMTL3's *translation framework*, an extensible translation pass generator that bridges the RTLIR and multiple low-level prototype languages.

The translation framework is built on the *RTLIR visitor*, a class that provides methods to recursively walk down the RTLIR types, behavioral RTLIR, and structural RTLIR. Each RTLIR node has a corresponding `visit` method that may be overridden to allow programmatic inspection of the RTLIR. Figure 5.7 presents the base translator and an example translator that converts RTLIR data types to strings. The translator base class defines a dispatch method `visit`, which inspects the type of its argument `rtype` and dispatch it to the corresponding handling method. For example, if the `rtype` is a `Struct` RTLIR type, the `visit_Struct` method will be called with the appropriate arguments. Figure 5.7(b) demonstrates how new backends can programmatically generate strings from translator APIs. For example, simple RTLIR data types such as `Vector` and `PackedArray` can easily fill in a template of definition with their arguments. More complex data types such as `Struct` can be handled by recursively translating the individual fields and assembly them together (note that Line 8 calls the dispatch method defined in the base translator class). The behavioral and structural RTLIR can be similarly translated into a new backend.

I have implemented two backends of the translation framework in PyMTL3: one for SystemVerilog and one for Verilog. The SystemVerilog backend naturally supports complex syntax such as structs, unpacked arrays, and the `always_comb` and `always_ff` keywords, which the Verilog backend does not support. The Verilog backend differs from the SystemVerilog backend mainly in the that it converts struct and arrays of signals into individual vector signals with a common prefix. Thanks to the reusable API design of the translation framework, the majority of the SystemVerilog backend could be reused and only APIs that deal with struct and array data types and signals need to be overridden. This demonstrates that the translation framework is an extensible and efficient approach to adding new translation backends in PyMTL3.

5.2.3 The Import Pass in PyMTL3

As shown in Figure 5.1(d), the co-simulation between the translated Verilog design and the native test benches requires a communication mechanism between Verilog and the host language. This section introduces the *import pass*, a PyMTL3 pass that enables seamless co-simulation by dynamically loading the Verilog design into the host Python environment (also known as *importing*). The remaining of this section describes the design and implementation of the import pass.

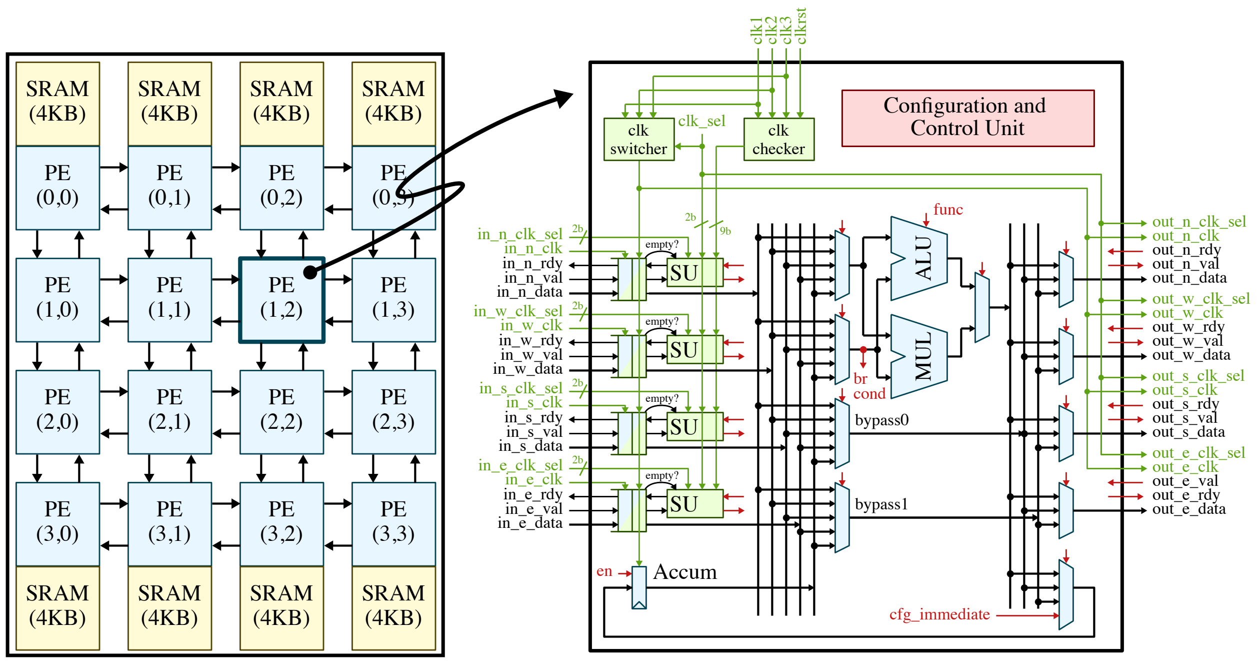
The import pass interacts with numerous files and tools to co-simulate Verilog with Python test benches. Figure 5.3 shows how the import pass generates and interacts with different kinds of files to import the translated SystemVerilog RTL back to Python. The import pass takes the RTLIR and the SystemVerilog RTL of the target model as input and performs the import in four steps. First, it generates the two wrappers (one in PyMTL3 and one in C) to allow communication between the translated RTL and the Python test bench. Second, it invokes Verilator, an open-source Verilog simulator, to compile the generated SystemVerilog RTL into C++ source code. Third, the import pass invokes a C++ compiler to compile the C wrapper and the generated C++ source file into a shared library. Finally, the import pass dynamically load the shared library using CFFI, a

Python Functions	**Called C Functions**	**Explanations**
`__init__`	`dlopen*`	Initializes the imported model; dynamically loads the compiled shared library into the current Python process.
`__del__`	`destroy_model`, `dlclose*`	Finalizes the imported model; deallocates heap space and closes the dynamically loaded shared library.
`construct`	`create_model`	Constructs the imported model; creates the Verilog model and allocates necessary heap space.
`comb_upblk`	`comb_eval`	Combinationally propagate the port values of the PyMTL3 wrapper into the compiled Verilog model.
`seq_upblk`	`seq_eval`, `has_assert_fired`	Ticks the compiled Verilog model and advances the clock by one cycle; also checks if any assertions has fired.
`assert_on`	`assert_on`	Enables assertion in the compiled Verilog model.
`line_trace`	`line_trace`	Retrieves the line trace of the compiled Verilog model.

Table 5.1: The Python and C Interface of a Imported Model – Each Python function in the left column calls the corresponding C functions in the right column. To implement these functions, the import pass generates a PyMTL3 wrapper and a C wrapper that define and implement these functions. *: this function is provided by the CFFI package and is not part of the C wrapper.

Python package for loading and interacting with shared libraries, and the Python test bench can then co-simulate with the generated RTL through the PyMTL3 wrapper.

Table 5.1 highlights the key functions exposed by the PyMTL3 wrapper (the **Python Functions** column) and the C functions called under the hood (the **Called C Functions** column). In a typical use case of the imported model, the Python functions are generally called in the following order. `__init__` initializes the imported model by dynamically loading the compiled Verilog model, which is done by calling the `dlopen` API of the CFFI package. After initialization, the `construct` method creates an instance of the imported model and allocates heap space for the model and associated metadata. At each cycle of co-simulation, the PyMTL3 simulator calls the `seq_upblk` method followed by `comb_upblk`. The former method evaluates the Verilog model before and after a clock edge and updates the model's states accordingly. The latter method then communicates the PyMTL3 and Verilog port values. The `assert_on` and the `line_trace` functions may be called during the co-simulation to enable assertions in the Verilog model or collect line trace from the Verilog line trace function. Finally, when the imported model goes out of scope in Python, the `__del__` method destroys the model, frees the allocated heap space, and unload the loaded shared library through `dlclose` CFFI API.



(a) 4x4 UE-CGRA PE Array (b) UE-CGRA PE

Figure 5.8: UE-CGRA Overview – (a) A 4x4 UE-CGRA with 32KB SRAM scratchpads. The top and bottom rows of PEs are directly connected to the scratchpads. (b) Overview of the UE-CGRA processing element (PE) design. The PE may operate at three different clock frequencies (sprint, nominal, and rest) and each input interface (`in_n/w/s/e`) is registered with a bisynchronous FIFO. Figure adapted from [TPO+21]

5.3 Case Study: Co-Simulation of an Ultra-Elastic CGRA

In Section 5.2, I described the design and implementation of the translation-import mechanism in PyMTL3. In this section, I present a case study of testing an ultra-elastic coarse-grain reconfigurable array (UE-CGRA) using the proposed translation-import mechanism. I demonstrate how translation-import improves the verification productivity of a complicated hardware generator, which incorporates mixed-language (Verilog) IPs, to verify the translated Verilog RTL using the native PyMTL3 test bench

5.3.1 UE-CGRA Overview

Coarse-grain reconfigurable arrays (CGRAs) are an efficient architecture generally consists of a grid of processing elements (PEs) connected with an interconnection network. Depending on the target workload, the PEs and the network may be reconfigured to implement different functionalities. CGRAs are capable of higher performance and energy efficiency than general-purpose processors *and* better programmability than fixed-function ASICs thanks to its reconfigurable nature.

However, irregular loops incur significant performance and energy inefficiencies on CGRAs because of the cross-iteration data dependency bottleneck. Ultra-elastic CGRAs (UE-CGRA) tackle the irregular loop specialization challenge with per-PE fine-grain DVFS and can selectively sprint or rest PEs to improve performance and energy efficiency.

Figure 5.8 provides an overview of the UE-CGRA architecture and the design of its processing elements (PEs). In UE-CGRAs, the PEs are interconnected with a mesh network. The top and bottom rows of PEs are also connected to an SRAM scratchpad. Each UE-CGRA PE can be reconfigured to accept input from two out of its four neighboring PEs and perform either ALU operations or a multiplication. Comparing to traditional CGRAs, UE-CGRAs allow individual PEs to be reconfigured and run at different frequencies: spring, which is the fastest and generally used for better performance; nominal, which is the default frequency; and rest, which is the slowest and generally used outside the critical path. To support this, each PE also a clock checker and a clock switcher to switch between different operating clocks.

UE-CGRA Verification challenges –UE-CGRA is a complex digital design with several components and can operate different clock frequencies. The reconfigurable nature of UE-CGRA imposes a significant challenge to verify the hardware using Verilog test benches. However, the large design size of UE-CGRA (typically including 64 PEs in an 8x8 arrangement) often significantly slows down the speed of native simulation in host languages like Python. In this section, I leverage the proposed seamless co-simulation methodology to tackle the above challenges and demonstrate how translation-import in PyMTL3 can significantly improve verification productivity.

5.3.2 Experiment Methodologies

In this section, I describe the methodology I use to perform co-simulation experiments on the UE-CGRA RTL design. I focus on explaining the design parameters and the microbenchmarks used in the experiments.

Design Specification –The UE-CGRA PE array is implemented mostly in PyMTL3 RTL and the clock switching and clock counting logic are implemented in Verilog. The target of verification is a representative 8x8 UE-CGRA array instance (64 PEs) and it assumes the clock period of the spring, nominal, and rest clock has a ratio of 2:3:9. The PE uses a ratiochronous clock-domain crossing scheme and registers the input from its four neighbours with bisynchronous FIFOs. Comparing to the traditional asynchronous crossing scheme, the ratiochronous approach selectively

```
while(hd) {
 if(hd->d==tgt)
  return hd->d;
 else
  hd=hd->nxt;
}
return -1;
```

(a) llist

```
for(i=0;i<N;++i){
 out = src[i]+err;
 if(out>127) {
  pixel = 0xFF;
  err = out-pixel;
 } else {
  pixel = 0;
  err = out;
 }
 dest[i] = pixel;
}
```

(b) dither

```
for(x=-S;x<=N;x++){
 bright=total+
  *ip++;
 tmp=*dpt++*
  *(cp-bright);
 area+=tmp;
 total+=tmp*bright;
}
```

(c) susan

```
for(k=0;k<G;++k){
 t_r=Wr*r[2*j*G+G+k]
  -Wi*i[2*j*G+G+k];
 t_i=Wi*r[2*j*G+G+k]
  +Wr*i[2*j*G+G+k];
 r[2*j*G+G+k]=
  r[2*j*G+k]-t_r;
 r[2*j*G+k]+=t_r;
 i[2*j*G+G+k]=
  i[2*j*G+k]-t_i;
 i[2*j*G+k]+=t_i;
}
```

(d) fft

```
for (i=0;i<21;++i){
 BF_ENC(right,left,
  s,p[i]);
 temp=right;
 right=left;
 left=temp;
}
```

(e) blowfish

Figure 5.9: UE-CGRA Microbenchmarks – All five microbenchmarks used in the experiments map the core irregular loop of the corresponding algorithm to a UE-CGRA PE configuration. Pseudo code adapted from [TPO+21].

suppresses unsafe clock domain crossings and reduces both verification complexity and synchronization latency penalties. The UE-CGRA PE supports a range of operations including integer multiplication and common ALU operations: comparison, addition, subtraction, left and right shifts, and bit-wise arithmetics. The PE also supports a merge operation `phi` and a branching operation `br`. These two operations enable mapping loops with branches onto UE-CGRAs.

Microbenchmarks –Figure 5.9 shows the pseudo code of the microbenchmarks used to verify the functionality of the UE-CGRA RTL. `llist` is a pointer-chasing microbenchmark. `dither` is commonly used in image processing. `susan` is an image processing algorithm for edge finding and noise filtering. `fft` implements the fast Fourier transform algorithm. `blowfish` is an encryption algorithm. To test the UE-CGRA with these microbenchmarks, I extract the innermost irregular loop and map it to UE-CGRA PE configurations. Each microbenchmark also corresponds to two different UE-CGRA power configurations: one *performance-optimized* and one *energy-optimized*, both of which are generated by the UE-CGRA power compiler [TPO+21]. At the end of co-simulations, the native PyMTL3 test bench compares the output in UE-CGRA PE array to the pre-computed reference output of the irregular loops.

5.3.3 Results and Analysis

Table 5.2 shows the simulation results of different UE-CGRA microbenchmarks using the translation-import mechanism in PyMTL3. The simulation statistics include the trip count of the target irregular loops, the number of SRAM reads and writes, and the simulation time measured in nominal clock cycles. Seamless co-simulation validates the idea that fine-grain DVFS in CGRAs can improve performance of irregular loops: comparing the simulation time of the energy-

Microbenchmark	Number of Tokens	SRAM Reads	SRAM Writes	EOpt Sim. Time	POpt Sim. Time
`llist`	1000	1999	1	8023	5359
`dither`	1000	1000	1000	8023	5662
`susan`	1000	1000	1	11026	7354
`fft`	1000	1000	1000	12043	8044
`blowfish`	32	32	1	865	523

Table 5.2: Simulation Results of UE-CGRA Microbenchmarks – Sim.: simulation. EOpt: energy-optimized. POpt: performance-optimized. EOpt and POpt refer to two different UE-CGRA power mappings to maximize energy efficiency or performance. Token count is equal to the trip count of the target irregular loop. Simulation time is measured by the number of nominal cycles in PyMTL3 seamless co-simulation.

optimized configuration against the performance-optimized configuration, it should be clear that selectively sprinting certain PEs in the critical path of the mapped loops can significantly speedup the application.

It is also worth noting that UE-CGRAs have multiple clock domains and deriving the accurate nominal cycle count is not trivial because PyMTL3 native simulation only supports one clock domain. Fortunately, translation-import provides a clean and systematic solution. The PyMTL3 Verilog translation pass generates Verilog code that includes both the translated PyMTL3 model and the Verilog clock switching and dividing logic. When the generated model is imported and simulated, the PyMTL3 simulator toggles the main clock, which will be divided by the Verilog clock divider into sprint, nominal, and rest clocks that drive UE-CGRA PEs. Since UE-CGRA clock-domain crossings are ratiochronous, dividing the number of PyMTL3 simulator cycles by the ratio of nominal clock periods yields the number of simulated nominal clock cycles.

Lastly, using the translation-import mechanism in PyMTL3 significantly improves the UE-CGRA verification productivity. Leveraging the Python host language, it is much more productive to implement the microbenchmarks, generate test inputs and reference outputs, and compare the reference outputs against the co-simulation results. Moreover, the Python environment facilitates debugging of the UE-CGRA. For example, identifying a UE-CGRA bug from waveform is tedious and time-consuming because it requires visualizing dataflow information from simulation waveforms. Using PyMTL3's line trace feature, I created a line trace function that visualizes UE-CGRA activities based on RTL co-simulation states, which is much more intuitive than the waveform.

5.4 Conclusion

This chapter presents translation-import, a solution to the seamless co-simulation verification challenges implemented in PyMTL3. The foundation of translation-import is the register-transfer level intermediate representation (RTLIR), which provides a canonical representation for synthesizable models in PyMTL3. Based on RTLIR, the translation framework provides clean interfaces to generate RTL from PyMTL3 models and can be easily extended to different backends. The import pass compiles the translated Verilog RTL into shared libraries and provides appropriate wrappers to dynamically load these libraries into the Python environment. This chapter concludes with a case study of co-simulating the ultra-elastic CGRA in PyMTL3. The translation-import mechanism provides a clean solution to co-simulating UE-CGRAs and significantly improves verification productivity.

CHAPTER 6
THE CGRA TAPE-OUT CASE STUDY

The previous chapters identified and addressed verification challenges in a typical agile hardware workflow: dynamic HDLs, generator development, instance composition, and co-simulation. This chapter presents my work on prototyping a coarse-grain reconfigurable array (CGRA) to demonstrate how the proposed techniques improve verification productivity. The chip was a DARPA-funded, multi-university project on developing the Bigblade SoC in GlobalFoundries 14nm technology. Professor Michael B. Taylor's group at University of Washington led the Bigblade tape-out and did the physical design of CGRA. I was the Cornell University student lead for the Bigblade project and led a team of graduate students to design, implement, and test the CGRA module.

This chapter is organized as follows. I start by describing the details of the CGRA module, including the overall architecture of the CGRA and preliminary physical design of the CGRA. Then I describe how the proposed verification techniques in this thesis helped improve CGRA verification velocity during the tape-out.

6.1 CGRA and Chip Details

The important application domains such as machine learning often demand high performance and energy efficiency for continuously evolving kernels and algorithms. Comparing to general-purpose processors or fixed-function ASICs, CGRAs provide a fabric for acceleration and efficiency without sacrificing flexibility. The architecture community has demonstrated that CGRAs can achieve significant performance and energy efficiency on important applications such as deep neural networks and imaging processing [BHME18, XZP+20, SZP+20, VBP+16, FWC+18].

In this section, I present the details of the CGRA accelerator. I describe the overall architecture of the CGRA and its preliminary physical design done at Cornell University.

CGRA Architecture – The main component of the taped out CGRA is an array of processing elements (PEs) interconnected with a mesh network. Figure 6.1 shows the architecture of a CGRA PE. It takes input from four sides (north, west, south, and east) and registers the inputs with a two-element FIFO. The use of FIFOs allows *elastic* communications between neighboring PEs: input messages can arrive at different cycles and they are only consumed when all required

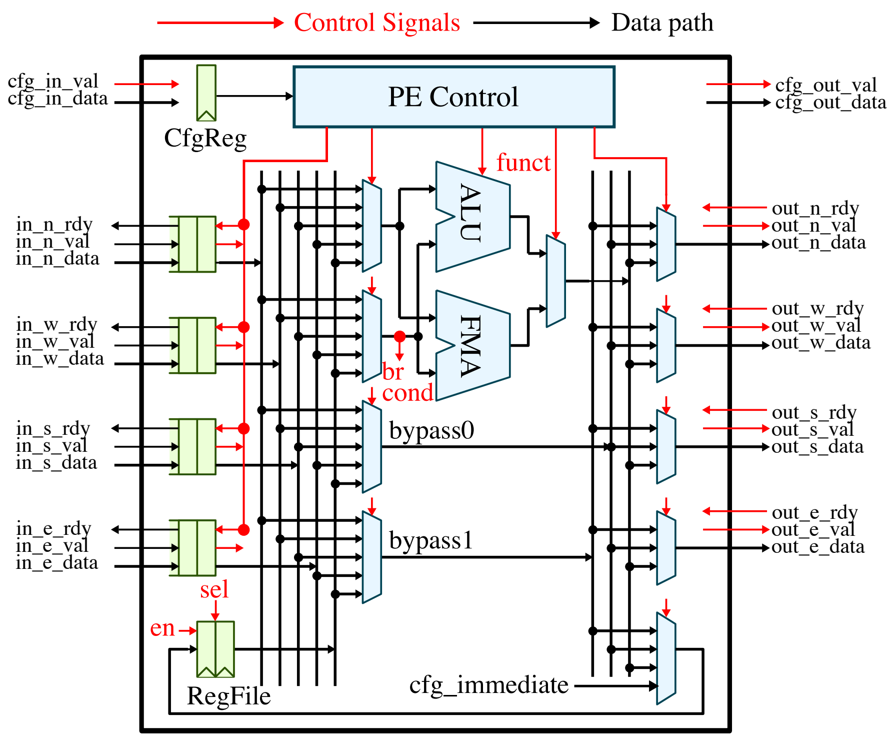


Figure 6.1: CGRA PE Architecture – ALU: arithmetic logic unit that supports common integer arithmetics, comparison, and bit-wise operations. FMA: fused multiply-add unit that supports fixed-width number multiplication and addition.

operands have arrived. The elastic nature of PE communications facilitates mapping kernels to the PE array. Depending on the content of the configuration register, the ALU (supports common integer operations) or FMA (supports fused fixed-width number multiplication addition) operates on two operands coming from either of the four sides and the internal register file. Depending on the configuration, the PE can put the ALU/FMA results or register file content on any of the four output interfaces. In addition, the PE also supports *bypassing* up to two input messages from any input interface to any output interface to enable more flexible mapping.

Figure 6.2 shows the architecture of a CGRA with a 4x4 PE array (the taped out CGRA accelerator includes an 8x8 PE array and four 4KB SRAM banks; the smaller PE array is shown here for simplicity). The CGRA module has three major components: the CGRA core, which consists of a PE array and associated memory and configuration engines (MEs and CEs); the scratchpads, which provides local storage of data; and the crossbar, which interconnects the CGRA core, accelerator request interface, and the scratchpads. CGRA scratchpads consist of SRAM banks that store both PE and ME configurations and data for computation. The MEs are small DMA units that, after configuration, fetch data at a specified scratchpad address with a specified stride and pass fetched

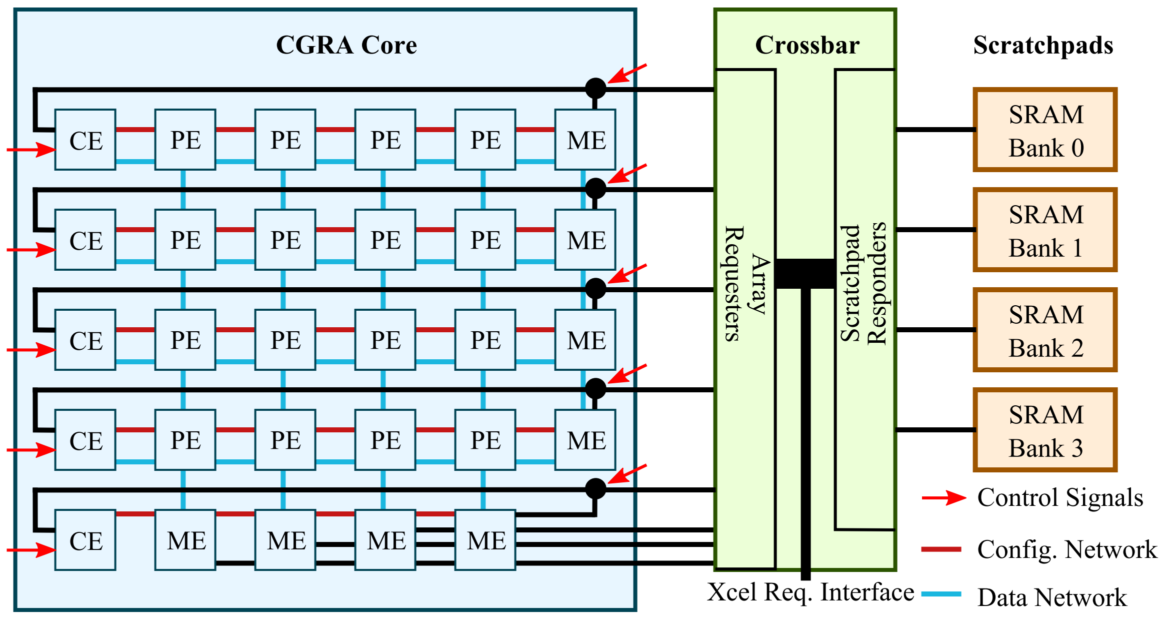
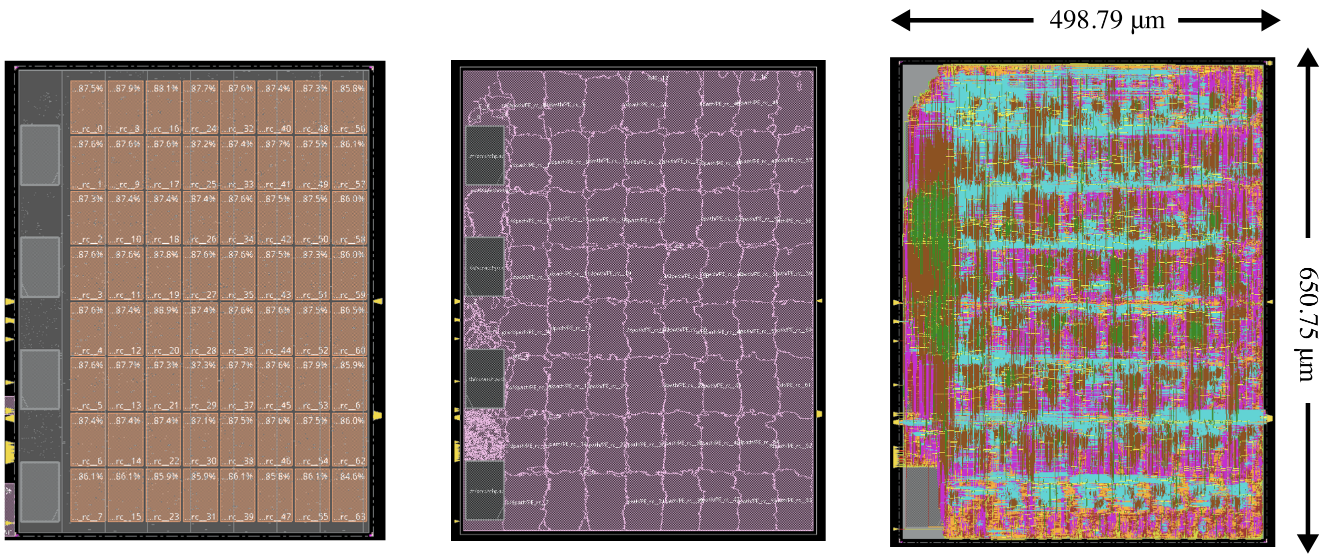


Figure 6.2: CGRA Architecture – CE: configuration engine. ME: memory engine. PE: processing element. The diagram shows the architecture of a 4x4 PE array CGRA accelerator, whereas the taped out CGRA accelerator contains an 8x8 PE array.

data to its neighboring PE. The CEs operate in a similar way and pass configuration messages to PEs and MEs through the configuration network.

CGRA Preliminary Physical Design – After implementing the target CGRA architecture in PyMTL3 RTL, I pushed the CGRA architecture through a GlobalFoundries 14nm ASIC flow with the help of Yanghui Ou from Batten Research Group. Figure 6.3 shows the preliminary results of floor planning, placement, and routing. As shown in Figure 6.3(a), each PE in the 8x8 PE array of the CGRA belongs to a soft grid of boxes in the floor plan, which significantly reduces the tool time required for placement. Figure 6.3(b) shows the amoeba plot of the post-placement CGRA, which shows that most standard cells in the PE are placed according to the floor plan. Finally, Figure 6.3(c) shows the routed CGRA, which has a dimension of about 650 µm high and 500 µm wide and was instantiated multiple times to create multiple CGRA accelerators. The preliminary routing result also reveals the existence of routing hotspots due to the crossbar and the large number (17) of requesters.



(a) CGRA Floorplan (b) Post-Placement Amoeba Plot of CGRA (c) Post-Routing CGRA

Figure 6.3: CGRA Preliminary Physical Design – The target CGRA has an 8x8 PE array an four 4KB SRAM banks. The PE macro is 48 μm wide and 73 μm high. Figures based on the work of Yanghui Ou, who did preliminary physical design for the CGRA.

6.2 Addressing the CGRA Verification Challenges

The target CGRA hardware has high complexity because of its reconfigurability and elasticity. This complexity adds to the difficulty of verification in every step of the agile development of the CGRA module. In this section, I describe how the techniques proposed in this thesis helped improve the verification productivity of a small team of graduate students that were responsible for the designing and testing of the CGRA module in the SDH tape-out.

6.2.1 GT-HDL for Safe and Performant CGRA Test Harness Composition

The first verification challenge arises in the mixed-typed composition of the CGRA instance, where the statically typed CGRA is composed with a dynamically typed test harness. As described in Section 2.1, existing dynamic HDLs suffer from the tension between providing safety guarantees of mixed-typed compositions and simulation performance. Using the naive approach to safe guard the mixed-typed CGRA test harness incurs high simulation overhead because of the large number of connections within the CGRA PE array.

Solution – Inspired by the efficient mixed-typed compositions in GT-HDL, I leverage the type-based simulation optimization techniques proposed in Section 2.4.2 to improve the CGRA test

```python
class CGRA(Component):
 def construct(s, r, c, W):
  s.recv_n = [ RecvIfc(W) for _ in range(c) ]
  s.pes = [ PE(W) for _ in range(r * c) ]
  for y in range(r):
   for x in range(c):
    # Connecting north most row of PEs to CGRA
    # interfaces on the north
    if y == r - 1:
     # Index out-of-range bug; only triggered if r > c
     # Correct index: y*c+x
     connect(s.pes[y*r+x].recv[N].msg, s.recv_n[x].msg)
```

(a) CGRA Generator with Bug

```python
dut1 = CGRA(8, 8, Bits32)
dut1.elaborate() # Pass!

dut2 = CGRA(16, 16, Bits32)
dut2.elaborate() # Pass!

dut3 = CGRA(16, 8, Bits32)
dut3.elaborate() # Fail!

dut4 = CGRA(8, 16, Bits32)
dut4.elaborate() # Fail!
```

(b) Failing and Passing CGRA Examples

Figure 6.4: Example CGRA Generator Bug – (a) the CGRA generator has three parameters: `r` and `c`, the number of rows and columns in the PE array; `W`, the width of the PE array data path. The code in the shown for-loop tries to connect the north most row of PEs to the north side interfaces of the CGRA. (b) the out-of-bound array indexing bug occurs when a non-square PE array is instantiated.

harness simulation performance without compromising safety guarantees. With the simulation type check pruning technique, GT-HDL removes all simulation-time type checks inside the CGRA instance and only preserves the checks at the test harness and CGRA DUT interface. Signal coalescing further removes redundant computation by setting up references between a statically typed writer signal and several statically typed readers.

Results – To demonstrate the simulation performance of applying GT-HDL techniques, I compare the simulation performance (measured in cycles/second) of the same CGRA test harness in PyMTL3 versus in GT-HDL. On an 8x8 CGRA instance, GT-HDL achieves 36.35% better simulation performance than PyMTL3 (12.85 cycles/second vs 9.42 cycles/second). The improved simulation performance enables shorter design-debug iteration cycles, which results in improved verification productivity.

6.2.2 Symbolic Elaboration for CGRA Generator Development

The second CGRA verification challenge arises in generator development, where the enormous generator parameter space often hides subtle design bugs that are difficult to discover with dynamic verification techniques. Figure 6.4 shows an example of such CGRA bugs. Figure 6.4(a) details the CGRA generator code that connects the north most row of PEs to the north side interfaces of the CGRA. The generator contains a design bug where the designer indexes into the PE array `s.pes` with an incorrect expression. As shown in Figure 6.4(b), this may cause index out-of-bound issues but only with non-square PE arrays. Since the SDH tape-out targets a square CGRA PE array, this bug is difficult to discover through simulation.

Solution – I leverage symbolic elaboration (Chapter 3) to tackle the CGRA verification challenges in generator development. Unlike dynamic testing techniques such as simulations, symbolic elaboration does not rely on concrete generator parameters nor test vectors to discover bugs. In addition, symbolic elaboration is capable of reasoning about arithmetics among bitwidths, array lengths, and expressions, making it an ideal choice for subtle CGRA generator bugs.

When applying symbolic elaboration to the code in Figure 6.4(a), the symbolic elaborator picks up the length information of the PE array `s.pes` from its definition and records the length to be `r * c`, a symbolic expression. From the for-loop syntax, the symbolic elaborator also understands that `y` ranges from 0 to `r`-1 and `x` ranges from 0 to `c`-1. When the elaborator reaches the index expression `y*r+x` on line 12, it determines that the maximum index is `(r-1)*r+c-1`, which could be larger than the maximum index of the array `r*c-1`. Based on this reasoning, the symbolic elaborator points out the potential out-of-bound error on line 12 and provides possible combinations of parameters to trigger this error (e.g., `r=16`, `c=8`).

Results – I applied the symbolic elaborator on the CGRA generator in PyMTL3 to thoroughly validate the common generator properties including matching bitwidths, correct local port directions, bounded array indexing, and valid hierarchical references. The symbolic elaborator reported several bitwidth mismatches where a fixed width 32-bit signal is assigned to a signal of parameterized bitwidth. As the designer of the CGRA generator, I can confirm these design bugs were introduced because the designer had a 32-bit CGRA target in mind and the unit tests failed to detect these bugs because the tests targeted a 32-bit instance of CGRA. After fixing the bitwidth mismatches, the symbolic elaborator was able to validate all generator properties for the CGRA generator, which provides strong correctness guarantees not jut for the taped-out CGRA instance but any instances produced by the CGRA generator.

6.2.3 Latency Equivalence Checking for PE Instance Composition

The third CGRA verification challenge happens during instance composition, where individual PEs with latency-insensitive (LI) interfaces are connected to form an elastic reconfigurable computing substrate. However, subtle bugs could appear in the complex handshake control logic of the PE's LI interfaces. Detecting these bugs with simulations requires a comprehensive test suite

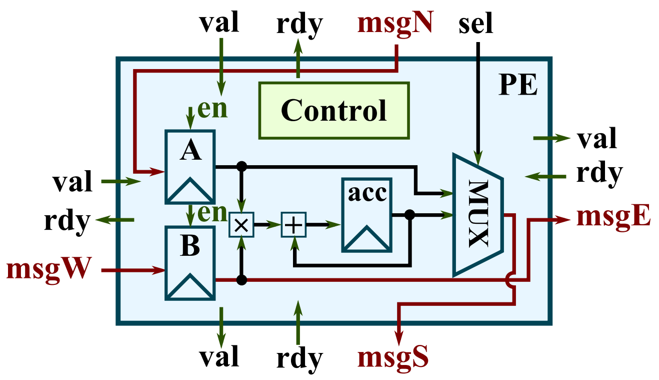


Figure 6.5: Example of PE Composition Verification Challenge – N, W: ingress interface on the north and west side of PE; E, S: egress interface on the east and south side of PE. acc: accumulation register. The control unit has to make sure the messages from N and W LI interfaces are correctly registered into A and B, and that contents of A and B are consumed when E and S LI interfaces are ready. This PE represents a simpler PE design than the actual reconfigurable PE used in the SDH CGRA tape-out.

that covers almost all stall events on the LI interfaces, which is prohibitively expensive for a small team of agile hardware designers. Figure 6.5 shows a latency-insensitive PE with four LI interfaces. The PE control unit has to make sure that (1) the messages from the N and W LI interfaces are registered into register A and B in pair and (2) the contents of A and B are consumed when the E and S LI interfaces are ready.

Solution – I adopted the latency equivalence checking technique from Chapter 4 and applied it to a simplified PE shown in Figure 6.5. Using the algorithm described in Section 4.3.2, I constructed a verification harness that perturbs the ingress N and W LI interfaces and formally verifies if the egress E and S LI interfaces produce the same sequences of messages. Since the exact results of the accumulation is not critical for verifying the correctness of the LI handshake logic, I replaced the PE multiplier with a bit-wise XOR unit to speedup the formal tool's convergence.

Results – I used JasperGold FPV 2023.03 on a commodity server with 72 cores of Intel Xeon E7-8867 v4 CPU and 256 GiB of main memory to carry out the formal verification of the PE. JasperGold found a LI handshake bug in the initial PE design where the contents of B register could get overwritten under certain delay conditions. After fixing this handshake logic error, JasperGold was able to prove all formal assertions in the constructed test harness, which indicates that the PE is stall invariant.

Unit Tests	Functional units, processing elements, memory engines, configuration engines, eager fork adapter, CGRA core, CGRA accelerator
Integration Tests	**Kernels:** conv1d, vvadd, i-GEMM, f-GEMM, dither, FFT, FIR, latnrm, llist
	Features: reconfig, bypass, reduction, rf-writeback, sram read, termination
	Operations: add, sub, fadd, fsub, fmul, fma, eq, lt, gt, sgt, and, or, xor, sll, slr

Table 6.1: CGRA Functional Verification using Seamless Co-Simulation – Each item under **Unit Tests** corresponds to a CGRA component that has directed unit tests. Each item under **Integration Tests** corresponds to a test case that runs on the RTL CGRA module on the Bigblade silicon. Kernel refers to microbenchmarks that run on the CGRA. Features correspond to test cases that stress test CGRA features (such as reconfiguration, termination, etc.). Operations correspond to the supported arithmetic operations by CGRA.

6.2.4 Translation-Import for Functional Verification

Functional verification is the final and most challenging verification problem for the agile development CGRA. Due to the reconfigurable PEs and interconnection networks, CGRA functional verification needs a systematic approach to navigate through an enormous test input space and validate the functionalities of CGRA.

Solution – I leveraged the seamless co-simulation technique from Chapter 5 to create a complete test suite in Python for the target CGRA model. Table 6.1 shows the CGRA test suite, which includes both unit tests and integration tests. For each test case shown in the table, I used the translation-import framework to reuse the Python test bench for both the native PyMTL3 CGRA and the translated Verilog CGRA.

The unit tests of CGRA include directed test cases for CGRA components such as the PEs, MEs, and CEs. The unit tests also include directed test cases that can be run on the CGRA core (only CGRA PE array) and the CGRA accelerator (only PE array, crossbar, and scratchpads). Thanks to seamless co-simulation, the same test cases that were developed for the CGRA core can be reused for the CGRA accelerator (both the PyMTL3 and Verilog versions). The integration tests of CGRA leverage a test harness that connects a non-CGRA Bigblade mock-up and the CGRA module, which provides detailed and accurate simulations for functional verification. There are three categories of test cases in integration tests: kernels, which are microbenchmarks running on the CGRA; features, which are critical CGRA features such as reconfiguration and termination; operations, which are supported arithmetic operations by the CGRA. Each operation test case is also randomized to allow the operation to be placed onto different PEs, which helps validate the functionality of different PEs and network connections.

Results – The Cornell SDH CGRA tape-out team performed extensive simulations using the test suite described in the previous section. The unit tests in the test suite enabled rapid feedbacks for the agile hardware designers, which allowed the hardware designers to continue the implementation of features productively. The integration tests provided significant functional coverage for the CGRA module thanks to the extensive operation test cases and its randomized counterparts. Using the test suite at RTL did not reveal functional bugs in the CGRA design. However, the CGRA tape-out team discovered functional bugs that broke the gate-level simulations of the CGRA driven by the input traces of the operation test cases. For example, the operation test case for integer addition revealed that the PE ALU was not data-gated, which leads to inconsistent results. Overall, the test suite enabled by the translation-import mechanism introduced in Chapter 5 helped the agile designers achieve high design and verification productivity and cover subtle design bugs which could have failed the tape-out.

6.3 Conclusion

In this section, I presented a case study of the SDH CGRA tape-out. I discussed the details of the chip and how the techniques proposed in this thesis helped the Cornell SDH CGRA tape-out team achieve higher verification productivity. Facing the unique challenges of the CGRA development, I leveraged GT-HDL's simulation type check pruning and signal coalescing techniques to improve CGRA test bench's simulation performance; I applied the symbolic elaborator to validate CGRA generator properties in generator development; I proposed to leverage latency equivalence checking to safeguard the complex latency-insensitive handshake logic in PEs to facilitate PE instance composition; and the Cornell team leveraged seamless co-simulation to create a test suite including comprehensive unit and integration tests. Putting together, the techniques proposed in this thesis allowed for fast design iterations on the CGRA prototype, increased the design team's confidence of the correctness of the CGRA, and revealed subtle design bugs.

CHAPTER 7
CONCLUSION

This thesis presented my work on addressing the verification challenges in typical agile hardware workflows. I argued that innovations across HDLs, generator development, instance composition, and co-simulation are necessary to augment existing agile hardware methodologies with productive verification. I presented a case study of a CGRA tape-out to demonstrate the effectiveness of the proposed techniques. In this chapter, I summarize my thesis contributions and describe future research directions.

7.1 Thesis Summary and Contribution

This thesis began with the observation that specialized hardware systems generally have high NRE costs that hinder the development of promising systems. Different from traditional hardware methodologies which assume a monolithic workflow, agile hardware methodologies promise to improve development productivity by iterating on an incomplete yet working prototype. However, verification challenges often arise from different steps of agile hardware workflows: HDLs, generator development, instance composition, and co-simulation. I then presented four different techniques to address the challenges in these steps.

I first introduced GT-HDL, an embedded HDL that enables safe and performant mixed-typed compositions. The increasing adoption of dynamically typed test harnesses reveals the challenge of having to sacrifice simulation performance to safeguard mixed-typed compositions. I propose GT-HDL, an embedded HDL that leverage a combination of optional type checkers, guarded generator parameters, and type-based simulation optimizations to improve simulation performance of dynamic HDLs without compromising safety.

I then introduced symbolic elaboration, an SMT solving-based technique to validate generator properties. Verifying hardware generator properties becomes increasingly more challenging because highly sophisticated generators typically have a prohibitively large parameter space, which makes enumerating parameter combinations intractable. Symbolic elaboration statically analyzes hardware generators and converts generator properties into SMT-solvable integer constraints. Symbolic elaboration can safeguard generators in agile hardware and significantly improve generator verification productivity.

I focused on instance composition and proposed latency equivalence checking to verify latency-insensitive handshake logics with formal verification. Latency-insensitive (LI) interfaces are ubiquitous in agile hardware but the LI handshake logic is error-prone and difficult to debug. I defined the stall invariant property of LI interfaces and introduced an algorithm to construct verification harnesses for given LI RTL modules that proves or falsifies the stall invariant property. Latency equivalence checking mitigates the instance composition verification challenge with formal verification.

Shifting focus to the co-simulation verification challenges, I introduced the translation-import mechanism to enable seamless co-simulation, which verifies the generated Verilog RTL using native Python test benches. The translation-import mechanism is based on RTLIR, a canonical representation of synthesizable hardware models that enables a extensible translation framework. I also presented a case study of co-simulating a UE-CGRA with Python test benches. Translation-import enables seamless co-simulation to productively verify generated Verilog RTL using Python.

Lastly, I described a CGRA tape-out case study in GlobalFoundries 14nm technology to demonstrate the effectiveness of the techniques proposed in this thesis. I leverage GT-HDL type-based optimizations to improve CGRA test harness simulation performance without compromising safety. During CGRA generator development, symbolic elaboration identifies a subtle design bug and validates the correctness of CGRA generator properties after fixing the bug. Latency equivalence checking detects a potential PE LI handshake bug and provides waveform of a counterexample to facilitate debugging. Translation-import enables productive random testing and hypothesis testing of the CGRA Verilog RTL. Overall, the proposed techniques significantly speed up the CGRA verification.

The major contributions of this thesis are as follows:

- I proposed GT-HDL, an embedded HDL that enables safe and performant mixed-typed compositions.
- I proposed symbolic elaboration, an SMT solving-based static analysis approach to validating hardware generator properties.
- I proposed the stall invariant property and latency equivalence checking, a formal verification technique that proves or falsifies the stall invariant property of latency-insensitive handshake interfaces.

- I proposed a translation-import mechanism in PyMTL3 to enable seamless co-simulation of generated RTL and the native Python test benches. I demonstrate how translation-import improves verification productivity with a UE-CGRA co-simulation case study.
- I presented a case study of a CGRA tape-out in GlobalFoundries 14nm technology and apply the techniques proposed in this thesis to overcome the verification challenges in the agile development of the CGRA.

7.2 Future Work

In this section, I identify several future research directions based on the works in this thesis. For each research direction, I describe the motivation, the possible approaches to the research question, and its potential impacts.

7.2.1 Formalizing Safety Guarantees for Mixed-Typed Compositions

In Chapter 2, I describe GT-HDL, an embedded HDL that supports safe and performant mixed-typed compositions. To provide safety guarantees for mixed-typed compositions, GT-HDL adopts guarded generator parameters and simulation-time type checking, which insert runtime type checks to prevent ill-typed parameters and values from propagating into statically typed instances. Despite its success of preventing type errors in an engineering context, GT-HDL does not formalize the safety guarantees for mixed-typed compositions, which leaves room for potential misunderstandings between HDL experts and agile hardware designers.

Possible Approaches – Inspired by the type system research in the programming language community, a type system typically represents a complete and unambiguous specification of safety guarantees. Researchers have already explored languages with novel type system capabilities to safe guard hardware designs. Filament [NdAS23] adopts timeline types to encode the latencies of hardware modules, which allow compilers to verify if multiple modules can be composed safely to form a statically scheduled pipeline. Parafil [NGLS24] takes the idea of timeline types one step further and applies it to hardware generators to provide correctness guarantees for all instances derived from a generator. Dahlia [NAT+20] uses time-sensitive affine types to enforce that memory read and write accesses do not exceed the number of memory ports at any cycle at the source code level.

Existing research works generally target one specific aspect of hardware design, and significant research and development efforts remain to fully close the type system gap in practical HDLs. To formulate and implement a generic and fully capable type system for hardware designs, researchers may start by formalizing the semantics and type systems of modern HDLs before eventually reaching the formal specification of mixed-typed compositions. To establish such specifications, researchers might need to establish the formal semantics of HDLs, which lays the foundation of building powerful type systems for HDLs. The fundamental difference between the semantics of hardware description languages versus software programs may create significant challenges for formalizing the semantics and type systems of HDLs.

7.2.2 Type Checking for Embedded Hardware Description Languages

Symbolic elaboration performs SMT solving-based static analysis to validate hardware generator properties implemented in PyMTL3. As a result, the PyMTL3 embedded HDL needs a static checking step before elaborating and simulating the target models. This resembles significant similarities to static type checking in statically typed programming languages, where the type checker scans through source code and searches for type errors. Type checking is a promising approach to systematically validate and enforce hardware generator properties. There has been some research on type checking (e.g., BlueSpec [Nik04]) hardware generators, but they generally expose a different abstraction than RTL and require significant efforts to design the HDL from ground up. Therefore, designing and implementing effective type systems for embedded HDLs remains an open research question.

Possible Approaches – One possible approach to type checking embedded HDLs is to leverage optional type checkers such as Mypy for Python. Unfortunately, these optional type checkers typically lack the capabilities to encode generator properties. For example, Mypy cannot encode parametrized bitwidths, which are commonly used in PyMTL3 generators. Research in this direction can explore extending existing host languages to support type checking of embedded DSLs. This may require support of first-class embedded DSLs, which includes a specification of the target type system that can be enforced with the native type checker of the host language.

Existing research works have already created representations for hardware development (typically in the form of HLS) that supports rigorous type checking. For example, Allo [CZX+24] is an accelerator design language (ADL) that supports both transformations and type checking of the functional specification of accelerators. Allo incorporates a simple type systems for arrays partitioned with different strategies and uses a linear-time unification algorithm to assign memory layout properties to each node in its representation.

7.2.3 Democratizing Formal Verification for Agile Hardware Methodologies

Formal verification is a powerful tool to tackle subtle and difficult bugs in hardware designs. Despite recent efforts on democratizing formal verification [MMB+18], formal verification still has relatively low adoption in agile hardware. There are two main challenges to democratizing formal verification in agile hardware: (1) the lack of full-featured open-source formal verification tools and (2) the needs of full specifications of the expected behaviors.

Possible Approaches – To improve the features of open-source formal verification tools, researchers may want to prioritize the most-wanted features of hardware designers. For example, temporal logic is a useful component of formal verification but has little support from open-source tools. The continuous improvement of high-quality and openly available tools will surely boost the adoption of formal verification.

Another possible approach to democratizing formal verification is to find out how to reduce the needs of manual property specifications. For example, specifications can be generated from hardware models that implement certain communication protocols; the property specifications may be templated on certain parameters, which can be inferred from the RTL implementation. The stall invariant property proposed in Chapter 4 can be conveniently inferred from RTL modules that use val-rdy latency insensitive handshakes. Under limited decoupling through finite FIFOs, violations of the stall invariant property can be effectively detected using commercial formal verification tools. However, it remains an open research question to prove the full stall invariant property without decoupling between the DUVs. Future research may investigate into alternative verification harnesses radically different from the one proposed in Chapter 4 to achieve the full proof of the stall invariant property.

7.2.4 Extensive and Productive Co-Simulation

Seamless co-simulation is an effective technique to boost verification productivity by reusing the native test benches on translated RTL. In Chapter 5, I described translation-import, a technique to co-simulate generated Verilog RTL (compiled with Verilator) and native PyMTL3 test benches. However, it remains an open research question to extend co-simulation to other backends than Verilog RTL and Verilator. For example, the Python test benches should be able to co-simulate with C++ high-level behavioral models, gate-level hardware models backed by commercial simulators, hardware on FPGAs, or ASIC prototypes.

Possible Approaches – One possible approach to enable extensive and productive co-simulation is to design and implement a common interface between high-level test benches and other backends. Then similar to the import pass in PyMTL3, the backend hardware can be imported and exposed to the high-level programming language process. It may be challenging to design a unified interface for multiple backends, since some commercial simulators may require the control of the main process and cannot be dynamically loaded as a shared library. In that case, more engineering efforts are necessary to achieve seamless co-simulation.

BIBLIOGRAPHY

[AAB$^+$16] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[ABG$^+$20] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro*, May 2020.

[AHY$^+$15] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2015.

[AK95] D. P. Appenzeller and A. Kuehlmann. Formal Verification of a PowerPC Microprocessor. *Int'l Conf. on Computer Design (ICCD)*, 1995.

[Bal24] M. Ballance. PyVSC: SystemVerilog-Style Constraints, and Coverage in Python. Online Webpage, 2024 (accessed Feb 23rd, 2024). `https://github.com/fvutils/pyvsc`.

[BAT14] G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. *European Conf. on Object-Oriented Programming (ECOOP)*, 2014.

[BCD$^+$18] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys*, 2018.

[BCE$^+$03] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. *Proc. of the IEEE*, 2003.

[BCJ$^+$20] J. Balkind, T.-J. Chang, P. J. Jackson, G. Tziantzioulis, A. Li, F. Gao, A. Lavrov, G. Chirkov, J. Tu, M. Shahrad, , and D. Wentzlaff. OpenPiton at 5: A Nexus for Open and Agile Hardware Design. *IEEE Micro*, May 2020.

[BCRZ99] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying Safety Properties of a PowerPC-Microprocessor Using Symbolic Model Checking without BDDs. *Int'l Conf. on Computer-Aided Verification (CAV)*, 1999.

[BCSS98] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. *Int'l Conf. on Functional Programming (ICFP)*, Sep 1998.

[BHME18] I. Bae, B. Harris, H. Min, and B. Egger. Auto-Tuning CNNs for Coarse-Grained Reconfigurable Array-Based Accelerators. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, Nov 2018.

[BKK$^+$10] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. Cλash: Structural Descriptions of Synchronous Hardware Using Haskell. *Euromicro Conf. on Digital System Design (DSD)*, Sep 2010.

[BLM01] P. Bjesse, T. Leonard, and A. Mokkedem. Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers. *Int'l Conf. on Computer-Aided Verification (CAV)*, 2001.

[BM02] L. Benini and G. D. Micheli. Networks on Chips: A New SoC Paradigm. *IEEE Computer*, 2002.

[BSDTH16] A. Bonnaire-Sergeant, R. Davies, and S. Tobin-Hochstadt. Practical Optional Types for Clojure. *European Symposium on Programming (ESOP)*, 2016.

[BVR$^+$12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. *Design Automation Conf. (DAC)*, Jun 2012.

[Car15] L. P. Carloni. From Latency-Insensitive Design to Communication-Based System-Level Design. *Proc. of the IEEE*, 2015.

[CBRZ01] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 2001.

[CCA$^+$11] 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. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, Feb 2011.

[CDHK15] E. Cerny, S. Dudani, J. Havlicek, and D. Korchemny. *SVA: The Power of Assertions in SystemVerilog*. Springer International Publishing, 2015.

[CFF$^+$01] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of Bounded Model Checking at an Industrial Setting. *Int'l Conf. on Computer-Aided Verification (CAV)*, 2001.

[CGK$^+$11] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. *International Conference on Software Engineering (ICSE)*, 2011.

[CKES17] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss - An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(1):127–138, Jan 2017.

[CLN$^+$11] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(4):473–491, Mar 2011.

[CLX$^+$16] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, and Y. W. andand Yuan Xie. PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2016.

[CMSSV99] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A Methodology for Correct-by-Construction Latency Insensitive Design. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 1999.

[CMSV99] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Latency Insensitive Protocols. *Int'l Conf. on Computer-Aided Verification (CAV)*, 1999.

[CMSV01] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, Sep 2001.

[CTD$^+$17] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood. A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation. *Int'l Conf. on Field Programmable Logic (FPL)*, Sep 2017.

[CTN$^+$22] A. Carsello, J. Thomas, A. Nayak, P.-H. Chen, M. Horowitz, P. Raina, and C. Torng. mflowgen: A Modular Flow Generator and Ecosystem for Community-Driven Physical Design. *Design Automation Conf. (DAC)*, Jul 2022.

[CZX$^+$24] H. Chen, N. Zhang, S. Xiang, Z. Zeng, M. Dai, and Z. Zhang. Allo: A Programming Model for Composable Accelerator Design. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2024.

[Dec04] J. Decaluwe. MyHDL: A Python-based Hardware Description Language. *Linux Journal*, Nov 2004.

[DKR$^+$21] S. Dai, A. Klinefelter, H. Ren, R. Venkatesan, B. Keller, N. Pinckney, and B. Khailany. Verifying High-Level Latency-Insensitive Designs with Formal Model Checking. *cs.LO arXiv:2102.06326*, Feb 2021.

[DT01] W. J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. *Design Automation Conf. (DAC)*, 2001.

[DTS20] D. Dangwal, G. Tzimpragos, and T. Sherwood. Agile Hardware Development and Instrumentation With PyRTL. *IEEE Micro*, May 2020.

[Dub05] M. Dubash. Moore's Law is Dead, Says Gordon Moore. *Techwold*, Apr 2005.

[FAP+12] K. E. Fleming, M. Adler, M. Pellauer, A. Parashar, A. Mithal, and J. Emer. Leveraging Latency-Insensitivity to Ease Multiple FPGA Design. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, 2012.

[FF02] R. B. Findler and M. Felleisen. Contracts for Higher-Order Functions. *Int'l Conf. on Functional Programming (ICFP)*, 2002.

[FNT15] M. Faldborg, T. L. Nielsen, and B. Thomsen. Type Systems and Programmers: A Look at Optional Typing in Dart. *Master's Thesis, Aalborg University*, 2015.

[FWC+18] X. Fan, D. Wu, W. Cao, W. Luk, and L. Wang. Stream Processing Dual-Track CGRA for Object Inference. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, Jun 2018.

[FZ03] S. Fine and A. Ziv. Coverage Directed Test Generation for Functional Verification using Bayesian Networks. *Design Automation Conf. (DAC)*, Jun 2003.

[Git24] cocotb. Online Webpage, 2024 (accessed Feb 23rd, 2024). `https://github.com/cocotb/cocotb`.

[Han24] P. Hanrahan. Magma. Online Webpage, 2024 (accessed Feb 23rd, 2024). `https://github.com/phanrahan/magma/`.

[HIT+13] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu. Elastic CGRAs. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, Feb 2013.

[HLM+16] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2016.

[HS07] T. A. Henzinger and J. Sifakis. The Discipline of Embedded Systems Design. *IEEE Computer*, 2007.

[iee09] IEEE Standard VHDL Language Reference Manual. Online Webpage, 2009. `https://ieeexplore.ieee.org/document/5981354`.

[iee17] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. Online Webpage, 2017. `https://ieeexplore.ieee.org/document/8299595`.

[IKL+17] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2017.

[JB99] J. Jennings and E. Beuscher. Verischemelog: Verilog Embedded in Scheme. *Conf. on Domain-Specific Languages (DSL)*, Oct 1999.

[JOP+21] S. Jiang, Y. Ou, P. Pan, K. Cheng, Y. Zhang, and C. Batten. PyH2: Using PyMTL3 to Create Productive and Open-Source Hardware Testing Methodologies. *IEEE Design Test*, 38:53–61, Apr 2021.

[JPOB20] S. Jiang, P. Pan, Y. Ou, and C. Batten. PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification. *IEEE Micro*, 40:58–66, May 2020.

[Kin76] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 1976.

[KMK+18] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanović. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2018.

[KZVT17] M. Khazraee, L. Zhang, L. Vega, and M. B. Taylor. Moonwalk: NRE Optimization in ASIC Clouds. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 2017.

[LCH+19] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, Feb 2019.

[Leh23] J. Lehtosalo. Mypy - Optional Static Typing for Python. Online Webpage, 2017 (accessed Dec., 2023). `http://mypy-lang.org`.

[LGW+22] H. Liew, D. Grubb, J. Wright, C. Schmidt, N. Krzysztofowicz, A. Izraelevitz, E. Wang, K. Asanović, J. Bachrach, and B. Nikolić. Hammer: A Modular and Reusable Physical Design Flow Tool. *Design Automation Conf. (DAC)*, Jul 2022.

[LKK+18] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2018.

[LSC10] C.-H. Li, S. Sonalkar, and L. P. Carloni. Exploiting Local Logic Structures to Optimize Multi-Core SoC Floorplanning. *Design Automation Conf. (DAC)*, 2010.

[Ltd11] A. Ltd. AMBA AXI and ACE Protocol Specification, 2011.

[LW21] A. Li and D. Wentzlaff. PRGA: An Open-Source FPGA Research and Prototyping Framework. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, 2021.

[LWC$^+$16] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtić, S. Bailey, M. Blagojević, P.-F. Chiu, R. Avižienis, B. Richards, J. Bachrach, D. Patterson, E. Alon, B. Nikolić, and K. Asanović. An Agile Approach to Building RISC-V Microprocessors. *IEEE Micro*, Mar 2016.

[LZB14] D. Lockhart, G. Zibrat, and C. Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[MB08] L. D. Moura and N. Bjørner. Z3: an Efficient SMT Solver. *Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Mar 2008.

[Mig23] Migen: A Python Toolbox For Building Complex Digital Hardware. Online Webpage, 2013 (accessed Dec., 2023). `https://m-labs.hk/gateware/migen/`.

[MMB$^+$18] C. Mattarei, M. Mann, C. Barrett, R. G. Daly, D. Huff, and P. Hanrahan. CoSA: Integrated Verification for Agile Hardware Design. *Formal Methods in Computer Aided Design (FMCAD)*, Oct 2018.

[MMI14] A. M. Maidl, F. Mascarenhas, and R. Ierusalimschy. Typed Lua: An Optional Type System for Lua. *Workshop on Dynamic Languages and Applications (DYLA)*, 2014.

[myh14] MyHDL: From Python to Silicon. Online Webpage, 2014 (accessed Oct 1, 2014). `http://www.myhdl.org`.

[NAT$^+$20] 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. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2020.

[NdAS23] R. Nigam, P. H. A. de Amorim, and A. Sampson. Modular Hardware Design with Timeline Types. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2023.

[NGLS24] R. Nigam, E. Gabizon, E. Lam, and A. Sampson. Correct and Compositional Hardware Generators. *cs.PL arXiv:2401.02570*, Jan 2024.

[Nik04] N. Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High-Level Specifications. *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, Jun 2004.

[NM15] M. Naylor and S. Moore. A Generic Synthesisable Test Bench. *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, Sep 2015.

[NTLS21] R. Nigam, S. Thomas, Z. Li, and A. Sampson. A Compiler Infrastructure for Accelerator Generators. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 2021.

[NZP12] A. Nahir, A. Ziv, and S. Panda. Optimizing Test-Generation to the Execution Platform. *Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2012.

[ORM22] A. Olofsson, W. Ransohoff, and N. Moroze. A Distributed Approach to Silicon Compilation. *Design Automation Conf. (DAC)*, Jul 2022.

[PB23] P. Pan and C. Batten. Formal Verification of the Stall Invariant Property for Latency-Insensitive RTL Modules. *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, Sep 2023.

[PGW$^+$20] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, T. Guarino, A. Joshi, M. Oskin, and M. B. Taylor. BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro*, May 2020.

[PJOB23] P. Pan, S. Jiang, Y. Ou, and C. Batten. Symbolic Elaboration: Checking Generator Properties in Dynamic Hardware Description Languages. *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, Sep 2023.

[POJB23] P. Pan, Y. Ou, S. Jiang, and C. Batten. The Case for Gradually Typed Hardware Description Languages. *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE)*, Mar 2023.

[py23] Python 3.7 Documentation - Abstract Syntax Trees. Online Webpage, 2021 (accessed Dec., 2023). `https://docs.python.org/3.7/library/ast.html`.

[RKJ08] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid Types. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2008.

[rvv23] RISC-V Verification Interface. Online Webpage, 2023. `https://github.com/riscv-verification/RVVI`.

[SAB10] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask). *IEEE Symposium on Security and privacy*, 2010.

[SGS$^+$16] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing through Selective Symbolic Execution. *Network and Distributed System Security Symposium*, 2016.

[SMBS06] S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla. Validating Families of Latency Insensitive Protocols. *IEEE Trans. on Computers (TC)*, 55(11):1391–1401, Nov 2006.

[SMT$^+$11] C. Salama, G. Malecha, W. Taha, J. Grundy, and J. O'Leary. Static Consistency Checking for Verilog Wire Interconnects. *Higher-Order and Symbolic Computation*, 2011.

[Spi23] SpinalHDL. Online Webpage, 2013 (accessed Dec., 2023). `https://spinalhdl.github.io/SpinalDoc-RTD/`.

[ST06] J. G. Siek and W. Taha. Gradual Typing for Functional Languages. *Scheme and Functional Programming Workshop (SFP)*, Sep 2006.

[ST07] J. G. Siek and W. Taha. Gradual Typing for Objects. *European Conf. on Object-Oriented Programming (ECOOP)*, Jul 2007.

[SVCB15] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. *Summit on Advances in Programming Languages (SNAPL)*, 2015.

[SWD+12] O. Shacham, M. Wachs, A. Danowitz, S. Galal, J. Brunhaver, W. Qadeer, S. Sankaranarayanan, A. Vassilev, S. Richardson, and M. Horowitz. Avoiding Game Over: Bringing Design to the Next Level. *Design Automation Conf. (DAC)*, Jun 2012.

[SZP+20] A. Soorishetty, J. Zhou, S. Pal, D. Blaauw, H.-S. Kim, T. Mudge, R. Dreslinski, and C. Chakrabarti. Accelerating Linear Algebra Kernels on a Massively Parallel Reconfigurable Architecture. *Int'l Conf. on Acoustics Speech and Signal Processing*, May 2020.

[Tay18a] M. B. Taylor. BaseJump STL: SystemVerilog Needs a Standard Template Library for Hardware Design. *Design Automation Conf. (DAC)*, Jun 2018.

[Tay18b] M. B. Taylor. Basejump STL: SystemVerilog Needs a Standard Template Library for Hardware Design. *Design Automation Conf. (DAC)*, 2018.

[TAZ+21] C. Tan, N. B. Agostini, J. Zhang, M. Minutoli, V. G. Castellana, C. Xie, T. Geng, A. Li, K. Barker, and A. Tumeo. OpenCGRA: An Open-Source Unified Framework for Modeling, Testing, and Evaluating CGRAs. *Int'l Conf. on Application-Specific Systems, Architectures, and Processors (ASAP)*, Jul 2021.

[TFG+16] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is Sound Gradual Typing Dead? *Annual Symp. on Principles of Programming Languages (POPL)*, Jan 2016.

[TGC+20] X. Tang, E. Giacomin, B. Chauviere, A. Alacchi, and P.-E. Gaillardon. OpenFPGA: An Open-Source Framework for Agile Prototyping Customizable FPGAs. *IEEE Micro*, Jul 2020.

[TH19] L. Truong and P. Hanrahan. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. *Summit on Advances in Programming Languages (SNAPL)*, May 2019.

[THF08] S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. *Annual Symp. on Principles of Programming Languages (POPL)*, 2008.

[THFF+17] S. Tobin-Hochstadt, M. Felleisen, R. Findler, M. Flatt, B. Greenman, A. M. Kent, V. St-Amour, T. S. Strickland, and A. Takikawa. Migratory Typing: Ten Years Later. *Summit on Advances in Programming Languages (SNAPL)*, May 2017.

[THS+20] L. Truong, S. Herbst, R. Setaluri, M. Mann, R. Daly, K. Zhang, C. Donovick, D. Stanley, M. Horowitz, C. Barrett, and P. Hanrahan. fault: A Python Embedded Domain-Specific Language for Metaprogramming Portable Hardware Verification Components. *Int'l Conf. on Computer-Aided Verification (CAV)*, Jul 2020.

[TOJ+19] C. Tan, Y. Ou, S. Jiang, P. Pan, C. Torng, S. Agwa, and C. Batten. PyOCN: A Unified Framework for Modeling, Testing, and Evaluating On-Chip Networks. *Int'l Conf. on Computer Design (ICCD)*, 2019.

[TPO+21] C. Torng, P. Pan, Y. Ou, C. Tan, and C. Batten. Ultra-Elastic CGRAs for Irregular Loop Specialization. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2021.

[TXL+20] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo. OpenCGRA: An Open-Source Unified Framework for Modeling, Testing, and Evaluating CGRAs. *Int'l Conf. on Computer Design (ICCD)*, Oct 2020.

[VBP+16] A. Vasilyev, N. Bhagdikar, A. Pedram, S. Richardson, S. Kvatinsky, and M. Horotiwz. Evaluating Programmable Architectures for Imaging and Vision Applications. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2016.

[VKSB14] M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and Evaluation of Gradual Typing for Python. *Symp. on Dynamic Languages*, Oct 2014.

[WaK97] D. C. Wang, A. W. appel, and J. L. Korn. The Zephyr Abstract Syntax Description Language. *Conf. on Domain-Specific Languages (DSL)*, Oct 1997.

[Wij16] V. M. Wijayasekara. *Equivalence Verification for NULL Convention Logic and Latency-Insensitive Circuits*. Ph.D. Thesis, Department of Electrical and Computer Engineering, North Dakota State University, 2016.

[WLP+14] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The Architecture and Design of a Database Processing Unit. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2014.

[XYT+22] Y. Xu, Z. Yu, D. Tang, G. Chen, L. Chen, L. Gou, Y. Jin, Q. Li, X. Li, Z. Li, J. Lin, T. Liu, Z. Liu, J. Tan, H. Wang, H. Wang, K. Wang, C. Zhang, F. Zhang, L. Zhang, Z. Zhang, Y. Zhao, Y. Zhou, Y. Zhou, J. Zou, Y. Cai, D. Huan, Z. Li, J. Zhao, Z. Chen, W. He, Q. Quan, X. Liu, S. Wang, K. Shi, N. Sun, and Y. Bao. Towards Developing High Performance RISC-V Processors Using Agile Methodology. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2022.

[XZP$^+$20] Y. Xiong, J. Zhou, S. Pal, D. Blaauw, H.-S. Kim, T. Mudge, R. Dreslinski, and C. Chakrabarti. Accelerating Deep Neural Network Computation on a Low Power Reconfigurable Architecture. *Int'l Conf. on Circuits and Systems (ISCAS)*, Oct 2020.

[ZANA22] J. Zhao, A. Agrawal, B. Nikolic, and K. Asanović. Constellation: An Open-Source SoC-Capable NoC Generator. *International Workshop on Network on Chip Architectures (NOCARC)*, Oct 2022.