Unlocking Design Reuse with Hardware Compiler Frameworks

Adam Izraelevitz
Jonathan Bachrach, Ed.
Krste Asanović, Ed.
Simon Schleicher, Ed.
Jonathan Ragan-Kelley, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-168
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-168.html

December 5, 2019

Unlocking Design Reuse with Hardware Compiler Frameworks

by

Adam Izraelevitz

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Adjunct Professor Jonathan Richard Bachrach, Co-chair
Professor Krste Asanović, Co-chair
Professor Simon Schleicher
Professor Jonathan Ragan-Kelley

Fall 2019

Unlocking Design Reuse with Hardware Compiler Frameworks

Abstract

Unlocking Design Reuse with Hardware Compiler Frameworks

by

Adam Izraelevitz

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Adjunct Professor Jonathan Richard Bachrach, Co-chair

Professor Krste Asanović, Co-chair

Emerging applications from the edge to the cloud are constantly increasing demand for energy efficient and performant computation. While specialized hardware can meet these power and performance goals, the high non-reoccurring engineering (NRE) costs of designing, testing, and verifying custom hardware severely hinders its supply. Hardware construction languages such as Chisel enable hardware designers to write parameterized hardware libraries which increase design reuse by turning NRE effort into reusable solutions for future specialized chips. This thesis introduces FIRRTL, Chisel's hardware compiler framework, which enables automatic and custom RTL-transformations including logic optimization and design instrumentation. In addition, this thesis proposes an aspect-oriented-inspired paradigm, Colla-Gen, as a mechanism to improve design collateral reuse (e.g. physical design floorplanning or verification instrumentation), which forms another large portion of chip NRE costs.

To my parents David and Terry, who taught me to look forward.

To my brothers Joe and Jacob, who taught me to appreciate the moment.

To my partner Christine, who taught me that things outside my thesis matter most of all.

Contents

List of Figures

List of Tables

Acknowledgments

This dissertation could not have been conceived, achieved, nor written without the help of many of the faculty and students in the ASPIRE/ADEPT lab, and the University of California, Berkeley, as well as many of my own personal friends and family whose support was immense.

I would like to thank the following individuals in particular for their contributions to the Chisel and FIRRTL ecosystem, upon which this entire thesis is built:

- Patrick Li: for designing and implementing the first prototype of the FIRRTL language, writing the initial specification for the language, and jointly writing the current specification for the language. Thank you for your constant mentorship and belief in me; I am immensely grateful for our collaboration and friendship. I divide my PhD into three parts: before, during, and after Patrick. You are a landmark in my life.

- Jack Koenig: for numerous and substantial contributions to the FIRRTL and Chisel projects including asynchronous reset, parameterized black boxes, ANTLR parser, Protobuf parser, dontTouch annotation, refactoring annotations, Travis+Yosys continuous integration, improving optimization passes, improving compiler speed and memory usage, improving the ExpandWhens pass, reworking name uniquification, mappers implementation, clock/reset scoping, the record type, printable, along with many bugfixes and community/governance/outreach contributions. Thank you for being such an amazing collaborator full of new ideas, endless motivation, and uplifting spirit. Keeping your contributions open source is a difficult yet often thankless job: thank you. Finally, thank you for teaching me that projects could be shared and are better for it, as well as making me a better programmer (given you review most of my code, I'm not sure that was purely selfless...).

- Edward Wang: for collaborating with me on your Hammer project and the floorplan DSL, as well as the MixedVec implementation and significant contributions to the bootcamp and other documentation. Our conversations about collaboration challenged, clarified and improved my own views about the subject. Thank you for teaching me so much about how to be a good mentor, friend, and listener; thank you also for listening to my good advice and discarding my bad advice.

- Schuyler Eldridge: for numerous and substantial contributions to the FIRRTL and Chisel projects including stage refactoring, dependency API, improving wiring transform, and library registration; for running Chisel outreach and massive improvements to the website. You are a selfless collaborator both in the time you contribute and the ideas you share. Once you grab on you never let go, and your ability to identify and embrace what you enjoy serves as an inspiration to me when I'm discouraged.

- Chick Markley: for numerous and substantial contributions including the FIRRTL interpreter, Treadle, Chisel intervals and fixed-point implementations, the logger, the

load memory feature, creating the new testing harness, porting the Chisel tutorial, overseeing the GUI and visualizer along with many bug-fixes and community/governance/outreach contributions. Thank you for your consistent work ethic yet easy-going attitude; it often kept me focused while still seeing life's big picture.

- Albert Magyar: for numerous and substantial contributions to the FIRRTL and Chisel projects including the FPGA memory transform and data collection, combinational loop detection, specification clarifications, improving constant propagation, digraph library, FIRRTL memory utilities, and clone module. It was wonderful being your lab partner in class and life. Your unique perspective on complex problems always helped clarify my research direction. Thank you for the unwavering support in both the technical and emotional sides of my PhD.

- Richard Lin: for numerous and substantial contributions to Chisel including improving the Chisel 3.0 code base for release quality, improving testing infrastructure, significant bootcamp contributions, bundle literals, naming annotations, autoclonetype, bindings refactor, module hierarchy refactor, Chisel ranges, along with many bug-fixes and community/governance/outreach contributions. Thank you for your unwavering belief in, love of, and sacrifices for this project, and for teaching me that a user's experience is paramount.

- Jim Lawson: for numerous and substantial contributions to Chisel and FIRRTL including improving the Chisel 3.0 code base for release quality, continuous integration features, compatibility mode, code releases, Verilog front-end, along with many bug-fixes and community/governance/outreach contributions. Thank you for your willingness to do the unglamorous tasks with consistent enthusiasm and completeness; I will always appreciate your humor, which is rarely shown but treasured when found.

- Andrew Waterman: for contributions to the design of FIRRTL's constructs and Chisel 3.0, porting architecture research infrastructure, and FIRRTL optimization passes. Thank you for your eager and excellent advice, but more personally for your willingness to place your trust into me as a new and inexperienced graduate student.

- Henry Cook: for porting and cleaning up many aspects of Chisel 3.0, including the testing infrastructure and the parameterization library. Thank you for your early mentorship and for providing me with an amazing graduate student role model.

- Angie Wang: for your contributions to FIRRTL including the ReplSeqMem transforms and memory utilities, as well as your original insights into interval analysis. Your encouragement of my values regarding team social dynamics allowed me to trust my own voice and be a better advocate for my colleagues.

- Paul Rigge: for your contribution and leadership in the Chisel bootcamp project, as well as your significant contributions to our collaboration in DSP tools and interval

analysis. Your willingness to collaborate and discuss any topic with a deep and measured perspective made our conversations uniquely valuable and inspirational.

- Donggyu Kim: for your contributions to the FIRRTL project including the latency pipe memory transformation, the inference of readwrite memory ports, and memory utility library contributions. Thank you also for our early Strober collaboration and the passion, drive, and hard-working spirit you bring to your work and collaborations; you take the "make it work" mantra to a new level.

- Stephen Twigg: for insights into the design of FIRRTL, and for many foundational improvements to the Chisel language. Thank you for challenging my conceptions about hardware intermediate representations, educating me in proper software development practices and the Scala language. Most importantly, thank you for teaching me to listen, stay humble, and reconsider my actions.

- Stevo Bailey: for contributions to the bootcamp, the CRAFT project, and for your advocacy of an agile design methodology. Thank you for bringing such a positive spirit and focused work ethic to our collaborations.

- Albert Chen: for contributions to the FIRRTL project including the clarification and implementation of the renaming algorithm, as well as implementing Verilog comments. I appreciate your reserved approach of active listening and precision of words, which kept our collaboration on topic, productive, and efficient.

- John Buchan: for the original contribution of the context dependent environment parameterization implementation. Our collaboration was one of my first attempts at research, and your willingness to mentor me and share your ideas set an excellent example for an impressionable young PhD student.

- Chris Celio: for converting BOOM to Chisel 3.0 and collecting data for motivating the project through code reuse. You were always by my side (figuratively and literally) and your mix of passion and perspective that life exists outside of a PhD served as an excellent role model that I have tried to emulate.

The road taken throughout a PhD takes many turns, and I've often relied on the support of my colleagues for both their technical expertise as well as their compassion. I'd like to additionally thank Orianna Demasi, Michael Driscoll, Yunsup Lee, Lisa Wu, Sarah Bird, Scott Beamer, Palmer Dabbelt, Eric Love, Martin Maas, David Biancolin, Ben Keller, Jenny Huang, Arya Reais-Parsi, Nathan Pemberton, John Wright, Kevin Laeufer, Howard Mao, Ameer Haj Ali, Alon Amid, David Bruns-Smith, Hasan Genc, Abraham Gonzalez, Sagar Karandikar, Ben Korpan, Dayeol Lee, Albert Ou, Jerry Zhao, Brendan Sweeney, Kyle Kovacs, and Danny Tang. You all made my work a second home through your willingness to commiserate in our failures and celebrate in our successes.

I would not be here without such consistent and excellent mentorship to help me navigate the confusing, non-linear, and difficult trek through academia. Mike Hamada and Christine Anderson-Cook took me under their wings at Los Alamos National Laboratory and surprised an intimidated high school student with their unwavering encouragement of my creativity - I learned from you that creative projects spark passion, and rules (while not made to be broken) are certainly made to be questioned. David Albonesi and Paula Petrica gave me the opportunity to work in their lab at Cornell, and with them I found a collaboration which valued my contributions, a mentorship which gave me the perfect mix of freedom and guidance, and opened my eyes to the beautiful world of computer architecture and hardware design methodologies.

My path through Berkeley was helped through by many staff and faculty. Ria Briggs, Tamille Chouteau, Kostadin Ilov, Roxana Infante, and Shirley Salanio are amazing administrators who constantly tackled logistical, bureaucratic, and technical obstacles to create a distraction-free and productive environment for graduate student research; I have zero doubts that my PhD would have been significantly longer and more painful without their constant and timely interventions. Borivoje Nikolić was an unofficial advisor to me and fully embraced both the Chisel ecosystem and my contributions to it. Jonathan Ragan-Kelley and Simon Schleicher provided me with novel and useful perspectives on the impact of my research and the choices I made when designing the FIRRTL compiler. To my secondary advisor, Krste Asanović: you are a man of few words but each is packed with meaning. Thank you for the advice - the technical but more importantly, the personal.

This thesis would not have been possible without the continuous feedback and encouragement of my primary advisor Jonathan Bachrach. His leadership in vision and implementation of Chisel's ecosystem has been constant from the start. Thank you for believing in me, investing in me, and giving me the room to grow (and for encouraging me to take a programming languages class).

While on paper a PhD is a solitary achievement, in reality it is the culmination of decades of personal investment into me as a person by those individuals who care and sacrifice the most. To my friends outside of Soda Hall, thank you for the closeness when I needed it, the space when I didn't, and the support always. To my friend and lab partner Colin Schmidt: you are an extraordinary individual with equal parts passion and compassion. Your single focus towards impact coupled with your love of humanity continues to be an inspiration. Keep on trucking; your friends and family are here for you.

To my parents David and Terry, brothers Jacob and Joe, sisters-in-law Leah and Lauren (and nephew Nathan): what a family. Your support for me on the sidelines has kept me going when the tank was empty. I am eagerly looking forward to our family's journey together through more of life's joys and challenges.

To my partner Christine: you are witness to my darkest and brightest moments, and have stood proud anyways. I love you dearly; thank you for always helping us build a joyous life - a life I am so privileged to share with you.

Chapter 1

Introduction

The end of Dennard scaling and the slowing of technology advances are eliminating the associated “free" power, performance, and area improvements for digital circuits. Since specialized hardware implementations have enormous energy and performance improvements over software on a general-purpose processor, specialization is hardware design's path forward for the foreseeable future. This trend manifests in an increasing demand for diverse products containing different specialized digital logic descriptions, also known as RTL (register-transfer-level). Meeting this demand with existing methodologies is difficult.

In contrast, the software industry has faster design cycles than the hardware industry; a small team can go from idea to profitable software in under two weeks. *What can the hardware industry learn from the software community?*

A key contributor to software industry's productivity is reusable libraries, which amortize development and verification costs of new applications. These libraries are built upon expressive languages with retargetable compilers that perform platform-specific optimizations on general-purpose code.

In comparison, hardware reuse is relatively rare; no widely-used reusable hardware library exists. However, if hardware projects reused more code, engineers might spend less time designing and, more importantly, less time verifying the new design. Since the benefits of reusing code are clear, *why don't hardware engineers write reusable libraries?*

This thesis attempts to answer this question, as well as describe the infrastructure necessary to develop reusable hardware libraries.

1.1 The Trend of Hardware Specialization

Novel applications are discovered daily which demand faster computation and lower power requirements. To meet this need, the hardware community proposes specializing hardware to the demands of the application; instead of accelerating general-purpose computation, companies design processors which quickly and efficiently execute a single, specific application (specialization). For example in autonomous driving, a car must quickly recognize

obstacles such as pedestrians in their path - this low latency, high complexity calculation requires significant computation embedded within the car. Enhancing a car with a specialized image-processing chip running image-processing code meets these requirements; the chip's lack of general-purpose usability is immaterial given the restricted application domain. However, the market's relatively low volume demand of per-application chips restricts hardware manufacturers because they cannot offset their custom chip designing and engineering costs through selling high volumes of that chip.

From Idea to Product

Prior to specialization, a company addressed an application by selling customized products containing commodity general-purpose chips. This previous era enabled large chip companies such as Intel to design a single generic chip that was manufactured in bulk and sold to a wide variety of per-application companies who integrated the chip within their product. Today in the post-Moore's Law age of computing, this business model breaks down because one general-purpose chip is neither fast enough nor efficient enough to address the full spectrum of modern application demands.

Without this market for general-purpose chips, application companies must accommodate the custom chip design effort within their process of transitioning from idea to product. Now, a company undergoes the following steps where the time, effort and cost of designing a new specialized chip directly delays the time-to-market:

1. Identify new application
2. Design new algorithm
3. Write software for new algorithm
4. Design hardware that runs that software quickly[1]
5. Produce chip, deploy design, sell product

Using current methods to design, verify, and produce a chip, there are staggering amounts of non-reusable-engineering (NRE) costs (e.g. the engineers' salaries). While previously chip companies amortized the large NRE cost bottleneck by selling millions of identical chips and reducing the per-chip NRE cost, the current market lacks sufficient demand to sufficiently amortize this cost. **In the age of specialization, the per-chip NRE cost becomes the dominating cost of producing a specialized chip**.

[1] Ideally hardware/software are designed in tandem

1.2 Reducing Per-Chip NRE Costs

Writing reusable designs, rather than one-off designs, allows greater amortization of NRE costs into recurring engineering costs. This approach requires a shift in thinking from traditional hardware engineers; in addition to designing their product, engineers consider how their design is reused in a future design. When a higher percentage of their work is directly usable in future designs, NRE cost of the next chip is reduced. Over time, the cost of developing and verifying a design is amortized over each future usage. In summary, the goal of designing a chip is to finish a design as well as contribute to a reusable chip-design ecosystem.

Suppose an existing design almost solves a new need; rather than starting afresh or copy-paste-modifying (both of which create new designs and restart the NRE-amortization), an engineer "grows" the existing design to address their new use case and preserve existing use cases; all existing designs are then updated to the new version. This approach reduces the growth of code whose maintenance, design, and verification costs contribute to the total NRE cost, while constant design sharing continuously amortizes the original NRE effort. Ideally, bringing up a new chip uses existing designs, existing verification infrastructure, and existing physical designs. Any new code an engineer writes is refactored and contributed back to the ecosystem to be used by others.

Recently, the hardware industry has increased its reuse of large complex custom IP blocks at the system-on-chip (SoC) level, which has had many benefits including faster time to market and reduced verification effort. However, custom IP blocks are usually very specialized, as opposed to being basic building blocks of hardware like queues, arithmetic units, multipliers, caches, and so on, and pose more integration challenges than a typical reusable library.

Take an analogy to software - if an open source library almost addresses a new use case, upstreaming a new API (application programming interface) or feature increases the power of the library, while rewriting the library or modifying it without contributing back increases the total amount of code to design and verify. Analogous to hardware IP (intellectual property) blocks, hand-optimized assembly routines are reusable but limited in scope, which distinguishes them from these wide-ranging and fully featured libraries. In short, reduce NRE costs by designing hardware and software for reuse and growing their codebases with new features.

A large majority of the NRE costs of designing a chip are in the verification and physical design of the chip, not just the digital logic of the hardware. Here is where the previous analogy to software breaks down. First, most software requires significantly less verification than hardware, as software bugfixes are relatively cheap to deploy. Secondly, compilers successfully increase code reusability by isolating code from computational platforms. Thirdly, slow but correct code can be iteratively improved upon via updates, making software's initial performance a lesser concern.

Hardware is different. Bugs are extremely costly to fix, if downright impossible. Most

hardware is very coupled to the underlying technology, exacerbating the lack of reusability. Finally, the production of chips is so expensive that slow chips are effectively useless and companies must invest significantly into early optimization of their design.

Verification infrastructure and physical design tools require additional design collateral which must also be reused. A hardware design is not simply the digital logic description of a chip; rather the design includes the digital logic + verification tests/infrastructure + simulation mapping collateral + physical design scripts + input/output cell placements + SRAM macro compiler calls + etc.

As demonstrated in Chapter 2, the existing flaws of verification technologies, design languages, and physical design tools limit the degree of design reusability; an engineer cannot write reusable designs given these tools. For example, the parameters supported by a design language directly influence a design's reusability; if a desired parameterization is not supported, then the code must be duplicated and modified. Another example is how the mixing of concerns within a codebase reduces its reuse potential. If a verification flow implicitly depends on a transistor technology, then every future design that uses a different transistor technology cannot use this flow, regardless of other design similarities.

Take another analogy to software - if a library is written only using x86 assembly, it cannot run on an ARM machine. Similarly, a design that directly instantiates 14nm SRAMs can never be reused in a 7nm process.

The existing software paradigm of aspect-oriented programming is well-suited as inspiration to solve the problem of reusing hardware design collateral.

Software aspect-oriented programming (AOP) is not widely adopted because the benefits of reusing software code collateral do not offset the downsides of aspect-oriented programming. In contrast, the degree of necessary hardware design collateral significantly improves the overall outcome of applying an AOP-inspired approach; the benefits from reusing design collateral outweigh the added downsides. In fact, the hardware design collateral problem is so severe that many predominant physical design and verification flows are aspect-oriented in spirit; if rewritten with a consideration of aspect-oriented programming, these flows would enable clearer semantics, more reuse, and more powerful features. In all, introducing an aspect-oriented approach brings reusability to hardware design collateral and enables a healthy chip design ecosystem.

1.3 Summary of Research Contributions

This thesis contributes the following:

- Chapter 1: An introduction to hardware specialization, non-reusable engineering costs and its impact on hardware design reusability
- Chapter 2: Four hypotheses accounting for the stagnation of hardware library development, as well as an overview of the Hammer framework which enables the abstraction of physical design concerns

- Chapter 3: An introduction and evaluation of Chisel, a hardware construction language which enables the ability to express reusable hardware designs. This chapter includes the following: (1) an analysis of existing hardware description languages and their flaws, (2) an analysis of Chisel as a primary tool for hardware libraries, and (3) an evaluation of Chisel's support for hardware libraries

- Chapter 4: A complete description of FIRRTL, a new hardware compiler framework and intermediate platform-agnostic representation, and how it isolates the digital logic design from the underlying physical design/verification concerns through an ecosystem of automatic digital logic transformations. This chapter includes the following: (1) analysis of LLVM, an existing software compiler framework; (2) introduction of FIRRTL's intermediate representation (IR); (3) description of value inference (e.g. widths) and its implementation; (4) support for arbitrary metadata throughout the compilation process; (5) the mechanisms for transforms to inspect and modify a design; (6) a description of interesting transformations; (7) an evaluation of the FIRRTL compiler framework

- Chapter 5: An introduction of Colla-Gen, an AOP-inspired approach as a user-facing language to express reusable design collateral, thus enabling an ecosystem of design collateral libraries. This chapter includes the following: (1) a detailed description of aspect-oriented programming (AOP), an analysis of its flaws, and an argument for its consideration within a hardware context; (2) an introduction to Colla-Gen illustrated with a physical design floorplanning example; (3) Colla-Gen's implementation and additional reusable libraries for generating design collateral are presented and discussed

- Chapter 6: A conclusion on the results of this thesis and a discussion of future directions of this research

- Appendix: A full specification of the FIRRTL intermediate representation

1.4 Previous Publication, Collaboration, and Funding

Some of the content and figures in this thesis are adapted from previous paper submissions that are the result of multiple collaborations. While the majority of the research content, design, and implementation was done by myself, other collaborators made direct contributions to content discussed in this thesis. The following details their contributions.

The first submission is "Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations" [23]. Jack Koenig improved FIRRTL's optimization passes and provided its evaluation in Chapter 4. Patrick Li contributed the idea of simplification transformations and a preliminary design of the FIRRTL language, the final form of which is included in the Appendix. Richard Lin, Chick Markley, and Jim Lawson contributed heavily to improvements to Chisel, the hardware construction language analyzed

in this project. Angie Wang contributed a version of the memory-replacement transformation and its evaluation in Chapter 4. Albert Magyar contributed the double-pumped FPGA memory transformation and its evaluation in Chapter 4. Donggyu Kim contributed the decoupling and snapshotting transformations and their evaluation in Chapter 4, as well as collecting the data about RocketChip and OpenPiton source code used in Chapter 3. Colin Schmidt contributed the details of the case study and coverage results in Chapter 4. Krste Asanović and Jonathan Bachrach provided guidance and feedback in all stages of the paper.

The second submission is "ACED: A hardware library for generating DSP systems" [45]. Angie Wang contributed the background of interval analysis in Chapter 4. Paul Rigge, Chick Markley, Jonathan Bachrach, and Borivoje Nikolić contributed to the theory, conception, and implementation of the interval analysis.

The third submission is "Hammer: Enabling Reusable Physical Design" [47]. Edward Wang contributed the first draft of the description of the Hammer project in Chapter 2 as well as being a major collaborator to the floorplan collateral generator described in Chapter 5. Colin Schmidt had multiple significant contributions to the Hammer project and paper, while Borivoje Nikolić, Elad Alon, and Jonathan Bachrach provided valuable feedback to the final draft of the paper.

Research partially funded by DARPA CRAFT HR0011-16-C-0052 and HR0011-12-2-0016; Intel Science and Technology Center for Agile Design; Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA; NSF-GRFP (DGE-1106400); ADEPT Lab industrial sponsors and affiliates Intel, Google, Siemens and SK Hynix; ASPIRE Lab industrial sponsors and affiliates Intel, Google, HPE, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung. The author would also like to acknowledge the sponsors, students, and faculty of the Berkeley Wireless Research Center, ADEPT Lab and ASPIRE Lab. Any opinions, findings, conclusions, or recommendations in this thesis are solely those of the author and do not necessarily reflect the position or the policy of the sponsors.

Chapter 2

An Analysis and Proposal for Hardware Design Methodologies

Existing solutions cannot reduce the per-chip NRE costs because they fail to sufficiently support a hardware design ecosystem. This chapter provides an analysis of current methodologies including hardware languages, compilers, and CAD tools, as well as outlines their flaws which inhibit effective reuse of hardware designs and increase non-recurring engineering costs. In addition, a cursory description is included of Hammer, a sister project which provides useful and reusable tool abstractions; however, keeping design collateral in-sync with a generated design remains a problem.

This chapter is organized through a discussion of the following three hypotheses that account for the stagnation of hardware library development (which are preceded by addressing alternative but incorrect hypotheses):

1. *Existing hardware description languages lack the expressivity to support hardware libraries*
2. *Diverse underlying implementations require RTL customization, limiting a design's reusability*
3. *Effective physical design, verification, emulation and instrumentation require additional design collateral which is too tool/platform/technology dependent and too brittle in face of design modifications*

2.1 Incorrect Hypotheses

Software libraries are pervasive in software development because, through code reuse, they reduce development and verification costs of new applications. Modern software relies on thousands of libraries—Ubuntu 14.04 has approximately 35,000 packages installed natively.

In direct comparison, hardware designers do not commonly reuse modules from project to project, let alone develop extensive and reusable libraries. As mentioned earlier, the

hardware industry has increased its reuse of large complex custom IP blocks at the SoC-level, but these custom IP blocks are specialized and pose more integration challenges than a typical reusable library. The amount of reusable hardware IP is a far cry from the ubiquity and usability of software libraries.

One could claim the lack of hardware libraries is from a lack of effort; yet in this author's experience, many companies try, but fail, to establish internal reusable libraries of hardware components.

One could also claim the lack of hardware libraries is from a lack of an open-source community; yet, popular open-source software is often written by one or two contributors. D3[7], the popular JavaScript visualization library, was primarily written by a single engineer, but has still seen widespread use.

2.2 Hypothesis 1—Existing HDLs lack expressivity

Programming languages have seen significant improvements since the 1980s when the majority of popular hardware description languages (HDLs) were designed (Verilog, VHDL). Modern advancements in mainstream programming languages have made languages like Java, C++, Python, Perl, and Ruby very powerful. Object-orientation, polymorphism, and higher-order functions enable the use of good software engineering principles like abstraction, separation of concerns, and modularity; these ultimately encourage and enable code reuse. HDLs have been very slow to adopt these paradigms.

An adder reduction tree illustrates this problem: Verilog and VHDL cannot express recursive generate statements, so a designer must manually unroll the loop and calculate indices for every instance. The lack of parameterization precludes re-use when a tree of different width is required.

Another example is a module that filters packets. With current HDLs, either the filter module or an external module must encode the filter condition. The first approach violates the principle of separation of concerns, while the second violates encapsulation. However, higher-order functions provide an elegant software engineering solution to the problem.

SystemVerilog, created in 2002, attempts to improve on existing HDLs by mixing in modern ideas like object-oriented programming with classic Verilog elements. The result is an extremely complicated language—intractable to support and confusing to learn—that is still missing other modern features like higher-order functions. To the author's knowledge, no commercial SystemVerilog compiler implements the entire specification.

High-level synthesis (HLS) takes a different approach by having the user design in a higher level language, with a compiler translating down to RTL. The input language can be C-like [50][31][11][8][44], a parallel C-like language [13][36][37], general purpose [2], or domain specific [18][20][32][21][33][22]. Many HLS tools are evaluated on simplicity of use, performance relative to a hand-coded implementation, succinctness, and resource footprint; their ability to foster reusable hardware libraries is not usually considered.

Unfortunately, HLS approaches suffer from two competing concerns: (1) a more expressive source language enables better software engineering (and thus more reusability); (2) a more expressive source language is more difficult to translate to hardware and creates more compilation/abstraction layers that the user must understand to fine-tune their design.

2.3 Hypothesis 2—Underlying complexity requires RTL customization

In spite of the success of logic synthesis, many underlying constraints still influence RTL design.

ASIC implementations often require RTL customizations. For example, Verilog lacks an explicit memory construct; users must use a register array. In modern technologies, SRAMs are provided by the fabrication company because large memories often contribute to a design's critical path, area, and power. RTL designers must rewrite their design to replace these register arrays with black-boxed SRAMs; this eliminates any future reuse that does not use this ASIC technology or performance envelope.

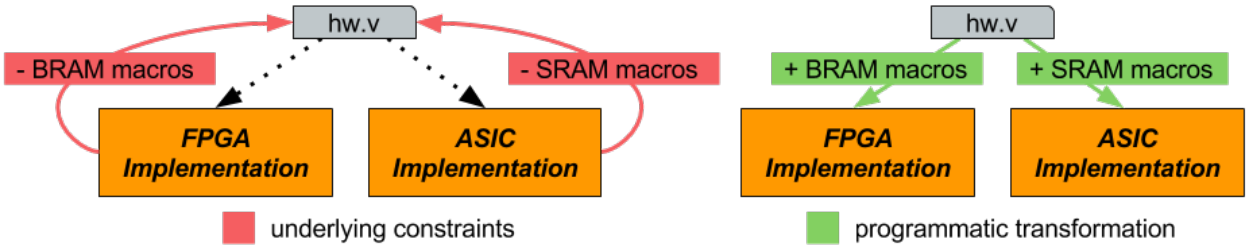

Figure 2.1: Underlying constraints for ASIC versus FPGA implementations means the same RTL cannot get good results on both platforms. This limits the reusability of any RTL design. To solve this problem, programmatic RTL transformations must take generic RTL and specialize it for a given platform.

FPGA implementations are no different; many FPGAs have hardened logic blocks to improve design quality. A designer can receive significant performance, power, or utilization advantages by modifying their RTL to be friendlier to a particular FPGA's synthesis tool. These changes, however, may be detrimental to an ASIC implementation or another FPGA implementation.

To solve this problem, some designers write a collection of custom scripts to do ad-hoc programmatic RTL modifications; these scripts are neither reusable, robust, nor composable.

Commercial CAD tools do not completely solve this problem either. While some do contain RTL-to-RTL transformations, CAD tools primarily focus on the separate problem of synthesis and place-and-route. Additionally, they are not organized in an open-source

compiler framework and are insufficient for custom flows that may have unsupported use cases.

One exception is Yosys[48], which is an open-source framework for Verilog RTL synthesis, and maps Verilog to ASIC standard cell libraries or Xilinx FPGAs. The main focus of Yosys is logic synthesis, not RTL to RTL transformations, and thus its internal design representation is very low level, and cannot represent higher-level constructs like aggregate types, width inference, and conditional assignment.

Separate from CAD tools, there exist stand-alone RTL modifiers, but many are closed source[6] and like commercial CAD tools cannot support custom flows. One exception is PyVerilog[42], which is an RTL-to-RTL modifier tailored specifically to Verilog. As such, it makes it difficult to act upon designer intent that is not directly represented in a Verilog construct. PyVerilog does not support SRAM inference or aggregate types, and these features would be very difficult to support given its internal circuit representation.

2.4 Hypothesis 3—Design collateral is necessary but not reusable

To obtain an ASIC implementation of a hardware design, its RTL must undergo many necessary and guided steps to obtain a tape-out-ready GDS. For example, one of these steps is place and route, the class of algorithms underlying physical design (physical realization of a logical design) which is generally considered to be NP-hard [40] [12]. As a result, designer intervention is often required to achieve high quality of results (QoR) within a reasonable timeframe. The dominant mode of designer intervention and interaction with CAD/EDA tools is through tool-specific TCL scripts which set constraints and run various physical design tasks in the tool. Place and route is one of many necessary-and-guided steps including standard cell synthesis, clock-tree synthesis, macro placement, IO cell selection and placement, and power strap specifications.

While existing CAD tools can perform all of these tasks, each step requires a significant amount of customization dependent on the following **orthogonal concerns**: the **logical design features** (SRAM size and number, signal fan-outs/fan-ins, bus bandwidths etc.), the **physical design features** (area constraints, shape constraints, desired frequency, floorplan) the **transistor technology** (restrictions on metal layers, SRAM macros and IO standard cells), and **CAD tools used** (specific placement directives, tool settings, flow structure).

Given the current status of interacting with EDA tools, the traditional approach towards "re-using" physical design effort is manual customization of tool vendor provided reference methodologies, essentially templates of physical design flow scripts e.g. To add a modicum of reuse, an approach is to use macro/string preprocessing, either within TCL itself or through other tools, to customize reference methodologies and generate tool scripts e.g. [35] [10] [14]. Macro/string preprocessing, however, has no awareness of the underlying physical design

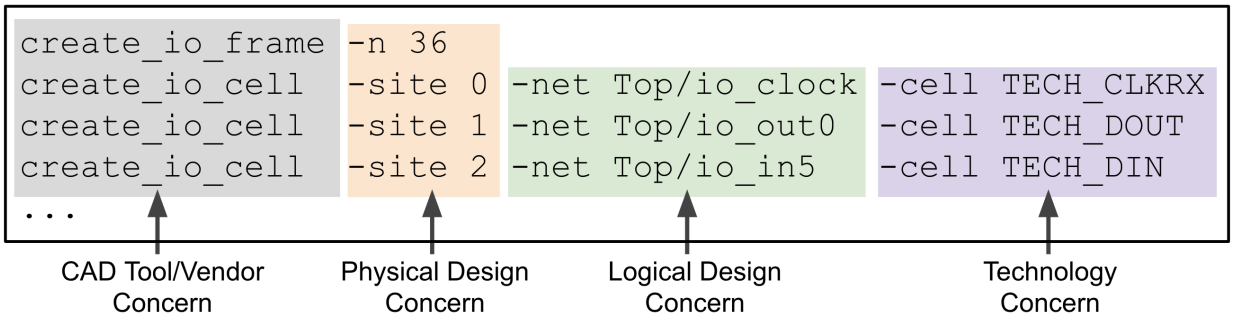

Figure 2.2: To create pad frames for an ASIC, the CAD tool expects to be told to create an IO frame. For each IO cell in the frame, one must use a tool-specific command to indicate its physical design-specific placement, its corresponding logical design-specific signal, and the technology-specific IO cell. Because this design collateral lacks any abstraction of concerns, these commands are impossible to reuse under technology, logical design, physical design, or CAD tool changes.

concepts; this approach gives rise to ad-hoc approaches which make it difficult to achieve safe programmatic re-use.

This lack of reusable design collateral is also present for both FPGA-implementations and efficient simulators. FPGA development requires many of the same steps as ASIC designs (synthesis, place and route) as well as requiring specific FPGA CAD tool directives in order to improve its clock frequency, LUT usage, and hard-block usage. Software simulators have an endless list of settings, flags, and debug features which improve debuggability, simulation speed, or both. Verification flows also require specific tools, commands, and other collateral in order to target the design (e.g. constrained-random criteria for improved coverage, or directives reducing the valid state-space for formal tools). All of this design collateral is heavily dependent on which tool or design used, regardless of function (e.g. feature-equivalent formal tools still employ unique API's which influence the associated design collateral).

Most manually created design collateral is contained within design tool scripts (see Figure 2.2). Because these scripts do not abstract these orthogonal **concerns**, changing one concern requires an entire rewrite of this design collateral (e.g. switching technologies, but preserving the tools, logical design and physical design). Figure 2.2 contains a snippet from the design collateral of a 28nm ASIC tapeout; every CAD tool command is directly dependent on every orthogonal concern, eliminating any reuse of this code if any one concern changes.

Design collateral is necessary for real products in ASIC, FPGA, simulator and verification flows. At the same time, it lacks abstraction and is manually written to a given design instance; this makes the design collateral non-reusable to changes in technology, design logic, physical design, or tools used.

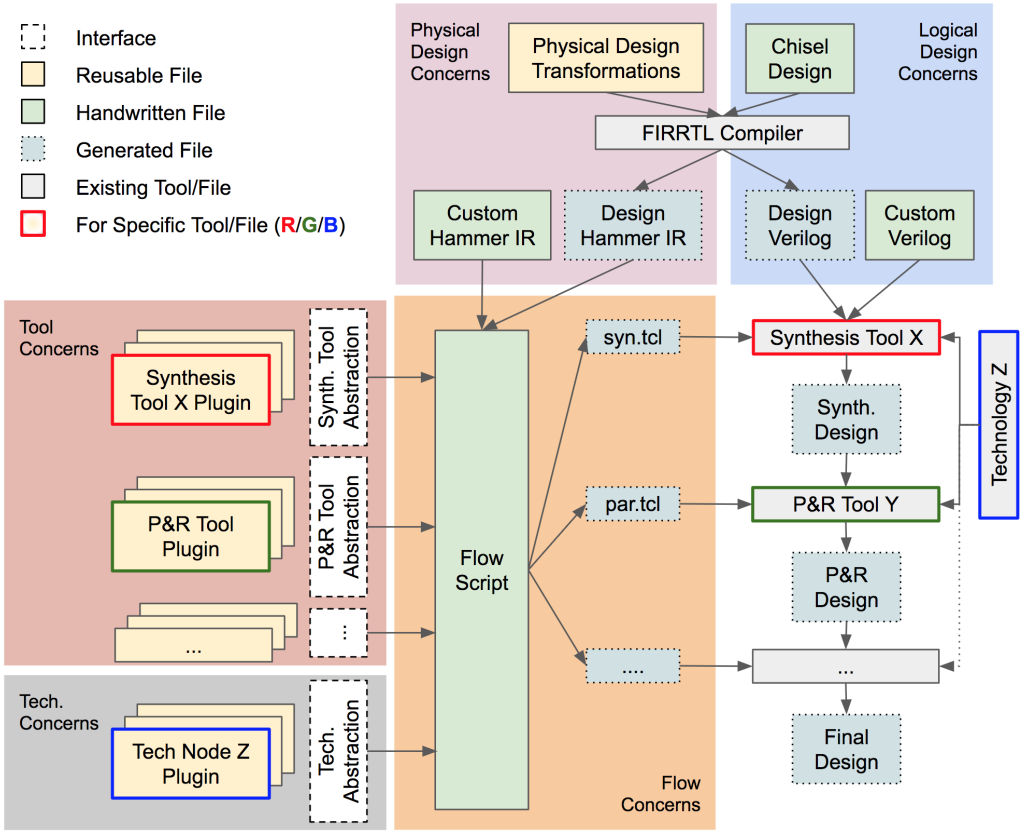

Figure 2.3: The overall architecture of the Hammer methodology showing the points of design entry, compilers/libraries, generated files/formats, and Hammer plugins, as well as the interactions between them.

The Hammer Framework

To enable design collateral reuse, the Hammer framework is designed to develop abstractions for tool, technology, and physical design concerns; this enables writing design collateral that is only dependent on design features. Like how programming languages provide abstract classes as an implementation-agnostic interface (and thus make a program reusable across class implementations), Hammer provides an abstraction for these concerns. Hammer scripts are independent of physical design, technology, and tool concerns. This section is a brief overview of the main components of Hammer; see Figure 5.4 for a diagram of the components discussed below.

The Hammer IR is the primary standardized data exchange format of Hammer. The IR standardizes physical design constraints such as placement constraints and clock constraints. In addition, the Hammer IR also standardizes communication among and to Hammer plugins, including tool control (e.g. loading tools, etc.) and configuration options (e.g. number of CPUs). Users can generate Hammer IR directly from a logical design generator like

Chisel (e.g. retiming) by triggering a custom physical design transformation; these compiler transforms ingest annotations and emit the appropriate Hammer IR to implement the desired feature. Users can still explicitly pass physical design features by writing Hammer IR or writing a TCL hook.

The Hammer Tool Abstractions create a CAD-tool abstraction layer which consists of APIs for performing various physical design tasks, including synthesis and place and route. Different plugins implement the same interface for interoperability. To use Hammer tool plugins, physical design information and settings using the Hammer IR must be provided. Developers implement Hammer tool plugins as Python classes which consume Hammer IR and emit the appropriate TCL fragments to implement those features for a certain tool. TCL hooks allow expert users to bypass the abstraction by injecting TCL code directly into the generated flow, similar to how inline assembly allows injection of assembly into C.

The Hammer Technology Abstraction provides a standard data interchange format and corresponding Python library to encapsulate technology concerns. Hammer tool plugins are linked with a technology library so that they can perform actions involving technologies, like reading timing libraries or inserting filler cells. Creating a new technology plugin involves describing paths for components of the foundry-provided PDK, including standard cell libraries, timing databases, memories, layouts, and design rules. The interface can be bypassed by using TCL hooks to inject arbitrary TCL code that may include technology-specific references/functions.

The Hammer Flow Driver, Hammer's interface for flow concerns (concerns about scheduling builds, build dependencies, tracking/managing build outputs, etc), orchestrates the ingestion of inputs/outputs and loads/calls tools, all via programmatic Python and JSON APIs. This allows users to build their own customized flow solutions using Hammer while decoupling build/flow concerns from the other four concerns, making it possible to use a variety of build tools (e.g. shell scripts, bazel, Make, SCons).

While Hammer scripts provide much of the reusability in ASIC flows, Hammer IR (Hammer's design collateral) is still heavily dependent on the design it is paired with. Manually writing Hammer IR to indicate a module's place-and-route floorplan will not work if the same module is later re-parameterized. *Design collateral must be generated in-sync with the generated design.* For this, another solution must be considered.

Chapter 3

Hardware Construction Languages

Expressive languages and programmatic customizations are a key component to enabling the development of reusable libraries. Previously, many influential works have introduced and expanded upon the concept of a hardware construction language, but this thesis revisits them in the sole context of providing a platform in which to develop hardware libraries. This chapter contains a discussion of hardware description languages through (1) an analysis of existing hardware description languages and their flaws, (2) an analysis of Chisel, a hardware construction language, as a primary tool for hardware libraries, and (3) an evaluation.

3.1 Limitations of Hardware Description Languages

As described in Chapter 2, hardware description languages like Verilog or VHDL were designed in the 1980s and have been slow to adopt modern programming improvements such as object-orientation, polymorphism, and higher-order functions. These features enable the use of good software engineering principles like abstraction, separation of concerns, and modularity; these ultimately encourage and enable more hardware design reuse. While SystemVerilog attempts to improve on existing HDLs by mixing in modern ideas like object-oriented programming with classic Verilog elements, the result is an extremely complicated language that is still missing other modern features like higher-order functions.

To illustrate this, consider a deeper dive into the example of a hardware module which filters incoming packets from a network (Figure 3.1). First, the module reads packets from the network; then, it checks if the packet violates its filtering condition. If true, the module writes the packet to the network. If false, it writes a zero'd packet to the network. There is one caveat however - the filtering condition depends on where the hardware module is instantiated.

Unfortunately, Verilog, VHDL, and SystemVerilog only support String, Integer, and Boolean parameters; our example requires a parameter that can generate the proper filtering condition hardware based on where the module is instantiated. Because these HDL's lack the capacity to express a higher-order parameter, the user cannot create this module

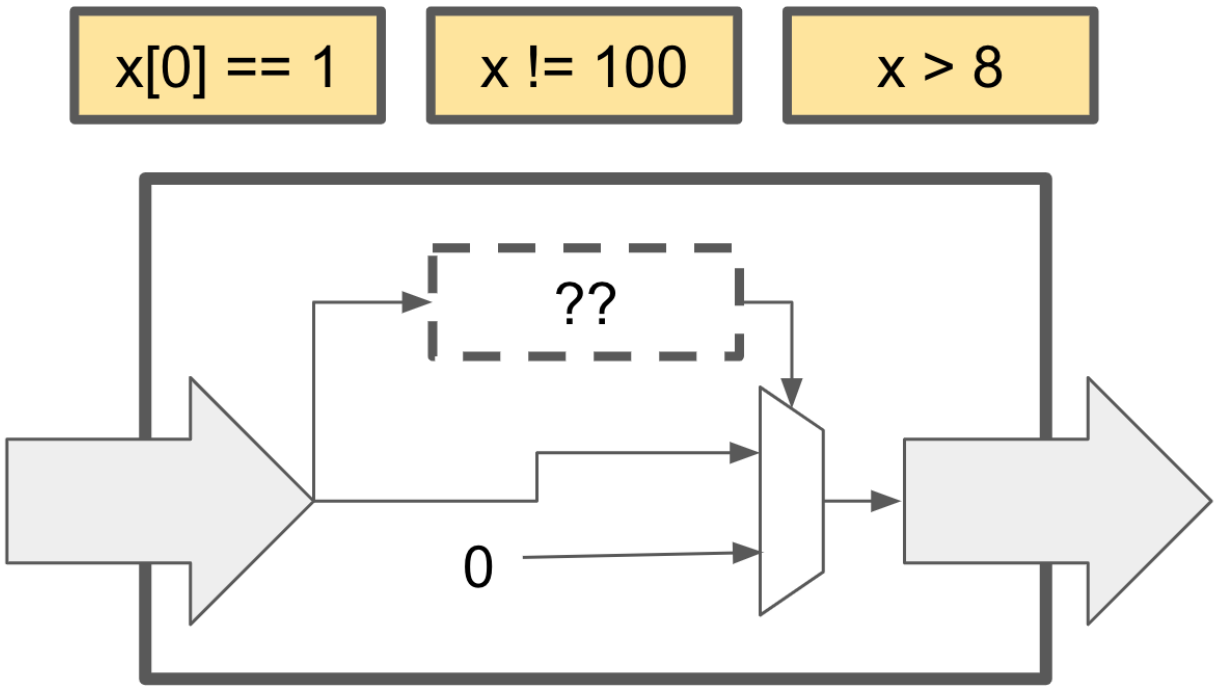

Figure 3.1: A module that filters packets by receiving them from a network, checking a filtering condition, and processing the packet accordingly. For different instantiations of this module, the filtering condition should be (1) 0th-bit is zero, (2) the signal is not 100, or (3) the signal is greater than 8. Because the design requires multiple of these filtering modules with unique filtering conditions, users cannot use an HDL because they lack these powerful parameters.

that is parameterized by the filtering condition; instead, a user must create separate modules for each separate filtering use case, even if the majority of the hardware block is reusable.

In industry, a common approach to address these types of examples is to write a custom program in a scripting language like Perl or Python which accepts these powerful parameters and generates a corresponding text file containing the corresponding HDL design. Unfortunately, this approach does not compose and requires special consideration within any build environment.

3.2 Hardware Construction Languages for Hardware Libraries

Hardware construction languages (HCLs) are a slightly different paradigm than HDLs for describing RTL circuits. Instead of expressing an RTL design directly, a user instead writes a program to construct the desired RTL design. Because HCL's require this programmatic approach, they are often created through embedding hardware construction capabilities in an existing programming language. Instead of being constrained to the limited generative capabilities of an HDL, designers have access to the rich control structures and abstractions of a general-purpose language, allowing modular, parameterizable, and reusable designs.

Chisel[3] is one of many embedded HCLs [15][38][43][30][39][5][29] and is hosted in Scala[34], a modern object-oriented and functional language.

Embedded HCLs, including Chisel, are software libraries with interfaces for constructing synthesizable RTL. For example, an object-oriented HCL might have classes representing registers and muxes:

```
// Represents synthesizable piece of hardware
abstract class HW {
  // Emits corresponding HDL representation
  def emit: String
}
class Register(name: String, width: Int)
   extends HW { ...
  def connect(r: HW) = ...
  def emit = s"reg [${width-1}:0] $name;"
}
class Mux(cond: HW, ifTrue: HW, ifFalse: HW)
   extends HW {...}
```

A designer can then create a register and hook it up by instantiating the `Register` object and calling its `connect` method:

```
class Top { ... // Start of program
  val my_reg = new Register("my_reg", 32)
  my_reg.connect(my_mux)
}
```

Language features like operator overloading can also cut verbosity:

```
class Top { ... // Start of program
  // Equivalent to my_reg.connect(my_mux)
  my_reg := my_mux
}
```

To generate the complete design, the user simply executes their HCL code; this process is called *elaboration*. Each HCL method call, including instantiations, builds up a data structure representing the hardware design instance. This design can then be emitted to an existing HDL.

Developing in a well-designed HCL can closely mimic the experience of writing in an HDL.

Enabling Hardware Libraries

HCLs *by themselves* do not provide any new hardware abstractions. However, host language features allow designs to be more parameterizable and modular.

For example, Chisel users can write a recursive Scala function to construct an adder-reduction tree, parameterized on bit-width. Unlike the explicitly unrolled version necessary in Verilog, the same generator could be re-used anywhere an adder tree is desired.

Similarly, a Chisel designer can write a filter module which takes, as a parameter, a higher-order-function that creates the condition-checking hardware. The user of this module then only needs to write the filtering condition, re-using the base filter structure.

Ultimately, the benefit of any HCL is the expressiveness provided by the host language; this opens the door for reusable hardware libraries.

3.3 Evaluating Chisel's Support for Hardware Libraries

An expressive language requires fewer lines of code to more fully parameterize a design. This parameterization enables reusing the same code in different contexts with different parameters, potentially generating radically different hardware.

The following evaluates Chisel with regards to its expressiveness, parameterizability, and ultimately its reusability.

Expressiveness

By using software engineering methods enabled by modern programming languages, one should expect fewer lines of code to express similar projects.

RocketChip[1] is an open-source hardware library, written in Chisel, that can generate many different instantiations of a symmetric multi-processor system (SMP). OpenPiton[4] is a research project, written primarily in Verilog and enhanced with some Python-Verilog generation scripts, that uses OpenSPARC cores with a custom interconnect and coherency framework.

OpenPiton and RocketChip have many similarities from 10,000 feet – both are SOC generators, containing cores, caches, network protocols, coherency domains, tests, and much more. Both are used for computer architecture research, have been realized in silicon, and boot Linux.

While clearly an apples-to-oranges comparison, Figure 3.2 depicts a comparison between the code bases. OpenPiton takes 3x and 10x more code to express similar hardware structures; the sheer magnitude of code size differences between OpenPiton and RocketChip cannot be explained solely by their differing feature sets. In addition, to the authors' knowledge, RocketChip's out-of-order core, BOOM[9], requires the fewest lines of code of any open-source out-of-order core implementation.

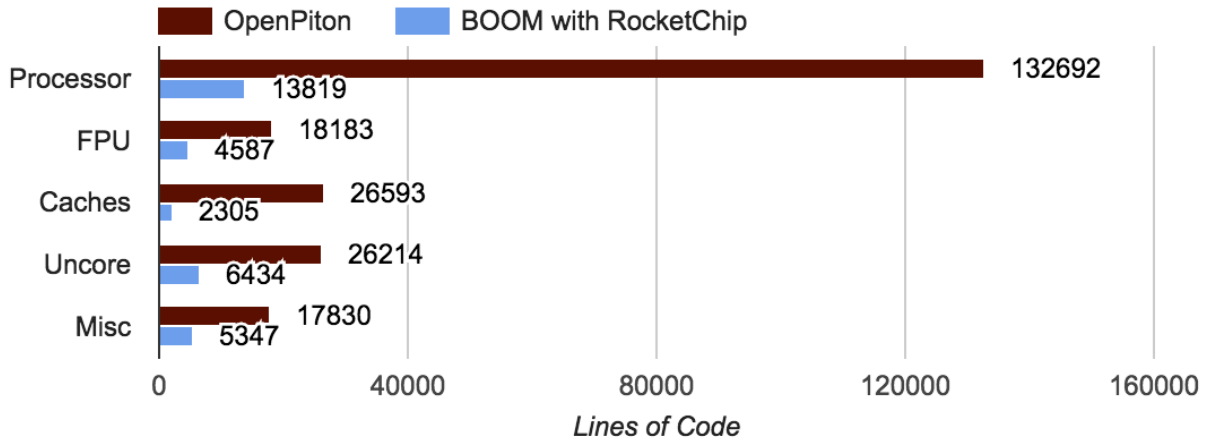

Figure 3.2: Similar hardware structures show significant differences in code size, ranging from between 3x to 10x. Because of their differing feature sets, this evaluation should not be taken as a strict comparison, but rather interpreted as a general trend that using Chisel enables a more expressive coding style.

While much of the OpenSPARC core was likely not entirely hand-written (tools like editor extensions could have been used), the comparison of language expressivity remains valid: Chisel is clearly more expressive than Verilog because RocketChip requires significantly less source code.

Parameterizability

Parameterization precedes effective reusability - a flexibly parameterized module is more useful, and thus more reusable.

While it is difficult to quantitatively evaluate the flexibility, magnitude, and degree of parameterization that a general-purpose programming language provides an embedded HCL, the type and degree of RocketChip's parameterizability is described qualitatively:

- **Out-of-order parameters:** fetch width (1, 2, 4), issue width (1, 2, 3, 4), branch predictors (BTB, GShare, TAGE)
- **Data parallelism:** number of parallel data operations (4 through 32), precision (half, word, double)
- **Multi-core:** number of cores (1, 2, 4, 8, 16)
- **Cache:** size (64KB to 2MB), associativity (direct-mapped, two-way), type (scratch-pad, blocking, non-blocking), coherence policy (MSI, MESI)

Note that the cross product of these parameters are all valid, and many (but not all) of these design points have been experimented with or even realized in silicon.

Furthermore, many of these parameters are not simply bit-widths, but impact the control logic, interface definitions, and communication protocols. As shown in Figure 3.3, the different parameterizations can generate vastly different designs with very different microarchitectures, Coremark[17] performance results, and area numbers.

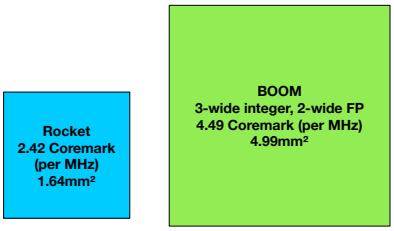

Figure 3.3: Two different configurations, a 3-issue out-of-order core and an in-order core, were pushed through the FIRRTL compiler framework and synthesized them with the Synopsis SAED educational standard cell library[19]. These designs are very different in their area footprint and Coremark performance; they are in fact very different processors generated using a large percentage of shared source code.

Reusability

Three processors written in Chisel are analyzed next: (1) BOOM[9], RocketChip's out-of-order machine, (2) Rocket, a single-issue in-order core, and (3) DecVec, a decoupled vector co-processor, to understand whether parameterized designs foster reusability. As shown in Figure 3.4, approximately 5000 lines of code are shared with all three designs, and even more is shared between pairs of designs. In all, the three designs share half or more of their codebases with one another.

3.4 Summary

Hardware construction languages provide additional expressibility and parameterizability to hardware designers, greatly encouraging the development of reusable hardware libraries. While successful in their own right, HCL's lack the ability to fully separate their source code from underlying platforms or technologies. The next chapter discusses how a hardware compiler framework enables this separation, as well as providing additional capabilities to Chisel hardware libraries.

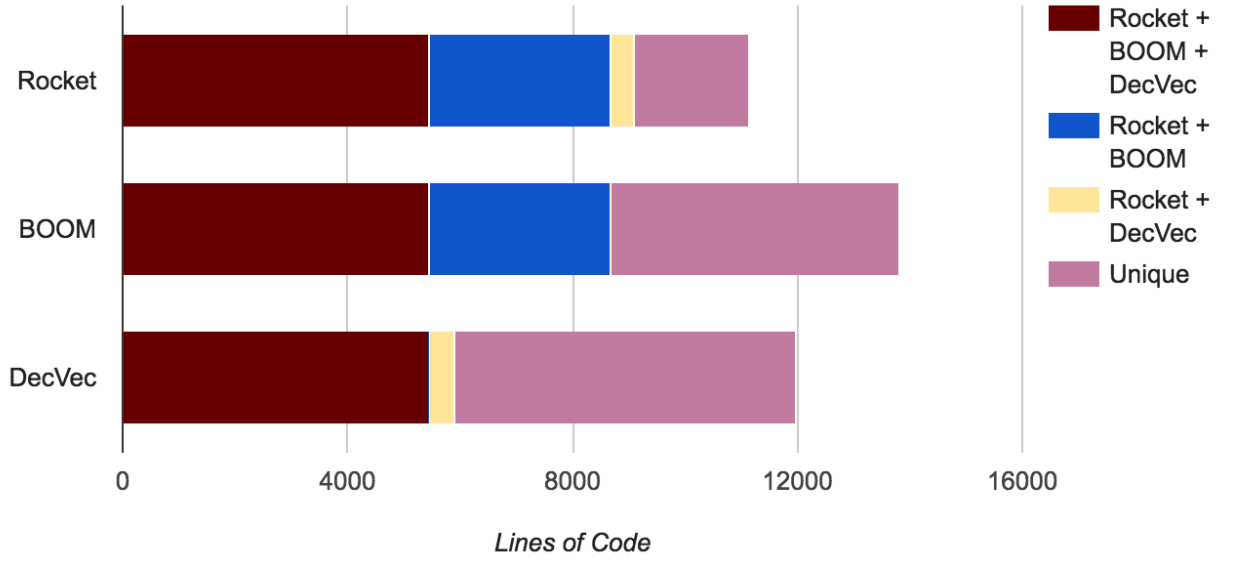

Figure 3.4: Three processors Rocket, BOOM and DecVec reuse each other's code. Modules used by all three designs include an ALU, a MulDiv unit, an ICache, a TLB, a Decoder, and an FPU. Modules used by Rocket and BOOM include a non-blocking data cache, a PTW, a CSR, and a BTB.

Chapter 4

FIRRTL: A Hardware Intermediate Representation and Compiler Framework

Like how software compilers transform general-purpose code into specialized assembly, a hardware compiler transforms general RTL into specialized RTL. The FIRRTL compiler enables this automatic transformation of a design, unlocking a huge amount of potential through optimizations and other generic transformations. This potential is best understood through two main features of the FIRRTL compiler framework: (1) its reusable transformations, and (2) its extensibility for customizations.

At first glance, writing an RTL transformation may seem like over-engineering; if a user wants to inline a module, why write an entire transformation when inlining manually is not very difficult? The key observation is that inlining is a common procedure required for most physical design implementations, and thus automation via transformation saves significant future manual effort. Indeed, writing a transformation only saves effort if its use-case is common enough that all future uses amortize the cost of the initial transformation development. This argument is similar to the argument in Chapter 1 for how reusable hardware libraries amortize their development costs over time. By making transformations easy to write and integrate within a compiler framework, the upfront development cost of a transformation is reduced and the number of worthwhile automatable tasks increases.

To motivate the need for an extendable hardware compiler infrastructure, consider the following example: a streaming digital-signal processing (DSP) hardware library. Every component in this library has a decoupled interface, where a queue of unknown size could be instantiated between each component. An unfortunate and unavoidable consequence of this library is that, if the queue size is zero, then the decoupled ready and valid signals between the components are vestigial yet form a combinational path. Ideally these signals would never be generated, but detecting this circumstance depends on knowing a neighbor's configuration; the local information available to a given library component generator is not sufficient.

Using an extendable hardware compiler framework enables this streaming DSP library to analyze the entire design topology and remove these vestigial combinational loops au-

tomatically by writing and integrating their own custom transformation. Supporting this use-case requires the compiler framework to inspect and modify a design, be extendable for custom transformations, and support a robust mechanism for communicate information throughout the compilation process (e.g. which signals were generated by the library so other combinational paths remain untouched.)

This chapter contains a discussion of FIRRTL's hardware compiler framework (HCF) through the following topics: (1) analysis of LLVM, an existing software compiler framework; (2) introduction of FIRRTL's intermediate representation (IR); (3) description of value inference (e.g. widths) and its implementation; (4) support for arbitrary metadata throughout the compilation process; (5) the mechanisms for transforms to inspect and modify a design; (6) a description of interesting transformations; (7) an evaluation of the FIRRTL compiler framework.

4.1 Background

Modern software compiler frameworks, such as LLVM[27], consist of (1) frontends, (2) transformations, and (3) backends. A frontend parses programs written in a specific programming language (e.g. C++ or Rust) into a compiler-specific IR. IR-to-IR transformations such as optimization passes then can operate on and modify the program's structure. Finally, a backend converts the IR into a program in the target ISA, e.g. ARM or x86. This structure of translating an input language into an IR enables reusing transformations among multiple designs and languages.

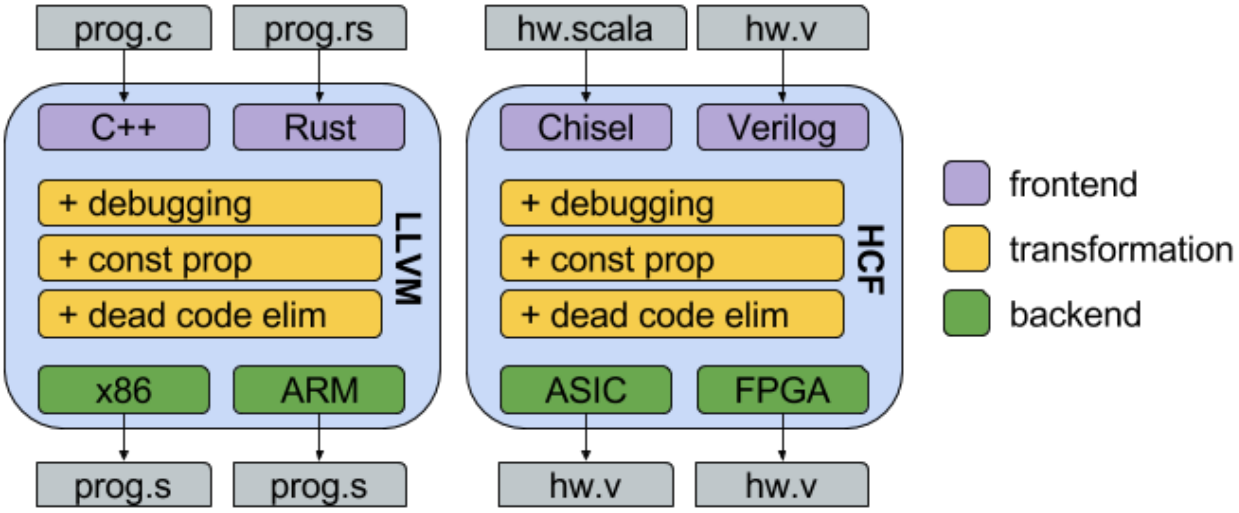

Figure 4.1: LLVM can create a C++-to-x86 compiler or a Rust-to-ARM compiler, yet share internal transformations on LLVM-IR. Similarly, our HCF can create a Chisel-to-ASIC-Verilog compiler or Verilog-to-FPGA-Verilog compiler and share internal transformations.

LLVM originated, like FIRRTL, as an academic project by Chris Lattner and advised by Vikram Adve in 2000 at the University of Illinois at Urbana-Champaign. Since these beginnings, the "compiler infrastructure project" has revolutionized compiler research by providing an open source, modular, and modern compiler.

The LLVM compiler has a modular design of many passes that operate on a common and well-defined intermediate representation (IR) of a program. This IR is independent of source program and target machine.

The LLVM's compiler infrastructure is composed of passes which operate in sequence on a program's IR. Each pass accepts a program's IR and returns a modified IR. They pipe together until the program is optimized, simplified, and instrumented.

Some passes require analyzing the program before modifying it, and many of these analyses can be shared among passes. However, other passes invalidate previously run analyses, which must be rerun. This presents an interesting challenge - how does the compiler know when to recompute analyses? Another challenge is pass dependency - some passes expect and require being run after other passes - how is this ordering done?

LLVM solves both of these challenges with a mechanism called pass scheduling and registration. As part of their interface, passes specify the following:

- any prerequisite passes (default is no other passes)
- any passes they invalidate/preserve (default is invalidating all other passes)

Note that references to prerequisite or invalidated passes is by name, which can be brittle to code modifications. Additionally, incorrect specification of prerequisites or invalidations can cause undetermined runtime behavior.

After passes are declared, they must be registered (either statically or dynamically) to a global *Pass Manager* with the following:

- Command-line option name
- Name of the pass
- Whether it walks and modifies the control-flow-graph
- Whether it is an analysis pass

LLVM has three pass categories: analysis passes, transform passes, and utility passes. Analysis passes compute information that other passes can use, and can be reused multiple times for multiple passes. Transform passes mutate the program in some way, and can use (or invalidate) analysis passes. Utility passes provide some utility that don't otherwise fit categorization, e.g. passes to extract functions to bitcode.

Each pass can take on one of a variety of traversal types. An *ImmutablePass* doesn't traverse the program but just reports statistics or other information. A *ModulePass* operates on the entire program and thus cannot optimize its execution. A *CallGraphSCCPass* traverses the program IR bottom up and can only access local information. Due to its traversal

behavior, it is possible to optimize its execution, but is tough to write one that is correct conceptually. A *FunctionPass* visits each function, independent of visiting other functions. This makes it easily parallelizable, conceptually simple, but has limited functionality. Finally, a *LoopPass* executes on each loop in the function, independent of all the other loops in the function.

The LLVM compiler infrastructure is well designed and has some important takeaways that can be applied to designing the FIRRTL compiler infrastructure. First, there can exist a user-enforced dependency between a pass's actual and specified behavior; a disconnect here would be difficult to debug. In addition, the variety of program traversal APIs can restrict passes but also enable optimizations such as inter-mixing the execution of multiple passes for better cache behavior from data locality. It is also important to keep the window open for multithreaded compilation. Finally, be sure to implement lots of useful infrastructure to simplify writing (and integrating) a compiler pass.

FIRRTL's compiler framework is similarly structured: Chisel and Verilog frontends parse into its IR, transformation passes provide simplification, optimization, and instrumentation, and the resulting IR can either be simulated directly or passed to one of many Verilog backends tailored for simulators, FPGAs, or ASIC technology processes. Dependencies between transformations are specified, enabling a total ordering of tranformations to be determined prior to running the compiler. Custom transformations are integrated automatically through this dependency interface.

4.2 FIRRTL's Intermediate Representation

To support an ecosystem of transformations which can consume, transform, and produce hardware designs, clear and exact behavior of the hardware design's representation at each stage of transformation must be defined. This collection of hardware element definitions is the FIRRTL specification. Its clean but powerful design is one reason why the FIRRTL compiler project has been successful.

The original goal of the project was for Chisel, when elaborating a design, to emit a FIRRTL representation. However, while FIRRTL was designed to resemble the Chisel HDL after all meta-programming has executed, it is not fundamentally tied to Chisel. In fact, additional HDLs written in other languages can target FIRRTL and reuse the majority of the compiler toolchain.

This section first introduces FIRRTL and its concrete syntax, as well as the FIRRTL forms which provide the basis for the lowering transformations.

FIRRTL Concrete Syntax

FIRRTL represents a standardized non-parameterized digital logic design, and consists of hardware modules for encapsulation, registers and memories for state elements, and primitive operations and muxes for combinational logic.

FIRRTL also has first-class support for high-level constructs such as vector types, bundle types, conditional statements, partial connects, and modules. These high-level constructs are then gradually removed by a sequence of *lowering* transformations. Eventually the circuit is simplified to resemble a structured netlist, which can then be translated into any output language (e.g. Verilog). All FIRRTL constructs interoperate with one another; a simplified circuit is expressed using a subset of the FIRRTL specification, rather than a separate specification.

Productions in the following concrete syntax tree are *italicized* and keywords are written in `monospaced` font. The special productions *id*, *int*, and *string*, indicates an identifier, an integer literal, and a string respectively. The notation $[\![e]\!]_{...}$ is used to indicate that e is repeated zero or more times, and the notation $[\![e]\!]_{?}$ is used to indicate that including e is optional.

For a detailed explanation of all FIRRTL components, see Appendix A.

circuit	=	`circuit` *id* `:` ⟦*info*⟧? `(`⟦*module*⟧...`)`	Circuit
module	=	`module` *id* `:` ⟦*info*⟧? `(`⟦*port*⟧... *stmt*`)`	Module
	|	`extmodule` *id* `:` ⟦*info*⟧? `(`⟦*port*⟧...`)`	External Module
port	=	*dir id* `:` *type* ⟦*info*⟧?	Port
dir	=	`input` | `output`	Port Direction
type	=	`UInt`⟦`<`*int*`>`⟧?	Unsigned Integer
	|	`SInt`⟦`<`*int*`>`⟧?	Signed Integer
	|	`Fixed`⟦`<`*int*`>`⟧?⟦`«`*int*`»`⟧?	Fixed Point
	|	`Interval`⟦*lb*,*bval*,*bval*,*ub*⟧?⟦`«`*int*`»`⟧?	Interval
	|	`Clock`	Clock
	|	`Analog`⟦`<`*int*`>`⟧?	Analog
	|	`{`⟦*field*⟧...`}`	Bundle
	|	*type*`[`*int*`]`	Vector
lb	=	`(`	Open Lower Bound
	=	`[`	Closed Lower Bound
ub	=	`)`	Open Upper Bound
	=	`]`	Closed Upper Bound
bval	=	`?`	Unknown Bound Value
	=	*int*⟦`.` *int*⟧?	Known Bound Value
field	=	⟦`flip`⟧? *id* `:` *type*	Bundle Field
stmt	=	`wire` *id* `:` *type* ⟦*info*⟧?	Wire
	|	`reg` *id* `:` *type exp* ⟦`(with: {reset => (`*exp*`,` *exp*`)})`⟧? ⟦*info*⟧?	Register
	|	`mem` *id* `:` ⟦*info*⟧? `(` `data-type =>` *type* `depth =>` *int* `read-latency =>` *int* `write-latency =>` *int* `read-under-write =>` *ruw* ⟦`reader =>` *id*⟧... ⟦`writer =>` *id*⟧... ⟦`readwriter =>` *id*⟧...`)`	Memory
	|	`inst` *id* `of` *id* ⟦*info*⟧?	Instance
	|	`node` *id* `=` *exp* ⟦*info*⟧?	Node
	|	*exp* `<=` *exp* ⟦*info*⟧?	Connect
	|	*exp* `<-` *exp* ⟦*info*⟧?	Partial Connect
	|	*exp* `is invalid` ⟦*info*⟧?	Invalidate
	|	`attach(`⟦*exp*⟧...`)` ⟦*info*⟧?	Attach
	|	`when` *exp* `:` ⟦*info*⟧? *stmt* ⟦`else :` *stmt*⟧?	Conditional
	|	`stop(`*exp*, *exp*, *int*`)`⟦*info*⟧?	Stop
	|	`printf(`*exp*, *exp*, *string*, ⟦*exp*⟧...`)` ⟦*info*⟧?	Printf
	|	`skip` ⟦*info*⟧?	Empty
	|	`(`⟦*stmt*⟧...`)`	Statement Group
ruw	=	`old` | `new` | `undefined`	Read Under Write Flag
info	=	`@[`*string*`]`	File Information Token

exp	=	`UInt`⟦`<`*int*`>`⟧?`(`*int*`)`	Literal Unsigned Integer
	|	`UInt`⟦`<`*int*`>`⟧?`(`*string*`)`	Literal Unsigned Integer From Bits
	|	`SInt`⟦`<`*int*`>`⟧?`(`*int*`)`	Literal Signed Integer
	|	`SInt`⟦`<`*int*`>`⟧?`(`*string*`)`	Literal Signed Integer From Bits
	|	*id*	Reference
	|	*exp*`.`*id*	Subfield
	|	*exp*`[`*int*`]`	Subindex
	|	*exp*`[`*exp*`]`	Subaccess
	|	`mux(`*exp*, *exp*, *exp*`)`	Multiplexor
	|	`validif(`*exp*, *exp*`)`	Conditionally Valid
	|	*primop*`(`⟦*exp*⟧..., ⟦*int*⟧...`)`	Primitive Operation
primop	=	`add`	Add
	|	`sub`	Subtract
	|	`mul`	Multiply
	|	`div`	Divide
	|	`rem`	Remainder
	|	`lt`	Less Than
	|	`leq`	Less or Equal
	|	`gt`	Greater Than
	|	`geq`	Greater or Equal
	|	`eq`	Equal
	|	`neq`	Not-Equal
	|	`pad`	Pad
	|	`asUInt`	Interpret Bits as UInt
	|	`asSInt`	Interpret Bits as SInt
	|	`asClock`	Interpret as Clock
	|	`asFixedPoint`	Interpret as Fixed Point
	|	`asInterval`	Interpret as Interval
	|	`asAnalog`	Interpret as Analog
	|	`shl`	Shift Left
	|	`shr`	Shift Right
	|	`dshl`	Dynamic Shift Left
	|	`dshr`	Dynamic Shift Right
	|	`incp`	Increase Precision
	|	`decp`	Decrease Precision
	|	`setp`	Set Precision
	|	`cvt`	Arithmetic Convert to Signed
	|	`neg`	Negate
	|	`not`	Not
	|	`and`	And
	|	`or`	Or
	|	`xor`	Xor
	|	`andr`	And Reduce
	|	`orr`	Or Reduce
	|	`xorr`	Xor Reduce
	|	`cat`	Concatenation
	|	`bits`	Bit Extraction
	|	`head`	Head
	|	`tail`	Tail
	|	`wrap`	Wrap
	|	`clip`	Clip
	|	`squz`	Squeeze

The Lowered FIRRTL Forms

FIRRTL contains richer elements than a pure netlist design to capture as much user intent as possible within FIRRTL's representation, without drastically increasing the number of constructs. However, this requires any FIRRTL compiler to rewrite a circuit with richer constructs into an equivalent circuit with simpler, lower-level constructs.

FIRRTL's simplification process is standardized into three well-defined forms, where each uses a smaller, stricter and simpler subset of FIRRTL features than the previous form.

The lowered FIRRTL forms, MidFIRRTL and LoFIRRTL, are increasingly restrictive subsets of the FIRRTL language that omit many of the higher level constructs. All conformant FIRRTL compilers must provide a *lowering transformation* that transforms arbitrary FIRRTL circuits into equivalent LoFIRRTL circuits. However, there are no additional requirements related to accepting or producing MidFIRRTL, as the LoFIRRTL output of the lowering transformation will already be a legal subset of MidFIRRTL.

Any transformation can specify which FIRRTL form it consumes, but can always emit a higher form that is subsequently lowered. While less-rich inputs have fewer corner cases, generating and modifying IR is simpler with richer features.

MidFIRRTL

A FIRRTL circuit is defined to be a valid MidFIRRTL circuit if it obeys the following restrictions:

- All unknown values must have been inferred or explicitly defined.
- The conditional statement is not used.
- The partial connect statement is not used.
- All components are connected to exactly once.

LoFIRRTL

A FIRRTL circuit is defined to be a valid LoFIRRTL circuit if it obeys the following restriction, in addition to the MidFIRRTL restrictions:

- All components must be declared with a ground type.

The additional restriction gives LoFIRRTL a direct correspondence to a circuit netlist.

Low level circuit transformations can be conveniently written by first lowering a circuit to its LoFIRRTL form, then operating on the restricted (and thus simpler) subset of constructs. Note that circuit transformations are still free to generate high level constructs as they can simply be lowered again.

The following module:

Example 4.1:

```
module MyModule :
   input in: {a:UInt<1>, b:UInt<2>[3]}
   input clk: Clock
   output out: UInt
   wire c: UInt
   c <= in.a
   reg r: UInt[3], clk
   r <= in.b
   when c :
      r[1] <= in.a
   out <= r[0]
```

is rewritten as the following equivalent LoFIRRTL circuit by the lowering transform.

Example 4.2:

```
module MyModule :
   input in_a: UInt<1>
   input in_b_0: UInt<2>
   input in_b_1: UInt<2>
   input in_b_2: UInt<2>
   input clk: Clock
   output out: UInt<2>
   wire c: UInt<1>
   c <= in_a
   reg r_0: UInt<2>, clk
   reg r_1: UInt<2>, clk
   reg r_2: UInt<2>, clk
   r_0 <= in_b_0
   r_1 <= mux(c, in_a, in_b_1)
   r_2 <= in_b_2
   out <= r_0
```

4.3 Width, Bound, and Precision Inference

Many hardware design languages require the hardware designer to manually specify the number of bits of every signal in their design. This requires a significant amount of boilerplate, as well as requiring a designer to "figure-out" the size of signals for which they don't care about, as long as it's "big enough" to not lose information carried on the wire.

Instead, FIRRTL has significant support for inferring these signal widths automatically, enabling the designer to specify the important signal widths, and let the compiler infer the rest. In addition, FIRRTL supports datatypes other than integers which require additional characteristics; fixed-point datatype signals require a width and precision, and interval datatype signals require a precision, upper bound, and lower bound.

This section discusses the motivation, behavior, and implementation of value inference, i.e. the inference of widths, precision, and interval-bounds. Included are the details of how

the FIRRTL compiler derives constraints on these unknown values from the design, as well as the algorithm behind solving these constraints.

Motivation

For all signal types, hardware designers often desire this value inference for primitive operations, enabling intermediate signal widths, precisions and intervals to be automatically determined. In addition to reducing boilerplate, it reduces the possibility of incorrect width specifications. Manually specifying the precision of intermediate signals is especially difficult to do properly and is more convenient to be inferred.

In addition, interval type signals are useful for domains of RTL design such as digital signal processing where a module's input/output ranges are known, even though the hardware design language requires a bitwidth. Since a signal's range cannot be inferred optimally from its bitwidth, the resulting overly-conservative width inference causes suboptimal power and area results. Manual bitwidth optimization, especially for generators, is tricky, inconvenient, and error-prone. Designer intent can be better captured by directly encoding input ranges into the design, allowing automatic range propagation and bitwidth reductions.

These techniques have been used in high-level synthesis flows[46] but require users to express their designs using non-zero-cost abstractions. The benefits of these optimizations are offset by the difficulties encountered when porting existing RTL to a new flow and/or application. Ideally, a hardware description language would directly support interval types and these bitwidth optimization to enable hardware designers to use them within an RTL abstraction.

Overview

FIRRTL supports width, precision, and interval-bound inference. The width inference is used for the following types: unsigned integers, signed integers, analog, and fixed-point. The precision inference is used for the fixed-point and interval types. The interval-bound inference is only used for the interval type, which represents a range of fixed-point values with a given precision.

All width, precision, and bound inference is conservative; they grow to never lose accuracy. If the desired width or precision or bound is less conservative than the inferred value, the value must be manually specified.

Our implementation of the interval type uses simple forward/backward range propagation [24] [41]; while still conservative, this approach still offers improved bitwidth inference over non-interval approaches without complex symbolic analysis of ranges. Potential ranges are tracked for each operation, and the solver finds the worst-case bitwidth. A different interval inference uses affine analysis of ranges [28], which can cancel correlated terms (e.g., $A - A$ has a range of [0, 0]). While enabling more powerful inference, affine analysis has non-local effects; the resulting range of an arithmetic operation is no longer solely determined by its

input arguments. As a standard supported datatype, this non-locality is undesirable for transformations attempting to reason about local behavior.

Specification

For all circuit components declared with unspecified width, bound, or precision, the FIRRTL compiler will attempt to infer the strictest possible value which can still represent all possible values of its incoming connections. If a component has no incoming connections, and the width/bound/precision is unspecified, then an error is thrown to indicate that the value could not be inferred.

For module input ports with an unspecified width/bound/precision, the inferred value is the strictest possible value that maintains the legality of all incoming connections to all instantiations of the module.

For the specific (and local) width, precision or bound inference rules for each expression, see their corresponding section in the Expression section of Section A.

To resolve all unknown widths, bounds and precisions, the FIRRTL compiler performs the following circuit transformations:

1. Resolve unknown precisions - constraints on unknown precisions for all signals are derived from the design and solved for

2. Trim interval-bounds to known precision - all known open interval upper/lower bounds are converted to closed bounds based on the (now) known precision of the interval

3. Resolve unknown widths and bounds - all constraints on upper bounds, lower bounds, and widths are derived from the design and solved for

Collecting Constraints

Every node in the FIRRTL graph is visited and unknown width, precision and bound values are replaced with a unique variable. Then, depending on the stage of the compiler, either precision or bound/width constraint expressions are built by visiting every FIRRTL expression and statement. When an assignment statement is encountered, a variable constraint is created from the right-side constraint expression and the left-side variable. See Table 4.3 for an example demonstrating this process.

Precision and bound constraint expressions are formed based on the FIRRTL primitive operation. For precision, many primitive operations simply take the maximum precision of its input arguments. Others explicitly set the precision or shift the binary point position. Finally, some have unique rules, like multiplication which takes the sum of the input argument precisions.

The relationship between bound constraints and their FIRRTL expressions are more complex, but all these relationships are summarized in the Expression section in Section A.

Example 4.3:

```
wire x: Interval[0, 10].1
wire y: Interval[2, 7].2
wire z: Interval
z <= add(x, y)
```

Collected Constraints:

Lower Bound	$z_l <= 0 + 2$
Upper Bound	$z_u >= 10 + 7$
Precision	$z_p >= max(1, 2)$

Table 4.1: In this example, the constraints on the upper bound, lower bound, and precision of `z` are collected. The lower bound is the largest value that is less than the sum of the lower bounds of its incoming connection. In this case, the incoming connection is the `add` operation, whose lower bound is the sum of its inputs lower bounds (`0` and `2`). The upper bound and precision constraints are similarly calculated, but their value is the smallest value which is larger than the corresponding value of the incoming connection.

After all variable constraints are collected, all variable constraints on a given variable are checked to be either >= or <=, but not both. Due to how the constraints are structured, this monotonicity constraint will never be violated with a legal FIRRTL circuit.

Solving Constraints

The following description references Figure 4.2, where the bolded terms refer to the corresponding algorithm.

After collecting all variable constraints and ensuring monotonicity, the constraints are **solved** by first **merging** constraints on the same variable. Groups of >= are combined with a $max(...)$ constraint, while groups of <= are combined with a $min(...)$ constraint. This guarantees there is a single variable constraint per variable.

Next, **forward substitution** is performed by iterating *top to bottom* through all variable constraints, replacing any references to variables previously seen with their corresponding constraint (this is done recursively). This is a legal substitution because of monotonicity; each variable must take the smallest value (if increasing monotonically) or the largest value (if decreasing monotonically) that still abides its constraints. Thus, if $x >= 10$, then x should take value 10 (and 10 can be directly substituted for x). After forward substitution, all variable constraint expressions only reference variables below them.

Finally, **backward substitution** iterates *bottom to top* through all forward-solved variable constraints, again recursively replacing previously seen variables with their constraints. Since each forward-solved variable constraint only references variables below it, all constraints are guaranteed to be solved (if a solution exists) after backward substitution.

Forward-backward substitution theoretically solves a set of constraints in linear time, as it only iterates through a list of constraints twice. However, constraint-expression substitution causes an exponential growth in the size of the constraint expression. In practice, aggressive in-line optimizations immediately after substitution will dramatically reduce the constraint expression to a tractable size.

Algorithm Forward-Backward Substitution

```
procedure OPTIMIZE(c: Constraint) ...
procedure REMOVECYCLE(n: Name, c: Constraint) ...
procedure MERGE(con: Map[Name,Constraint]) ...
procedure SUBSTITUTE(sol: Map[Name,Constraint],c: Constraint)
    if c typeof Variable & sol.has(c.name) then
        c ← sol.get(c.name)
    if c typeof Operator then
        for i in (0 → c.children.length) do
            c.children[i] ← optimize(c.children[i])
            c.children[i] ← substitute(sol, c.children[i])
    return c
procedure FORWARD(con: Map[Name,Constraint])
    sol ← Map[Name, Constraint].empty
    for i in (0 → (con.size − 1)) do                ▷ in insertion order
        (name, c) ← con[i]                          ▷ get ith (key,value)
        sol[name] ← substitute(sol, c)
        sol[name] ← optimize(sol[name])
        sol[name] ← removecycle(name, sol[name])
    return sol
procedure BACKWARD(con: Map[Name,Constraint])
    sol ← Map[Name, Constraint].empty
    for i in ((con.size − 1) → 0) do                ▷ reverse insertion order
        (name, c) ← con[i]                          ▷ get ith (key,value)
        sol[name] ← substitute(sol, c)
        sol[name] ← optimize(sol[name])
    return sol
procedure SOLVE(con: Map[Name,Constraint])
    con ← merge(con)
    con ← forward(con)
    con ← backward(con)
    return con
```

Figure 4.2: Forward substitution populates *sol* with forward-solved variable constraints. It uses the substitution procedure which recursively visits constraint children and substitutes variables with their optimized constraints in *sol* (if they exist). It also attempts to remove cyclic constraints - failures are reported to the user. Backward substitution iterates through the forward-solved constraints in reverse order, calling *substitute* and optimizing the result. The solver ignores whether *Name* is >= or <= *Constraint* in *Map*, as all constraints are monotonic.

During both forward- and backward-substitution, constraint expressions are aggressively **optimized**. In addition, during forward-substitution, cyclic constraints are attempted to be **removed**; when not all cyclic constraints can be removed, the constraint solver moves on. For example, $w_1 >= max(w_1, 10)$ becomes $w_1 >= 10$, while $w_2 >= w_2 + 1$ is unsolvable. Any unsolved constraint (due to an unsolvable cyclic constraint or an underspecified constraint) is an error and is reported back to the user.

4.4 Specificity and Metadata

In their most general form, annotations are arbitrary metadata associated with an IR node. As illustrated by the DSP library example at the beginning of the chapter, a compiler framework's annotation system can be leveraged to support novel features and libraries. Common uses of annotations include marking signals to exclude from optimization, naming specific modules as targets for a topological transformation (e.g. flattening), or carrying command-line options like the target directory or debug flags. FIRRTL uses its annotation system as the single mechanism to pass information to and from a transformation, which simplifies its interface but adds two significant requirements: (1) robustness, meaning annotations do not get stale, and (2) completeness, meaning annotations can contain arbitrary information.

While at first glance supporting robustness and completeness may seem straightforward, these innocuous features require solutions for many subtle problems. For example, suppose a transformation deletes, renames, or modifies an IR node with an annotation - how should the annotation propagate to remain robust? Given that annotations are arbitrary, their desired behavior could include any of the following:

- Error (e.g. if annotation means 'don't delete this node', and the node got deleted)
- Delete the annotation
- Move the annotation to a different IR node
- Duplicate the annotation
- Track history of how that IR node changed

Additionally, annotations could refer to multiple IR nodes - what's the behavior then? For example, an annotation could specify pairs of signals - what happens if only one of them is deleted?

Another perspective is that supporting both arbitrary metadata and arbitrary transforms requires the compiler to make them robust to one another. However, the arbitrariness of behavior means paradoxical scenarios can be constructed; for example, a transform desires to deletes an IR node, but metadata desires to prevent deletion. Given the contradiction, how can the desired system behavior be robustly determined?

To address this question, first consider the following observations:

1. In the general case, a compiler must be able to detect this paradox and error
2. Only the system that uses both the annotation and the transformation knows how to resolve this (neither the transform nor the annotation has enough information)
3. Transforms that seek large interoperability should provide escape-hatch annotations to enable systems that use contradictory libraries to resolve the contradiction (e.g. optimization passes must provide a DontTouchAnnotation, if they are to interoperate with other transforms that may require some signals to not be deleted)

This section first delves into how other compilers tackle this problem and the shortcomings of their approaches. Then, the FIRRTL compiler's support for annotations is discussed via (1) the transform/annotation interface; (2) *target*, a mini-language for specifying named components within a FIRRTL circuit; and (3) support for tracking renaming/modifying signals.

Related Work

Most compiler infrastructures use ad hoc methods to attach metadata to their intermediate representation. This unfortunately can result in their annotation system being unreliable, as any other transformation could delete a node (and its metadata along with it), with no ability to detect whether this deletion was done in error. For example, a user may add a custom annotation to trigger a custom transform, but the transform never runs because the annotation was deleted sometime earlier in the compilation process. One examples of a compiler which uses this approach is Yosys [48].

LLVM has a slightly different approach. Instead of tackling the complete problem of arbitrary metadata behavior, it restricts it to classes of metadata which have different propagation properties. All transforms must indicate which class of metadata it preserves, and which it invalidates. While this does provide more robustness to the metadata system, it does not allow for more fine-grained control over annotation-propagation behavior. [27]

Annotations

The first important realization is that the metadata should not be kept in the AST, but instead in a separate datastructure. This separation enables the propagation behavior of IR nodes to be distinct from the propagation behavior of their metadata.

Secondly, each annotation contains its metadata, as well as references to IR nodes in the circuit. To support this, the FIRRTL compiler framework introduces the *target* language (see the next subsection). This separation also provides the added benefit of enabling one annotation to refer to multiple targets in the design.

Thirdly, changes to nodes in the AST (renaming, deletion, splitting, merging) must be reported by all transformations and passed to each annotation's update method, for which the annotation can implement its desired propagation behavior.

To support these realizations, the FIRRTL compiler passes a *CircuitState* data structure to/from each transformation; a *CircuitState* contains three values: the FIRRTL AST, the annotations in a *AnnotationSeq*, and a *RenameMap* which contains a representation of the AST changes of that transform. After the transformation is run, an *update* method of every annotation is called with the *RenameMap*, enabling annotations to dictate their own propagation behavior. Because transforms can also add or delete annotations, the *AnnotationSeq* prior to transformation is compared to the one returned to detect deleted annotations; for every deleted annotation, a *DeletedAnnotation* is added to keep a record of all deleted annotations (for debugging).

Although transforms can arbitrarily transform/modify IR AST, they must keep the *RenameMap* in sync with their changes to the AST. While future work could be to expose AST-modifying APIs that automatically updated the *RenameMap*, this requires significant work to support without limiting the power of arbitrary transformations.

Annotations can contain arbitrary fields, including any number of references to FIRRTL nodes through targets (see next section). The only requirement is for every annotation to implement an update method that accepts a RenameMap and returns a list of new annotations.

RenameMap is a structure that maps from old IR node targets to a list of new IR node targets. This mapping of one-to-many can represent renaming ('A' → ['B']), deleting ('A' → []), splitting ('A' → ['B', 'C']), and merging ('A' → ['C'], 'B' → ['C']).

To illustrate the capabilities of FIRRTL's annotation system, consider the following circumstances for resolving conflicting behavior between annotations and transformations.

First, one transform deletes a signal that another transform was relying on metadata to preserve. One possible solution is for the signal-preserving transform's annotation to error if it detects a deletion in the *RenameMap*. While this returns a valid error message, this transform doesn't support scaling to a larger ecosystem. A second solution is for the signal-deleting transform to provide an annotation which exempts a signal from being deleted; the user of both transformations can then use this annotation to enable both transforms to interoperate.

A second circumstance is for one transform to delete an annotation that is required for a second transform. A reasonable solution for the second transformation is to enforce its requirement by searching through the *DeletedAnnotation* list and error gracefully if it finds its annotation was deleted (rather than the default behavior of silently failing). A solution that is similar to a solution if the previous circumstance is to update the first transform to enable exempting the annotation deletion with an exemption annotation.

A final circumstance is for one transform to rename, split, or delete an IR node which has associated metadata. As described previously, an annotation can customize its propagation behavior via its update method.

The main limitation of FIRRTL's annotation system are that changes to the AST must be able to be represented as a deletion, a renaming, a split, or a merge of only nodes with names. This limitation means AST changes like reordering nodes, changing the node types (which have no name), updating signal widths, expanding unnamed nodes (e.g. the conditional when

statement), or changing the connections between IR nodes cannot be directly represented in the *RenameMap*. While intermediate expressions have no name, it is straightforward to rewrite that expression as a node statement (with a name), and a new reference to that statement, so this limitation in practice is not an issue. In the author's experience, this interface has been sufficient for all annotation use-cases. However, future work could be to expand the set of AST changes that could be recorded by a transformation, as well as for the *RenameMap* to also contain a reference to the transformation that generated these changes, enabling even more annotation-propagation customization.

Target

Thus far, this chapter has mentioned that annotations must refer by name to the FIRRTL components they annotate, but supporting mechanisms for representing names and their renaming semantics have yet to be formally introduced. This section introduces FIRRTL's naming mechanism *target*, its design justifications, the renaming semantics of a *target*, and many illustrative examples.

This section will often refer to the following FIRRTL example. In the top-level module `Top`, two instances of module `Leaf` called `foo` and `bar` are instantiated. `Top`'s input port `in` is connected to `Top`'s output port `out` through a path going through both `Leaf` instances.

Example 4.4:

```
circuit Top:
  module Leaf:
    input in: UInt<1>
    output out: UInt<1>
    out <= in
  module Top:
    input in: UInt<1>
    output out: UInt<1>
    inst foo of Leaf
    inst bar of Leaf
    foo.in <= in
    bar.in <= foo.out
    out <= bar.out
```

In the FIRRTL compiler, a *target* can refer to a *circuit*, a *module*, an *instance*, or a *reference* (meaning any named non-instance component contained in a module, including a subfield or subindex). In addition, a *target* may specify a path through the instance hierarchy to reference an instance-specific reference or instance. This instance-specific component, while conceptually distinct, has no distinct representation in the AST; *target*'s instance-specificity feature cannot be implemented by a different mechanism that directly annotates the AST with metadata.

A *target* has four subclasses: *CircuitTarget* refers to a circuit, *ModuleTarget* refers to a root module, *InstanceTarget* refers to an instance (starting from a root module), and *ReferenceTarget* refers to a named signal (or a subfield, subindex, or the special register

Target	Symbol	Example	Meaning
CircuitTarget	~	~Top	circuit Top
ModuleTarget	|	~Top|Leaf	module Leaf in circuit Top
InstanceTarget	/,:	~Top|Top/foo:Leaf	instance foo of module Leaf in module Top in circuit Top
ReferenceTarget	>	~Top|Top>foo	reference foo in module Top in circuit Top
		~Top|Leaf>out	reference out in module Leaf in circuit Top
		~Top|Top/foo:Leaf>out	reference out in instance foo of module Leaf in module Top in circuit Top
		~Top|Top>foo.in	reference foo.in in module Top in circuit Top

Table 4.2: Target concrete syntax for *CircuitTarget*, *ModuleTarget*, *InstanceTarget* and *ReferenceTarget*. Note that instances like `foo` in module `Top` could be referred to with either an *InstanceTarget* or an *ReferenceTarget*. All examples reference components within Example 4.4.

ports for clock, reset, or init). Table 4.4 contains the *target* concrete syntax, as well as more examples and descriptions. The *target* mechanism was designed with the following in mind.

Specify only component name and hierarchy. Other AST information such as a component's type (e.g. `UInt` vs `SInt`) or kind (e.g. `reg` vs `wire`) is not included in the target. Although useful, it duplicates information that is already specified in the AST and creates more opportunities for errors (i.e. updating the AST but not the target, and vice versa). Keeping the names and hierarchy information in sync is difficult enough; no reason to make it more difficult for minimal added gain.

No "local-only" *target* type. Originally, the authors considered capturing whether a reference was directly contained in the root module (versus instance-specific) into the type of the *target* (e.g. *LocalReferenceTarget* and *NonLocalReferenceTarget*). After experimentation they concluded that this "local-only" restriction significantly complicated the implementation and encouraged subtle anti-patterns. For example, an annotation might require local-only *targets*, which appears to be a reasonable thing to do. However, this requirement is never needed because an instance-specific component can always be converted to a local component through transforming the AST by duplicating all modules in the component's instance-specific hierarchy (and updating each instance in the hierarchy to instantiate the new duplicated module). Then, the instance-specific *target* can become a local *target* whose root is the final duplicated module (which is now only instantiated once). This duplicating circuit transformation is provided to enable easily converting instance-specific *target*'s to local ones, thus eliminating the need for an annotation to require a local *target* type.

Separate *InstanceTarget* from *ReferenceTarget*. A renaming corner-case is to change which module an instance refers to. This change must be captured in the renaming framework, thus requiring a separate *target* type for instances.

Renaming

As mentioned previously, every transform must populate a *RenameMap* with changes to the circuit; for example if a transform renames a module from `Leaf` to `Blah` in circuit `Top`, the returned *RenameMap* must have the entry mapping the old target to the new target: `~Top|Leaf => (~Top|Blah)`. After the transformation, every target in each annotation can be updated to the new target (or an error can be thrown if the rename is unexpected).

Because it can be difficult to figure out how a target is renamed when pieces of its path or components have changed, the *RenameMap* provides a `get` method which, given any target, returns a set of updated targets. The two rules for calculating a new target under changes are as follows:

1. All changes in *RenameMap* are final - no sub-piece can be renamed by another change contained in the *RenameMap*. For example, under the changes `~Top|Leaf => (~Top|Blah)` and `~Top => (~NewTop)`, the module `~Top|Leaf` is renamed to `~Top|Blah`, rather than `~NewTop|Blah`.

2. Check all relative paths to component (and subcomponents), before checking path to leaf module

Table 4.4 contains multiple examples illustrating this automatic renaming capability.

The following example demonstrates the order each sub-piece of a target is checked against the existing recorded changes. First the all relative paths to the component and subcomponents are checked. Then, all sub-pieces of the path to the component are checked:

Example 4.5:

```
; First check relative paths to component and subcomponent
~Top|Top/a:A>b.c
~Top|Top/a:A>b
~Top|A>b.c
~Top|A>b
; Next check all pieces of the path to leaf module
~Top|Top/a:A
~Top|A
~Top|Top
```

Multiple rename maps to be created from one transform and chained, enabling *RenameMaps* to be consecutively applied to a target to obtain a multiply-renamed target (if desired). For example, a deduplication transformation has one *RenameMap* containing how each original module is renamed to an instance, and a second *RenameMap* mapping the original modules to the deduped module.

The main limitation of this approach is its compute-intensive nature; changes must be calculated for every target in every annotation after every transformation. To speed this process up, renames are cached between multiple `get` calls to the same *RenameMap*. In addition, a quick check is first performed whether a given target is sensitive to a *RenameMap*, or shares a module in its path that is also in the *RenameMap*. These two optimizations significantly speed up the renaming process.

4.5 Circuit Traversals

The in-memory structure of a FIRRTL design significantly influences how easily transformations are written. As is commonly done in software compilers, a FIRRTL design is internally represented with an abstract syntax tree (AST) structure, where nested elements are walked recursively to modify the circuit. If non-local information is necessary, transformations first walk the tree to build a custom data structure, then walk the tree a second time to modify the circuit.

An AST representation for the IR was chosen over a graph representation because an AST can represent more complex nodes including when statements and bulk connects, which do not have a direct representation in a graph. In addition due to their non-cyclical nature, AST's have more predictable traversals, are easier to keep internally consistent, and are easy to convert to a human-readable form. All of these worked well given the goals of making the compiler infrastructure extendable.

However, some transformations do require other representations of the design to compute additional circuit information. Combinational loop detection, for example, requires a netlist-like directed graph to compute connectivity information. FIRRTL's compiler framework has an accompanying directed graph library which builds from the AST representation; transformations can use this library to build a graph and perform graph traversals.

AST Traversals

The FIRRTL AST consists of IR nodes represented by an in-memory object, each of which is a subclass of one of the following IR abstract classes: *circuit*, *module*, *port*, *statement*, *expression*, *type*. Each IR node can have children objects of other IR node classes, the relationship of which is shown in Figure 4.3. Figure 4.4 demonstrates how a FIRRTL circuit is represented in-memory as an AST of IR nodes.

The following recursive algorithm visits all *expression* nodes in a circuit: First, visit each module's *statement* nodes. For each visited *statement*, visit each of its children *statement* and *expression* nodes. For each visited *expression*, visit each of its children *expression* nodes.

All transformations use these recursive walks of the FIRRTL AST to modify the circuit.

Because transformations always consume and produce a well-defined AST circuit and easily pipe one-after-another, constraints on the design can be checked after each transfor-

Description	RenameMap Entry	Automatic Renames

Rename module `Leaf` to module `Blah`	`~Top|Leaf => (~Top|Blah)`	`~Top|Leaf => (~Top|Blah)`
		`~Top|Leaf>in => (`
		`  ~Top|Blah>in`
		`)`
		`~Top|Top/foo:Leaf>in => (`
		`  ~Top|Top/foo:Blah>in`
		`)`

Rename reference `in` in module `Leaf` to reference `x` in module `Leaf`	`~Top|Leaf>in => (~Top|Leaf>x)`	`~Top|Leaf => (~Top|Leaf)*`
		`~Top|Leaf>in => (`
		`  ~Top|Leaf>x`
		`)`
		`~Top|Top/foo:Leaf>in => (`
		`  ~Top|Top/foo:Leaf>x`
		`)`

Rename module `Leaf` to module `Blah` AND rename reference `foo` in module `Top` to reference `blah` in module `Top`	`~Top|Leaf => (~Top|Blah)`
	`~Top|Top>foo => (`
	`  ~Top|Top>blah`
	`)`
		`~Top|Leaf => (~Top|Blah)`
		`~Top|Leaf>in => (`
		`  ~Top|Blah>in`
		`)`
		`~Top|Top/foo:Leaf>in => (`
		`  ~Top|Top/blah:Blah>in`
		`)`

Rename reference `in` in module `Leaf` to references `xi`, `xj` in module `Leaf`	`~Top|Leaf>in => (`
	`  ~Top|Leaf>xi,`
	`  ~Top|Leaf>xj`
	`)`
		`~Top|Leaf => (~Top|Leaf)*`
		`~Top|Leaf>in => (`
		`  ~Top|Leaf>xi,`
		`  ~Top|Leaf>xj`
		`)`
		`~Top|Top/foo:Leaf>in => (`
		`  ~Top|Top/foo:Leaf>xi,`
		`  ~Top|Top/foo:Leaf>xj`
		`)`

Table 4.3: These examples demonstrate how rename changes entered into a *RenameMap* would be applied to other targets. Specifically, this example demonstrates how the three targets `~Top|Leaf`, `~Top|Leaf>in`, and `~Top|Top/foo:Leaf>in` are renamed under different *RenameMap* entries. All these examples reference components within Example 4.4. Rename results marked with * are unchanged by the *RenameMap*

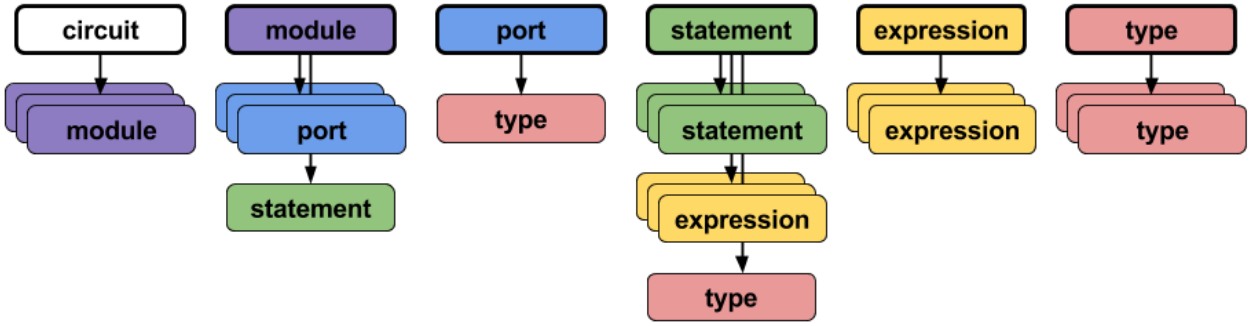

Figure 4.3: A FIRRTL circuit is represented using these AST nodes. Each can have one or many different children nodes of various types. For example, a FIRRTL *statement* can have children *statements*, *expressions*, and/or a *type*.

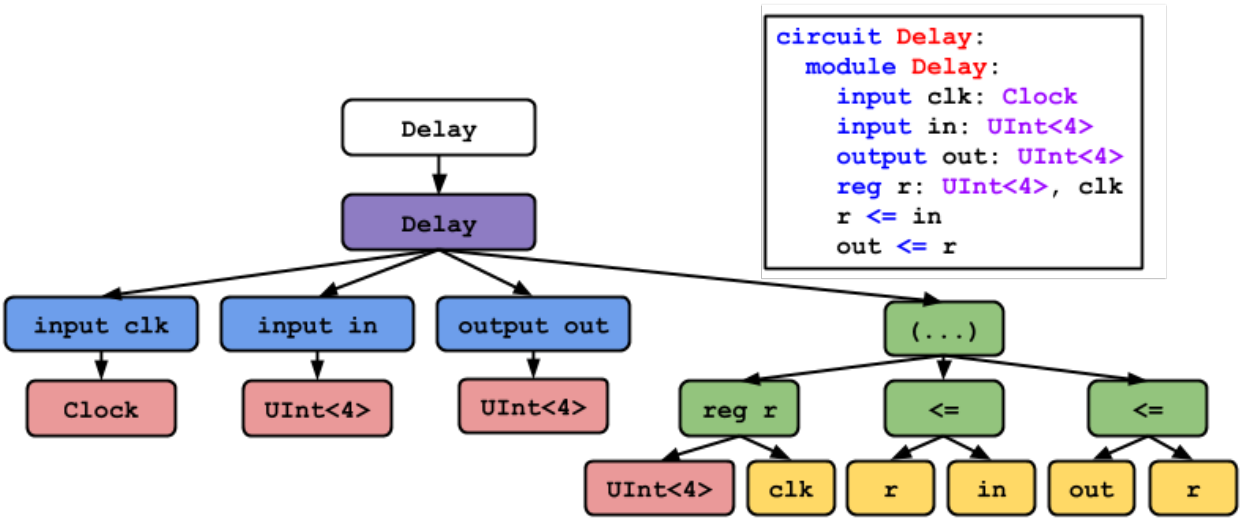

Figure 4.4: An example FIRRTL circuit in its AST versus textual representation. This circuit contains a single module that outputs the input signal delayed by one cycle. The `(...)` statement is a block statement that only contains multiple children statements - this node makes it easy to replace a single statement with multiple statements in a single walk of the AST.

mation. This structure makes inserting new transformations straightforward and safe, unlike the use of brittle, ad-hoc scripts.

If a transformation introduces a bug, it is straightforward to understand what happened, as the circuit state is visible between transformations. This is in direct opposition to existing non-compiler methodologies of python/perl scripts, which have no clear intermediate representation, and thus are extremely brittle.

To express a recursive walk, every IR node has implemented a custom **map** function; a node's **map** applies a user-specified function to the subset of children whose node-type matches the function's input and return node-type.

The following example demonstrates calling a *module*'s **map** with a function that accepts and returns a *port*, and with a function that returns a *statement*.

While simple, using **map** to recursively walk the FIRRTL AST is extremely powerful. The following example is an optimization transformation that uses the **map** pattern to perform constant propagation over muxes with constant predicates. First, walk all FIRRTL modules, statements, and expressions recursively by calling **map** on modules, statements, and expressions. For any mux seen, check the constant propagation condition and, if true, perform the optimization. Note that this code visits expressions in postorder traversal, requiring only one pass through the AST.

Graph Traversals

While the AST traversals are very powerful, there are some transformations, analyses, checks, or novel features which require the ability to know or query circuit topological information. This requires the FIRRTL compiler infrastructure to provide users a mechanism to view a FIRRTL circuit as a directed graph, where components are nodes and connections are directed edges between components.

The following use cases illustrate the need for a general graph-traversal solution:

- **Optimizations** - dead-code elimination, constant propagation (through modules).
- **Analyses/Checks** - determining a module's clock domain, determining clock crossings (e.g. "return a path, if it exists, between every pair of registers whose clocks are different.")
- **Novel Features** - structural assert (e.g. ensure all paths between two specified signals take exactly N number of cycles.)

While a generic graph-traversal solution is important to support these use cases, FIRRTL's solution cannot be too complicated to use, especially given the short supply of CAD developers with compiler knowledge. Enabling these specific topological queries requires a lightweight, fast, space-efficient, and semantically simple graph-traversal solution.

This section first goes into more detail about why a generic graph-traversal solution is difficult and why previous approaches fail to address the needs of this compiler. Introduced next is *ConnectivityGraph*, FIRRTL's solution and implementation of a generic graph-traversal library, followed by an explanation of additional implementation details. Finally, a fast and easy-to-use implementation working on a variety of use cases is demonstrated. The following sections will often reference FIRRTL example 4.8.

```
circuit D0:
  module D0:
    input in: UInt<8>
    output out: UInt<8>
    inst c1 of D1
    inst c2 of D1
    c1.in <= in
    c2.in <= c1.out
    out <= c2.out
  module D1:
    input in: UInt<8>
    output out: UInt<8>
    inst c1 of D2
    inst c2 of D2
    c1.in <= in
    c2.in <= c1.out
    out <= c2.out
  module D2:
    input in: UInt<8>
    output out: UInt<8>
    inst c1 of D3
    inst c2 of D3
    c1.in <= in
    c2.in <= c1.out
    out <= c2.out
  module D3:
    input in: UInt<8>
    output out: UInt<8>
    out <= in
```

Example 4.8: This example contains four modules named in the form `module D[N]`, where N is the module's depth in the instance hierarchy. Each non-leaf module instantiates two children instances, `c1` and `c2`, of `module D[N+1]`. The module then connects its input port `in` to its output port `out` through a path going through both child instances.

Challenges and Related Work

Existing CAD tools represent their netlists with graph structures. The range of internal representations of these graphs is not too large - they usually consist of an array-of-structs, where each net has forward and backward pointers to each net it drives/is driven by. In general this solution works well but requires the entire design to be flattened which expands the memory footprint and limits the tool's ability to exploit multiply-instantiated modules for performance. To avoid these downsides and determine a sufficient underlying representation, this section details the demands placed upon a hardware compiler framework's graph-traversal library.

(1) Efficient Hierarchical Node Representation. Representing a hierarchical design

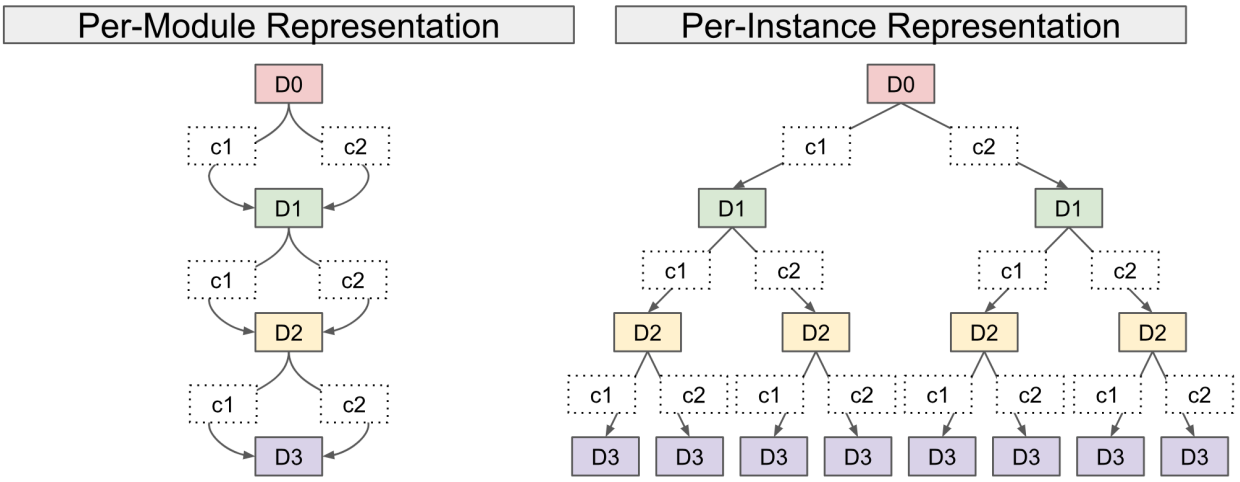

Figure 4.5: This diagram depicts two representations of FIRRTL example 4.8. The first is a per-module representation, where every module is represented once and instances of them all point to the same node. The second is a per-instance representation, where there is a distinct node for each instance of a module. A per-module representation is space-efficient, but could not resolve a connectivity query about a specific per-instance component. For example, querying the connectivity of `~D0|D0/c1:D1>out` is different than the connectivity `~D0|D0/c2:D1>out`. Resolving these per-instance queries would be impossible in a per-module representation as all instances share a node but have distinct connectivity. However, it is important to note that given a design with an instance hierarchy depth D and C children instances per module, the per-module representation size is $O(D * C)$, but per-instance representation size is a massive $O(C^{D+1})$!

without flattening has complications. Queries like determining a net's clock domain requires a per-instance, not per-module, query. In other words, determining the clock domain of a module's register is an under-specified problem if that module is instantiated multiple times. Thus, a hierarchical graph representation must be able to resolve per-instance queries, while still retaining a per-module representation.

Another complication to this problem of designing a graph connected component representation is the ability to represent subproblems efficiently. Often, RTL modules could instantiate a child module multiple times, and that module could also instantiate multiple child modules, etc. If the netlist can support this nested hierarchical structure, it can drastically reduce the size of the graph because it does not need to flatten the entire design.

(2) Flexible Edges. An edge of a graph-traversal library may represent something other than a straightforward hardware connection. For example, one transform may want to walk the circuit connectivity, while another may want to walk the instance hierarchy. Another example is determining path delays between registers versus dead-code elimination; the first does not include the path through the register, while the second requires traversing that

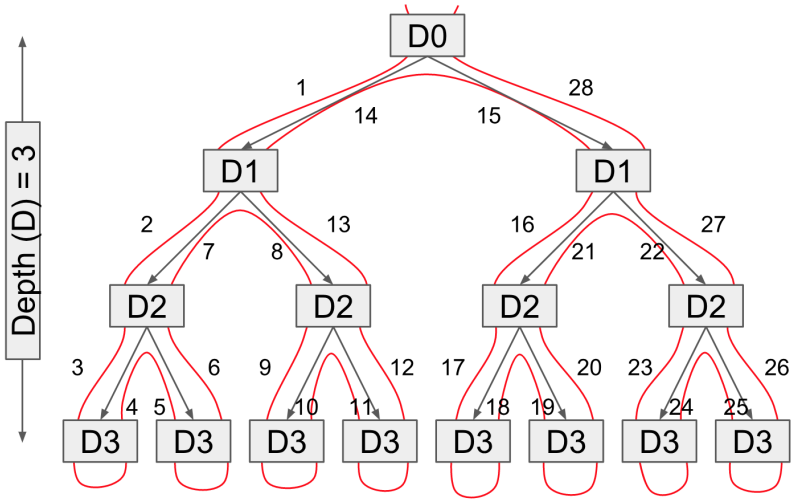

Figure 4.6: This diagram depicts the path length from FIRRTL example 4.8 between `~D0|D0>in` and `~D0|D0>out`. In this specific example, the worst case path length given $D = 3$ and $C = 2$ is 28 edges.

connection.

(3) Limit Exponential Path Lengths. Connectivity paths can be exponential in the depth of the instance hierarchy. For a generic FIRRTL circuit with an instance hierarchy depth D and C children instances per module, the worst-case path length is $(2 * C) * (1 - C^D)/(1 - C)$, which is $O(C^D)$.

An important note is that this path length will be present regardless of node representation - neither inlining every instance with a per-module representation nor a per-instance representation can address this length. While some connectivity queries are naturally limited (e.g. computing combinational delays between registers will naturally not have excessively long paths), other queries for use cases like dead-code elimination can have these long paths.

Finally, while a path delay is calculated on paths in the graph that in practice aren't too long because they are bounded by the frequency of the chip, other graph queries including dead-code elimination could walk paths which traverse up and down the hierarchy. While the nets in the path can have a hierarchical representation, the path itself may require traversing the same path through a sub-module multiple times. In the worst case, this path could be exponential with the depth of the hierarchy (see Figure 4.9).

(4) Understandable yet fast APIs. An extendable hardware compiler framework must allow new users who are not compiler experts to write their own analyses. However, there is often an implicit tradeoff between simple yet slow interfaces and fast yet complex interfaces; an ideal solution is both simple and fast.

Simple graph interfaces often enable a program to walk the graph using known algorithms (BFS, DFS) to resolve a query. For example, Yosys supports a find-connected-component query by walking the graph. However, this interface can be difficult to speed up; if a user

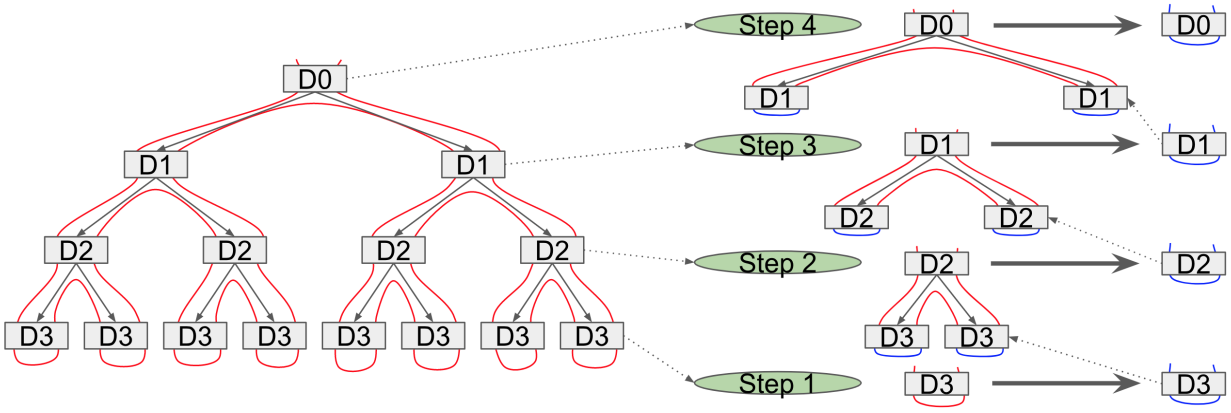

Figure 4.7: Topological search requires sorting FIRRTL modules by topological order leaf-to-root, solve a module's sub-problem, then use that solution to solve the next module's sub-problem. In this figure, the connectivity in FIRRTL example 4.8 from `~D0|D0>in` to `~D0|D0>out` is followed by first solving how D3's ports are connected, then use that solution to solve how D2's ports are connected, etc. While as a generic solution topological search would be fast and space efficient, it would also unnecessarily solve all sub-problems of point-to-point connectivity queries and force a user to explicitly (and often non-trivially) divide their circuit query into per-module queries.

wants to query many different components, each query walks the graph and causes a massive slowdown for large numbers of queries.

On the other-hand, complex interfaces can be very fast, but difficult to understand or reuse. *Topological search* is an interface that can be used to address challenges $1 \rightarrow 3$, but fails short on exposing a generic yet understandable API (Figure 4.7). Topological search works by sorting modules by topological order leaf-to-root. Then, a user solves the module-specific sub-problem of their general problem. Finally, the user uses this partial solution to solve the parent sub-problem (repeat until entire circuit is solved). The pros of this approach are that it has linear time O(number of modules), is straightforward to implement, and is good for global analyses. However, it unnecessarily solves all sub-problems which is bad for user-friendly point-to-point connectivity problems. The biggest downside, however, is the user must explicitly (and often non-trivially) divide problem into sub-problems, which is too much to handle for a novice user.

The remainder of this section details the FIRRTL hardware compiler framework's solution to these challenges which is encapsulated by a *Connectivity Graph*. First its underlying representation, *virtual occurrence graph*, is motivated and discussed. Then, its *separation from the AST* is motivated, followed by a traversal algorithm *memoizing search*. For examples of uses of this library, see Section 4.6.

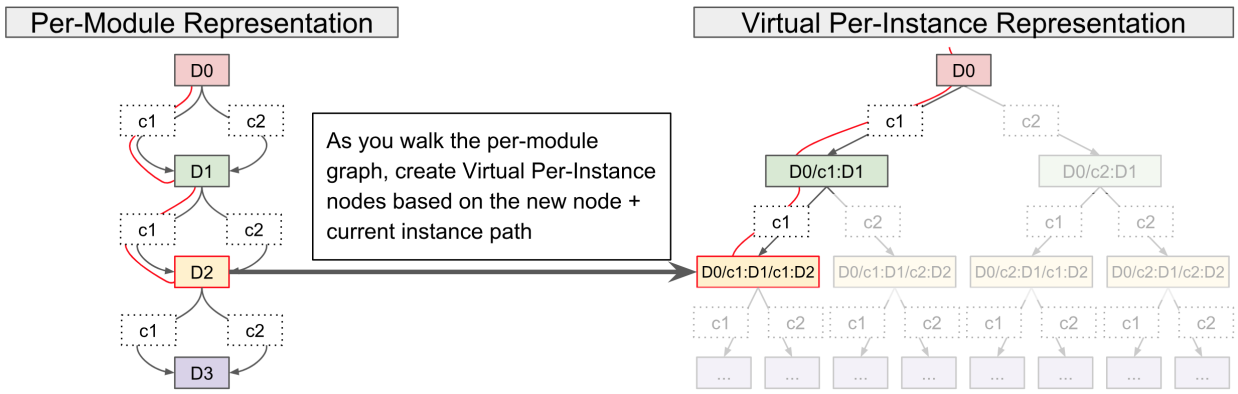

Figure 4.8: Virtual occurrence graphs maintains an underlying per-module graph and dynamically constructs the corresponding per-instance nodes along its search path. In this example circuit from 4.8, the traversal is started on a virtual occurrence graph from node `~D0|D0>in`. If a search traverses to the instance `c1`'s port in, the underlying per-module representation `~D0|D1>in` is converted to a new per-instance node `~D0|D0/c:D1>in`.

Virtual Occurrence Graph

While a per-instance graph representation is exponential in space, a virtual occurrence graph only represents the per-instance nodes along its search path. It does this by creating a graph with a per-module representation, but any query starts with the per-instance node. When descending into an instance, the new per-instance node is updated to include this instance and module in its instance hierarchy. When ascending from an instance, drop the last instance and module from the new node's instance hierarchy.

A virtual occurrence graph starts with a space-efficient representation and maintains its small size if queries do not have excessive path lengths. However, this type of graph does not address the time and space concerns if a search continues along a path that explores large parts of the design, as the virtual occurrence graph must still create a new node per unique instance component it traverses.

Separate Graph Representation from AST

Instead of creating a "standard" graph representation of a FIRRTL circuit, a design's AST structure is used as the "ground-truth" of the design from which to build a new and separate graph data-structure that can be queried. This separation provides a few benefits.

First, an AST structure is often conceptually easier to transform as well as keep internally consistent. If a circuit must be transformed based on the results of connectivity query, an AST traversal can be used make the desired modifications. FIRRTL's HCF already has good mechanisms to walk and modify ASTs - this allows users to only learn one concept for circuit transformation.

Secondly, the construction of a graph from an AST is a customizable process, enabling users to customize which 'edges' exist (not just AST connections). For example, suppose a user wanted a graph representation of the instance hierarchy, where nodes were instances and edges were instantiations. This would be straightforward to implement given this separation.

Thirdly, users can override the generic graph's *getEdges* function, giving the most amount of flexibility. This allows users to filter edges based on the from and to nodes, as well as external information (labeled edge information or previously searched nodes).

Finally, because there is a separate FIRRTL mechanism to represent references to named FIRRTL components called *target* (see Section 4.4, this mechanism also works perfectly to represent nodes in both per-module representations (as "local-only" targets) as well as per-instance representations.

Memoizing search

Memoizing search reduces the length of paths explored by only traversing each module-specific path once. To do this, a virtual occurrence graph is first traversed. When the search enters and leaves a module, the entrance and exit ports are stored in a separate data-structure. If the search enters a different instance of a module through a previously-explored entrance port, it can immediately jump to the known exit port. Since a memoizing search only visits each module entrance-port once, the corresponding virtual occurrence graph remains small in its representation. In addition, all worst case path lengths are reduced to be no greater than the size of the underlying per-module representation. Note that memoizing search only performs necessary work to find a solution, and is thus equally effective at solving both global connectivity analyses and point-to-point connectivity queries.

In order for memoizing search to work properly, a few subtle implementation details are necessary. First, not all searching algorithms can be supported; a search must be a priority-search where signals deeper in the instance hierarchy have priority over shallower signals. This preserves the invariant that, before ascending from an explored module, that all module port exits from the search's entrance port are known. Otherwise, it is possible for the search to encounter another instance of a module via an entrance port for which not all corresponding module exits are known.

Secondly, like how connectivity information can be effectively memoized, search results that consist of per-instance nodes (e.g. a clock domain search results in a per-instance clock source) must also be similarly memoized as a per-module solution, and reconstructed when visiting a new instance.

4.6 Transformation Evaluation

This section describes many different circuit transformations which are supported by the compiler, ranging from straightforward optimizations to complex circuit analyses. Ideally, users can rely on these transformations to keep their source-code more platform-agnostic;

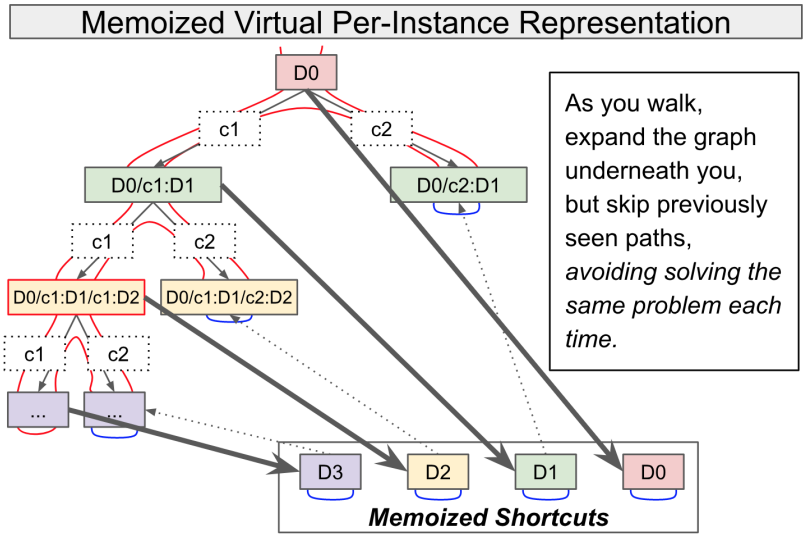

Figure 4.9: Memoizing search reduces the length of paths explored by only traversing each module-specific path once. A memoizing search on FIRRTL example 4.8 starts from `~D0|D0>in`. The search then enters `module D1` through `~D0|D0/c1:D1>in` and eventually exits through `~D0|D0/c1:D1>out`. At this time, the connectivity from `~D0|D1>in` to `~D0|D1>out` is memoized. When the search again enters `module D3` through `~D0|D0/c2:D1>in`, it matches the memoized connectivity and the search immediately returns `~D0|D0/c2:D1>out` without traversing `module D3`. Note that prior to memoizing `module D1`, `module D2` and `module D3` have already been memoized.

when backend-specific customizations are reflected in source code changes, it limits code reusability.

To demonstrate the wide-ranging applicability and utility of this hardware compiler framework, this section describes and evaluates the collection of transformations: (1) lowering transformations enabling FIRRTL's IR to capture user intent, but remain simple; (2) optimizations which support a similar degree of optimizations that CAD tools can employ; (3) analyses including clock-crossing detection; (4) topology transformations which can modify a design's instance hierarchy; (5) instrumentation transformations for facilitating test coverage; (6) FPGA/ASIC specialization transformations which enable a Chisel design to remain platform-agnostic; (7) a custom transform case study of a chip tape-out which employed custom transformations to solve physical design issues.

Lowering Transformations

Designing an IR is an important part of any compiler, and this section considers three desirable, yet sometimes competing, qualities:

- *clear*: semantically straightforward
- *simple*: small set of IR nodes
- *rich*: captures user-intent

All tools that manipulate RTL or gate-level designs have an IR that they operate on, whether rigorously defined or not. Each tool's IR makes differing tradeoffs depending on their use: an IR for operating only on behavioral Verilog-2005 should be more rich but less clear and simple than an IR operating solely on netlists.

FIRRTL's IR represents RTL digital circuits and is designed to specialize source RTL code from underlying implementations.

As such, FIRRTL first prioritizes richness to capture as much source RTL user intent as possible.

For example, FIRRTL contains explicit memory nodes, aggregate types, a clock type, and typesafe connections to enable other languages, like Chisel, to map to these constructs and capture the user's intent.

Since FIRRTL's compiler framework must eventually emit a less-rich representation for downstream simulators and tools, FIRRTL is also simple. Finally, FIRRTL is clear because it is rigorously defined and has straightforward width inference and type inference rules.

Lowering transformations take a FIRRTL circuit and simplify it to a lower form, enabling the IR to remain both rich and simple (see Section 4.2 for more information about FIRRTL forms). There are two lowering transformations: (1) *high-to-mid*, which takes in high form and emits middle form; (2) *mid-to-low*, which takes in middle form and emits low form.

One task of the high-to-mid transformation is to remove FIRRTL's bulk-connect operator. This operator allows components with aggregate types to be connected in a type-safe manner with a single statement, capturing user intent. However, lower forms only support connections between primitive types, so the high-to-mid transform rewrites the bulk-connect into a series of individual connections.

For example, the high-to-mid transformation would rewrite the conditional statement and aggregate-typed connections of the following module:

Example 4.9:

```
module MyModule :
   input in: {a:UInt<1>, b:UInt<2>[3]}
   input clk: Clock
   output out: UInt
   wire c: UInt
   c <= in.a
   reg r: UInt[3], clk
   r <= in.b
   when c :
      r[1] <= in.a
   r[0] <= in.b[0]
   r[1] <= mux(c, in.a, in.b[1])
```

```
    r[2] <= in.b[2]
    out <= r[0]
```

Then, the mid-to-low transformation would expand the aggregate typed components into multiple primitive-typed components, as well as inferring all unknown widths:

Example 4.10:

```
module MyModule :
    input in: {a:UInt<1>, b:UInt<2>[3]}
    input in_a: UInt<1>
    input in_b_0: UInt<2>
    input in_b_1: UInt<2>
    input in_b_2: UInt<2>
    input clk: Clock
    output out: UInt<2>
    wire c: UInt<1>
    c <= in.ain_a
    reg r: UInt[3], clk
    reg r_0: UInt<2>, clk
    reg r_1: UInt<2>, clk
    reg r_2: UInt<2>, clk
    r[0]r_0 <= in.b[0]in_b_0
    r[1]r_1 <= mux(c, in.ain_a, in.b[1]in_b_1)
    r[2]r_2 <= in.b[2]in_b_2
    out <= r[0]r_0
```

To demonstrate the utility of a rich IR, this section analyzes the following three designs: (1) a reorder-buffer, (2) a branch reorder-buffer, and (3) a register renaming free list. As the design's rich features are simplified into FIRRTL's middle and low forms, the lines of code required to represent the design are recorded. Finally, the FIRRTl compiler emits the design to Verilog. To ensure this compiler does not artificially inflate code size, the degree Yosys can reduce the Verilog line size through optimizations is also shown. Since emitters can emit different styles of Verilog, the code size of Yosys reading and writing Verilog without optimizations is shown. Finally, Yosys reads and writes the Verilog design with optimizations.

As shown in the Figure 4.10, some designs exhibit huge growths in code size during lowering, in spite of FIRRTL and Yosys optimizations; this illustrates how a rich IR can concisely express a design, if the designer or frontend chooses to use the rich features.

Optimization Transformations

The two major optimization transformations implemented are dead-code elimination, and constant propagation. Because downstream tools perform aggressive logic analysis and other optimizations, these two transformations have little effect on the gate-level design. They primarily affect the compilation performance of the HCF and backend CAD tools.

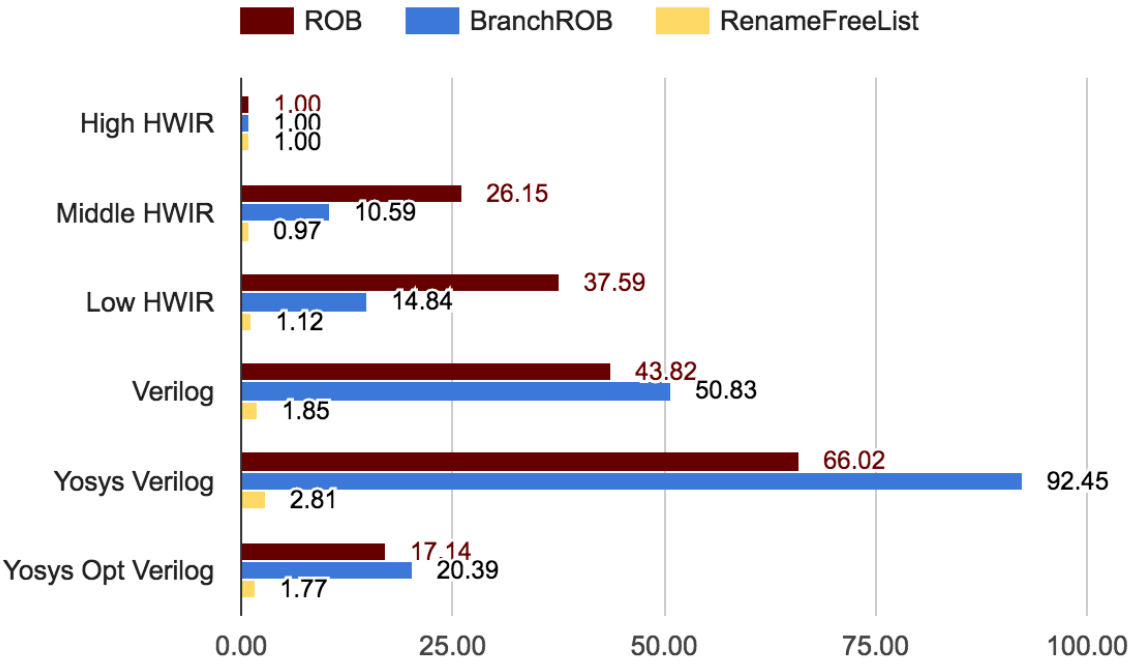

Figure 4.10: Code size normalized to size of representation in High FIRRTL. The ROB and BranchROB both use aggregate types, bulk connections, and memory nodes while the RenameFreeList is made primarily of logic and does not use rich FIRRTL features.

Dead-code elimination removes FIRRTL components which are never referenced in a design; for example, this transform removes registers whose outputs are never connected to another component. While useful in reducing the size of the design, it can be confusing for Chisel users who accidentally generate dead-code which is correctly, although unexpectedly, removed. To remedy this, this transformation also consumes a `DontTouchAnnotation` to indicate which components should not be eliminated.

The following example demonstrates a FIRRTL design before and after the dead-code elimination transformation, where the extraneous input port, output port, and connection in `module Leaf` are removed:

Example 4.11:

```
circuit Top:
  module Top:
    input in: UInt<1>
    output out: UInt<1>
    inst leaf0 of Leaf
    inst leaf1 of Leaf
    leaf0.in0 <= in
    leaf1.in0 <= leaf0.out0
    out <= leaf1.out0
```

```
  module Leaf:
    input in0: UInt<1>
    output out0: UInt<1>
    input in1: UInt<1>
    output out1: UInt<1>
    out0 <= in0
    out1 <= in1
```

Constant propagation is another optimization transformation which replaces signals with constant values, if a signal can be statically determined to always have one value. The following example demonstrates how this transformation can analyze the connectivity of signals through instances, and still propagate constant values; the constants propagates through the `leaf0` and `leaf1` instances, connecting constants `UInt(0)` and `UInt(1)` to the output ports in `module Top`:

Example 4.12:

```
circuit Top:
  module Top:
    output out0: UInt<1>
    output out1: UInt<1>
    inst leaf0 of Leaf
    inst leaf1 of Leaf
    leaf0.in0 <= UInt(0)
    leaf1.in0 <= leaf0.out0
    leaf0.in1 <= UInt(1)
    leaf1.in1 <= leaf0.out1
    out0 <= leaf1.out0UInt(0)
    out1 <= leaf1.out1UInt(1)
  module Leaf:
    input in0: UInt<1>
    output out0: UInt<1>
    input in1: UInt<1>
    output out1: UInt<1>
    out0 <= in0
    out1 <= in1
```

A synthesis tool like Yosys implements bit-level analysis and can thus perform more aggressive optimizations than FIRRTL can. However, as shown in Figure 4.11, these optimization passes ultimately reduce the synthesized standard cell count by up to 71% compared to up to 76% for Yosys. Running both FIRRTL's and Yosys's optimization passes results in even further cell count reduction. These optimization passes speed up execution time of backend CAD tools.

Analysis Transformations

Designers often desire insight into the compiler to understand the degree of optimizations taking place. Node-counting, early area estimations, and module hierarchy depictions are

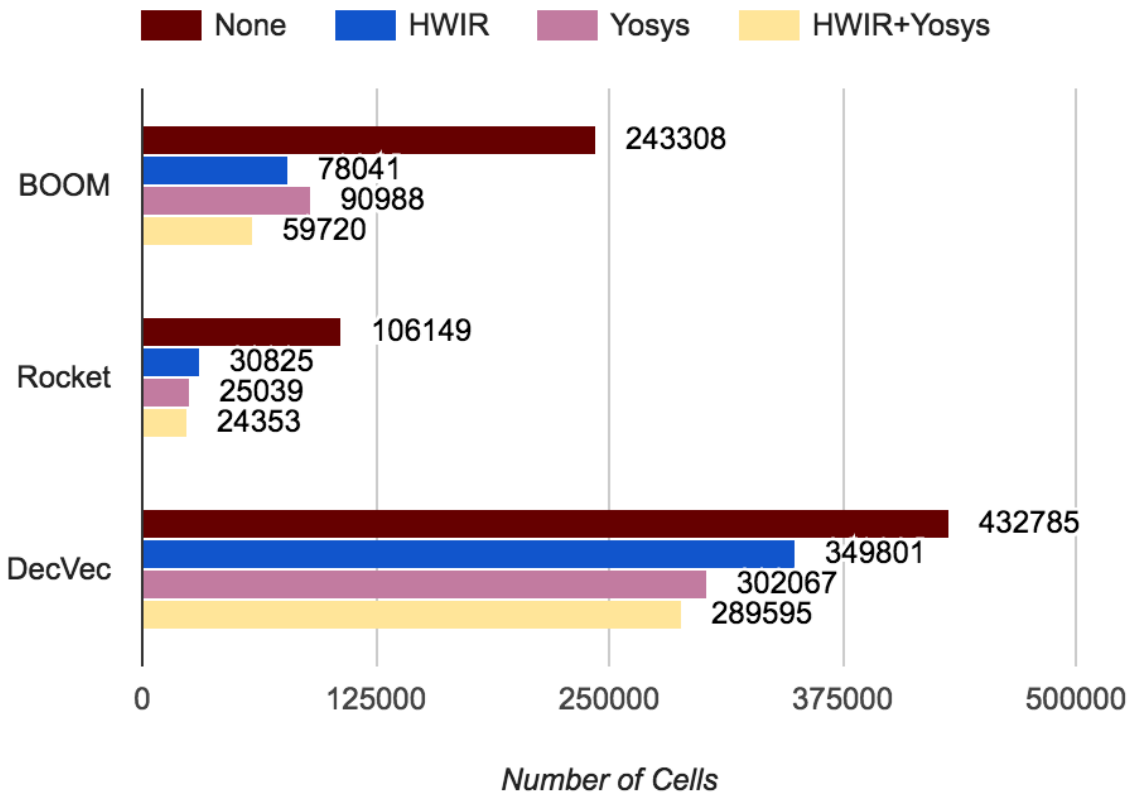

Figure 4.11: FIRRTL optimization passes ultimately reduce the number of synthesized standard cells by a similar degree as the optimization passes of Yosys, an open-source CAD synthesis tool.

three useful analysis transformations early in the design cycle.

A more complex analysis is to determine the clock sources of a signal in the design. This information can then be used to determine every circumstance where a register is driven by a signal which has a different clock source than the register's clock. When designing multi-clock circuits, every one of these clock-domain crossings is a potential source of bugs and any unintentional clock-crossing can be flagged by this analysis.

To demonstrate this analysis, the following example is used as well as the additional information that the top-level ports `~Top|Top>in` and `~Top|Top>clkSel` have the clock source of the other input port, `~Top|Top>clk`.

Example 4.13:

```
circuit Top:
  module Top:
    input clk: Clock
    input in: UInt<1>                    ; Has clock source clk
    input clkSel: UInt<1>                ; Has clock source clk
```

```
    output out: UInt<1>
    inst clkdiv of CLKDIVIDER
    clkdiv.clk <= clk
    inst leaf0 of Leaf
    inst leaf1 of Leaf
    inst leaf2 of Leaf
    leaf0.clk <= clk
    leaf0.in <= in
    leaf1.clk <= clkdiv.slow
    leaf1.in <= leaf0.in
    leaf2.clk <= mux(clkSel, clk, clkdiv.slow)
    leaf2.in <= leaf1.in
  extmodule CLKDIVIDER:
    input clk: Clock
    output slow: Clock
  module Leaf:
    input clk: Clock
    input in: UInt<1>
    output out: UInt<1>
    reg r: UInt<1>, clk
    r <= in
    out <= r
```

The clock-domain analysis transform analyzes this circuit and determines the unique clock source for each register in the design. Note that registers can be driven by multiple clocks:

- ~Top|Top/leaf0:Leaf>r → ~Top|Top>clk
- ~Top|Top/leaf1:Leaf>r → ~Top|Top/clkdiv:CLKDIVIDER>slow
- ~Top|Top/leaf2:Leaf>r → (~Top|Top/clkdiv:CLKDIVIDER>slow, ~Top|Top>clk)

The analysis then uses this information to determine all clock-crossings in the design, and returns a path, if it exists, between every pair of registers having distinct clock-domains.

For example, the registers ~Top|Top/leaf0:Leaf>r and ~Top|Top/leaf1:Leaf>r have distinct clock-domains, and thus the following path is returned from this analysis:

Example 4.14:

```
~Top|Top/leaf0:Leaf>r
~Top|Top/leaf0:Leaf>out
~Top|Top>leaf0.out
~Top|Top>leaf1.in
~Top|Top/leaf1:Leaf>in
~Top|Top/leaf1:Leaf>r
```

Topology Transformations

The following transformations modify a FIRRTL circuit by changing the instance hierarchy of modules. These transformations are often used to facilitate physical design, which may require additional constraints on modules that differ from the logical grouping of signals. Most of the transforms described in this section operate on the following FIRRTL example circuit, where a top-level module instantiates a leaf module twice:

Example 4.15:

```
circuit Top:
  module Leaf:
    input in: UInt<1>
    output out: UInt<1>
    out <= in
  module Top:
    input in: UInt<1>
    output out: UInt<1>
    inst foo of Leaf
    inst bar of Leaf
    foo.in <= in
    bar.in <= foo.out
    out <= bar.out
```

Inlining is the process of replacing an instance with all of the components contained in the instance. Inlining is necessary to represent the entire design as a single module. The following example inlines `~Top|Top/bar:Leaf`; note that all ports of an inlined module are replaced with wires, and the name of the instance is prefixed to each inlined component's name.

Example 4.16:

```
circuit Top:
  module Leaf:
    input in: UInt<1>
    output out: UInt<1>
    out <= in
  module Top:
    input in: UInt<1>
    output out: UInt<1>
    inst foo of Leaf
    inst bar of Leaf          ;Remove instance bar
    wire bar_in: UInt<1>      ;Insert bar port as prefixed wire
    bar_out <= bar_in         ;Insert bar connections with prefixes
    foo.in <= in
    bar._in <= foo.out
    out <= bar._out
```

Grouping is the process of creating a new module that contains a subset of a different module's components; this new module is then instantiated in place of this component subset.

Grouping is the inverse operation of inlining. The following example groups two instances, ~Top|Top/bar:Bar and ~Top|Top/foo:Foo, into a new module ~Top|FooBar, instantiated as ~Top|Top/foobar:FooBar:

Example 4.17:

```
circuit Top:
  module Leaf:
    input in: UInt<1>
    output out: UInt<1>
    out <= in
  module FooBar:              ;Insert new FooBar module
    input in: UInt<1>         ;Insert Top reference as port
    output out: UInt<1>       ;Insert Top reference as port
    inst foo of Leaf           ;Insert foo instance
    inst bar of Leaf           ;Insert bar instance
    foo.in <= in              ;Insert connection to foo
    bar.in <= foo.out         ;Insert connection between foo and bar
    out <= bar.out            ;Insert connection from bar
  module Top:
    input in: UInt<1>
    output out: UInt<1>
    inst foobar of LeafFooBar     ;Instantiate FooBar instead
    inst bar of Leaf          ;Remove bar instance
    foobar.in <= in           ;Rename foo to foobar
    bar.in <= foo.out         ;Remove connections betweem foo and bar
    out <= foobar.out         ;Rename bar to foobar
```

Renaming is simply the process of changing a module's name. This also requires updating all instances of the renamed module to refer to the new module name. In the following example, ~Top|Leaf is renamed to ~Top|PassThrough.

Example 4.18:

```
circuit Top:
  module LeafPassThrough:       ;Rename module name
    input in: UInt<1>
    output out: UInt<1>
    out <= in
  module Top:
    input in: UInt<1>
    output out: UInt<1>
    inst foo of LeafPassThrough       ;Rename foo's module name
    inst bar of LeafPassThrough       ;Rename bar's module name
    foo.in <= in
    bar.in <= foo.out
    out <= bar.out
```

Retopping redefines the top-level module of the design; any module that is no longer instantiated is removed. In this example, the top of the design changed from ~Top to ~Leaf; because ~Top is no longer instantiated, it is also removed:

Example 4.19:

```
circuit TopLeaf:                ;Rename circuit
  module Leaf:
    input in: UInt<1>
    output out: UInt<1>
    out <= in
  module Top:                   ;Remove unused module Top
    input in: UInt<1>
    output out: UInt<1>
    inst foo of Leaf
    inst bar of Leaf
    foo.in <= in
    bar.in <= foo.out
    out <= bar.out
```

Duplication is the process of duplicating a module such that multiple instances of the same module are instead instantiating different modules, which are duplicates of each other. This transformation has no behavioral change to the circuit and increases the size of the FIRRTL AST, but is required for downstream transformations which intend to transform these instances in different ways. The following example duplicates `~Top|Leaf` and changes its instances to point to different, duplicate versions:

Example 4.20:

```
circuit Top:
  module Leaf:
    input in: UInt<1>
    output out: UInt<1>
    out <= in
  module Leaf2:             ;Duplicate Leaf as Leaf2
    input in: UInt<1>
    output out: UInt<1>
    out <= in
  module Top:
    input in: UInt<1>
    output out: UInt<1>
    inst foo of Leaf
    inst bar of Leaf2       ;Rename bar's module to Leaf2
    foo.in <= in
    bar.in <= foo.out
    out <= bar.out
```

Deduplication is the process of identifying duplicate modules, removing the duplicates and creating multiple instances of the same module. It is the inverse operation of duplication. While deduplication in theory is a computationally complex problem, this transformation implementation requires the modules to contain identical FIRRTL components. The only difference between modules which does not prevent deduplication is component name differences. The following example, given the previous result of duplication, removes the duplicate module `~Top|Leaf2`.

Example 4.21:

```
circuit Top:
  module Leaf:
    input in: UInt<1>
    output out: UInt<1>
    out <= in
  module Leaf2:             ;Remove Leaf2
    input in: UInt<1>
    output out: UInt<1>
    out <= in
  module Top:
    input in: UInt<1>
    output out: UInt<1>
    inst foo of Leaf
    inst bar of Leaf2       ;Rename bar's module to Leaf
    foo.in <= in
    bar.in <= foo.out
    out <= bar.out
```

Instrumentation Transformations

The FIRRTL compiler framework's modular structure makes it straightforward to add simple instrumentation passes. These can include inserting hardware counters, hardware assertions, or even improving simulation line-coverage detection.

This FIRRTL line-coverage transform instruments the circuit to print coverage information as throughout its simulation execution. This instrumentation was necessary for Chisel because some of its constructs cannot map directly to Verilog, and so must first be simplified. This simplification destroys source-level information that Verilog line-coverage tools rely on, making them largely ineffective. This transformation works by associating high-level source-line information with low-level execution statements.

Figure 4.12 shows the percentage of modules in an instance of the RocketChip SoC binned by the percentage of those modules source lines covered. The results for three different configurations of the SoC are shown; modules with low coverage exist in all three, but larger designs have fewer low-coverage modules relative to their larger total number of modules. At the high end there is no clear trend in coverage based on configuration. In general, most modules have a high level of coverage on the given test-suite with a few modules that are very lightly tested. This enables the designer to specifically target these modules with new tests to improve verification.

Specialization Transformations

Different backend targets, especially FPGAs and ASIC process nodes, require RTL modifications to achieve good results.

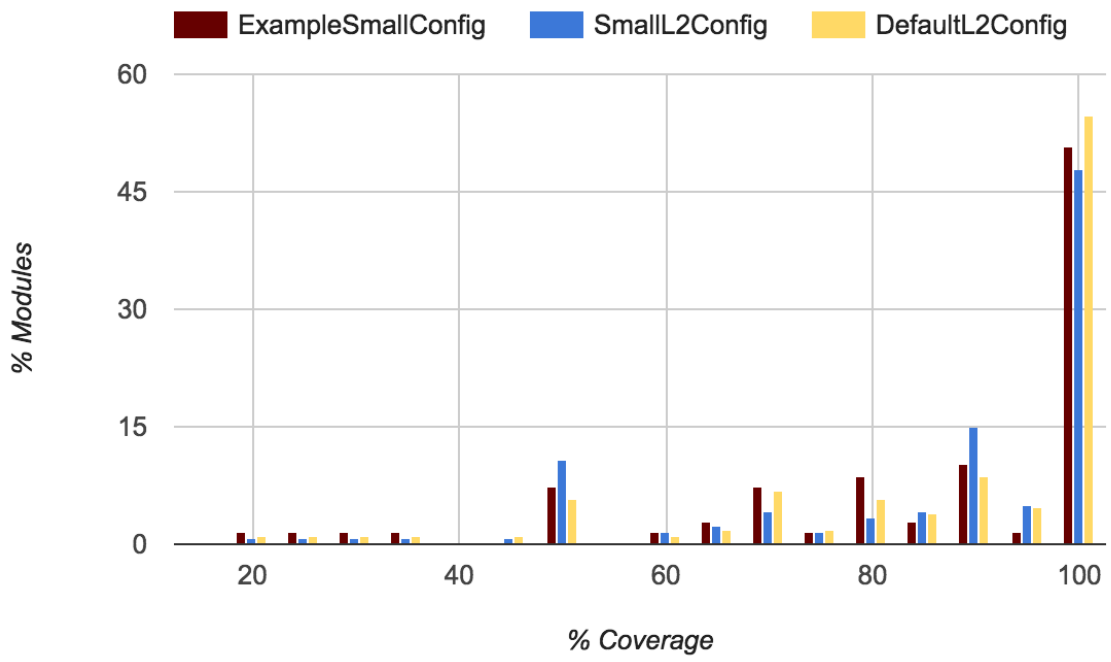

Figure 4.12: The results for three different configurations of the SoC are shown: a minimally sized configuration, ExampleSmallConfig, a moderately sized configuration with a small L2 cache, SmallL2Config, and the default sized configuration with a 256KB L2 cache, DefaultL2Config. The majority of modules have high coverage, but there remain a few which need targeted testing.

When simulating on an FPGA, there is little default visibility into a design. Commercial tools like Chipscope[49] enable real-time analysis, but require a long iteration cycle to select specific signals to target. In addition, Chipscope does not provide visibility into the BRAM memories on the FPGA, so cannot provide a full "snapshot" of the design at a given cycle.

A specialized FPGA transformation enables pausing a design on the FPGA, and another transformation enables reading out a state snapshot of the target design on the FPGA. These transformations involves threading an enable signal to all registers, inserting buffers to record input and output traces, inserting address-generation hardware to read out memory state, and attaching a custom daisy chain to scan out register and BRAM state from the FPGA.

In addition, other FPGA specializations aim to provide the most effective use of resources when mapping a design to an FPGA. In particular, BRAMs are a valuable resource that can easily be wasted through replication to accommodate high port counts; instead, another FPGA specialization transformation automatically replaces memories with high port counts with double-pumped memories with half as many ports by providing clock doubling and glue logic. Although this may reduce the maximum clock rate, the high speed potential of BRAM macros means that many microarchitectures will suffer much less than the worse case halving of throughput. In exchange for this trade off, **the pass attains a 3x reduction**

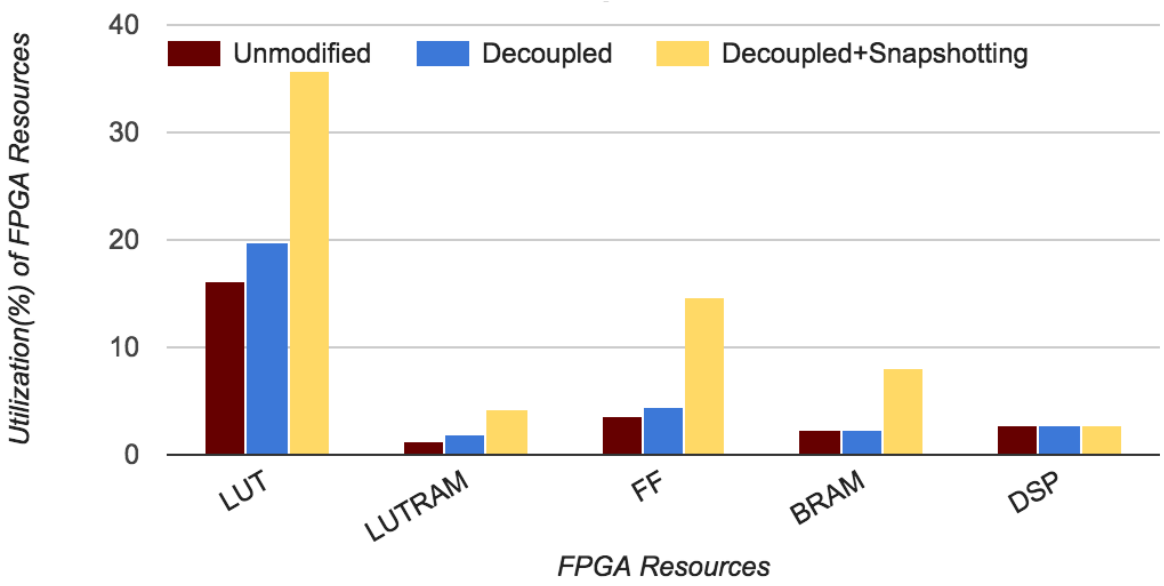

Figure 4.13: The decoupling and snapshotting transforms can add significant demands on FPGA resources, but do provide visibility into the design that was previously unobtainable. The baseline design can run at 50MHz, but the other two transformed designs run at 40MHz.

in BRAM utilization for a streaming vector arithmetic block consuming three operands and producing one result per cycle, as shown in Figure 4.14. This comes at a small **1.43% increase in logic slice utilization.**

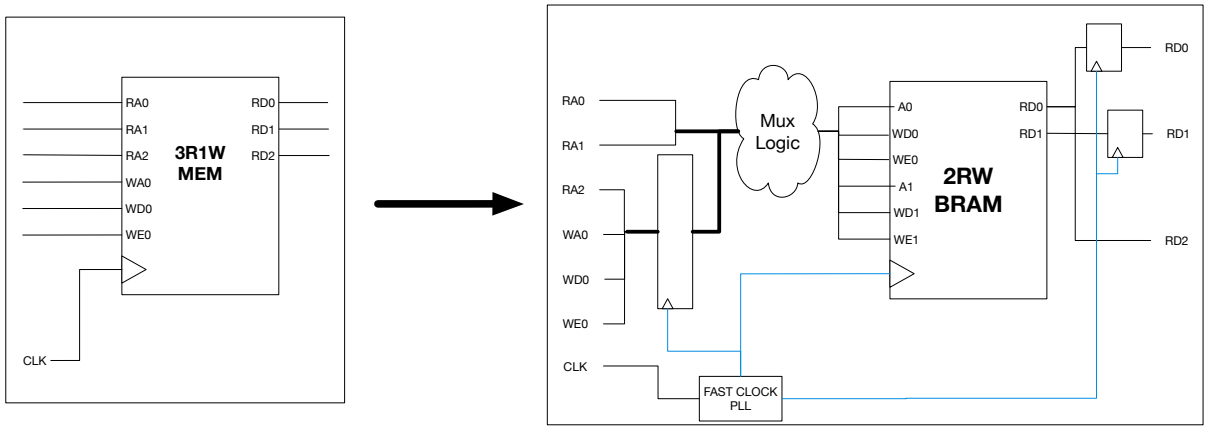

Figure 4.14: Automatic double-pumping saves FPGA resources by emulating expensive, highly-ported memories. This approach maintains abstractions that facilitate reuse.

ASIC designs benefit from the use of highly optimized hard-IP macros that are targeted towards specific functions. For example, large memory-based designs are typically mapped to vendor-provided SRAMs rather than registers, to improve QoR (in terms of area, power, and timing closure) and tool runtime. As seen in Figure 4.15, after synthesis a 2048-point memory-based FFT (20-bit real and imaginary) implemented with SRAMs (4 banks of 512-depth memories) is 6x smaller than the same FFT implemented with registers. The savings will increase after place and route due to excess routing penalties incurred in the register-based design. Additionally, synthesizing the SRAM-based design takes considerably less time than the register-based design, because the tools need to handle significantly fewer hardware instances.

However, specializing RTL to make use of these macros on a per-technology basis is non-trivial. Vendor-provided SRAMs often require additional pins that must be properly connected for initialization and verification but do not contribute to the functionality at a high-level. To address this issue, a memory-replacement transformation replaces a generic FIRRTL memory with a custom black-box that matches the ports of the vendor-provided SRAM. Without running the transformation, FIRRTL translates the generic FIRRTL memory into a large register array. When the transformation is run, design-specific memory signals (data, address, and enable) are connected to the ports of a vendor-provided SRAM instance, and any additional initialization and verification signals to and from the SRAM are automatically connected, across module boundaries, to top-level ports. This greatly reduces the design effort required to map generic RTL to optimized hardware.

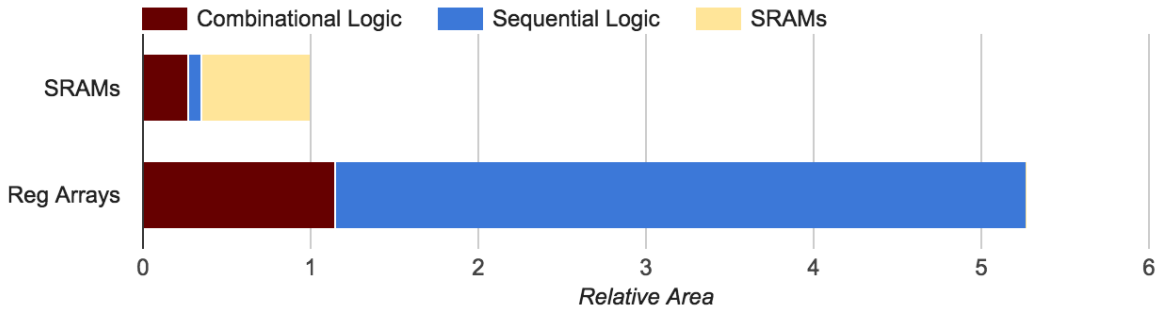

Figure 4.15: A hardware FFT Chisel design generated with and without the memory-replacement transformation was synthesized in a 16nm process. Using the transformation improved utilized area by 6x. Both transformed designs met timing at 800ps (1.25GHz). Synthesis took ∼6 minutes with SRAMs, compared to 1 hour 44 minutes without.

Custom Transform Case Study

To illustrate a workflow using Chisel and the FIRRTL hardware compiler framework, a custom parameterization of RocketChip was synthesized and place-and-routed on a 28nm process to DRC/LVS-clean GDS.

The design consists of two cores with a large data-parallel cache-coherent accelerator. The L2 cache is heavily banked, which requires multiple SRAMs. In addition, there are multiple clock and voltage domains, as well as multiple high speed off-chip IOs.

Because of the parameterization and reuse employed by the RocketChip hardware library, it is very easy to specify the desired design - only 1817 new lines of code were added, which consisted of specialized configuration parameters, top-level glue logic, and an associated test harness. Many modules had already been verified and evaluated in previous projects, and thus needed less verification and design effort. Almost all verification effort was spent on the new code as a result and this reusability was key to reducing design overhead.

Targeting the 28nm process reused the memory-transformation and the deduplication transformation described in Section 4.6. However, this process presented two new problems: (1) the synthesis tools required specifying which modules were in which clock and voltage domains; (2) the SRAMs had additional initialization and control pins that were unique to this process.

Due to the modularity of FIRRTL's compiler implementation, two custom FIRRTL transformations were written to solve these problems; additionally, they were added as part of an open-source FIRRTL transformation ecosystem and have since been used without modification for other designs, backends, and projects. In total, these customizations required only 680 new lines of code, demonstrating the principle of reusability and growing a hardware design ecosystem.

4.7 Summary

This chapter contains a discussion of FIRRTL's hardware compiler framework through an introduction of FIRRTL's intermediate representation (IR), a description of its value inferencing (width, bound, precision), its support for robust and arbitrary metadata, circuit traversal strategies, and an overview of many existing FIRRTL transformations. It has been used in a variety of ASIC tapeouts and FPGA emulation projects, and its ability to customize the RTL has accelerated the development of these projects. The development strategy of FIRRTL was generally forward looking; important infrastructure features like the annotation system were worked on early to enable its implementation to be well-architected for future use.

Chapter 5

Colla-Gen: A Chisel Interface for Hardware Collateral Generation

As Chapter 3 and Chapter 4 demonstrate, the language of Chisel and the hardware compiler infrastructure for FIRRTL accelerate the process of designing digital logic. However, a digital logic design is far removed from a silicon chip, requiring verification and physical design steps. As described in Chapter 2, these steps consist of manually writing design-, technology- and tool-specific collateral for verification and CAD programs. This manual process limits reuse and lags behind the now-accelerated digital logic generation. FPGA emulation of a digital logic design also requires additional collateral, such as adding instrumentation for design visibility or changing a design's memory structure to better synthesize on FPGA block RAMs. Arguably, even the process of simulating a design requires a sizable amount of code collateral; break-point selection, collected statistics specification, and traced signals are all design-specific collateral which designers and verification engineers spend significant time specifying manually.

All of these examples demonstrate the need for additional design collateral when delivering a usable product, regardless of the actual simulation, emulation, or implementation technology used. Because much of this collateral is specified manually, it lags behind the generator-based flows of Chisel and FIRRTL and limits the utility of this framework. To address this problem, the Chisel and FIRRTL hardware compiler framework must provide a solution for generating this code collateral such that it is in-sync with the generated design, while also maintaining reusability across designs and platforms.

This chapter contains a detailed description of aspect-oriented programming (AOP), an analysis of its flaws, and an argument for its consideration within a hardware context. Next, *Colla-Gen*, is introduced as an AOP-inspired interface for generating hardware collateral and is illustrated with a physical design floorplanning example. Finally, Colla-Gen's implementation and additional reusable libraries for generating design collateral are presented and discussed.

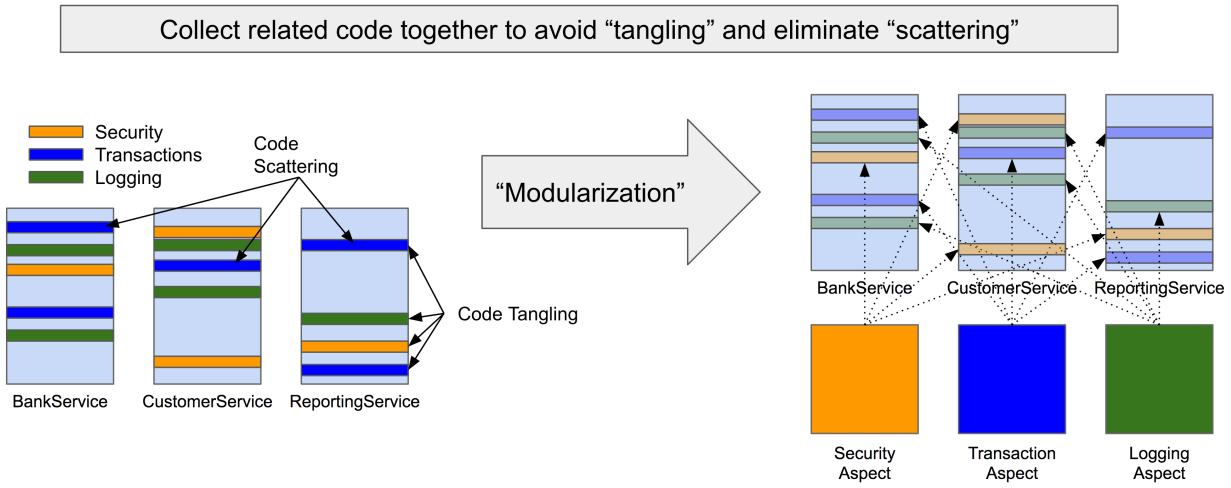

Figure 5.1: In many software programs, there are side concerns like security or logging, which permeate a program but are unrelated to the core functionality of a program. Aspect oriented programming solves this "scattering" of related code by collecting all concern-related code (modularization) in an **aspect**, and specifying where in the central program to inject itself.

5.1 Aspect-Oriented Programming

In software, a cross-cutting issue can cause significant problems when its code is scattered throughout a large project. For example, security code, transaction code, or logging code are separate concerns and tend to be scattered throughout a project. Due to this scattering, one simple conceptual change (e.g. logging on standard out, not standard err) requires updating code in many disparate program locations. Of note, these scattered code snippets are not usually related to the program's core functionality; usually, these snippets are a complementary but cross-cutting concern to the program.

Aspect oriented programming (AOP) is a programming paradigm designed to eliminate this scattering of cross-cutting issues with a unique set of language-supported programming constructs. Invented by Gregor Kiczales and colleagues at Xerox PARC, its seminal paper was published in 1997 and has over 9000 citations as of the writing of this thesis[26].

A programmer expresses their cross-cutting concern as AOP code in a separate and distinct location called an *aspect*; this code is later injected into the main program via a process called *weaving*. *Join points* are specific instances during the execution of a program where code can be injected; for example, both entering a method and leaving a method are valid join points. Each aspect can inject its executable code, called an *advice*, when the executing program arrives at a join point that matches the advice's *pointcut*, a regex-like selector of join points. For an example of aspect-oriented programming, see Figure 5.2.

While AOP made a large splash in the programming languages research community, almost no mainstream programming language has added support for an aspect-oriented

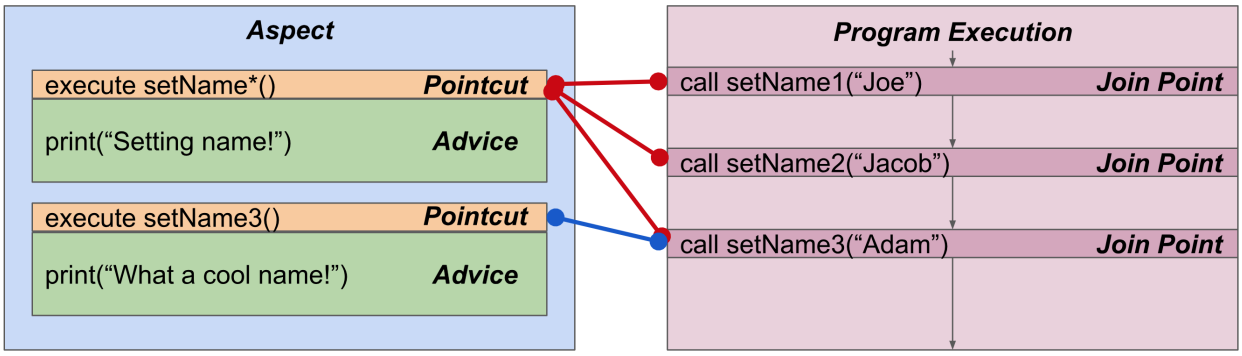

Figure 5.2: In this above example, the two aspects declare their (1) pointcut (when during the program's execution they take effect) and (2) advice (the behavior they are injecting). The first aspect triggers when any function matching the regex *setName** is called and executes its advice which prints "Setting name!". The second aspect triggers only when the function *setName3* is called, and prints "What a cool name!".

paradigm (one notable exception being the Spring library in Java). For many language developers, the downsides of aspect-oriented programming significantly outweighed any redeeming qualities of the paradigm.

Firstly, overusing an aspect-oriented paradigm will significantly obscure a program's control flow. Because aspects change program semantics, a reader must have *whole-program* knowledge to reason about the *local* dynamic execution of an aspect-oriented program. Aspects are in some ways analogous to the joke assembly instruction "come-from", which is the reverse of the much maligned (and equally harmful) "goto" instruction [16].

A second major criticism of the aspect-oriented paradigm is its lack of aspect-ordering semantics. When multiple aspects select the same join point, the default behavior is unclear regarding the order with which each aspect is weaved. AspectJ[25], an implementation of AOP, partially addresses this problem with a non-scalable ad-hoc numbering system. Without more formal treatment, this aspect-ordering problem continues to obscure an already unclear control flow.

Finally, the aspect-paradigm has a *fragile pointcut problem*, or the sensitivity of a pointcut to future modifications of the central program. A pointcut can become out-of-date by simple changes like renaming a method, and knowing which aspects are affected is difficult because pointcuts are separated from the modified code. Providing explicit flags or event annotations, a technique used by the Ptolemy language, can help mitigate this problem while still preventing scattering of cross-cutting concerns.

While AOP has many upsides and downsides, the following are lessons to takeaway. First, treat the program as data to modify; rather than viewing AOP as two running programs modifying each other (which leads to paradoxical behavior), the central program is data being transformed by an aspect. Second, use AOP for secondary concerns like debuggability, instrumentation, or specialization/optimization rather than for the main program behavior.

Third, be transparent about when weaving occurs; a human-readable representation of the program as it undergoes aspect modification can help debugging this process. Fourth, design robust mechanisms to select places to transform to avoid the fragile point cut problem that plagued other AOP implementations.

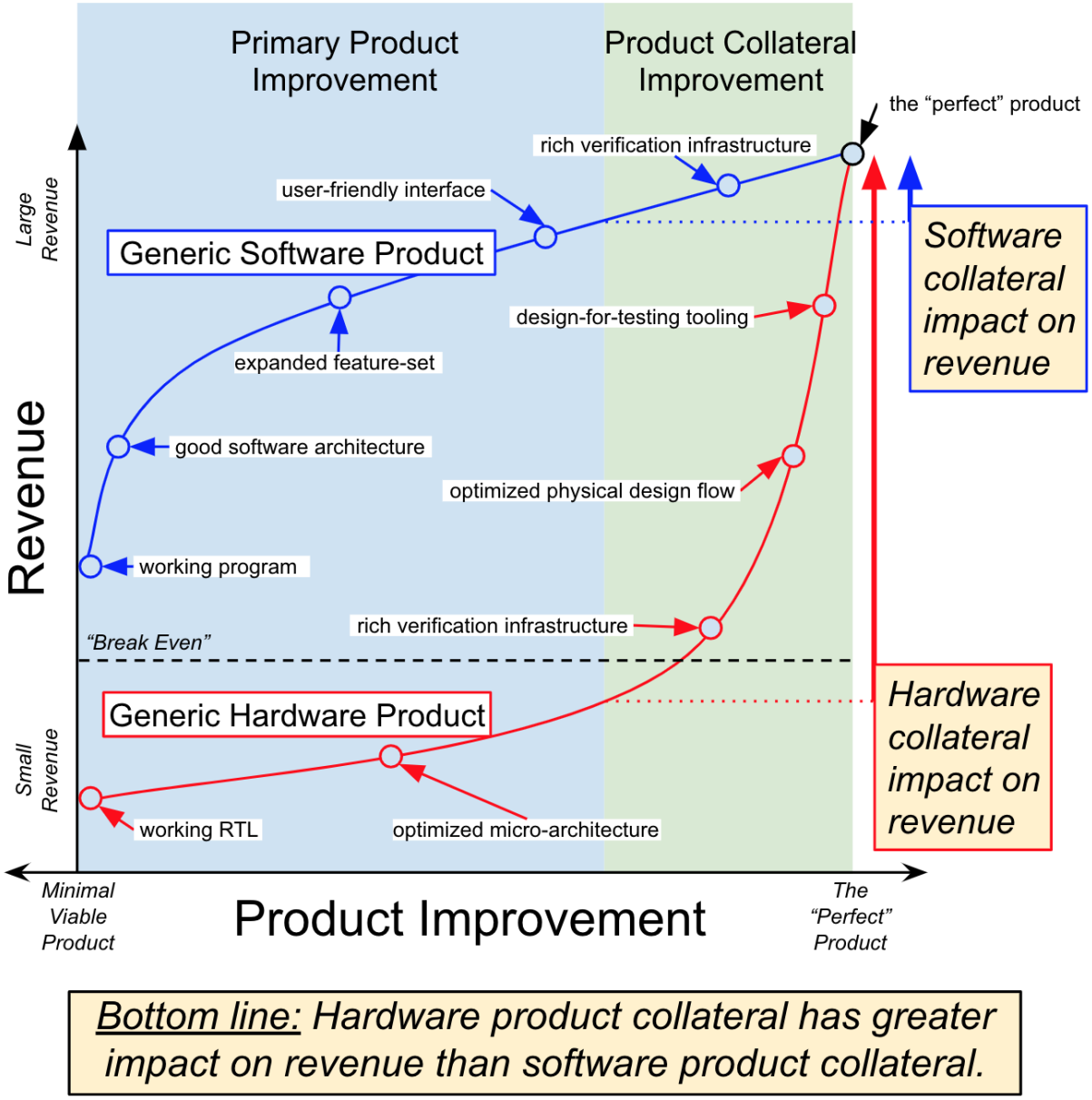

Figure 5.3: Note - this figure is illustrative in nature and does not represent any quantified data or measurements.

Given AOP has so many detriments, why is it useful inspiration for improving hardware

design methodology? There are two key differences between software languages and hardware construction languages which demonstrate why an AOP-inspired approach has more benefits for hardware design; see Figure 5.3 for an illustration of some of these differences.

First, hardware collateral is necessary and impactful. Hardware requires significant collateral for physical design, verification, and other steps in the hardware design process. Before any return on investment, upfront development of hardware collateral is often fruitful and necessary because hardware design flaws cannot be fixed after fabrication. In direct contrast, software collateral is often an afterthought because its secondary concerns can be addressed in the future and a project's primary application is by far the main determining factor in a project's success.

Secondly, AOP's paradigm of separating program and aspect fits naturally into hardware construction languages. Hardware construction languages already have a meta-programming construction paradigm that can be leveraged for an AOP-like framework; by not creating new constructions with new semantics, there is no additional conceptual barriers limiting adoption that have negatively affected software AOP implementations.

5.2 The Colla-Gen Interface

Adding a standard collateral generation mechanism to hardware construction languages addresses a necessary code collateral problem by leveraging an existing language paradigm. For these reasons, an AOP-inspired interface has been added to Chisel, *Colla-Gen*, to enable building aspect libraries for generating hardware code collateral.

This section first provides a background section that introduces floorplanning as a running example of hardware collateral. Then, the typical interfaces between a user and a Colla-Gen library, and between that library and the Chisel/FIRRTL framework, are described.

Background

This section first introduces RISC-V Mini, a simple RISC-V core, as an example Chisel design with which hardware collateral can be generated for. Next, this section describes a set of ASIC floorplanning elements which are later leveraged for a Colla-Gen floorplan library for generating floorplans for Chisel modules.

RISC-V Mini is an existing Chisel core generator that will be used in later examples. Its simple 3-stage pipeline implements RV32I of the User-level ISA Version 2.0 and the Machine-level ISA of the Privileged Architecture Version 1.7. Mini consists of a tile containing a core, an instruction cache, and a data cache. Many of its modules are parameterized, where providing different top-level parameters will generate different hardware instances.

Floorplanning is usually a manual process to specify the placement (via absolute coordinates) of large modules and macros. They are necessary to seed place-and-route tools, reducing tool run times from years to days. Generally, physical designers rely on their experience and trial-and-error to create effective floorplans, making early design space exploration

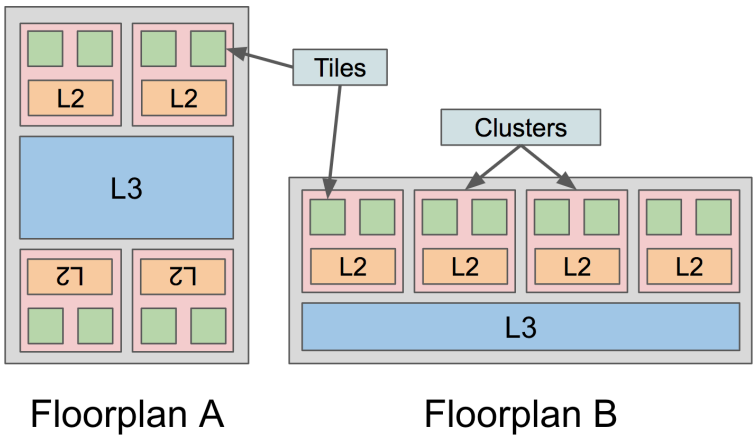

Figure 5.4: For this design of four clusters, each containing two tiles, two floorplans are depicted. Knowing which floorplan is better depends on the logical connections of the design and the physical characteristics of the transistor technology - the ability to quickly generate multiple floorplans can have significant effects on the resulting area, frequency, and chip power consumption.

an important step. Because floorplans are so design-, technology-, and tool-dependent, they are rarely reused from design to design, between vendors, or between technologies. There is little room for RTL design space exploration due to the manual effort involved; this makes Chisel generators less useful.

The floorplan elements are a hierarchical layout framework in Scala and consists of three major portions - a geometry API, references to FIRRTL modules/signals for interfacing with the RTL, and a numeric solver to concretize the design with absolute coordinates and dimensions to enable a tapeout. After composing these elements into a floorplan, they can be translated into Hammer IR and reused across technologies and tools (see Chapter 2). In addition, these floorplans can also be depicted with a JavaScript hierarchical visualization.

A variety of geometry constructs are provided to enable the construction of a floorplan. These constructs can be nested within one another and are independent of any numerical co-ordinates to enable re-use of floorplans across different technologies and chip projects.

These constructs include:

- `HBox` - tile its children elements horizontally.
- `VBox` - tile its children elements vertically.
- `AutoLayout` - do not specify any particular constraints to the backend tools for the module in this box.
- `Expander` - create a space/separate the given modules as much as possible.
- `HardMacro` - represents a hard macro, which has fixed dimension but can vary in position.

Each geometry element can be attached to a module or signal at the RTL level via a FIRRTL target (see Chapter 4), a reference for exactly one hardware component. Not every geometry element is required to have a FIRRTL target. For example, a `VBox` could be attached to a module and have two sub-elements to floorplan parts of that module and an `Expander` in between. In this case, the `Expander` would not need a RTL module attached to it. Geometry elements are attached to a targets via the `replaceTarget()` function.

Notably, the RTL hierarchy does not need to correspond to the physical hierarchy. For example, in the RTL, one module could exist as logical child of another, but in the floorplan hierarchy the two modules could be siblings.

Finally, a floorplan's elements must to be resolved into concrete positions and dimensions. Since the exact co-ordinates for a particular module or layout will differ wildly between different process nodes, numbers can be specified separately in a second stage as opposed to being required when the layout is created - while the hierarchy structure can be reused, the actual widths and sizes do not have to be.

Geometric elements can have dimensions entered to them via functions like `replaceHeight`, `replaceWidth`. All the co-ordinates can then be resolved in the system (e.g. for visualization or export to backend tools) via the `resolve()` function.

Using a Colla-Gen Library

While there are no limitations on the API a hardware collateral generator library can expose to a user, this interface typically follows the standard practice described below. Then, a concrete floorplanning example of this user-interface is described. Finally, the benefits of this interface are emphasized including its support for in-step hardware collateral generation, its between-generator reuse model, and its support for robust signal references.

A typical user-interface for a Colla-Gen library is a user-provided custom function that, given the top-level elaborated Chisel instance, returns their generated hardware collateral. For example, using the floorplanning Colla-Gen library on RISC-V Mini requires the user to provide a function that, given the top-level module of Mini, returns the nested geometry floorplan elements as described in the previous section.

After creating an instance of a Colla-Gen library for a project, it is passed in as an argument to the Chisel elaboration step (or a tester's elaboration step). Because the Colla-Gen class extends Annotation, it relies on the same metadata mechanism as all other annotations.

Hardware collateral usually requires references to their associated design (e.g. a floorplan requires associating a geometric element to a module or signal in the Chisel design). Like the 'fragile pointcut problem' of aspect-oriented programming, without proper treatment these design references can easily become outdated due to the separation of the design definition from the collateral definition.

Fortunately, hardware construction languages have a natural separation between the constructed design and the programmatic assembly of the design; this gives an intermediate and statically-typed representation of the post-elaboration constructed design which can be inspected. Note that this is different from a FIRRTL representation, which is an AST; this

elaborated statically-typed representation is the object directly constructed when executing a Chisel generator. While this object is later translated into a FIRRTL AST that is given to the compiler, prior to this it can be inspected to derive robust references to a design.

Inspecting this elaborated object has other benefits as well - the parameter values used to elaborate a Chisel module can also be inspected and used to also parameterize the collateral generation. The signals in this object are also type-safe; because Chisel is hosted in Scala (a statically-typed language), a signal reference is a statically-known member of a class. Any change to the name of a signal will trigger compile-time errors in any user-defined collateral generation function that references that signal. In essence, this solution converts the 'fragile pointcut problem' into a problem already solved by programming languages: robust references to class members.

While referencing signals by name is robust in this system, it may be inadequate for selecting all signals/modules of a certain kind. For example, a user may desire to generate collateral for all registers in a design, and reference all registers by name is fragile to a later user adding another register. To account for these more general selection use-cases, Colla-Gen provides a selector object *Select* which contains functions that select a module's signals by kind (logical operation, register/wire, ports, module), as well as iterate through all module instances in the design.

Because the user-facing interface of Colla-Gen libraries is to require a function that accepts the top-level module instance, this function cannot be directly reused by a different project that shares internal Chisel modules. Instead, the typical reuse model between projects is to implement the top-level hardware collateral generation by composing the results of other per-module collateral generator functions called on children modules. For example, a reusable cache-to-floorplan collateral generator function may accept a RISC-V Mini Cache and return the floorplan for that cache (see Figure 5.6. The top-level RISC-V Mini floorplan generator function will then call this cache-to-floorplan function and integrate the cache's floorplan into its top-level floorplan. This reuse module enables the cache-to-floorplan function to be reused with any project that uses the RISC-V Cache and desires a corresponding floorplan.

In summary, the typical user-facing interface for a Colla-Gen library is a custom function that maps the top-level Chisel instances to its hardware collateral. With access to the elaborated Chisel design, a user can inspect instance parameter values and parameterize their hardware collateral generation similarly. With robust by-name selectors and general selection mechanisms, a user can robustly select and associate their desired hardware logical components to their generated hardware collateral. Finally, a user can express their hardware collateral generation on a per-module basis, enabling reuse between projects sharing Chisel modules.

A concrete example of this user-facing interface is depicted in Figure 5.5. This floorplan Colla-Gen library's hardware collateral are nested geometric elements associated with Chisel modules/signals. However, the returned hardware collateral will depend on the Chisel RISC-V top-level instance and can generate different floorplans accordingly. For instance, RISC-V Mini's cache module is highly parameterized; by giving a different `nWords` parameter,

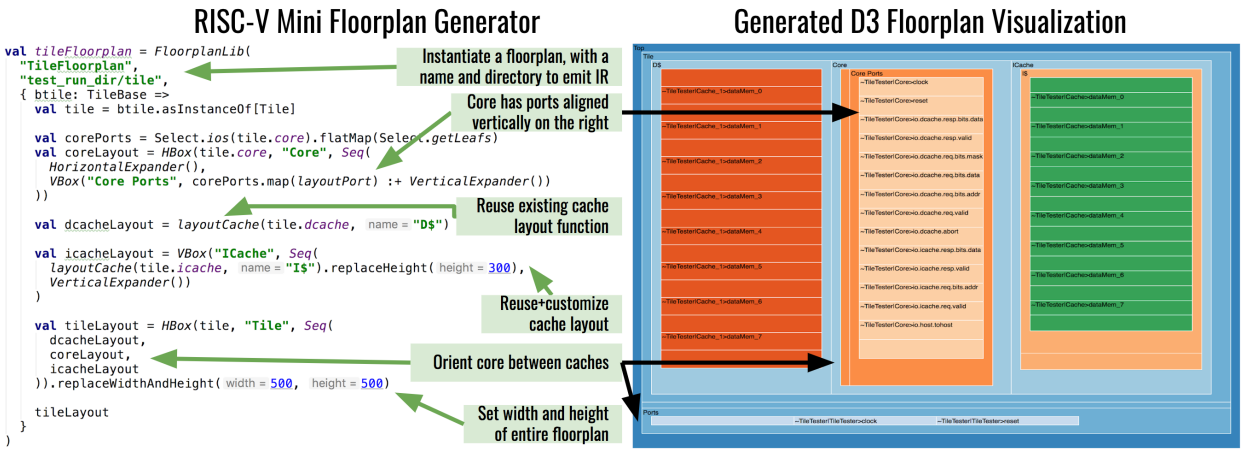

Figure 5.5: A RISC-V Mini floorplan generator and generates both Hammer IR and a d3 visualization of the floorplan hierarchy. Note that the boxes in this visualization are not size-accurate. The Select object is used to select all ports of the Core module.

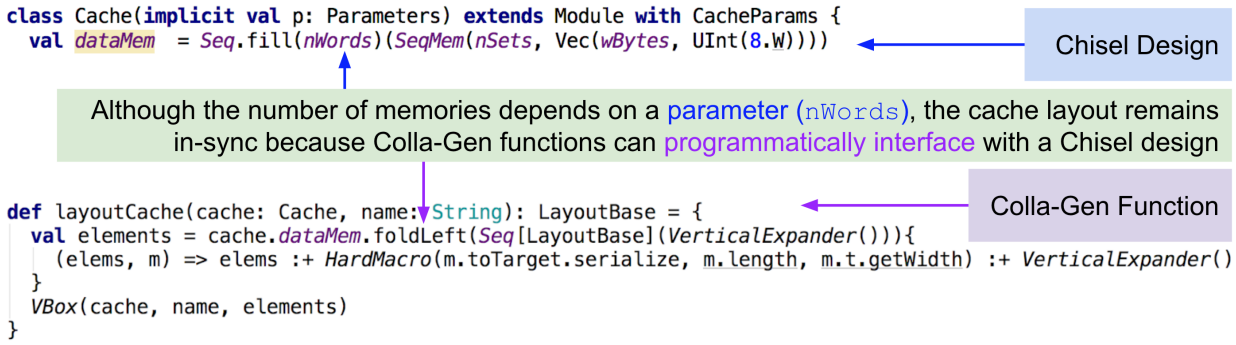

Figure 5.6: Our `layoutCache` function generates different floorplan objects (`VBox`, which subclasses `LayoutBase`) because it can programmatically iterate through the elaborated Chisel design's components, signals and modules. As a consequence, the generated floorplan is always in-sync with the generated Chisel instance.

the number of instantiated memories changes. This change affects the number of SRAMs eventually needed for physical design; the floorplan must adapt accordingly. The user-provided function calls `layoutCache`, the function in Figure 5.6 which iterates through all memories in `dataMem` (the memories affected by the `nWords` parameter) regardless of their number. Direct access to the elaborated Chisel design enables this synchronization between the floorplan and the Chisel instance.

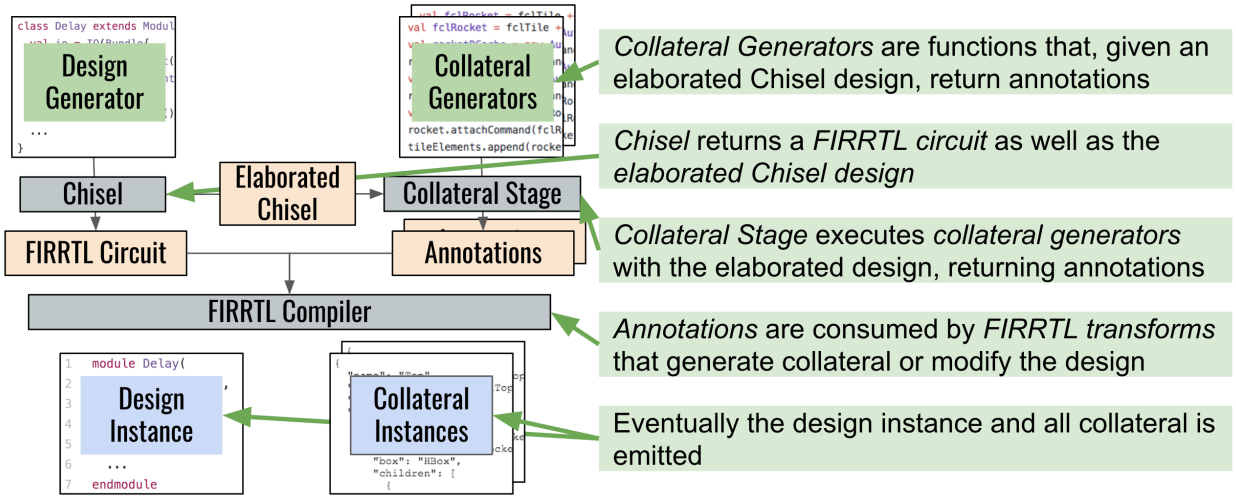

Figure 5.7: Colla-Gen generators are resolved prior to FIRRTL compilation but after Chisel elaborates the design. The *Collateral Stage* resolves each Colla-Gen generator by calling the Colla-Gen's `toAnnotations` function with the elaborated Chisel design. All returned annotations are consumed during FIRRTL compilation. A Colla-Gen library-specific FIRRTL transform can then write the collateral instances to a file, where all changes to a signal's name are tracked via FIRRTL's renaming capabilities.

Implementing a Colla-Gen Library

When designing a Colla-Gen library, a library writer must ensure their library integrates properly with the Chisel/FIRRTL framework. This section first describes this interface. Then, the ordering semantics of libraries and their propagation semantics under compiler transformations is discussed. Finally an example library implementation is provided.

As explained in the previous section, most libraries accept a user-defined function that, given the top-level Chisel instance, returns the generated hardware collateral. The interface between a Colla-Gen library and greater ecosystem, `toAnnotations`, is similar: given the user-defined function and the top-level design, return a list of FIRRTL annotations that will be passed downstream to the FIRRTL compiler. These annotations can consist of existing annotation types, or custom annotations which can trigger or be consumed by a custom transform. As shown in Figure 5.7, each Colla-Gen library is resolved after Chisel compilation but before FIRRTL compilation. This enables libraries to use the elaborated Chisel design while still providing annotations to affect the FIRRTL compilation.

When multiple libraries are used simultaneously by a project, Colla-Gen ordering semantics dictate the order of observable effects of these libraries. Most Colla-Gen libraries focus on generating collateral and thus have no design side-effect changes. However, other libraries may modify the design indirectly by triggering a transform. In this case, the order in which these transforms are applied are independent of the order of the libraries; instead, they are

applied according to the FIRRTL compiler infrastructure's dependency API. This is similar to Chisel's annotation mechanism; the order with which a signal is annotated is irrelevant to the order with which they are processed by a transformation.

Often, a Colla-Gen library will want their generated collateral to update itself under arbitrary compiler transformations; for example, a signal removed from dead code elimination may also require deleting its corresponding collateral. To customize this behavior, a library should track the changes to their referenced components with an annotation and the renaming API (see Chapter 4). Then, these annotations should be processed in a final custom transformation which can update the collateral accordingly and serialize it to disk.

To illustrate, consider the floorplanning example in Example 5.1. The floorplanning library takes the top-level design and calls the user-provided function with it, returning the geometric elements corresponding to the floorplan. Then, the library analyzes the geometric elements and generates custom floorplan annotations on each referenced module/signal to track its changes during compilation. Finally, these annotations are consumed downstream by a custom floorplan transformation which updates the floorplan and serializes it to either Hammer IR (see Chapter 2) or a javascript D3 visualization. See the three code blocks of Example 5.1 and Example 5.2 which demonstrates this library's implementation.

Overall, Colla-Gen provides a powerful mechanism for hardware collateral generation libraries to remain consistent with a design undergoing compiler transformations, trigger arbitrary compiler transformations, and define their ordering semantics for design modifications.

5.3 Colla-Gen Library Examples

Now that the motivation of generating hardware collateral, as well as the Colla-Gen library interfaces have been described, this section details a wide variety of different hardware design collateral libraries which leverage the Colla-Gen interface. In addition to the previous section's physical design theme of floorplanning, these libraries address collateral problems of a general-purpose, instrumentation, verification, or resiliency nature. In all, the diversity and power of these libraries demonstrate the usefulness and wide-reaching power of an AOP-inspired methodology for generating hardware design collateral.

General-Purpose Collateral

Many Colla-Gen libraries require similar infrastructure in addition to the interfaces defined in the previous sections. This subsection describes in detail a Colla-Gen library which provides a mechanism to inject custom Chisel code into an already-elaborated module. While most collateral libraries can rely on downstream tools to consume the collateral, in some cases a library writer may implement this feature directly in the FIRRTL compiler. For example, a verification library writer may create a library to generate coverage bins to be consumed by a downstream verification tool; alternatively, this library could also synthesize these coverage

Example 5.1: Floorplan Colla-Gen Library

```
/** A Colla-Gen Library
  * Use to generate a floorplan for a given Chisel design
  * Uses LayoutBase geometric primitives to represent floorplan
  *
  * @param name Name of the file to serialize floorplan to
  * @param dir  Directory to write file to
  * @param buildFloorplan User-defined function to build floorplan from
    top-level Chisel module
  * @param tTag Necessary detail to properly record type of top-level Chisel
    module, ignore if a user
  * @tparam T Type of top-level module
  */
case class FloorplanLib[T <: RawModule](
    name: String,
    dir: String,
    buildFloorplan: T => LayoutBase
)(implicit tTag: TypeTag[T]) extends CollaGen[T] {

  /** Search given layout to collect all signal/module references
    * Return a tracking annotation for each reference to record
    *   FIRRTL Compiler changes to these signals
    */
  def collectTrackers(layout: LayoutBase): Seq[MemberTracker] = ...

  /** Given the top-level elaborated Chisel module, generate FIRRTL
    Annotations */
  override def toAnnotation(top: T): AnnotationSeq = {
    val layout = buildFloorplan(top)
    val trackers = collectTrackers(layout)
    Seq(FloorplanInfo(layout, dir, name),
        RunFirrtlTransformAnnotation(new FloorplanTransform())
    ) ++ trackers
  }
}
```

bins and directly instrument the design. For similar cases where the process consuming the hardware collateral is implemented in FIRRTL, this generic hardware injection library proves very useful.

It is important to emphasize that this process of injecting hardware should be used with extreme caution. It should not be used to implement primary RTL behavior, as this would obscure the hardware generation process and result in the same problems as aspect-oriented programming. Instead of exposed to users, this library should only be used by other Colla-Gen libraries to implement these secondary, collateral-consuming processes.

Because this library is generally useful and requires additional calls to Chisel elaboration, its implementation required minor modifications to Chisel. However, it primarily uses the Colla-Gen interfaces and other Colla-Gen libraries which require similar features can use *InjectingLib* instead of modifying Chisel directly.

To use the injecting Colla-Gen library, simply provide a selector function that collects instances of a given Chisel module from the top-level elaborated design. Secondly, provide

Example 5.2: Floorplan Colla-Gen Library

```
/** Contains original floorplan generated from FloorplanLib
  * Also contains additional info required for serialization
  */
case class FloorplanInfo(
    layout: LayoutBase,
    dir: String = "test_run_dir/html",
    name: String = "layout"
) extends NoTargetAnnotation with Unserializable

/** Records changes to targeted nodes as FIRRTL compiles
  * Given the set of rename mappings optionally select the one to be the
    final renaming
  */
case class MemberTracker(
    name: String,
    targets: Seq[IsMember],
    finalSelection: Seq[IsMember] => Option[IsMember]
) extends Annotation with Unserializable {...}

/** Consumes [FloorplanInfo] and [MemberTracker] to update floorplan
  * Then, serializes floorplan to disk
  */
class FloorplanTransform extends Transform {...}
```

a function that, given a module to inject code into, creates Chisel components which will be injected into that module. This Chisel code can reference signals within the module directly, and this library will correctly resolve these references. The following example, if included when compiling RISC-V Mini, would inject a Chisel print statement into Mini's ALU:

Example 5.3: Injecting print into ALU

```
val inlinedLogging = InjectingLib(
  { top: Tile => Seq(top.core.dpath.alu) },
  { alu: ALU =>
    when(alu.io.alu_op === 15.U) {
      printf("A == %d, B == %d, opcode == %d\n", alu.io.A, alu.io.B,
    alu.io.alu_op)
    }
  }
)
```

This Colla-Gen library is implemented by using custom annotations to pass the injected hardware from the resolution of the Colla-Gen library to a custom FIRRTL transform which does the hardware injection. The two code blocks in Example 5.7 and Example 5.8, located at the end of this chapter, contain the source code for the Colla-Gen library *InjectingLib*, the custom annotation *InjectStmt*, and the custom transform *InjectingTransform*.

Verification Libraries

To help understand or debug existing hardware, a user may want additional visibility into the per-cycle execution behavior of the reused hardware; this is most often done with explicit addition of debugging RTL and loggers, or relying on the debugging features of the user's simulator. However, since one goal of reusing hardware is to reuse without modification, a user may become frustrated with the lack of per-cycle visibility and instead copy the source-code to enable custom simulation-specific instrumentation modifications.

To address this problem, we created a histogramming library to enable users to collect and bin per-cycle values of specified signals, and then when the test finishes, display the histogram values. With the ability to mix-in on a per-test basis, the user's visibility concern is separated from the actual hardware source code.

Below is a description of this histogramming library. First, a user can instantiate the library and mix it in with their tests as follows, as all subclasses of Colla-Gen also subclass Annotation:

Example 5.4: Using HistogramLib

```
object MiniHistograms {
  def selectALU(t: TileTester): Seq[ALU] = ...

  // Create a HistogramLib that instruments Mini's ALU
  val aluHistogram = HistogramLib[TileTester, ALU](
    selectALU, // Selects module to add histogram
    { alu: ALU => Seq(new HistogramSignal(alu.io.alu_op)) }, // Select signal
    to histogram, and wrap in HistogramSignal which enables bin customization
    { tester: TileTester => tester.isDone }, // Indicates when the test is
    finished
    { tester: TileTester => tester.setDone } // Indicates when the simulation
    is killed
  )
}

// To create custom tests with the histogrammer, pass the aluHistogram
//   to the tester as an annotation
class TileSimpleTestsWithHistogrammer extends TileTests(
  SimpleTests,
  annotations = Seq(MiniHistograms.aluHistogram)
)
```

When the `TileSimpleTestsWithHistogrammer` is executed, Example 5.5 contains what is printed at the end of the simulation.

Example 5.5: Test output with ALU's Opcode Histogram

```
Enabling waves...
Starting simulation!
Histogram for signal alu_op in ALU:
Bin  0 ( 0 until  1) -> 100
Bin  1 ( 1 until  2) ->   0
Bin  2 ( 2 until  3) ->   0
Bin  3 ( 3 until  4) ->   0
```

```
Bin  4 ( 4 until  5) ->   0
Bin  5 ( 5 until  6) ->   0
Bin  6 ( 6 until  7) ->   0
Bin  7 ( 7 until  8) ->   0
Bin  8 ( 8 until  9) ->   0
Bin  9 ( 9 until 10) ->   0
Bin 10 (10 until 11) ->   7
Bin 11 (11 until 12) ->   0
Bin 12 (12 until 13) ->   0
Bin 13 (13 until 14) ->   0
Bin 14 (14 until 15) ->   0
Bin 15 (15 until 16) ->  14
Process finished with exit code 0
```

This prototype can be used as the basis for other verification libraries such as a coverage library.

Resiliency Libraries

Some applications like computation upon a satellite or airplane controllers require ASIC chips to be more robust to random bit toggling. These applications drive a need for augmenting an existing design to improve its reliability.

Triple modular redundancy (TMR) is a robustness strategy where key elements are duplicated three times; if one element produces the wrong answer, it is overruled by the other two elements producing the correct answer. Because this strategy requires three times the amount of digital logic, it has a large area and power cost. As a trade-off between cost and reliability, TMR can be applied to the most vulnerable sub-components of a design, rather than the entire design. Exploring this design space of TMR versus cost can require large engineering costs if TMR is manually applied to the design; an attractive alternative is to automatically apply TMR to selected components.

A TMR Colla-Gen library *RedundancyLib* was designed to support automatically applying TMR on selected registers. To test this library, another Colla-Gen library *StuckFaultLib* was designed to automatically inject faults into a design. Through use of these two libraries, a designer can iterate applying TMR and evaluating its reduction in resilience, search for an optimal solution given the associated costs. The code in Example 5.6 demonstrates how Colla-Gen functions are reused to select the registers to apply TMR or stuck-faults, and how they are then passed to tests to confirm their correctness/incorrectness.

While these libraries only work with registers, future work is to support TMR on modules as well. In addition, these libraries provide a starting place for implementation of other resiliency strategies with similar hardware collateral requirements.

Example 5.6: TMR Example

```
object MiniTMR {
  def selectDpath(tester: TileTester): Datapath = ...

  // Selects all registers in the datapath module to apply TMR
  val redundantDPathRegs = RedundancyLib(
    { dut: TileTester => Select.getDeep(selectDpath(dut)) { Select.registers
    } }
  )

  // Selects one register in datapath to apply stuck fault to
  val faultyInst = StuckFaultLib(
    { dut: TileTester => Seq(selectDpath(dut).ew_inst) }
  )
}

// This will fail because it injects a fault
class TileSimpleTestsWithFault extends TileTests(SimpleTests, annotations =
    Seq(MiniTMR.faultyInst))

// This will pass even when it injects a fault because TMR is on every
    datapath register
class TileSimpleTestsWithRegRedundancyAndFault extends TileTests(SimpleTests,
  annotations = Seq(MiniTMR.redundantRegs, MiniTMR.faultyInst)
)
```

5.4 Summary

Colla-Gen provides a flexible yet powerful framework for generating hardware collateral in-sync with the generated design. It provides a reliable mechanism to reference design signals, as well as update these references under arbitrary FIRRTL transformations. By leveraging the FIRRTL hardware compiler framework's annotation system, its ordering semantics are no different than other compiler transformations. Its utility is demonstrated through the implementation of a wide variety of Colla-Gen libraries to tackle physical design, simulation, verification, and reliability issues. Overall, the Colla-Gen library interface provide a strong foundation for tackling the sizable and severe hardware collateral generation problem.

Example 5.7: Injecting Colla-Gen Library

```
/** Colla-Gen library to inject Chisel code into a module of type M */
case class InjectingLib[T <: RawModule,  // Type of top-level module
                        M <: RawModule]( // Type of module to inject HW
    selectRoots: T => Iterable[M], // Picks module instances from top-level
    to inject
    injection: M => Unit           // Generate Chisel hardware to inject in M
)(implicit tTag: TypeTag[T]) extends CollaGen[T] { // tTag prevents
    type-erasure

  final def toAnnotation(top: T): AnnotationSeq = {...} // Calls other
    toAnnotation

  /** Create annotations to trigger transformation which injects FIRRTL
    statements
    * Creates annotations by elaborating injection (Chisel code)
    */
  final def toAnnotation(modules: Iterable[M], circuit: String):
    AnnotationSeq = {
    val injectAnnos = modules.map {
      module =>
        // Elaborate hardware to inject by executing Chisel's builder
        // ModuleAspect is an internal detail to allow seamless Chisel
    generation
        val (chiselIR, _) = Builder.build(Module(new ModuleAspect(module) {
          module match {
            case x: experimental.MultiIOModule =>
              withClockAndReset(x.clock, x.reset) { injection(module) }
            case x: RawModule => injection(module)
          }
        }))
        // Collect injected hardware, contained in a DefModule
        val comps = chiselIR.components.map {
          case x: DefModule if x.name == module.name => x.copy(id = module)
          case other => other
        }

        // Collect any additional annotations generated from injected HW
        val annotations = chiselIR.annotations.map(_.toFirrtl).filterNot{ a
    => a.isInstanceOf[DesignAnnotation[_]] }

        // Convert elaborated injected HW into FIRRTL
        val stmts = mutable.ArrayBuffer[ir.Statement]()
        val modules = CollaGen.getFirrtl(chiselIR.copy(components =
    comps)).modules.flatMap {
          case m: firrtl.ir.Module if m.name == module.name =>
            stmts += m.body
            Nil
          case other =>Seq(other)
        }

        // Return annotations containing injection information
        InjectStatement(ModuleTarget(circuit, module.name), ir.Block(stmts),
    modules, annotations)
    }.toSeq
    Seq(RunFirrtlTransformAnnotation(new InjectingTransform)) ++ injectAnnos
  }
}
```

Example 5.8: Injecting Annotation and Transformation

```
/** Contains all information needed to inject statements into a module */
case class InjectStatement(
    module: ModuleTarget, // Module to inject code into at the end of the
    module
    s: firrtl.ir.Statement, // Statements to inject
    modules: Seq[firrtl.ir.DefModule], // Additional instantiated modules in s
    annotations: Seq[Annotation] // Additional annotations
) extends SingleTargetAnnotation[ModuleTarget] {...}

/** Appends statements contained in [[InjectStatement]] to corresponding
    module */
class InjectingTransform extends Transform {...}
```

Chapter 6

Research Contributions and Future Outlook

With Dennard scaling and Moore's law coming to a close, specialization is a promising approach to tackle the increasing demand for computation, but is limited by the time and cost of designing a custom chip. Designing for future reuse enables more amortization of cost to bring down the future cost of chips. However, unlike the software industry the hardware industry is inhibited by a lack of code reuse. Capturing reusable designs as hardware libraries requires a powerful programming language, as well as an abstraction layer to separate function from implementation.

This thesis outlines one approach to facilitating reuse through a hardware construction language, hardware compiler framework, and an aspect-oriented-programming (AOP) inspired paradigm for reusable design collateral. To enable hardware libraries, this thesis contributes the following: (1) a reemphasis on how hardware construction languages (HCLs) provide language expressivity to enable reusability, (2) hardware compiler frameworks as a mechanism for additional reusability via digital logic customization using a novel hardware intermediate representation, and (3) an AOP-inspired paradigm and implementation for tackling the design collateral reusability problem.

After a detailed introduction in Chapter 1 of the current landscape of the market and methodologies behind hardware design, Chapter 2 details the following three hypotheses for the current lack of hardware library development:

- Existing hardware description languages lack the expressivity to support hardware libraries
- Diverse underlying implementations require digital logic customization, limiting a design's reusability
- Effective physical design, verification, emulation and instrumentation require additional design collateral which is too tool/platform/technology dependent and brittle to design modifications

The remaining Chapters of the thesis strive to address the deficiencies highlighted by these hypotheses. Chapter 3 demonstrates how Chisel, a hardware construction language, provides the expressivity and power to support hardware libraries with a comparison between a Verilog-based project and a Chisel-based project, as well as multiple evaluations of its expressive power. Chapter 4 describes the details of a novel hardware compiler framework and intermediate representation (IR). It concludes with an extensive demonstration of its supported transformations and an evaluation of its optimization transformations as compared to another open-source synthesis tool. Finally, Chapter 5 introduces aspect-oriented programming as inspiration for a design collateral generator approach; many prototypes of collateral generator libraries are described and discussed, the central example enabling physical design floorplan generation. Through evaluating the reuse potential of Chisel, the demonstrating wide-ranging applications of a hardware compiler framework, and illustrating in-sync and reusable hardware collateral generators, this thesis demonstrates an approach to speeding the entire hardware design loop to bring custom hardware faster to market.

6.1 Status and Future Work

As of the writing of this thesis, many companies spanning from small start-ups to well-established companies have used Chisel and FIRRTL (but not yet Colla-Gen) to accelerate their custom hardware design processes. In this applied setting, the results were successful: hardware design was faster with this methodology. As also expected, future designs in the same family as the first design were significantly faster because large amounts of engineering effort were directly reused through the powerful parameterization that Chisel supports. However, their experience was not entirely rosy - these companies ran abruptly into the next bottlenecks in the design loop, namely verification and physical design. Verification was difficult but feasible with infrastructure additions, and physical design was mostly unchanged; however, neither could be reused for future designs. Given the newness of the aspect-inspired Colla-Gen framework, at this time no commercial entities have adopted it. As this uptake process continues, only time will tell whether it can live up to its promise of design collateral generation and reuse. However, this ecosystem's established following increases the likelihood of future adopters due to the reduced shared risk.

While Chisel, FIRRTL and Colla-Gen have many features and paths forward, there are certainly foreseeable limitations which require novel ideas and/or significant engineering. For Chisel, most analog design is outside its scope, as well as some digital constructs like latches; these features require using Chisel's blackbox module, but direct language support would be preferred by the user. Additionally, design testing is a large concern - existing Chisel testers are slow and difficult to create large reusable test benches; work is being done to improve the testing infrastructure but it remains to be seen whether it can address this large need. Another foreseeable Chisel limitation is its large requirement for memory size to compile large designs; these take up large amounts of heap, and thus often compile times are often slowed down. Worse, the Colla-Gen framework requires this design to be contained

in memory, which may limit its applicability to smaller designs or submodules of designs. For FIRRTL, its limitations are primarily in its friendliness for new users; standardization of FIRRTL transform writing, providing larger libraries of transformations, and a more detailed tutorial would ease the onboarding process for new FIRRTL transformation writers. Another current limitation is how FIRRTL transformations are scheduled; while a more powerful dependency feature is in the works, it remains to be seen whether it can capture relative transformation ordering constraints in a user-understandable way. One limitation of the Colla-Gen libraries is their resolution immediately after Chisel elaboration but before FIRRTL transformations, placing the restriction that generating collateral on signals which are added via a transformation requires non-trivial workarounds (as a generic approach to this problem has not yet been supported). As also detailed earlier, all Colla-Gen implementations are prototypes and have not been used for tapeouts or industry adoption - more development effort is required to make them accessible to users and for a real evaluation of their industry-ready potential.

The future challenges of this project are significant but still conquerable. The largest impediment is better integration with existing design ecosystems; learning Chisel is hard and most hardware engineers have little real software development experience (learning how to write hardware compiler transformations is even harder). Not every company can shift to this ecosystem and thus integration with non-Chisel digital logic designs gracefully is a major but important challenge to tackle. One approach which can ease this integration challenge is to build better tools for integration. There do exist some new solutions but are not widely adopted and its unknown whether they adequately tackle this problem. Another major challenge is verification in a generator-based design flow. While leveraging the Colla-Gen framework is an important first step, the exact verification constructs and approach is still unknown and an excellent area for additional research; lots of novelty is needed in this area to elevate verification to generator level. The next challenge is that this methodology still requires large run times for synthesis and place-and-route tools to get energy/area numbers. Faster evaluation of these metrics would further accelerate a designer's productivity. Additionally, generator-approaches increase the need for cloud-based regular regressions for which the per-user license model of current CAD (computer-aided design) companies doesn't address, and open-source CAD tools are still too premature to provide effective results (while hopefully this will change soon). A reasonable solution to this problem are FIRRTL-based estimations which would be less accurate but faster to compute and easily scalable to the cloud. Finally, managing a growing project and ecosystem is a future challenge; any open-source project must balance the disparate demands of many companies. Perhaps more difficult is finding and keeping the right contributors who match the specialized skill set required by this project. While the future challenges of Chisel, FIRRTL and Colla-Gen span open research problems, technical engineering effort, and human management, there are no fundamental challenges which block its increasing adoption into industry workflows.

This is a very exciting time for challenging the methodology status-quo. Market demands and current limitations point to a methodology solution, making tool designers in high demand and opening an opportunity to think big picture about how tools can make

hardware designers (and verification/physical design engineers) more effective and efficient. The impact potential has never been greater - there is a massive under-served market in custom silicon that can be torn wide open given the right tools and methodologies. As more companies begin building their own custom chips and run into the same methodology bottlenecks of the status-quo, the more attractive Chisel and other similar solutions become. Widely-used open-source solutions provide risk mitigation, as bugs are shared. Additionally, vertically-integrated companies have little incentive to bear the large upfront costs of building a competing custom tool when their individual market is smaller - sharing this infrastructure is a win-win for the industry because a large percentage of the players are not directly competing.

Of the future challenges mentioned, many of these problems require engineering solutions, which can be addressed with good integration with existing designs and tools/methodologies which are missing in the Chisel ecosystem. These missing features can then be incrementally supported directly in Chisel, enabling developers the time to properly design these features. Separate from the engineering problems is the project management, which requires active effort to maintain and grow the project, as well as finding dedicated contributors. Open-source contributions are best done by passionate individuals, and while the project is exciting, it is important to keep it an enjoyable experience as well.

Specialization is the future of hardware design, and increasing reusability within our hardware design methodologies is critical to meeting the incoming demand for chip diversity. Designers should focus on developing reusable hardware libraries, while researchers and developers should consider reusability as a primary focus of future languages and compilers. This thesis describes a Chisel ecosystem which is perfectly poised to support hardware libraries and tackle the hardware collateral barriers within the hardware design loop.

Appendix A

The FIRRTL Specification

The ideas for FIRRTL (Flexible Intermediate Representation for RTL) originated from work on Chisel, a hardware description language (HDL) embedded in Scala used for writing highly-parameterized circuit design generators. Chisel designers manipulate circuit components using Scala functions, encode their interfaces in Scala types, and use Scala's object-orientation features to write their own circuit libraries. This form of meta-programming enables expressive, reliable and type-safe generators that improve RTL design productivity and robustness.

A.1 Project History

The computer architecture research group at U.C. Berkeley relied critically on Chisel to allow small teams of graduate students to design sophisticated RTL circuits. Over a three year period with under twelve graduate students, the architecture group has taped-out over ten different designs.

Internally, the investment in developing and learning Chisel was rewarded with huge gains in productivity. At the time, Chisel's external rate of adoption was slow for the following reasons.

1. Writing custom circuit transformers requires intimate knowledge about the internals of the Chisel compiler.

2. Chisel semantics are underspecified and thus impossible to target from other languages.

3. Error checking is unprincipled due to underspecified semantics resulting in incomprehensible error messages.

4. Learning a functional programming language (Scala) is difficult for RTL designers with limited programming language experience.

5. Confounding the previous point, conceptually separating the embedded Chisel HDL from the host language is difficult for new users.

6. The output of Chisel (Verilog) is unreadable and slow to simulate.

As a consequence, Chisel needed to be redesigned from the ground up to standardize its semantics, modularize its compilation process, and cleanly separate its front-end, intermediate representation, and backends. A well defined intermediate representation (IR) allows the system to be targeted by other HDLs embedded in other host programming languages, making it possible for RTL designers to work within a language they are already comfortable with. A clearly defined IR with a concrete syntax also allows for inspection of the output of circuit generators and transformers thus making clear the distinction between the host language and the constructed circuit. Clearly defined semantics allow users without knowledge of the compiler implementation to write circuit transformers; examples include optimization of circuits for simulation speed, and automatic insertion of signal activity counters. An additional benefit of a well defined IR is the structural invariants that can be enforced before and after each compilation stage, resulting in a more robust compiler and structured mechanism for error checking.

A.2 Details about Syntax

FIRRTL's syntax is designed to be human-readable but easily algorithmically parsed.

The following characters are allowed in identifiers: upper and lower case letters, digits, and _. Identifiers cannot begin with a digit.

An integer literal in FIRRTL begins with one of the following, where '#' represents a digit between 0 and 9.

- '0x' : For indicating a hexadecimal number. The rest of the literal must consist of either digits or a letter between 'A' and 'F', or the separator '_'.
- '0o' : For indicating an octal number. The rest of the literal must consist of digits between 0 and 7, or the separator '_'.
- '0b' : For indicating a binary number. The rest of the literal must consist of either 0 or 1, or the separator '_'.
- '-#' : For indicating a negative decimal number. The rest of the literal must consist of digits between 0 and 9.
- '#' : For indicating a positive decimal number. The rest of the literal must consist of digits between 0 and 9.

Comments begin with a semicolon and extend until the end of the line. Commas are treated as whitespace, and may be used by the user for clarity if desired.

Block structuring is indicated using indentation. Statements are combined into statement groups by surrounding them with parentheses. A colon at the *end of a line* will automatically surround the next indented region with parenthesis and thus create a statement group.

The following statement:

Example A.1:

```
when c :
   a <= b
else :
   c <= d
   e <= f
```

can be equivalently expressed on a single line as follows.

Example A.2:

```
when c : (a <= b) else : (c <= d, e <= f)
```

All circuits, modules, ports and statements can optionally be followed with the info token `@[fileinfo]` where fileinfo is a string containing the source file information from where it was generated.

The following example shows the info tokens included:

Example A.3:

```
circuit Top : @["myfile.txt: 14, 8"]
   module Top : @["myfile.txt: 15, 2"]
     output out:UInt @["myfile.txt: 16, 3"]
     input b:UInt<32> @["myfile.txt: 17, 3"]
     input c:UInt<1> @["myfile.txt: 18, 3"]
     input d:UInt<16> @["myfile.txt: 19, 3"]
     wire a:UInt @["myfile.txt: 21, 8"]
     when c : @["myfile.txt: 24, 8"]
       a <= b @["myfile.txt: 27, 16"]
     else :
       a <= d @["myfile.txt: 29, 17"]
     out <= add(a,a) @["myfile.txt: 34, 4"]
```

A.3 Circuits and Modules

A.3.1 Circuits

Every FIRRTL circuit consists of a list of modules, each representing a hardware block that can be instantiated. The circuit must specify the name of the top-level module; because the circuit represents an entire hardware design, the top-level input/outputs are dictated by the top-level module.

Example A.4:

```
circuit MyTop :
   module MyTop :
      ...
   module MyModule :
      ...
```

A.3.2 Modules

Each module has a given name, a list of ports, and a statement representing the circuit connections within the module. A module port is specified by its *direction* (which may be *input* or *output*), a name, and the data type of the port.

The following example declares a module with one input port, one output port, and one statement connecting the input port to the output port. See section A.5.1 for details on the connect statement.

Example A.5:

```
module MyModule :
   input foo: UInt
   output bar: UInt
   bar <= foo
```

Note that a module definition does *not* indicate that the module will be physically present in the final circuit. Refer to the description of the instance statement for details on how to instantiate a module (section A.5.13).

A.3.3 Externally Defined Modules

Externally defined modules consist of a given name, and a list of ports, whose types and names must match its external definition.

Example A.6:

```
extmodule MyExternalModule :
   input foo: UInt
   output bar: UInt
   output baz: SInt
```

A.4 Types

Types are used to specify the structure of the data held by each circuit component. Every type in FIRRTL is either one of the fundamental ground types or built up by aggregating other types.

A.4.1 Ground Types

There are six ground types in FIRRTL: an unsigned integer type, a signed integer type, a fixed-point type, an interval type, a clock type, and an analog type.

Integer Types

Both unsigned and signed integer types may optionally be given a known positive integer bit width.

Example A.7:

```
UInt<10>
SInt<32>
```

Alternatively, if the bit width is omitted, it will be automatically inferred by FIRRTL's width inferencer, as detailed in section 4.3.

Example A.8:

```
UInt
SInt
```

Fixed-Point Type

Fixed-point types have both a width and a precision; a precision's value refers to the number of bits, from the full bit width, that represent fractional bits. The precision value can be negative, but cannot be greater than the bit width. The precision value is represented by the second integer surrounded by « and ».

Example A.9:

```
Fixed<10><<2>>
Fixed<4><<-1>>
```

Both the width and precision values can be inferred by FIRRTL's value inferencer, as detailed in section 4.3.

Interval Type

Interval types have an upper bound, a lower bound, and a precision. Their bound values can be either closed or open, and are represented by a *double* as opposed to an integer. Both their bounds and precision can be inferred by FIRRTL's value inferencer (section 4.3). Like the fixed point type, the interval type precision can be negative.

Example A.10:

```
Interval[0, 10.432].2
Interval(-1, 20].4
```

If the bounds are not evenly divided into their precision, they will be trimmed to the closest evenly-divided value within the interval. Open bounds indicate an interval boundary that does not include the specified value; this is useful when the precision is unknown and thus the true boundary value of an open bound is unknown.

Clock Type

The clock type is used to describe wires and ports meant for carrying clock signals. The usage of components with clock types is restricted. Clock signals cannot be used in most primitive operations, and clock signals can only be connected to components that have been declared with the clock type.

The clock type is specified as follows:

Example A.11:

```
Clock
```

Analog Type

The analog type is used to describe wires and ports meant for signals that lack directionality. The usage of components with analog types is restricted. Analog signals cannot be used in most primitive operations, nor can they be used in the connection statement. They can only be used in the attach statement and in partial connection statements to other analog-typed signals.

The analog type is specified as follows:

Example A.12:

```
Analog<10>
```

Ports can also be declared as an analog type; the direction of this port is irrelevant to the behavior of the signal.

A.4.2 Vector Types

A vector type is used to express an ordered sequence of elements of a given type. The length of the sequence must be greater than zero and known.

The following example specifies a ten-element vector of 16-bit unsigned integers.

Example A.13:

```
UInt<16>[10]
```

The next example specifies a ten-element vector of unsigned integers of omitted but identical bit widths.

Example A.14:

```
UInt[10]
```

Note that any type, including other aggregate types, may be used as the element type of the vector. The following example specifies a twenty-element vector, each of which is a ten element vector of 16-bit unsigned integers.

Example A.15:

```
UInt<16>[10][20]
```

A.4.3 Bundle Types

A bundle type is used to express a collection of nested and named types. Every field in a bundle type must have a given name and a type. No two fields in a bundle type may have the same name.

The following is an example of a possible type for representing a complex number. It has two fields, `real`, and `imag`, each a 10-bit signed integer.

Example A.16:

```
{real:SInt<10>, imag:SInt<10>}
```

Additionally, a field may optionally be declared with a *flipped* orientation.

Example A.17:

```
{word:UInt<32>, valid:UInt<1>, flip ready:UInt<1>}
```

In a connection from a circuit component with a bundle type to another circuit component with a bundle type, the data carried by the flipped fields flow in the opposite direction as the data carried by the non-flipped fields.

As an example, consider a module output port declared with the following type:

Example A.18:

```
output a: {word:UInt<32>, valid:UInt<1>, flip ready:UInt<1>}
```

In a connection to the `a` port, the data carried by the `word` and `valid` subfields will flow out of the module, while data carried by the `ready` subfield will flow into the module. More details about how the bundle field orientation affects connections are explained in section A.5.1.

As in the case of vector types, a bundle field may be declared with any type, including other aggregate types.

Example A.19:

```
{real: {word:UInt<32>, valid:UInt<1>, flip ready:UInt<1>}
 imag: {word:UInt<32>, valid:UInt<1>, flip ready:UInt<1>}}
```

When calculating the final direction of data flow, the orientation of a field is applied recursively to all nested types in the field. As an example, consider the following module port declared with a bundle type containing a nested bundle type.

Example A.20:

```
output myport: {a: UInt, flip b: {c: UInt, flip d:UInt}}
```

In a connection to `myport`, the `a` subfield flows out of the module. The `c` subfield contained in the `b` subfield flows into the module, and the `d` subfield contained in the `b` subfield flows out of the module.

A.4.4 Passive Types

It is inappropriate for some circuit components to be declared with a type that allows for data to flow in both directions. For example, all subelements in a memory should flow in the same direction. These components are restricted to only have a passive type.

Intuitively, a passive type is a type where all data flows in the same direction, and is defined to be a type that recursively contains no fields with flipped orientations. Thus every ground type is a passive type. A vector type is passive if and only if its element type is passive. A bundle type is passive if and only if every one of the bundle type's fields is not flipped and has a passive type.

A.4.5 Type Equivalence

The type equivalence relation is used to determine whether a connection is proper. See section A.5.1 for further details about connect statements.

An unsigned integer type is always equivalent to another unsigned integer type regardless of bit width, and is not equivalent to any other type. Similarly, a signed integer type is always equivalent to another signed integer type regardless of bit width, and is not equivalent to any other type.

Clock types are equivalent to clock types, and are not equivalent to any other type.

Two vector types are equivalent if they have the same length, and if their element types are equivalent.

Two bundle types are equivalent if they have the same number of fields, and both the bundles' i'th fields have matching names and orientations, as well as equivalent types. Consequently, `{a:UInt, b:UInt}` is not equivalent to `{b:UInt, a:UInt}`, and `{a: {flip b:UInt}}` is not equivalent to `{flip a: {b: UInt}}`.

A.4.6 Weak Type Equivalence

The weak type equivalence relation is used to determine whether a partial connection is proper. See section A.5.2 for further details about partial connect statements.

Two types are weakly equivalent if their corresponding oriented types are equivalent.

Oriented Types

The weak type equivalence relation requires first a definition of *oriented types*. Intuitively, an oriented type is a type where all orientation information is collated and coupled with the leaf ground types instead of in bundle fields.

An oriented ground type is an orientation coupled with a ground type. An oriented vector type is an ordered sequence of positive length of elements of a given oriented type. An oriented bundle type is a collection of oriented fields, each containing a name and an oriented type, but no orientation.

Applying a flip orientation to an oriented type reverses the orientation of every oriented ground type contained within. Applying a non-flip orientation to an oriented type does nothing.

Conversion to Oriented Types

To convert a ground type to an oriented ground type, attach a non-flip orientation to the ground type.

To convert a vector type to an oriented vector type, convert its element type to an oriented type, and retain its length.

To convert a bundle field to an oriented field, convert its type to an oriented type, apply the field orientation, and combine this with the original field's name to create the oriented field. To convert a bundle type to an oriented bundle type, convert each field to an oriented field.

Oriented Type Equivalence

Two oriented ground types are equivalent if their orientations match and their types are equivalent.

Two oriented vector types are equivalent if their element types are equivalent.

Two oriented bundle types are not equivalent if there exist two fields, one from each oriented bundle type, that have identical names but whose oriented types are not equivalent. Otherwise, the oriented bundle types are equivalent.

As stated earlier, two types are weakly equivalent if their corresponding oriented types are equivalent.

A.5 Statements

Statements are used to describe the components within a module and how they interact.

A.5.1 Connect Statement

The connect statement is used to specify a physically wired connection from one circuit component to another circuit component.

The following example demonstrates connecting a module's input port to its output port, where port `myinput` is connected to port `myoutput`.

Example A.21:

```
module MyModule :
   input myinput: UInt
   output myoutput: UInt
   myoutput <= myinput
```

In order for a connection to be proper the following conditions must hold:

1. The types of the left-hand and right-hand side expressions must be equivalent (see section A.4.5 for details).
2. The bit widths of the two expressions must allow for data to always flow from a smaller bit width to an equal size or larger bit width.
3. The flow of the left-hand side expression must be sink or duplex (see section A.8 for an explanation of flow).
4. Either the flow of the right-hand side expression is source or duplex, or the right-hand side expression has a passive type.

If a connect statement happens to connect a ground-type component to a wider component, then the ground type must be an integer type, and the value that flows through that connection will automatically be sign-extended (if it is a signed integer) or zero-extended (if it is an unsigned integer). The behavior of a connect statement from one circuit component with an aggregate type to another with an aggregate type is defined by the connection algorithm in section A.5.3.

A.5.2 Partial Connect Statement

Like the connect statement, the partial connect statement is also used to specify a physically wired connection from one one circuit component to another. However, it enforces fewer restrictions on the types and widths of the circuit components it connects.

In order for a partial connect to be legal the following conditions must hold:

1. The types of the left-hand and right-hand side expressions must be weakly equivalent (see section A.4.6 for details).
2. The flow of the left-hand side expression must be sink or duplex (see section A.8 for an explanation of flow).

3. Either the flow of the right-hand side expression is source or duplex, or the right-hand side expression has a passive type.

If a partial connect statement happens to connect a ground-type component to a wider component, then the ground type must be an integer type, and the value that flows through that connection will automatically be sign-extended (if it is a signed integer) or zero-extended (if it is an unsigned integer). Partial connect statements from a wider ground type component to a narrower ground type component will have its value automatically truncated to fit the smaller bit width.

Intuitively, bundle fields with matching names will be connected appropriately, while bundle fields not present in both types will be ignored. Similarly, vectors with mismatched lengths will be connected up to the shorter length, and the remaining subelements are ignored.

The following example demonstrates partially connecting a module's input port to its output port, where port `myinput` is connected to port `myoutput`.

Example A.22:

```
module MyModule :
   input myinput: {flip a:UInt, b:UInt[2]}
   output myoutput: {flip a:UInt, b:UInt[3], c:UInt}
   myoutput <- myinput
```

The above example is equivalent to the following:

Example A.23:

```
module MyModule :
   input myinput: {flip a:UInt, b:UInt[2]}
   output myoutput: {flip a:UInt, b:UInt[3], c:UInt}
   myinput.a <- myoutput.a
   myoutput.b[0] <- myinput.b[0]
   myoutput.b[1] <- myinput.b[1]
```

For details on the syntax and semantics of the subfield expression, subindex expression, and statement groups, see sections A.6.6, A.6.7, and A.5.5.

A.5.3 The Connection Algorithm

Both connection statements and partial connection statements (both denoted as a *connection*) follow the following algorithm to determine whether each subelement of the left-hand side of the *connection* is connected to or connected from a corresponding subelement of the right-hand side of the *connection*. Any reference to a *connection* implies the original statement type, either a connection statement or a partial connection statement.

A *connection* from one ground typed component to another ground typed component connects the right-hand side component to the left-hand side component.

A *connection* from a vector typed component to another vector typed component recursively applies a *connection* from each of the first n indexed subelements in the right-hand side component to each of the first n corresponding indexed subelements in the left-hand side component, where n is the length of the shorter vector. Note that the vector lengths can only mismatch in a partial connection statement.

A *connection* from a bundle typed component to another bundle typed component considers all pairs of subelement fields with matching names, where the first field in the pair is from the left-hand side component and the second field in the pair is from the right-hand side component. If the first field in the pair *is not flipped*, then a *connection* is applied from the *second* field in the pair (from the right-hand side component) to the *first* field in the pair (from the left-hand side component). However, if the first field in the pair *is flipped*, then a *connection* is applied from the *first* field in the pair (from the left-hand side component) to the *second* field in the pair (from the right-hand side component). Note that unmatched fields or pairs of differently-ordered fields within their respective bundle types can only occur in a partial connection statement.

A.5.4 Attach

A sequence of analog-typed signals can be connected to one another with the attach statement. Multiple attach statements can reference the same signal; if this occurs, it is as if the set of both attach statement's analog signals are all attached to one another. Only analog types of equivalent widths can be attached.

Example A.24:

```
module MyModule :
   input a: Analog<8>
   input b: Analog<8>
   output c: Analog<8>
   output d: Analog<8>
   attach(a, b, c)
   attach(c, d)
```

In the previous example, signal `d` is attached to `c` which is in turn attached to signals `a` and `b`. Due to the nature of the attach statement, `d` is also attached to `a` and `b`.

A.5.5 Statement Group

An ordered sequence of one or more statements can be grouped into a single statement, called a statement group. The following example demonstrates a statement group composed of three connect statements.

Example A.25:

```
module MyModule :
   input a: UInt
   input b: UInt
   output myport1: UInt
   output myport2: UInt
   myport1 <= a
   myport1 <= b
   myport2 <= a
```

Last Connect Semantics

Ordering of statements is significant in a statement group. Intuitively, during elaboration, statements execute in order, and the effects of later statements take precedence over earlier ones. In the previous example, in the resultant circuit, port `b` will be connected to `myport1`, and port `a` will be connected to `myport2`.

Note that if a user desires to enforce single-connection semantics, in some cases they may instead use a node statement (see A.5.10).

Note that connect and partial connect statements have equal priority, and later connect or partial connect statements always take priority over earlier connect or partial connect statements. Conditional statements are also affected by last connect semantics, and for details see section A.5.11.

In the case where a connection to a circuit component with an aggregate type is followed by a connection to a subelement of that component, only the connection to the subelement is overwritten. Connections to the other subelements remain unaffected. In the following example, in the resultant circuit, the `c` subelement of port `portx` will be connected to the `c` subelement of `myport`, and port `porty` will be connected to the `b` subelement of `myport`.

Example A.26:

```
module MyModule :
   input portx: {b:UInt, c:UInt}
   input porty: UInt
   output myport: {b:UInt, c:UInt}
   myport <= portx
   myport.b <= porty
```

The above circuit can be rewritten equivalently as follows.

In the case where a connection to a subelement of an aggregate circuit component is followed by a connection to the entire circuit component, the later connection overwrites the earlier connections completely.

The above circuit can be rewritten equivalently as follows.

See section A.6.6 for more details about subfield expressions.

Example A.27:

```
module MyModule :
   input portx: {b:UInt, c:UInt}
   input porty: UInt
   output myport: {b:UInt, c:UInt}
   myport.b <= porty
   myport.c <= portx.c
```

Example A.28:

```
module MyModule :
   input portx: {b:UInt, c:UInt}
   input porty: UInt
   output myport: {b:UInt, c:UInt}
   myport.b <= porty
   myport <= portx
```

Example A.29:

```
module MyModule :
   input portx: {b:UInt, c:UInt}
   input porty: UInt
   output myport: {b:UInt, c:UInt}
   myport <= portx
```

A.5.6 Skip Statement

The skip statement does nothing and is used simply as a placeholder where a statement is expected. It is specified using the `skip` keyword.

The following example:

Example A.30:

```
a <= b
skip
c <= d
```

can be equivalently expressed as:

Example A.31:

```
a <= b
c <= d
```

The skip statement is most often used as a convenient placeholder for removed components during transformational passes, or can also be used as the `else` branch in a conditional

statement (a `when` statement with no `else` clause is syntactic sugar for an `else` branch with a `skip` statement). See section A.5.11 for details on the conditional statement.

A.5.7 Wire Declaration

A wire is a named combinational circuit component that can be connected to and from using connect and partial connect statements.

The following example demonstrates instantiating a wire with the given name `mywire` and type `UInt`.

Example A.32:

```
wire mywire : UInt
```

A wire entirely equivalent to an instance of a module with one input port and one output port of the same type, such that the input port is connected to the output port, except for the syntactic shorthand that one can connect directly to and from the wire using just the wire name without having to explicitly mention its port names.

A.5.8 Register Declaration

A register is a named stateful circuit component.

The following example demonstrates instantiating a register with the given name `myreg` and type `SInt`, and is driven by the clock signal `myclock`. A register must be declared with a passive type.

Example A.33:

```
wire myclock: Clock
reg myreg: SInt, myclock
...
```

Optionally, for the purposes of circuit initialization, a register can be declared with a reset signal and value. In the following example, `myreg` is assigned the value `myinit` when the signal `myreset` is high.

Example A.34:

```
wire myclk: Clock
wire myrst: UInt<1>
wire myinit: SInt
reg myreg: SInt, myclk with: (reset => (myrst, myinit))
...
```

Note that the clock signal for a register must be of type `clock`, the reset signal must be a single-bit `UInt`, and the type of initialization value must match the declared type of the register.

A.5.9 Invalidation Statement

An invalidation statement is used to indicate that a circuit component contains indeterminate values. It is specified as follows:

Example A.35:

```
wire w:UInt
w is invalid
```

Invalidation statements can be applied to any circuit component of any type. However, if the circuit component cannot be connected to (for example, an input port of a module), then the statement has no effect on the component. This allows the invalidation statement to be applied to any component, to explicitly ignore initialization coverage errors.

The following example demonstrates the effect of invalidating a variety of circuit components with aggregate types. See section A.5.9 for details on the algorithm for determining what is invalidated.

Example A.36:

```
module MyModule :
   input in: {flip a:UInt, b:UInt}
   output out: {flip a:UInt, b:UInt}
   wire w: {flip a:UInt, b:UInt}
   in is invalid
   out is invalid
   w is invalid
```

is equivalent to the following:

Example A.37:

```
module MyModule :
   input in: {flip a:UInt, b:UInt}
   output out: {flip a:UInt, b:UInt}
   wire w: {flip a:UInt, b:UInt}
   in.a is invalid
   out.b is invalid
   w.a is invalid
   w.b is invalid
```

For the purposes of simulation, invalidated components could either be initialized to random values or supported directly if the simulator supports three-valued bits (0, 1, and invalid). In terms of its specification behavior, operations on indeterminate values produce undefined behavior.

The Invalidate Algorithm

Invalidating a component with a ground type indicates that the component's value is indetermined if the component is sink or duplex (see section A.8). Otherwise, the component is unaffected.

Invalidating a component with a vector type recursively invalidates each subelement in the vector.

Invalidating a component with a bundle type recursively invalidates each subelement in the bundle.

A.5.10 Nodes

A node is simply a named intermediate value in a circuit. The node must be initialized to a value with a passive type and cannot be connected to. Nodes are often used to split a complicated compound expression into named subexpressions.

The following example demonstrates instantiating a node with the given name `mynode` initialized with the output of a multiplexor (see section A.6.9).

Example A.38:

```
wire pred: UInt<1>
wire a: SInt
wire b: SInt
node mynode = mux(pred, a, b)
...
```

A.5.11 Conditionals

Connections within the first substatement (or statement group) of a conditional statement that connect to components declared prior to the conditional statement hold only when the given condition is high. If a conditional statement has an "else" clause, then connections within the substatement (or statement group) after the "else" keyword that connect to components declared prior to the conditional statement hold only when the given condition is low. The condition must have a 1-bit unsigned integer type.

In the following example, the wire `x` is connected to the input `a` only when the `en` signal is high. Otherwise, the wire `x` is connected to the input `b`.

Example A.39:

```
module MyModule :
   input a: UInt
   input b: UInt
   input en: UInt<1>
   wire x: UInt
   when en :
      x <= a
   else :
      x <= b
```

Syntactic Shorthands

The `else` branch of a conditional statement may be omitted, in which case a default `else` branch is supplied consisting of the empty statement.

Thus the following example:

Example A.40:

```
module MyModule :
   input a: UInt
   input b: UInt
   input en: UInt<1>
   wire x: UInt
   when en :
      x <= a
```

can be equivalently expressed as:

Example A.41:

```
module MyModule :
   input a: UInt
   input b: UInt
   input en: UInt<1>
   wire x: UInt
   when en :
      x <= a
   else :
      skip
```

To aid readability of long chains of conditional statements, the colon following the `else` keyword may be omitted if the `else` branch consists of a single conditional statement.

Thus the following example:

Example A.42:

```
module MyModule :
   input a: UInt
   input b: UInt
   input c: UInt
   input d: UInt
   input c1: UInt<1>
   input c2: UInt<1>
   input c3: UInt<1>
   wire x: UInt
   when c1 :
      x <= a
   else :
      when c2 :
         x <= b
      else :
         when c3 :
```

```
            x <= c
         else :
            x <= d
```

can be equivalently written as:

Example A.43:

```
module MyModule :
   input a: UInt
   input b: UInt
   input c: UInt
   input d: UInt
   input c1: UInt<1>
   input c2: UInt<1>
   input c3: UInt<1>
   wire x: UInt
   when c1 :
      x <= a
   else when c2 :
      x <= b
   else when c3 :
      x <= c
   else :
      x <= d
```

Declarations Nested Within Conditional Statements

If a component is declared within a conditional statement, connections to the component are unaffected by the condition. In the following example, `a` is always connected to register `myreg1`, and `b` is always connected to register `myreg2`.

Example A.44:

```
module MyModule :
   input a: UInt
   input b: UInt
   input en: UInt<1>
   input clk : Clock
   when en :
      reg myreg1 : UInt, clk
      myreg1 <= a
   else :
      reg myreg2 : UInt, clk
      myreg2 <= b
```

Intuitively, a line can be drawn from the component on the left-hand side of a connection (or partial connection) to that component's declaration. All conditional statements that are crossed by the line apply to that connection (or partial connection).

Initialization Coverage

Because of the conditional statement, it is possible to syntactically express circuits containing wires that have not been connected to under all conditions.

In the following example, the wire `a` is connected to the wire `w` when `en` is high, but it is not specified what is connected to `w` when `en` is low.

Example A.45:

```
module MyModule :
   input en: UInt<1>
   input a: UInt
   wire w: UInt
   when en :
      w <= a
```

This is an improper FIRRTL circuit and an error will be reported during compilation. All wires, memory ports, instance ports, and module ports that can be connected to must be connected to under all conditions. A register does not need to be connected to under all conditions, as it will keep its previous value if unconnected.

Scoping

The conditional statement creates a new *scope* within each of its `when` and `else` branches. It is an error to refer to any component declared within a branch after the branch has ended. This is unlike a statement group, which does not create a new scope.

Conditional Last Connect Semantics

In the case where a connection to a circuit component is followed by a conditional statement containing a connection to the same component, the connection is overwritten only when the condition holds. Intuitively, a multiplexor is generated such that when the condition is low, the multiplexor returns the old value, and otherwise returns the new value. For details about the multiplexor, see section A.6.9.

The following example:

Example A.46:

```
wire a: UInt
wire b: UInt
wire c: UInt<1>
wire w: UInt
w <= a
when c :
   w <= b
...
```

can be rewritten equivalently using a multiplexor as follows:

Example A.47:

```
wire a: UInt
wire b: UInt
wire c: UInt<1>
wire w: UInt
w <= mux(c, b, a)
...
```

In the case where a component is first invalided with an invalidation statement, and is then followed by a conditional statement containing a connection to this component, the resulting connection to the component can be expressed using a conditionally valid expression. See section A.6.10 for more details about the conditionally valid expression.

Example A.48:

```
wire a: UInt
wire c: UInt<1>
wire w: UInt
w is invalid
when c :
   w <= a
...
```

can be rewritten equivalently as follows:

Example A.49:

```
wire a: UInt
wire c: UInt<1>
wire w: UInt
w <= validif(c, a)
...
```

The behavior of conditional connections to circuit components with aggregate types can be modeled by first expanding each connect into individual connect statements on its ground elements (see section A.5.3 for the connection and partial connection algorithms) and then applying the conditional last connect semantics.

For example, the following snippet:

Example A.50:

```
wire x: {a:UInt, b:UInt}
wire y: {a:UInt, b:UInt}
wire c: UInt<1>
wire w: {a:UInt, b:UInt}
w <= x
when c :
   w <= y
...
```

can be rewritten equivalently as follows:

Example A.51:

```
wire x: {a:UInt, b:UInt}
wire y: {a:UInt, b:UInt}
wire c: UInt<1>
wire w: {a:UInt, b:UInt}
w.a <= mux(c, y.a, x.a)
w.b <= mux(c, y.b, x.b)
...
```

Similar to the behavior of aggregate types under last connect semantics (see section A.5.5), the conditional connects to a subelement of an aggregate component only generates a multiplexor for the subelement that is overwritten.

For example, the following snippet:

Example A.52:

```
wire x: {a:UInt, b:UInt}
wire y: UInt
wire c: UInt<1>
wire w: {a:UInt, b:UInt}
w <= x
when c :
   w.a <= y
...
```

can be rewritten equivalently as follows:

Example A.53:

```
wire x: {a:UInt, b:UInt}
wire y: UInt
wire c: UInt<1>
wire w: {a:UInt, b:UInt}
w.a <= mux(c, y, x.a)
w.b <= x.b
...
```

A.5.12 Memories

A memory is an abstract representation of a hardware memory. It is characterized by the following parameters.

1. A passive type representing the type of each element in the memory.
2. A positive integer representing the number of elements in the memory.
3. A variable number of named ports, each being a read port, a write port, or readwrite port.

4. A non-negative integer indicating the read latency, which is the number of cycles after setting the port's read address before the corresponding element's value can be read from the port's data field.

5. A non-negative integer indicating the write latency, which is the number of cycles after setting the port's write address and data before the corresponding element within the memory holds the new value.

6. A read-under-write flag indicating the behavior when a memory location is written to while a read to that location is in progress.

The following example demonstrates instantiating a memory containing 256 complex numbers, each with 16-bit signed integer fields for its real and imaginary components. It has two read ports, `r1` and `r2`, and one write port, `w`. It is combinationally read (read latency is zero cycles) and has a write latency of one cycle. Finally, its read-under-write behavior is undefined.

Example A.54:

```
mem mymem :
  data-type => {real:SInt<16>, imag:SInt<16>}
  depth => 256
  reader => r1
  reader => r2
  writer => w
  read-latency => 0
  write-latency => 1
  read-under-write => undefined
```

In the example above, the type of `mymem` is:

Example A.55:

```
{flip r1: {flip data: {real:SInt<16>, imag:SInt<16>},
           addr: UInt<8>,
           en: UInt<1>,
           clk: Clock}
 flip r2: {flip data: {real:SInt<16>, imag:SInt<16>},
           addr: UInt<8>,
           en: UInt<1>,
           clk: Clock}
 flip w: {data: {real:SInt<16>, imag:SInt<16>},
          mask: {real:UInt<1>, imag:UInt<1>},
          addr: UInt<8>,
          en: UInt<1>,
          clk: Clock}}
```

The following sections describe how a memory's field types are calculated and the behavior of each type of memory port.

Reader Ports

If a memory is declared with element type `T`, has a size less than or equal to 2^N, then its reader ports have type:

Example A.56:

```
{flip data:T, addr:UInt<N>, en:UInt<1>, clk:Clock}
```

If the `en` field is high, then the element value associated with the address in the `addr` field can be retrieved by reading from the `data` field after the appropriate read latency. If the `en` field is low, then the value in the `data` field, after the appropriate read latency, is undefined. The port is driven by the clock signal in the `clk` field.

Writer Ports

If a memory is declared with element type `T`, has a size less than or equal to 2^N, then its writer ports have type:

Example A.57:

```
{data:T, mask:M, addr:UInt<N>, en:UInt<1>, clk:Clock}
```

where `M` is the mask type calculated from the element type `T`. Intuitively, the mask type mirrors the aggregate structure of the element type except with all ground types replaced with a single bit unsigned integer type. The *non-masked portion* of the data value is defined as the set of data value leaf subelements where the corresponding mask leaf subelement is high.

If the `en` field is high, then the non-masked portion of the `data` field value is written, after the appropriate write latency, to the location indicated by the `addr` field. If the `en` field is low, then no value is written after the appropriate write latency. The port is driven by the clock signal in the `clk` field.

Readwriter Ports

Finally, the readwriter ports have type:

Example A.58:

```
{wmode:UInt<1>, flip rdata:T, wdata:T, wmask:M,
 addr:UInt<N>, en:UInt<1>, clk:Clock}
```

A readwriter port is a single port that, on a given cycle, can be used either as a read or a write port. If the readwriter port is not in write mode (the `wmode` field is low), then the `rdata`, `addr`, `en`, and `clk` fields constitute its read port fields, and should be used accordingly. If the readwriter port is in write mode (the `wmode` field is high), then the `wdata`, `wmask`, `addr`, `en`, and `clk` fields constitute its write port fields, and should be used accordingly.

Read Under Write behavior

The read-under-write flag indicates the value held on a read port's `data` field if its memory location is written to while it is reading. The flag may take on three settings: `old`, `new`, and `undefined`.

If the read-under-write flag is set to `old`, then a read port always returns the value existing in the memory on the same cycle that the read was requested.

Assuming that a combinational read always returns the value stored in the memory (no write forwarding), then intuitively, this is modeled as a combinational read from the memory that is then delayed by the appropriate read latency.

If the read-under-write flag is set to `new`, then a read port always returns the value existing in the memory on the same cycle that the read was made available. Intuitively, this is modeled as a combinational read from the memory after delaying the read address by the appropriate read latency.

If the read-under-write flag is set to `undefined`, then the value held by the read port after the appropriate read latency is undefined.

In all cases, if a memory location is written to by more than one port on the same cycle, the stored value is undefined.

A.5.13 Instance Declaration

FIRRTL modules are instantiated with the instance statement. The following example demonstrates creating an instance named `myinstance` of the `MyModule` module within the top level module `Top`.

Example A.59:

```
circuit Top :
   module MyModule :
      input a: UInt
      output b: UInt
      b <= a
   module Top :
      inst myinstance of MyModule
```

The resulting instance has a bundle type. Each port of the instantiated module is represented by a field in the bundle with the same name and type as the port. The fields corresponding to input ports are flipped to indicate their data flows in the opposite direction as the output ports. The `myinstance` instance in the example above has type `{flip a:UInt, b:UInt}`.

Modules have the property that instances can always be *inlined* into the parent module without affecting the semantics of the circuit.

To disallow infinitely recursive hardware, a module must not contain instances of itself, either directly, or indirectly through instances of other modules it instantiates.

A.5.14 Stops

The stop statement is used to halt simulations of the circuit. Backends are free to generate hardware to stop a running circuit for the purpose of debugging, but this is not required by the FIRRTL specification.

A stop statement requires a clock signal, a halt condition signal that has a single bit unsigned integer type, and an integer exit code.

Example A.60:

```
wire clk:Clock
wire halt:UInt<1>
stop(clk,halt,42)
...
```

A.5.15 Formatted Prints

The formatted print statement is used to print a formatted string during simulations of the circuit. Backends are free to generate hardware that relays this information to a hardware test harness, but this is not required by the FIRRTL specification.

A printf statement requires a clock signal, a print condition signal, a format string, and a variable list of argument signals. The condition signal must be a single bit unsigned integer type, and the argument signals must each have a ground type.

Example A.61:

```
wire clk:Clock
wire condition:UInt<1>
wire a:UInt
wire b:UInt
printf(clk, condition, "a=0x%x, b=%d.\n", a, b)
...
```

On each positive clock edge, when the condition signal is high, the printf statement prints out the format string where its argument placeholders are substituted with the value of the corresponding argument.

Format Strings

Format strings support the following argument placeholders:

- `%b` : Prints the argument in binary
- `%d` : Prints the argument in decimal
- `%x` : Prints the argument in hexadecimal
- `%%` : Prints a single `%` character

Format strings support the following escape characters:

- `\n` : New line
- `\t` : Tab
- `\\` : Back slash
- `\"` : Double quote
- `\'` : Single quote

A.6 Expressions

FIRRTL expressions are used for creating literal unsigned and signed integers, for referring to a declared circuit component, for statically and dynamically accessing a nested element within a component, for creating multiplexors and conditionally valid signals, and for performing primitive operations.

A.6.1 Unsigned Integers

A literal unsigned integer can be created given a non-negative integer value and an optional positive bit width. The following example creates a 10-bit unsigned integer representing the number 42.

Example A.62:

```
UInt<10>(42)
```

Note that it is an error to supply a bit width that is not large enough to fit the given value. If the bit width is omitted, then the minimum number of bits necessary to fit the given value will be inferred.

Example A.63:

```
UInt(42)
```

A.6.2 Unsigned Integers from Literal Bits

A literal unsigned integer can alternatively be created given a string representing its bit representation and an optional bit width.

The following radices are supported:

1. `b` : For representing binary numbers.
2. `o` : For representing octal numbers.

3. `h` : For representing hexadecimal numbers.

If a bit width is not given, the number of bits in the bit representation is directly represented by the string. The following examples create a 8-bit integer representing the number 13.

Example A.64:

```
UInt("b00001101")
UInt("h0D")
```

If the provided bit width is larger than the number of bits required to represent the string's value, then the resulting value is equivalent to the string zero-extended up to the provided bit width. If the provided bit width is smaller than the number of bits represented by the string, then the resulting value is equivalent to the string truncated down to the provided bit width. All truncated bits must be zero.

The following examples create a 7-bit integer representing the number 13.

Example A.65:

```
UInt<7>("b00001101")
UInt<7>("o015")
UInt<7>("hD")
```

A.6.3 Signed Integers

Similar to unsigned integers, a literal signed integer can be created given an integer value and an optional positive bit width. The following example creates a 10-bit signed integer representing the number -42.

Example A.66:

```
SInt<10>(-42)
```

Note that it is an error to supply a bit width that is not large enough to fit the given value using two's complement representation. If the bit width is omitted, then the minimum number of bits necessary to fit the given value will be inferred.

Example A.67:

```
SInt(-42)
```

A.6.4 Signed Integers from Literal Bits

Similar to unsigned integers, a literal signed integer can alternatively be created given a string representing its bit representation and an optional bit width.

The bit representation contains a binary, octal or hex indicator, followed by an optional sign, followed by the value.

If a bit width is not given, the number of bits in the bit representation is the minimal bitwidth to represent the value represented by the string. The following examples create a 8-bit integer representing the number -13.

Example A.68:

```
SInt("b-1101")
SInt("h-d")
```

If the provided bit width is larger than the number of bits represented by the string, then the resulting value is unchanged. It is an error to provide a bit width smaller than the number of bits required to represent the string's value.

A.6.5 References

A reference is simply a name that refers to a previously declared circuit component. It may refer to a module port, node, wire, register, instance, or memory.

The following example connects two ports with a connection statement whose left-hand side expression is a reference expression `in` (referring to the previously declared port `in`) and whose right-hand side expression is the reference expression `out` (referring to the previously declared port `out`).

Example A.69:

```
module MyModule :
   input in: UInt
   output out: UInt
   out <= in
```

A.6.6 Subfields

The subfield expression refers to a subelement of an expression with a bundle type.

The following example connects the `in` port to the `a` subelement of the `out` port.

Example A.70:

```
module MyModule :
   input in: UInt
   output out: {a:UInt, b:UInt}
   out.a <= in
```

A.6.7 Subindices

The subindex expression statically refers, by index, to a subelement of an expression with a vector type. The index must be a non-negative integer and cannot be equal to or exceed the length of the vector it indexes.

The following example connects the `in` port to the fifth subelement of the `out` port.

Example A.71:

```
module MyModule :
   input in: UInt
   output out: UInt[10]
   out[4] <= in
```

A.6.8 Subaccesses

The subaccess expression dynamically refers to a subelement of a vector-typed expression using a calculated index. The index must be an expression with an unsigned integer type.

The following example connects the n'th subelement of the `in` port to the `out` port.

Example A.72:

```
module MyModule :
   input in: UInt[3]
   input n: UInt<2>
   output out: UInt
   out <= in[n]
```

A connection from a subaccess expression can be modeled by conditionally connecting from every subelement in the vector, where the condition holds when the dynamic index is equal to the subelement's static index.

Example A.73:

```
module MyModule :
   input in: UInt[3]
   input n: UInt<2>
   output out: UInt
   when eq(n, UInt(0)) :
      out <= in[0]
   else when eq(n, UInt(1)) :
      out <= in[1]
   else when eq(n, UInt(2)) :
      out <= in[2]
   else :
      out is invalid
```

The following example connects the `in` port to the n'th subelement of the `out` port. All other subelements of the `out` port are connected from the corresponding subelements of the `default` port.

A connection to a subaccess expression can be modeled by conditionally connecting to every subelement in the vector, where the condition holds when the dynamic index is equal to the subelement's static index.

The following example connects the `in` port to the m'th `UInt` subelement of the n'th vector-typed subelement of the `out` port. All other subelements of the `out` port are connected from the corresponding subelements of the `default` port.

Example A.74:

```
module MyModule :
   input in: UInt
   input default: UInt[3]
   input n: UInt<2>
   output out: UInt[3]
   out <= default
   out[n] <= in
```

Example A.75:

```
module MyModule :
   input in: UInt
   input default: UInt[3]
   input n: UInt<2>
   output out: UInt[3]
   out <= default
   when eq(n, UInt(0)) :
      out[0] <= in
   else when eq(n, UInt(1)) :
      out[1] <= in
   else when eq(n, UInt(2)) :
      out[2] <= in
```

Example A.76:

```
module MyModule :
   input in: UInt
   input default: UInt[2][2]
   input n: UInt<1>
   input m: UInt<1>
   output out: UInt[2][2]
   out <= default
   out[n][m] <= in
```

A connection to an expression containing multiple nested subaccess expressions can also be modeled by conditionally connecting to every subelement in the expression. However the condition holds only when all dynamic indices are equal to all of the subelement's static indices.

A.6.9 Multiplexors

A multiplexor outputs one of two input expressions depending on the value of an unsigned single bit selection signal.

Example A.77:

```
module MyModule :
   input in: UInt
   input default: UInt[2][2]
   input n: UInt<1>
   input m: UInt<1>
   output out: UInt[2][2]
   out <= default
   when and(eq(n, UInt(0)), eq(m, UInt(0))) :
      out[0][0] <= in
   else when and(eq(n, UInt(0)), eq(m, UInt(1))) :
      out[0][1] <= in
   else when and(eq(n, UInt(1)), eq(m, UInt(0))) :
      out[1][0] <= in
   else when and(eq(n, UInt(1)), eq(m, UInt(1))) :
      out[1][1] <= in
```

The following example connects to the `c` port the result of selecting between the `a` and `b` ports. The `a` port is selected when the `sel` signal is high, otherwise the `b` port is selected.

Example A.78:

```
module MyModule :
   input a: UInt
   input b: UInt
   input sel: UInt<1>
   output c: UInt
   c <= mux(sel, a, b)
```

A multiplexor expression is legal only if the following holds.

1. The type of the selection signal is a single bit unsigned integer.
2. The types of the two input expressions are equivalent.
3. The types of the two input expressions are passive (see section A.4.4).

For multi-way multiplexors and for non-passive connections, the `when` statement provides the appropriate functionality.

The width, precision and bounds of a ground-typed multiplexor expression is the strictest of its two corresponding input values. For multiplexing aggregate-typed expressions, the resulting values of each leaf subelement is the maximum of its corresponding two input leaf subelement values.

A.6.10 Conditionally Valids

A conditionally valid expression is expressed as an input expression guarded with an unsigned single-bit valid signal. It outputs the input expression when the valid signal is high; otherwise the result is undefined.

The following example connects the `a` port to the `c` port when the `valid` signal is high. Otherwise, the value of the `c` port is undefined.

Example A.79:

```
module MyModule :
   input a: UInt
   input valid: UInt<1>
   output c: UInt
   c <= validif(valid, a)
```

A conditionally valid expression is legal only if the following holds.

1. The type of the valid signal is a single bit unsigned integer.
2. The type of the input expression is passive (see section A.4.4).

Conditional statements can be equivalently expressed as multiplexors and conditionally valid expressions. See section A.5.11 for details.

The width, precision or bound of a conditionally valid expression is the corresponding value of its input expression.

A.6.11 Primitive Operations

Every fundamental operation on ground types is expressed as a FIRRTL primitive operation. In general, each operation takes some number of argument expressions, along with some number of static integer literal parameters.

The general form of a primitive operation is expressed as follows:

Example A.80:

```
op(arg0, arg1, ..., argn, int0, int1, ..., intm)
```

The following examples of primitive operations demonstrate adding two expressions, `a` and `b`, shifting expression `a` left by 3 bits, selecting the fourth bit through and including the seventh bit in the `a` expression, and interpreting the expression `x` as a Clock typed signal.

Example A.81:

```
add(a, b)
shl(a, 3)
bits(a, 7, 4)
asClock(x)
```

Section A.7 will describe the format and semantics of each primitive operation.

A.7 Primitive Operations

The arguments of all primitive operations must be expressions with ground types, while their parameters are static integer literals. Each specific operation can place additional restrictions on the number and types of their arguments and parameters.

Notationally, an argument `e` has their width, bounds, or precision represented as follows: width — w_e, upper bound — u_e, lower bound — l_e, precision — p_e.

A.7.1 Add Operation

The add operation result is the sum of `e1` and `e2` without loss of precision.

Name: add	**Arguments:** (e1,e2)	**Parameters:** ()
Argument Types:	**Result Type:**	**Inference Rules:**
(UInt,UInt)	UInt	$w = max(w_{e1},w_{e2})+1$
(SInt,SInt)	SInt	$w = max(w_{e1},w_{e2})+1$
(Fixed,Fixed)	Fixed	$w = max(w_{e1}-p_{e1},w_{e2}-p_{e2})+max(p_{e1},p_{e2})+1$ $p = max(p_{e1},p_{e2})$
(Interval,Interval)	Interval	$l = l_{e1}+l_{e2}$ $u = u_{e1}+u_{e2}$ $p = max(p_{e1},p_{e2})$

A.7.2 Subtract Operation

The subtract operation result is `e2` subtracted from `e1`, without loss of precision.

Name: sub	**Arguments:** (e1,e2)	**Parameters:** ()
Argument Types:	**Result Type:**	**Inference Rules:**
(UInt,UInt)	UInt	$w = max(w_{e1},w_{e2})+1$
(SInt,SInt)	SInt	$w = max(w_{e1},w_{e2})+1$
(Fixed,Fixed)	Fixed	$w = max(w_{e1}-p_{e1},w_{e2}-p_{e2})+max(p_{e1},p_{e2})+1$ $p = max(p_{e1},p_{e2})$
(Interval,Interval)	Interval	$l = l_{e1}-u_{e2}$ $u = u_{e1}-l_{e2}$ $p = max(p_{e1},p_{e2})$

A.7.3 Multiply Operation

The multiply operation result is the product of `e1` and `e2`, without loss of precision.

Name: mul	Arguments: (e1,e2)	Parameters: ()
Argument Types:	**Result Type:**	**Inference Rules:**
(UInt,UInt)	UInt	$w = w_{e1}+w_{e2}$
(SInt,SInt)	SInt	$w = w_{e1}+w_{e2}$
(Fixed,Fixed)	Fixed	$w = w_{e1}+w_{e2}$ $p = p_{e1}+p_{e2}$
(Interval,Interval)	Interval	$l = \min(l_{e1}*l_{e2}, l_{e1}*u_{e2}, u_{e1}*l_{e2}, u_{e1}*u_{e2})$ $u = \max(l_{e1}*l_{e2}, l_{e1}*u_{e2}, u_{e1}*l_{e2}, u_{e1}*u_{e2})$ $p = p_{e1}+p_{e2}$

A.7.4 Divide Operation

The divide operation divides numerator `num` by denominator `den`, truncating the fractional portion of the result. This is equivalent to rounding the result towards zero.

Name: div	Arguments: (num,den)	Parameters: ()
Argument Types:	**Result Type:**	**Inference Rules:**
(UInt,UInt)	UInt	$w = w_{num}$
(SInt,SInt)	SInt	$w = w_{num}+1$

A.7.5 Remainder Operation

The remainder operation yields the remainder from dividing numerator `num` by denominator `den`, keeping the sign of the numerator. Together with the divide operator, the remainder operator satisfies the relationship: `num = add(mul(den,div(num,den)),rem(num,den))`

Name: rem	Arguments: (num,den)	Parameters: ()
Argument Types:	**Result Type:**	**Inference Rules:**
(UInt,UInt)	UInt	$w = \min(w_{num}, w_{den})$
(SInt,SInt)	SInt	$w = \min(w_{num}, w_{den})$

A.7.6 Comparison Operations

The comparison operations return an unsigned 1 bit signal with value one if `e1` is less than (`lt`), less than or equal to (`leq`), greater than (`gt`), greater than or equal to (`geq`), equal to (`eq`), or not equal to (`neq`) `e2`. The operation returns a value of zero otherwise.

Name: lt,leq,gt,geq,eq,neq	Arguments: (e1,e1)	Parameters: ()
Argument Types:	Result Type:	Inference Rules:
(UInt,UInt)	UInt	w = 1
(SInt,SInt)	UInt	w = 1
(Fixed,Fixed)	UInt	w = 1
(Interval,Interval)	UInt	w = 1

A.7.7 Padding Operations

If e's bit width is smaller than n, then the pad operation zero-extends or sign-extends e up to the given width n. Otherwise, the result is simply e. n must be non-negative.

Name: pad	Arguments: (e)	Parameters: (n)
Argument Types:	Result Type:	Inference Rules:
(UInt)	UInt	$w = max(w_e,n)$
(SInt)	SInt	$w = max(w_e,n)$
(Fixed)	Fixed	$w = max(w_e,n)$
		$p = p_e$

A.7.8 Shift Left Operation

The shift left operation concatenates n zero bits to the least significant end of e. n must be non-negative. The number of fractional bits remains constant.

Name: shl	Arguments: (e)	Parameters: (n)
Argument Types:	Result Type:	Inference Rules:
(UInt)	UInt	$w = w_e+n$
(SInt)	SInt	$w = w_e+n$
(Fixed)	Fixed	$w = w_e+n$
		$p = p_e$
(Interval)	Interval	$l = l_e*2^n$
		$u = u_e*2^n$
		$p = p_e$

A.7.9 Shift Right Operation

The shift right operation truncates the least significant n bits from e. If n is greater than or equal to the bit-width of e, the resulting value will be zero for unsigned types, the sign bit for signed types, and the precision bits for fixed/interval types. n must be non-negative.

Name: shr Argument Types:	Arguments: (e) Result Type:	Parameters: (n) Inference Rules:
(UInt)	UInt	$w = max(w_e - n, 1)$
(SInt)	SInt	$w = max(w_e - n, 1)$
(Fixed)	Fixed	$w = max(max(w_e - n, 1), p_e)$ $p = p_e$
(Interval)	Interval	$l = floor(l_e * 2^{p_e - n}) / 2^{p_e}$ $u = floor(u_e * 2^{p_e - n}) / 2^{p_e}$ $p = p_e$

A.7.10 Dynamic Shift Left Operation

The dynamic shift left operation shifts the bits in `e` `n` places towards the most significant bit. `n` zeroes are shifted in to the least significant bits. For intervals, the range grows accordingly

Name: dshl Argument Types:	Arguments: (e, a) Result Type:	Parameters: () Inference Rules:
(UInt,UInt)	UInt	$w = w_e + 2^{w_a} - 1$
(SInt,UInt)	SInt	$w = w_e + 2^{w_a} - 1$
(Fixed,UInt)	Fixed	$w = w_e + 2^{w_a} - 1$ $p = p_e$
(Interval,UInt)	Interval	$l = min(l_e, l_e * 2^{2^{w_a} - 1})$ $u = max(u_e, u_e * 2^{2^{w_a} - 1})$ $p = p_e$

A.7.11 Dynamic Shift Right Operation

The dynamic shift right operation shifts the bits in `e` `n` places towards the least significant bit. `n` signed or zeroed bits are shifted in to the most significant bits, and the `n` least significant bits are truncated.

Name: dshr Argument Types:	Arguments: (e, a) Result Type:	Parameters: () Inference Rules:
(UInt,UInt)	UInt	$w = w_e$
(SInt,UInt)	SInt	$w = w_e$
(Fixed,UInt)	Fixed	$w = w_e$ $p = p_e$
(Interval,UInt)	Interval	$l = l_e$ $u = u_e$ $p = p_e$

A.7.12 Increase Precision Operation

The increase precision operation increases the precision of `e` by `n`. The value of non-fractional bits remain the same. `n` must be non-negative.

Name: incp Argument Types:	Arguments: (e) Result Type:	Parameters: (n) Inference Rules:
(Fixed)	Fixed	$w = w_e+n$ $p = p_e+n$
(Interval)	Interval	$l = l_e$ $u = u_e$ $p = p_e+n$

A.7.13 Decrease Precision Operation

The decrease precision operation decreases the precision of `e` by `n`. The value of non-fractional bits remain the same. `n` must be non-negative.

Name: decp Argument Types:	Arguments: (e) Result Type:	Parameters: (n) Inference Rules:
(Fixed)	Fixed	$w = w_e-n$ $p = p_e-n$
(Interval)	Interval	$l = \text{floor}(l_e*2^{p_e-n})/2^{n-p_e}$ $u = \text{floor}(u_e*2^{p_e-n})/2^{n-p_e}$ $p = p_e-n$

A.7.14 Set Precision Operation

The set precision operation sets the precision of `e` to `n`. The value of non-fractional bits remain the same. `n` must be non-negative.

Name: setp Argument Types:	Arguments: (e) Result Type:	Parameters: (n) Inference Rules:
(Fixed)	Fixed	$w = w_e-p_e+n$ $p = n$
(Interval)	Interval	$l = \text{floor}(l_e*2^n)/2^n$ $u = \text{floor}(u_e*2^n)/2^n$ $p = n$

A.7.15 Wrap Operation

The wrap operation wraps the first argument's value around the range of the second argument. If the range of the first argument is significantly larger than the range of the second argument (such that more than one wrap around the range could be required), the operator will error as the required hardware implementation is too costly. The precision of the result is the same as the first argument's precision.

Name: wrap	Arguments: (e1,e2)	Parameters: ()
Argument Types:	**Result Type:**	**Inference Rules:**
(Interval)	Interval	$l = l_{e2}$ $u = u_{e2}$ $p = p_{e1}$

A.7.16 Clip Operation

The clip operation clips the first argument's value to remain within the range of the second argument. If the value of the first argument is larger or smaller than the range of the second argument, the returned value is the corresponding upper/lower bound of the second argument (regardless of whether the ranges are disjoint). The precision of the result is the same as the first argument's precision.

Name: clip	Arguments: (e1,e2)	Parameters: ()
Argument Types:	**Result Type:**	**Inference Rules:**
(Interval)	Interval	$l = \min(\max(l_{e1},l_{e2}),u_{e2})$ $u = \max(\min(u_{e1},u_{e2}),l_{e2})$ $p = p_{e1}$

A.7.17 Squeeze Operation

The squeeze operation tries to fit the first argument's value into the smallest range given the bounds of both arguments. If the value of the first argument outside of this range, the value of the result is undefined. If defined behavior is needed, use the `clip` or `wrap` operator.

Name: squz	Arguments: (e1,e2)	Parameters: ()
Argument Types:	**Result Type:**	**Inference Rules:**
(Interval)	Interval	$l = \min(\max(l_{e1},l_{e2}),u_{e2})$ $u = \max(\min(u_{e1},u_{e2}),l_{e2})$ $p = p_{e1}$

A.7.18 Arithmetic Convert to Signed Operation

The result of the arithmetic convert to signed operation is a signed integer representing the same numerical value as `e`.

Name: `cvt`	Arguments: `(e)`	Parameters: `()`
Argument Types:	**Result Type:**	**Inference Rules:**
`(UInt)`	`SInt`	$w = w_e+1$
`(SInt)`	`SInt`	$w = w_e$

A.7.19 Negate Operation

The result of the negate operation is a signed integer representing the negated numerical value of `e`.

Name: `neg`	Arguments: `(e)`	Parameters: `()`
Argument Types:	**Result Type:**	**Inference Rules:**
`(UInt)`	`SInt`	$w = w_e+1$
`(SInt)`	`SInt`	$w = w_e+1$

A.7.20 Bitwise Complement Operation

The bitwise complement operation performs a logical not on each bit in `e`.

Name: `not`	Arguments: `(e)`	Parameters: `()`
Argument Types:	**Result Type:**	**Inference Rules:**
`(UInt)`	`UInt`	$w = w_e$
`(SInt)`	`UInt`	$w = w_e$

A.7.21 Binary Bitwise Operations

The above bitwise operations perform a bitwise and, or, or exclusive or on `e1` and `e2`. The result has the same width as its widest argument, and any narrower arguments are automatically zero-extended or sign-extended to match the width of the result before performing the operation.

Name: and,or,xor	Arguments: (e1,e2)	Parameters: ()
Argument Types:	Result Type:	Inference Rules:
(UInt,UInt)	UInt	w = max(w_{e1},w_{e2})
(UInt,SInt)	UInt	w = max(w_{e1},w_{e2})
(SInt,UInt)	UInt	w = max(w_{e1},w_{e2})
(SInt,SInt)	UInt	w = max(w_{e1},w_{e2})

A.7.22 Bitwise Reduction Operations

The bitwise reduction operations correspond to a bitwise and, or, and exclusive or operation, reduced over every bit in `e`.

Name: andr,orr,xorr	Arguments: (e)	Parameters: ()
Argument Types:	Result Type:	Inference Rules:
(UInt)	UInt	w = 1
(SInt)	UInt	w = 1

A.7.23 Concatenate Operation

The result of the concatenate operation is the bits of `e1` concatenated to the most significant end of the bits of `e2`. Note that `Any*` refers to `UInt`, `SInt`, `Fixed`, or `Interval` types.

Name: cat	Arguments: (e1,e2)	Parameters: ()
Argument Types:	Result Type:	Inference Rules:
(Any*, Any*)	UInt	w = w_{e1}+w_{e2}

A.7.24 Bit Extraction Operation

The result of the bit extraction operation are the bits of `e` between `lo` (inclusive) and `hi` (inclusive). `hi` must be greater than or equal to `lo`. Both `hi` and `lo` must be non-negative and strictly less than the bit width of `e`.

Name: bits	Arguments: (e)	Parameters: (hi,lo)
Argument Types:	Result Type:	Inference Rules:
(UInt)	UInt	w = hi-lo+1
(SInt)	UInt	w = hi-lo+1
(Fixed)	UInt	w = hi-lo+1
(Interval)	UInt	w = hi-lo+1

A.7.25 Head

The result of the head operation are the `n` most significant bits of `e`. `n` must be positive and less than or equal to the bit width of `e`.

Name: head	Arguments: (e)	Parameters: (n)
Argument Types:	**Result Type:**	**Inference Rules:**
(UInt)	UInt	w = n
(SInt)	UInt	w = n

A.7.26 Tail

The tail operation truncates the `n` most significant bits from `e`. `n` must be non-negative and strictly less than the bit width of `e`.

Name: tail	Arguments: (e)	Parameters: (n)
Argument Types:	**Result Type:**	**Inference Rules:**
(UInt)	UInt	$w = w_e - n$
(SInt)	UInt	$w = w_e - n$

A.7.27 Interpret As UInt

The interpret as UInt operation reinterprets `e`'s bits as an unsigned integer.

Name: asUInt	Arguments: (e)	Parameters: ()
Argument Types:	**Result Type:**	**Inference Rules:**
(UInt)	UInt	$w = w_e$
(SInt)	UInt	$w = w_e$
(Clock)	UInt	w = 1
(Analog)	UInt	$w = w_e$
(Fixed)	UInt	$w = w_e$
(Interval)	UInt	$w = w_e$

A.7.28 Interpret As SInt

The interpret as SInt operation reinterprets `e`'s bits as a signed integer according to two's complement representation.

Name: asSInt	Arguments: (e)	Parameters: ()
Argument Types:	Result Type:	Inference Rules:
(UInt)	SInt	$w = w_e$
(SInt)	SInt	$w = w_e$
(Clock)	SInt	$w = 1$
(Analog)	SInt	$w = w_e$
(Fixed)	SInt	$w = w_e$
(Interval)	SInt	$w = w_e$

A.7.29 Interpret as Clock

The result of the interpret as clock operation is the Clock typed signal obtained from interpreting a single bit integer as a clock signal.

Name: asClock	Arguments: (e)	Parameters: ()
Argument Types:	Result Type:	Inference Rules:
(UInt)	Clock	n/a
(SInt)	Clock	n/a
(Clock)	Clock	n/a
(Analog)	Clock	n/a
(Fixed)	Clock	n/a
(Interval)	Clock	n/a

A.7.30 Interpret as Analog

The result of the interpret as analog operation is the Analog typed signal obtained from interpreting e's bits as an analog signal.

Name: asAnalog	Arguments: (e)	Parameters: ()
Argument Types:	Result Type:	Inference Rules:
(UInt)	Analog	w_e
(SInt)	Analog	w_e
(Clock)	Analog	1
(Analog)	Analog	w_e
(Fixed)	Analog	w_e
(Interval)	Analog	w_e

A.7.31 Interpret as Fixed

The result of the interpret as fixed operation is the Fixed typed signal obtained from interpreting `e`'s bits as a fixed point signal, with `p`'s precision.

Name: `asFixed` **Argument Types:**	**Arguments:** `(e)` **Result Type:**	**Parameters:** `(p)` **Inference Rules:**
`(UInt)`	`Fixed`	$w = w_e$ $p = p$
`(SInt)`	`Fixed`	$w = w_e$ $p = p$
`(Clock)`	`Fixed`	$w = 1$ $p = p$
`(Analog)`	`Fixed`	$w = w_e$ $p = p$
`(Fixed)`	`Fixed`	$w = w_e$ $p = p$
`(Interval)`	`Fixed`	$w = w_e$ $p = p$

A.7.32 Interpret as Interval

The result of the interpret as interval operation is the Interval typed signal obtained from interpreting `e`'s bits as an interval signal with precision of `p`. The provided lower and upper bounds of `l` and `u` are the scaled value of the bound; the value of the returned Interval's bounds are $l/2^p$ and $u/2^p$. This is done to avoid adding another parameter list of doubles to the primop ast.

Name: asInterval	Arguments: (e)	Parameters: (l,u,p)
Argument Types:	Result Type:	Inference Rules:
(UInt)	Interval	$l = l/2^p$ $u = u/2^p$ $p = p$
(SInt)	Interval	$l = l/2^p$ $u = u/2^p$ $p = p$
(Clock)	Interval	$l = l/2^p$ $u = u/2^p$ $p = p$
(Analog)	Interval	$l = l/2^p$ $u = u/2^p$ $p = p$
(Fixed)	Interval	$l = l/2^p$ $u = u/2^p$ $p = p$
(Interval)	Interval	$l = l/2^p$ $u = u/2^p$ $p = p$

A.8 Flows

An expression's flow partially determines the legality of connecting to and from the expression. Every expression is classified as either *source*, *sink*, or *duplex*. For details on connection rules refer back to sections A.5.1 and A.5.2.

The flow of a reference to a declared circuit component depends on the kind of circuit component. A reference to an input port, an instance, a memory, or a node is source. A reference to an output port is sink. A reference to a wire or register is duplex.

The flow of a subindex or subaccess expression is the flow of the vector-typed expression it indexes or accesses.

The flow of a subfield expression depends upon the orientation of the field. If the field is not flipped, its flow is the same flow as the bundle-typed expression it selects its field from. If the field is flipped, then its flow is the reverse of the flow of the bundle-typed expression it selects its field from. The reverse of source is sink, and vice-versa. The reverse of duplex remains duplex.

The flow of all other expressions are source.

A.9 Namespaces

All modules in a circuit exist in the same module namespace, and thus must all have a unique name. Each module has an identifier namespace containing the names of all port and circuit component declarations. Thus, all declarations within a module must have unique names. Within a bundle type declaration, all field names must be unique. Within a memory declaration, all port names must be unique.

During the lowering transformation, all circuit component declarations with aggregate types are rewritten as a group of component declarations, each with a ground type. After the lowering transformation, the names of the lowered circuit components are usually determined by the name expansion algorithm. A name-uniquification transform occurs during type and width inference to ensure that the lowered version of the names remain unique.

Bibliography

[1] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016.

[2] Joshua Auerbach et al. "Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 89–108. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869469.

[3] Jonathan Bachrach et al. "Chisel: Constructing Hardware in a Scala Embedded Language". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. New York, NY, USA: ACM, 2012, pp. 1216–1225. ISBN: 978-1-4503-1199-1. DOI: 10.1145/2228360.2228584.

[4] Jonathan Balkind et al. "OpenPiton: An Open Source Manycore Research Framework". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 217–232. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872414.

[5] P. Bellows and B. Hutchings. "JHDL-an HDL for reconfigurable systems". In: *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines*. Apr. 1998, pp. 175–184. DOI: 10.1109/FPGA.1998.707895.

[6] J. P. Bergmann and M. A. Horowitz. "Vex-a CAD toolbox". In: *Proceedings 1999 Design Automation Conference*. 1999, pp. 523–528. DOI: 10.1109/DAC.1999.781371.

[7] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. "D3 Data-Driven Documents". In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (Dec. 2011), pp. 2301–2309. ISSN: 1077-2626. DOI: 10.1109/TVCG.2011.185.

[8] Andrew Canis et al. "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36. ISBN: 978-1-4503-0554-9. DOI: 10.1145/1950413.1950423.

[9] Christopher Celio, David A. Patterson, and Krste Asanović. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Tech. rep. UCB/EECS-2015-167. EECS Department, University of California, Berkeley, June 2015.

[10] Christopher Torng. *alloy-asic - The Modular VLSI Build System*. https://github.com/cornell-brg/alloy-asic.

[11] Philippe Coussy et al. "GAUT: A High-Level Synthesis Tool for DSP Applications". en. In: *High-Level Synthesis*. Ed. by Philippe Coussy and Adam Morawiec. Springer Netherlands, 2008, pp. 147–169. ISBN: 978-1-4020-8587-1 978-1-4020-8588-8.

[12] Mark R. Cramer and Jan Van Leeuwen. *Wire Routing is NP-Complete*. Tech. rep. RUU-CS-82-4. Department of Computer Science, University of Utrecht, Feb. 1982. URL: http://web.archive.org/web/20180504191319/https://dspace.library.uu.nl/bitstream/handle/1874/16302/kramer_82_wire_routing.pdf.

[13] T. S. Czajkowski et al. "From opencl to high-performance hardware on FPGAS". In: *22nd International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2012, pp. 531–534. DOI: 10.1109/FPL.2012.6339272.

[14] Palmer Dabbelt. "PLSI: A Portable VLSI Flow". MA thesis. EECS Department, University of California, Berkeley, 2017.

[15] Jan Decaluwe. "MyHDL: A Python-based Hardware Description Language". In: *Linux J.* 2004.127 (Nov. 2004), p. 5. ISSN: 1075-3583.

[16] Edsger W Dijkstra. "Go to statement considered harmful". In: *Communications of the ACM* 11.3 (1968), pp. 147–148.

[17] EEMBC. *CoreMark: an EEMBC Benchmark*. URL: https://www.eembc.org/coremark/.

[18] N. George et al. "Hardware system synthesis from Domain-Specific Languages". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927454.

[19] Richard Goldman et al. "Synopsys' open educational design kit: Capabilities, deployment and future". In: *IEEE International Conference on Microelectronic Systems Education, MSE '09, San Francisco, CA, USA, July 25-27, 2009*. IEEE Computer Society, 2009, pp. 20–24. ISBN: 978-1-4244-4407-6. DOI: 10.1109/MSE.2009.5270840.

[20] Frank Hannig et al. "PARO: Synthesis of Hardware Accelerators for Multi-dimensional Dataflow-Intensive Applications". en. In: *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, Berlin, Heidelberg, Mar. 2008, pp. 287–293. DOI: 10.1007/978-3-540-78610-8_30.

[21] Amir Hormati et al. "Optimus: Efficient Realization of Streaming Applications on FPGAs". In: *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '08. New York, NY, USA: ACM, 2008, pp. 41–50. ISBN: 978-1-60558-469-0. DOI: 10.1145/1450095.1450105.

[22] "IEEE Standard for Standard SystemC Language Reference Manual". In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (Jan. 2012), pp. 1–638. DOI: 10.1109/IEEESTD.2012.6134619.

[23] Adam Izraelevitz et al. "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations". In: *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press. 2017, pp. 209–216.

[24] Holger Keding et al. "FRIDGE: a fixed-point design and simulation environment". In: *Proceedings of the conference on Design, automation and test in Europe*. IEEE Computer Society. 1998, pp. 429–435.

[25] Gregor Kiczales et al. "An overview of AspectJ". In: *European Conference on Object-Oriented Programming*. Springer. 2001, pp. 327–354.

[26] Gregor Kiczales et al. "Aspect-oriented programming". In: *European conference on object-oriented programming*. Springer. 1997, pp. 220–242.

[27] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: *CGO*. San Jose, CA, USA, Mar. 2004, pp. 75–88.

[28] Dong-U Lee et al. "MiniBit: bit-width optimization via affine arithmetic". In: *Proceedings of the 42nd annual Design Automation Conference*. ACM. 2005, pp. 837–840.

[29] Yanbing Li and M. Leeser. "HML, a novel hardware description language and its translation to VHDL". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 8.1 (Feb. 2000), pp. 1–8. ISSN: 1063-8210. DOI: 10.1109/92.820756.

[30] D. Lockhart, G. Zibrat, and C. Batten. "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2014, pp. 280–292. DOI: 10.1109/MICRO.2014.50.

[31] Mentor. *Catapult and PowerPro: High-Level Synthesis and RTL Low-Power*. URL: https://www.mentor.com/hls-lp/ (visited on 04/12/2017).

[32] Peter Milder et al. "Computer Generation of Hardware for Linear Digital Signal Processing Transforms". In: *ACM Trans. Des. Autom. Electron. Syst.* 17.2 (Apr. 2012), 15:1–15:33. ISSN: 1084-4309. DOI: 10.1145/2159542.2159547.

[33] R. Nikhil. "Bluespec System Verilog: efficient, correct RTL from high level specifications". In: *Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings*. June 2004, pp. 69–70. DOI: 10.1109/MEMCOD.2004.1459818.

[34] Martin Odersky et al. *An overview of the Scala programming language*. Tech. rep. 2004.

[35] OpenPiton. *OpenPiton-ZC706*. https://github.com/s117/OpenPiton-ZC706/tree/zc706_port/piton/tools/synopsys.

[36] M. Owaida et al. "Synthesis of Platform Architectures from OpenCL Programs". In: *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. May 2011, pp. 186–193. DOI: 10.1109/FCCM.2011.19.

[37] A. Papakonstantinou et al. "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs". In: *2009 IEEE 7th Symposium on Application Specific Processors*. July 2009, pp. 35–42. DOI: 10.1109/SASP.2009.5226333.

[38] Shimpei Sato and Kenji Kise. "ArchHDL: A Novel Hardware RTL Design Environment in C++". en. In: *Applied Reconfigurable Computing*. Springer, Cham, Apr. 2015, pp. 53–64. DOI: 10.1007/978-3-319-16214-0_5.

[39] O. Shacham et al. "Avoiding game over: Bringing design to the next level". In: *DAC Design Automation Conference 2012*. June 2012. DOI: 10.1145/2228360.2228472.

[40] K. Shahookar and P. Mazumder. "VLSI cell placement techniques". In: *ACM Computing Surveys* 23.2 (June 1991), pp. 143–220. ISSN: 03600300. DOI: 10.1145/103724.103725. URL: http://portal.acm.org/citation.cfm?doid=103724.103725 (visited on 05/04/2018).

[41] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. "Bidwidth analysis with application to silicon compilation". In: *ACM SIGPLAN Notices*. Vol. 35. 5. ACM. 2000, pp. 108–120.

[42] Shinya Takamaeda-Yamazaki. "Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL". en. In: *Applied Reconfigurable Computing*. Springer, Apr. 2015, pp. 451–460. DOI: 10.1007/978-3-319-16214-0_42.

[43] Shinya Takamaeda-Yamazaki. *Veriloggen: A library for constructing a Verilog HDL source code in Python*. URL: https://github.com/PyHDI/veriloggen.

[44] Justin L. Tripp, Maya B. Gokhale, and Kristopher D. Peterson. "Trident: From High-Level Language to Hardware Circuitry". In: *Computer* 40.3 (2007), pp. 28–37. ISSN: 0018-9162. DOI: http://dx.doi.org/10.1109/MC.2007.107.

[45] Angie Wang et al. "ACED: A hardware library for generating DSP systems". In: *Proceedings of the 55th Annual Design Automation Conference*. ACM. 2018, p. 61.

[46] Cheng C Wang et al. "An automated fixed-point optimization tool in MATLAB XSG/SynDSP environment". In: *ISRN Signal Processing* 2011 (2011).

[47] Edward Wang et al. "Hammer: Enabling Reusable Physical Design". In: ().

[48] Clifford Wolf. *Yosys Open SYnthesis Suite*. URL: http://www.clifford.at/yosys/.

[49] Xilinx. *ChipScope Pro Debugging Overview*. URL: https://www.xilinx.com/itp/xilinx10/isehelp/ise_c_process_analyze_design_using_chipscope.htm (visited on 04/13/2017).

[50] Xilinx. *Vivado High-Level Synthesis*. URL: http://www.xilinx.com/products/silicon-devices.html (visited on 04/12/2017).