Feature-Oriented Design for Hardware Generation*
Dissertation Proposal

Justin Deters

Department of Computer Science and Engineering
James McKelvey School of Engineering
Washington University in St. Louis
St. Louis, Missouri USA

December 15, 2020

*Supported under NSF CISE award CNS-1763503 Performant Architecturally Diverse Systems via Aspect-oriented Programming

Contents

1 Introduction

Hardware designers frequently build monolithic designs with different hardware features entangled with each other. Because of this, subsetting or extending features becomes difficult. For the same reason, evaluating and integrating features between chip designs is arduous. Traditional hardware description languages force hardware designers to explicitly lay out chips in their designs. Hardware generation languages, such as Chisel, were introduced to bring more flexibility to hardware design. These frequently come as domain-specific languages embedded in a general-purpose programming language. Despite the increased flexibility, designers use these languages just like traditional HDLs, resulting in the same monolithic and entangled designs. We propose analyzing the benefits of applying the software design technique of Feature-Oriented Design to these hardware generation languages. This technique isolates software features and allows them to be optionally included or exchanged with others. We believe that this approach will alleviate the issues with hardware design discussed above.

2 Background

2.1 Prior Work

2.1.1 Feature-Oriented Programming

When customers have a single set of features that they wish their software to contain, widely used software paradigms such as procedural or object oriented programming are commonly used to implement software. However, there are some situations where a single set of features does not suit every end-user. Software designers could take a "kitchen sink" approach where every possible feature is implemented in the final software design. This can lead to code bloat and can make human understanding of the codebase difficult. One programming paradigm that attempts to solve this problem is **Feature-Oriented Programming (FOP)**. In FOP, features are implemented separately and then composed together to create a design for the end-user.

In [13, 14] Hunleth et al. demonstrate this approach in the context of middleware. Their approach uses **Aspect-Oriented Programming (AOP)** to compose features. Generally, AOP [17] is used to implement **crosscutting concerns**. These are concerns that do not fit cleanly into a module, such as a function or class, but instead span over multiple modules in a program. **Aspects** capture these crosscutting concerns. Aspects, at minimum, have two components: **pointcuts** and **advice.** A pointcut represents all the points (called **join points**) in the codebase where the advice will be applied. Advice defines what code will be applied and how it will be applied. The three types of advice are **before**, **after**, and **around** which are applied before a join point, after a join point, or replaces a join point, respectively.

Hunleth et al. created a single base implementation of the middleware with functionality common to all features. They then captured different features in aspects. This allowed them to design an architecture in which the features the end-user desired could be woven into the middleware. By generating different feature sets, they were able to reduce the

code complexity and the footprint of the middleware. Similarly, we wish to apply this programming paradigm to achieve reductions in complexity and footprint, but in hardware instead of software.

2.1.2 Hardware Description versus Hardware Construction Languages

While it is not impossible to use FOP with **hardware description languages** (HDLs), their limited features make the application of FOP difficult. Fortunately, a class of hardware languages, **hardware construction languages**, have been developed that attempt to do for hardware languages what high-level programming languages have done for software development. These make the application of FOP to hardware more attainable.

Historically, chip designers have used hardware description languages, such as VHDL and Verilog, to define the layout of their chips. Unlike their programming language counterparts, VHDL and Verilog lack high-level language features that enable modular, parameterizable, and reusable code. To overcome this, a class of languages called **hardware construction languages** have developed. These either borrow syntax from high-level languages or are a domain-specific language embedded within a general-purpose programming language. Rather than these directly specifying a circuit, they instead generate a circuit as an output of their execution. Examples of these languages include JHDL [4], Clash [2], MyHDL [8], PyMTL [20], Hardware ML [20], and Chisel [3].

Chisel An example of a popular hardware construction language is **Chisel** [3]. Chisel is a domain-specific language embedded in Scala [23] for constructing hardware. Unlike HDLs, Chisel code is not directly synthesizable. Instead, execution of a Chisel program results in generating a hardware design as output. Thus, a Chisel program can be thought of as collection of code that *manages* the generation of a hardware design. This allows hardware designers to create more generalized designs that can be parametrized and reused.

Listing 1[1] demonstrates powerful hardware construction features that Chisel provides. To begin, both `Adder` and the subcomponent `OneBitAdder` are defined as normal Scala classes. Thus, as seen on line 10, a programmer can instantiate as many of these classes as they would like. Furthermore, because both of these modules are plain Scala classes, they can be parameterized. On line 1, the number of bits is parameterized. This adder can be an *arbitrary* number of bits long. Because Chisel generates synthesizable hardware designs, we can use hardware tools to get space and power estimations of different parameterizations of designs. Thus, we can directly compare different hardware configurations to each other.

Unlike traditional HDLs, hardware construction languages like Chisel can utilize the rich ecosystem of software design patterns. This flexibility allows designers to create complex hardware generators that can then be packaged as libraries and reused by other designers. This is especially easy for Chisel since in reality it is just Scala and thus comes with all the same packaging features. Ideally, this will lead to hardware designers creating libraries for frequently needed hardware constructs and enable more open hardware design. The flexibility that Chisel has can make applying FOP to hardware much easier.

[1] All listings from here forward will appear in the Appendix.

2.2 Current Problems with Hardware Design

2.2.1 Monolithic hardware designs are difficult to reuse.

Consider a company that needs a new microcontroller. Given their specific use cases, they may not need things such as floating point units or other types of instructions. Any licensed design will likely be monolithic with limited modularity because the structure of traditional HDLs forces designers to explicitly lay out hardware. Despite the introduction of hardware construction languages which do not have this same requirement, this practice remains common. This results in the continuation of monolithic designs without the ability to remove hardware that supports different parts of the ISA without difficulty. This can add significant development time and effort for their product.

In a similar situation, a company may need to save space or power in their chip design, but still require the full ISA. Here microarchitectural features, such as data and instruction prefetchers, cache, and branch predictors, might be targeted for removal or changes. However, the chip designers will encounter the same situation as the previous scenario. Microarchitectural features are also explicitly laid out and are entangled into the overall design. In addition, when extending microarchitecture features, designers may spend large amounts of time refactoring due to the entangled and explicit nature of HDLs.

2.2.2 Hardware designs are difficult to integrate into each other.

Consider another situation, in which a research group has created several designs for accelerators. They wish to have other research groups use their designs in their own research. Currently, if one wishes to use an accelerator, co-processor, or other existing hardware design, they must have the physical hardware it was designed for. Hardware designs are usually tied to a specific hardware interface provided by the device. Often, these interfaces are not modular, and this limits the exchange of accelerator designs between researchers. Conversely, if a chip designer has several accelerators they wish to integrate into their chip, they will run into the same issue. Changing each of the interfaces to match the destination interface can be tedious. The interfaces in general can be limiting in both how the accelerator must communicate with the rest of the data path and how many accelerators can be integrated into the chip. Furthermore, these interfaces themselves add overhead, space, and power consumption to a processor design.

2.2.3 Comparing microarchitecture designs between chips is difficult.

Monolithic chip designs increase the difficulties of comparing two or more feature implementations. Say a research group has created a design for a new branch predictor and wants to directly compare it to other branch predictor designs. Because microarchitectural features are frequently entangled with each other, replacing one implementation with another may require substantial refactoring. This introduces more variability into the design of the chip, making direct comparisons between different implementations of microarchitectural features harder.

2.2.4 Performance tuning a processor requires substantial effort.

Naturally, if designers wish to compare implementations of features, they will also wish to compare different *combinations* of implementations in order to determine which arrangement will work best for their purposes. This adds yet another level of complexity above the previous problem. Performance tuning can take a lot of refactoring to integrate multiple microarchitectural designs for comparison. Thus doing large amounts of design space exploration and performance tuning can add significant time and effort to the design process of a chip.

2.2.5 Monolithic designs obscure negative feature interaction.

The two well-known microarchitecture security exploits, Spectre [18] and Meltdown [19], are the direct result of the difficulty of feature interaction analysis in hardware design. Independently, the two features of out-of-order execution and caching seem innocuous. However, when *combined* they created an exploit where sensitive data could be moved into cache by speculative execution and then read later by legal instructions. If features could be analyzed for negative interactions, then these exploits could possibly be avoided in the future.

3 Proposed Research Questions

Given the state of affairs in subsection 2.2 we propose applying feature-oriented design to hardware generation using an aspect-oriented approach to alleviate the problems described above. In order to determine if this approach is feasible, we need to answer several questions.

3.1 To what extent does FOP save on development time?

If FOP is a viable alternative to monolithic hardware design techniques, then we need to show that it actually reduces the development burden on hardware designers. Given two or more different implementations of features, does FOP reduce the time to switch them out? If developers want to remove a feature altogether, is FOP faster than doing it by hand? When navigating design spaces, can FOP iterate through designs faster than manual reconfiguration? When a new hardware design needs to be integrated into an existing design, can FOP make that integration faster?

3.2 How much better can we utilize hardware resources by conditionally weaving hardware features?

We want to show that designers can use FOP to utilize hardware resources better, especially in terms of circuit area, power consumption, and latency. How much space and power can be reclaimed by removing the hardware that supports unnecessary parts of the ISA? By how much can the critical path of a circuit be reduced? Similarly, when certain microarchitecture features are unneeded, how much space and power can be saved by stripping them from the design? Instead of using predefined hardware interfaces, can space and power be reduced by just weaving the new design into the chip? Finally, if FOP allows designers to directly

compare different features or sets of features together, can space and power be saved overall by choosing the more optimal design configurations?

3.3 To what extent does FOP introduce penalties in the hardware design toolchain?

Since FOP is not already accounted for in any hardware design toolchains, we need to investigate which penalties exist and to what magnitude. For instance, FOP moves away from a monolithic design so the hardware elaboration time might increase. Furthermore, the hardware will most likely end up being laid out in code much differently. To what extent does this affect the optimization process? Does place and route for hardware generated with FOP take any longer than monolithic designs? Ideally, FOP has the same or fewer penalties in the toolchain as monolithic designs. If FOP does have penalties, understanding what they are and their extent can help determine where FOP is most useful.

4 Methodology

4.1 RISC-V

In their 2018 Turing Award, Lecture John Hennessy and David Patterson outlined why they believed that their open ISA would pave the way for community development of hardware. They envisioned RISC-V as the basis for open development of hardware. Rather than having hardware designers work away in enclaves, designers could publish, compare, audit, and stress-test designs as a community. The openness and extendability of RISC-V would also allow communities of hardware designers to agree on additions to the ISA for specific domains, giving a common language for these domains to operate on hardware. Our goal for FOP in hardware is to increase the modularity and shareability of hardware designs, so RISC-V is a natural choice for a target hardware architecture.

4.2 Programming Infrastructure

4.2.1 Chisel

We have chosen Chisel as the primary programming infrastructure for our experiments. Earlier we discussed how Chisel can parameterize circuit designs. We demonstrate the parameterization in Diagrams 1 and 2. These show what Chisel generates when our parameterized adder design is configured as a four-bit adder. Recall that Chisel is embedded within Scala. This enables a level of expressiveness far and above that of a traditional HDL and allows us to utilize existing libraries and tools within the Scala community.

We can demonstrate this expressiveness with a second adder example. This time, we will examine a carry look-ahead adder. The circuitry needed to build the carry component of this adder is much more complex and can become unwieldy given a high number of bits. However, because we can *algorithmically* generate hardware in Chisel, we can abstract out the necessary logic to generate the proper circuit. Listing 2 demonstrates such an algorithm. By leveraging Scala's functional programming features, we can generate a carry look-ahead

component *of an arbitrary length.* Diagram 4 shows the carry component which is generated for a four-bit adder. This expressiveness directly assists us in exploring our research questions and eases the adoption of FOP.

Chisel also offers many other perks. A growing community of hardware designers have adopted Chisel, giving us a rich set of hardware components to investigate. Furthermore, Chisel offers a very robust toolchain that is open to this community. The FIRRTL[2] compiler offers robust type checking, optimization of circuits, and can directly emit Verilog or build a emulator for a circuit. As such, we have been able to synthesize both of our Chisel adders on an Xilinx Artix-7 FPGA.

Chisel Aspects Currently, Chisel does contain a library for aspect-oriented programming. The primary work on this library was completed by Adam Izraelevits for his PhD thesis [15]. Despite basing this library on aspect-oriented programming, Izraelevits specifically warns programmers against using it to implement primary functionality. Rather he proposes that the library be used for inserting hardware design collateral, such as floor plans, into Chisel designs. To our knowledge, there is no functional reason for this warning, but it is instead based on the author's own opinions on aspect-oriented programming.

In the preparation for this proposal, we evaluated this library and explicitly ignored the author's warning. The two adder designs we have explored in this proposal are actually the same design, but an aspect controls which version of the carry is generated. Listing 3 shows two aspects we created to insert the circuitry needed for each type of carry. Importantly, both of these aspects are agnostic to the number of bits in the adder. Both of these use the InjectingAspect interface provided by Chisel. The first parameter is an anonymous function that provides a list of modules that the aspect will be applied to (the joinpoint). The second parameter is an anonymous function with the actual code that will be applied at the end of the module. Colla-Gen then produces a series of FIRRTL annotations [15] which are passed to the FIRRTL compiler with the rest of the circuit design.

Limitations of Chisel Aspects In our preliminary investigation of Chisel's aspect library, we found its abilities to be too limiting for our experiments. First, Chisel Aspects only provide one joinpoint, the module. This means that all RLT code inserted into the design ends up at the *bottom* of the module. This coarse granularity does not afford us much flexibility. Furthermore, around advice can only be faked by leveraging Chisel's **last connection syntax**, where, during elaboration, the last connection in depth first traversal of the code will be the generated connection.

Second, Chisel Aspects cannot influence the IO of a module. This means that every feature a designer would want to include must be accounted for in the IO of the module. This creates inflexible cluttered interfaces for IO that could otherwise be simplified to conserve space and power.

Third, Chisel Aspects are applied after circuit elaboration during compilation in the FIRRTL compiler. This means that the Scala compiler is completely unaware of any new code that would be inserted into the design. Thus any aspect that would create legal Scala code if applied prior to compilation creates an error. By extension, these aspects cannot

[2]FIRRTL is the intermediate representation used in the Chisel tool chain.

influence each other. If a joinpoint is introduced in one aspect, the corresponding aspect for that joinpoint will not be applied in a later phase. For these reasons, we looked elsewhere for a way to apply aspects.

4.2.2 Scalameta

Fortunately, Scala includes a powerful meta-programming library, Scalameta [26]. The Scalameta library can parse Scala code into abstract syntax trees (ASTs) that can be manipulated by removing, changing, or generating new subtrees. We have used Scalameta to create a rudimentary aspect compiler that overcomes many of the limitations of Chisel Aspects. The transformations are applied to the source code *prior to the Scala compilation phase*. The Scala compiler is aware of all new features that have been woven into the code and can perform syntax and type checking on them. Furthermore, because the code still follows the same toolchain, we keep the Chisel type checking and the optimizations that occur in the FIRRTL compiler. But now, the smallest joinpoint granularity is on the scale of the individual tokens. This offers us far more flexibility than Chisel Aspects.

4.2.3 Rocket Chip

Rocket Chip [1] is a system-on-a-chip generator written in Chisel that implements the RISC-V ISA. Rocket Chip is still under active development and is used as the chip environment for many different RISC-V research projects. Rocket Chip not only provides the ability to generate a complete chip, but also provides a robust test suite, easing the development of new features for the chip. The designers have included parameterization within Rocket Chip, but the overall chip design still suffers from the problems that we outlined in subsection 2.2.

We have chosen Rocket Chip over other RISC-V designs for two reasons. First, the robust development community and its use in research projects gives us plenty of examples of hardware features to investigate and compare. Second, Rocket Chip is written in Chisel, which makes the application of FOP much easier than a RISC-V design written in an HDL.

5 Experiments

5.1 Feature-Oriented Performance Counters for Rocket Chip

Chip designers commonly include performance counters in their designs to give useful information to end-users about the behavior of their chips. To collect the necessary information across the chip, designers must create circuitry that crosscuts multiple modules. Which performance counters are available to end-users may differ greatly between chip designs. For instance, a server-grade multi-core processor may provide counters for cache misses, stall cycles, executed instructions, and others for each core. However, an embedded processor may not include any performance counters at all.

As a first foray into this technique, we propose developing a feature-oriented performance counter system for Rocket Chip [1]. Given the crosscutting nature of performance counters and their variability over designs, they are perfect candidates for feature-oriented design. Currently, if a user wishes to implement a new performance counter in Rocket Chip, they

would need to manually route all signals through multiple modules. The current hardware performance counters are hard coded into the design with no infrastructure for other designers to add their own counter to the list or remove the ones they do not need.

Refactoring the existing performance counter system in Rocket Chip into a feature-oriented design has several advantages. First, we will be able demonstrate how FOP eases the development of new performance counters and the selection of counters in the final design (Question 3.1). Second, as Rocket Chip is a fully synthesizable design, we can easily compare space and power requirements of the two different implementations with different sets of performance counters (Question 3.2). Along the same lines, Rocket Chip utilizes the full Chisel toolchain giving us the ability to determine if any performance penalties from FOP exist in the toolchain (Question 3.3).

Apart from a direct comparison to the current performance counter system, we would also like to contribute new functionality while demonstrating the advantages of FOP (Question 3.1); namely, automatically generating different circuitry based on the number of different performance counters the designer would like to have in the design. Currently, the RISC-V specification [25] allows for 28 user-configurable performance counters[3]. If a chip designer has 28 or fewer events they wish to include, then a simple hardware structure to associate the event to a counter is sufficient. In the event that more than 28 events are included, we wish to weave in a different hardware structure that will handle the swapping of events for the end-user. Using FOP, these changes could be applied automatically without any designer intervention.

5.2 Accelerator and Co-Processor Interfaces

In Section 2.2 we discussed how hard-coded hardware interfaces present a challenge to integrating different designs. More concretely, Rocket Chip provides two separate interfaces for connecting accelerators to the core generator: the Rocket Custom Coprocessor (RoCC) and the TileLink-Attached Accelerator. RoCC is a tightly coupled interface that Rocket Chip provides. The general-purpose processor can directly communicate with the accelerator via custom instructions reserved in the RISC-V ISA. A RoCC accelerator also has direct access to a core's L1 private data cache [12]. However, RoCC limits users to only four custom accelerators [24]. A TileLink-Attached Accelerator uses the TileLink [7] interface to connect the accelerators to the LLC of the processor [12]. Communication with the accelerator is achieved via memory-mapped registers [24]. Although these interfaces are useful, they are limited in that users can only use the predefined interface (RoCC), or the information must travel over the chip interconnect to get to its destination (TileLink-Attached Accelerator).

We propose contributing a new feature-oriented interface to Rocket Chip for co-processors and accelerators. Instead of limiting designers to a single monolithic interface, we wish to refactor both of the existing interfaces so that designers can choose what features best fit their designs' needs. Rocket Chip [1] comes packaged with a set of example accelerators and an ecosystem of accelerators designed for Rocket Chip has begun to develop [22, 27, 9, 5, 21, 11, 10]. This collection of co-processors and accelerators can show us the feature

[3]The Counter subsection of the specification is still a draft, so the number of performance counters may change in the future.

sets that designers actually ended up using or what features they had to add themselves. From this point, we can determine how much effort it would take to generate the useful feature set for each accelerator and how much effort is necessary to extend the feature set (Question 3.1). Since we will be working within the existing ecosystem, we can then make direct comparisons between implementations in terms of space and power consumption (Question 3.2). Depending on what features are included in the interface, there might be a fair amount of weaving that cross cuts many different modules in the chip or accelerator design. This will help us further determine how FOP affects the chip generator toolchain (Question 3.3).

As a second phase of this experiment, we want to push our FOP integration technique even further. The primary goal of these interfaces is to create a structure for the ISA and the hardware features to interact. This raises the question, if we can use FOP to weave in the necessary logic to extend the ISA and handle special instructions in the general-purpose core's data path, do we need the interface at all? We propose using FOP to dissolve the co-processors and accelerators themselves into the data path of processors. We will be able to use our prior work on the interfaces to directly compare the development effort between the two different approaches. This will give us more insight into how FOP can increase or decrease development effort (Question 3.1). Furthermore, since our prior work should represent minimal implementations of the necessary interfaces, we will know how much space and power we can save by eliminating them altogether (Question 3.2).

5.3 Design Space Exploration in Rocket Chip

It is apparent that Rocket Chip offers fertile ground for open source hardware exploration. It is a complete system for hardware designers and researchers to build and test new designs. On top of that, Rocket Chip provides parameterization to users so that they can customize Rocket Chip either by changing hardware features or in some cases, removing them altogether. On the surface this would seem like an ideal situation for chip designers. However, like we described in Section 2.2, Rocket Chip suffers from the same monolithic design that most other hardware designs do. This limits Rocket Chip in two important ways.

1. Designers are limited to the parameters that Rocket Chip provides.

2. Designers cannot swap different implementation of features without manual refactoring.

Rocket Chip achieves parameterization through the use of **implicit parameters.** In Scala a parameter in any function may be declared as implicit. When the function is used, that parameter does not have to be present in the parameter list to be passed. A variable is passed to the function implicitly if it is defined in a parent scope, is declare as implicit, and shares the same name. This means that parameters must crosscut many different modules to actually end up where their hardware definition is elaborated. If a designer wishes to extend the parameters, they must take care to pass them through the design in the same way. In order to demonstrate FOP's ability to ease development (Question 3.1) we propose creating a feature-oriented design exploration framework. Instead of splaying the parameters through the whole system, we envision weaving them into the design, making them easier to configure and extend.

The two main implementations of processing cores in Rocket Chip are Rocket Core [16], a five stage in order pipeline core, and Rocket BOOM [6], an out-of-order superscalar core. Both Rocket Core and Rocket BOOM share libraries provided by Rocket Chip in their implementations. However, they both contain their own non-swappable micorarchitectural features such as branch predictors and prefetchers. Ironically, despite both of these cores implementing the necessary features, no configuration of either or combination of the two can create a superscalar in order core. Finally, even though the RISC-V ISA is modular by design, neither of these two cores has the ability to easily drop parts of or extend the ISA. The ability to test different designs, or combinations of designs, or remove parts of the design altogether would be extremely beneficial for chip development. Thus, we propose using FOP to make this swapping possible within our design exploration framework (Question 3.1).

Doing this work would allow us to explore both Questions 3.2 and 3.3 on the level of a whole chip design. Luckily, Rocket Chip comes packaged with Verilator, a cycle accurate simulation tool. Additionally, Rocket Chip comes with the standard RISC-V benchmark suite created by the RISC-V Foundation. Given this toolchain, we can performance tune Rocket Chip for the provided benchmarks. We want to show that by quickly generating chip designs, we can find where space and power can be saved faster (Question 3.2), hopefully with little overhead on the toolchain overall (Question 3.3).

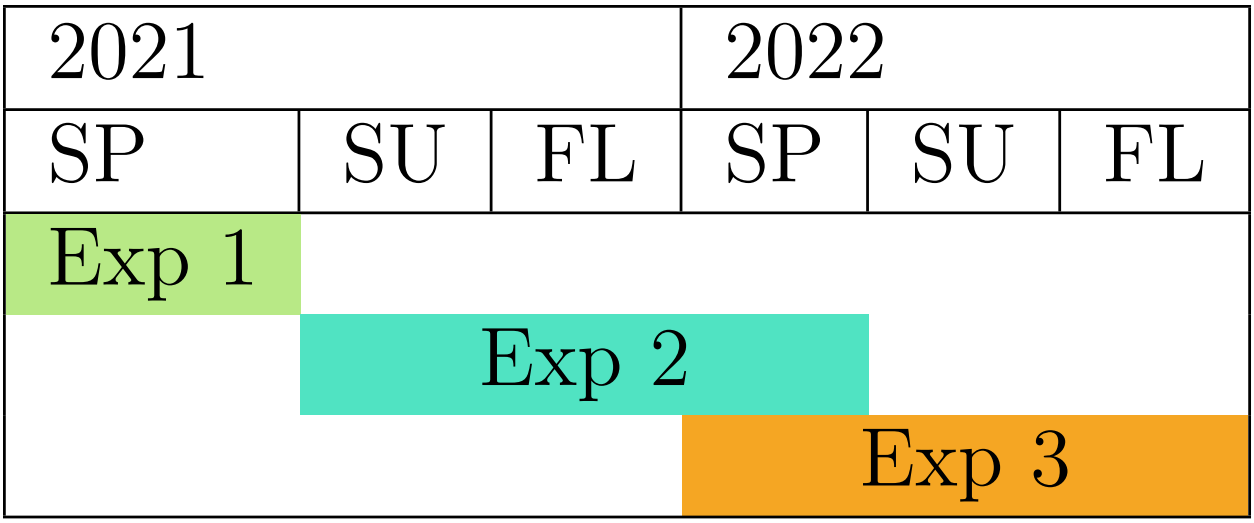
6 Future Work

The work in this proposal opens up the ability to explore the last problem in Section 2.2. This problem poses a new research question that we have not planned to explore, "To what extent can we quantify and analyze interaction of features in a chip design?" What information would we need to capture from a feature? What models would be appropriate for feature analyzing feature interaction? What kinds of interactions, positive or negative, are possible to uncover? Answering these questions could have large implications for quickly discovering problems with both performance and security in chip designs. Demonstrating the advantages of FOP and separating out features from a monolithic design is the first step to exploring this problem, which we have proposed.

7 Research Plan

Possible Places for Publication

- IEEE/ACM International Symposium on Microarchitecture
- Workshop on Computer Architecture Research with RISC-V
- International Symposium on Computer Architecture
- Design Automation Conference



Acknowledgements

I would like to thank Emery Cox IV, Sam Fisher, and Luke Hunt for their proofreading efforts for this proposal.

8 References

[1] Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C., Twigg, S., Vo, H., and Waterman, A. The rocket chip generator. Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[2] Baaij, C. Cλash: From haskell to hardware. Master's thesis, University of Twente, 2009.

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

[4] Bellows, P., and Hutchings, B. Jhdl-an hdl for reconfigurable systems. In *Proceedings. IEEE symposium on FPGAs for custom computing machines (Cat. No. 98TB100251)* (1998), IEEE, pp. 175–184.

[5] Bocco, A., Durand, Y., and De Dinechin, F. Smurf: Scalar multiple-precision unum risc-v floating-point accelerator for scientific computing. In *Proceedings of the Conference for Next Generation Arithmetic 2019* (New York, NY, USA, 2019), CoNGA'19, Association for Computing Machinery.

[6] Celio, C., Patterson, D. A., and Asanović, K. 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, Jun 2015.

[7] Cook, H. M., Waterman, A. S., and Lee, Y. Tilelink cache coherence protocol implementation. *White Paper* (2015).

[8] Decaluwe, J. Myhdl: a python-based hardware description language. *Linux journal*, 127 (2004), 84–87.

[9] Delshadtehrani, L., Eldridge, S., Canakci, S., Egele, M., and Joshi, A. Nile: A programmable monitoring coprocessor. *IEEE Computer Architecture Letters 17*, 1 (2018), 92–95.

[10] Eldridge, S., Waterland, A., Seltzer, M., Appavoo, J., and Joshi, A. Towards general-purpose neural network computing. In *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015* (2015), pp. 99–112.

[11] Genc, H., Haj-Ali, A., Iyer, V., Amid, A., Mao, H., Wright, J., Schmidt, C., Zhao, J., Ou, A., Banister, M., Shao, Y. S., Nikolic, B., Stoica, I., and Asanovic, K. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *arXiv preprint arXiv:1911.09925* (2019).

[12] Huang, Q., Yarp, C., Karandikar, S., Pemberton, N., Brock, B., Ma, L., Dai, G., Quitt, R., Asanovic, K., and Wawrzynek, J. Centrifuge: Evaluating full-system hls-generated heterogenous-accelerator socs using fpga-acceleration. *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2019), 1–8.

[13] Hunleth, F. Building Customizable Middleware using Aspect-Oriented Programming - Master's Thesis, May 2002. Master's thesis, Washington University in St. Louis, 2002.

[14] Hunleth, F., and Cytron, R. K. Footprint and feature management using aspect-oriented programming techniques. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems* (New York, NY, USA, 2002), LCTES/SCOPES '02, Association for Computing Machinery, p. 38–45.

[15] Izraelevitz, A. *Unlocking Design Reuse with Hardware Compiler Frameworks*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2019.

[16] Keller, B. *RISC-V, Spike, and the Rocket Core*. Berkeley Architecture Group, 2013.

[17] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming* (Berlin, Heidelberg, 1997), M. Akşit and S. Matsuoka, Eds., Springer Berlin Heidelberg, pp. 220–242.

[18] Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)* (2019).

[19] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)* (2018).

[20] Lockhart, D., Zibrat, G., and Batten, C. Pymtl: A unified framework for vertically integrated computer architecture research. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), IEEE, pp. 280–292.

[21] Mao, H. Hardware acceleration for memory to memory copies. Master's thesis, EECS Department, University of California, Berkeley, Jan 2017.

[22] Nasahl, P., Schilling, R., Werner, M., and Mangard, S. Hector-v: A heterogeneous cpu architecture for a secure risc-v execution environment. *ArXiv abs/2009.05262* (2020).

[23] Odersky, M., and Rompf, T. Unifying functional and object-oriented programming with scala. *Commun. ACM 57*, 4 (Apr. 2014), 76–86.

[24] Research, B. A. 6.3. RoCC vs MMIO — Chipyard documentation.

[25] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121*, December 2019.

[26] Scalameta. Scalameta · Library to read, analyze, transform and generate Scala programs.

[27] Tiwari, S., Gala, N., Rebeiro, C., and Kamakoti, V. Peri: A posit enabled risc-v core, 2019.

A Code Listings

```
class Adder (bitWidth: Int) extends Module {
  val io = IO(new Bundle{
    val a = Input(UInt(bitWidth.W))
    val b = Input(UInt(bitWidth.W))
    val sum = Output(UInt(bitWidth.W + 1))
  })

  //create each adder and wire them to their inputs
  val adders = for (i <- 0 until bitWidth) yield {
    val unit = Module(new OneBitAdder())
    //wire up the inputs
    unit.io.a := io.a(i)
    unit.io.b := io.b(i)

    unit
  }

  //the first adder needs to have a false input to carryIn
  adders(0).io.carryIn := false.B

  //wire the carryOut from n-1 to the carryIn of n
  for(i <- 1 until bitWidth) {
    adders(i).io.carryIn := adders(i-1).io.carryOut
  }

  //create a set of wires to collect the sums
  val sums = Wire(Vec(bitWidth + 1, Bool()))

  //collect the sums
  for(i <- 0 until bitWidth) sums(i) := adders(i).io.sum

  //get the carry out of the last adder to complete the sum
  sums.last := adders.last.io.carryOut

  //wire up the full sum to the output
  io.sum := sums.asUInt
}

class OneBitAdder extends Module {
  val io = IO(new Bundle{
    val a = Input(Bool())
    val b = Input(Bool())
    val carryIn = Input(Bool())
    val sum = Output(Bool())
    val carryOut = Output(Bool())
  })

  val p = io.a ^ io.b
  io.sum := p ^ io.carryIn

  val g = io.a & io.b
```

```
  val p_c = io.carryIn & p
  io.carryOut := g | p_c
}
```

Listing 1: An example of an adder in Chisel. Special classes for hardware description are shown in red.

```
class CarryLookaheadGenerator (bitWidth: Int) extends MultiIOModule{
  val pIn = IO(Input(Vec(bitWidth, Bool())))
  val gIn = IO(Input(Vec(bitWidth, Bool())))
  val cOut = IO(Output(Vec(bitWidth, Bool())))

  for(stage <- 0 until bitWidth){
    cOut(stage) := generateStage(stage)
  }

  def generateStage (stage: Int): Bool = {
    generateOR(stage, stage)
  }

  def generateOR(stage: Int, level: Int): Bool = {
    level match {
      //the last thing that need to be OR'd is the g of the stage
      case -1 => gIn(stage)
      //otherwise, OR with the level's AND, then generate another level
      case x if x == stage => (false.B & generateAND(stage, level)) |
        generateOR(stage, level-1)
      case _ => (gIn(stage-level-1) & generateAND(stage, level)) |
        generateOR(stage, level-1)
    }
  }

  def generateAND(stage: Int, level: Int) : Bool = {
    level match {
      case 0 => pIn(stage-level)
      case _ => pIn(stage-level) & generateAND(stage, level-1)
    }
  }
}
```

Listing 2: A class that generates the carry-lookahead component for an adder.

```
class AdderAspects(bitWidth: Int) {
  val rippleCarry = Seq(
    InjectingAspect(
      //this function has to point to the actual path of the objects
      //top.adders is the *actual* list of OneBitAdders in the Adder class
      {top: Adder => top.adders},
      {adder: OneBitAdder =>
        val g = adder.io.a & adder.io.b
        // p is actually defined in OneBitAdder
        val p_c = adder.io.carryIn & adder.p
        adder.io.carryOut := g | p_c
      }
```

```
    )
  )

  val carryLookahead = Seq(
    InjectingAspect(
      {top: Adder => top.adders},
      {adder: OneBitAdder =>
        val g = adder.io.a & adder.io.b
        adder.io.pOut := adder.p
        adder.io.gOut := g
      }
    ),
    InjectingAspect(
      {top: Adder => Seq(top)},
      {adder: Adder =>
        val carryLookaheadModule = Module(new CarryLookaheadGenerator(
    bitWidth))

        //pull the p and g out of each module and connect it to the
    carryLookahead module
        for(i <- 0 until bitWidth){
          carryLookaheadModule.io.pIn(i) := adder.adders(i).io.pOut
          carryLookaheadModule.io.gIn(i) := adder.adders(i).io.gOut
        }
        //reassign the carry ins to be from the carryLookahead module
    instead
        for(i <- 1 until bitWidth){
            adder.adders(i).io.carryIn := carryLookaheadModule.io.cOut(i
    -1)
        }

        //reassign the last bit in sums to the carry out of the
    carryLookahead module
        adder.sums.last := carryLookaheadModule.io.cOut.last
      }
    )
  )
}
```

Listing 3: Two aspects that can switch between two different carry circuits.

```
val alu = Module(new ALU)
alu.io.dw := ex_ctrl.alu_dw
alu.io.fn := ex_ctrl.alu_fn
alu.io.in2 := ex_op2.asUInt
alu.io.in1 := ex_op1.asUInt

//163 lines later

mem_reg_wdata := Mux(ex_scie_unpipelined, ex_scie_unpipelined_wdata, alu.
    io.out)
mem_br_taken := alu.io.cmp_out

//291 lines later
```

```
13
14 io.dmem.req.bits.addr := encodeVirtualAddress(ex_rs(0), alu.io.adder_out)
```

Listing 4: An example of how the ALU is splayed out over the Rocket Core.

B Component Diagrams

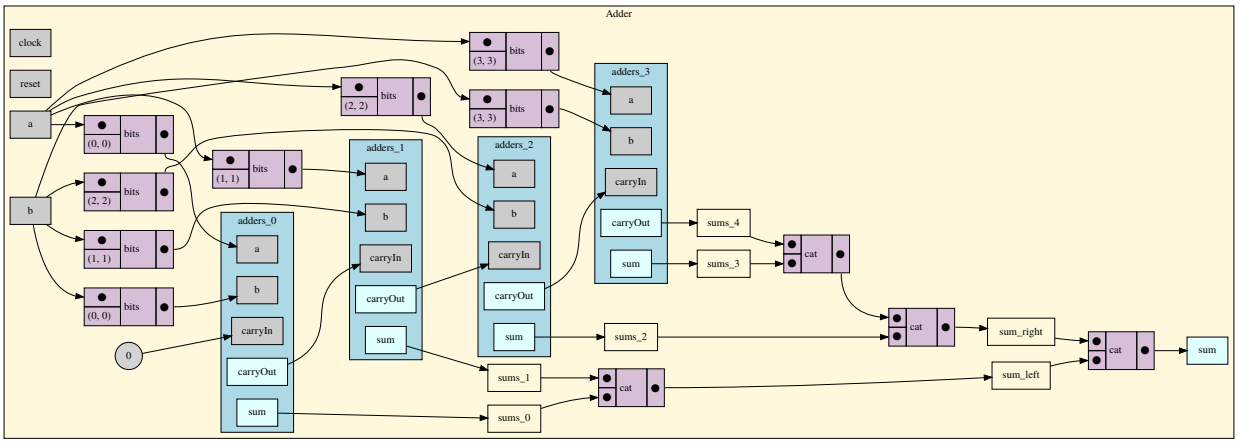


Figure 1: The overview of the generated four bit ripple carry adder.

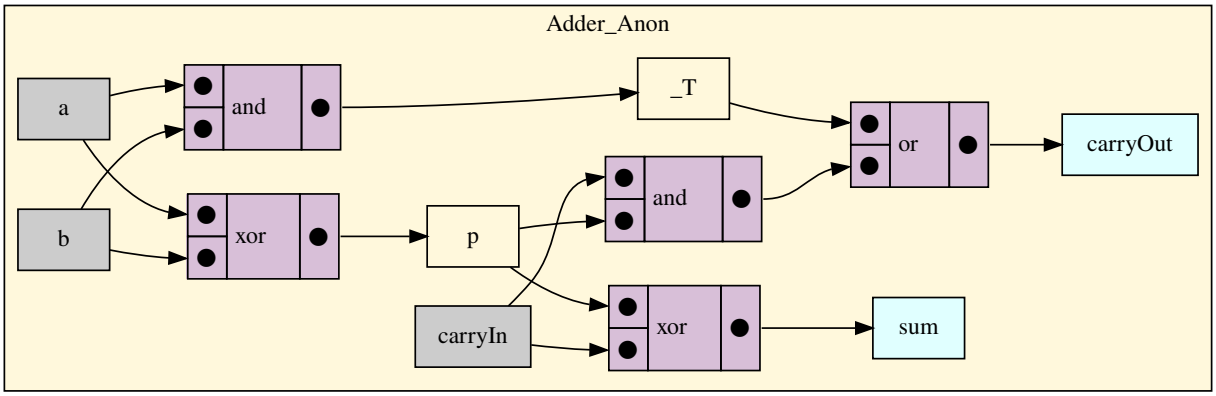


Figure 2: The internals of the one bit adder generated for the ripple carry design.

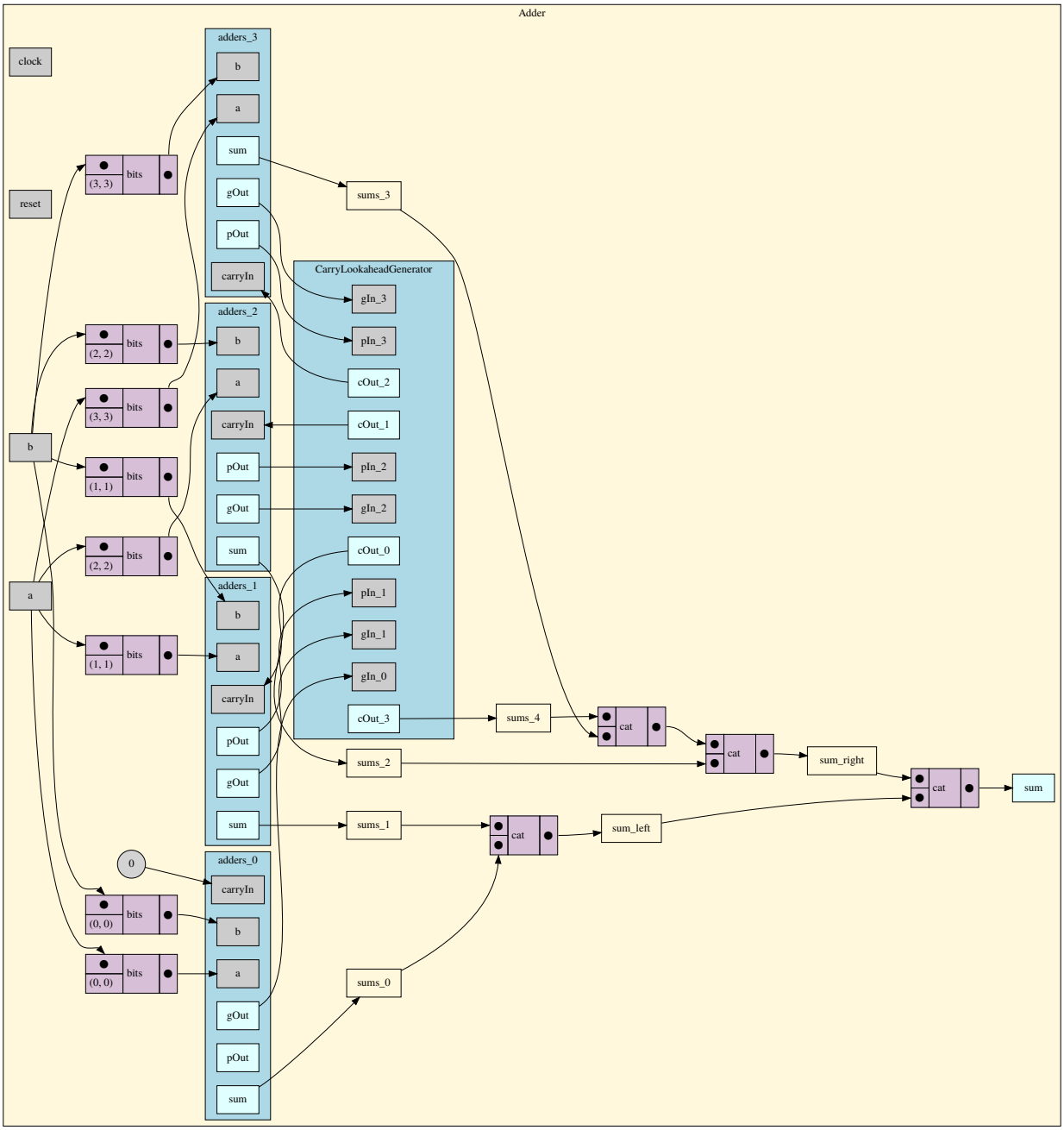


Figure 3: The overview of the generated four bit carry lookahead adder.

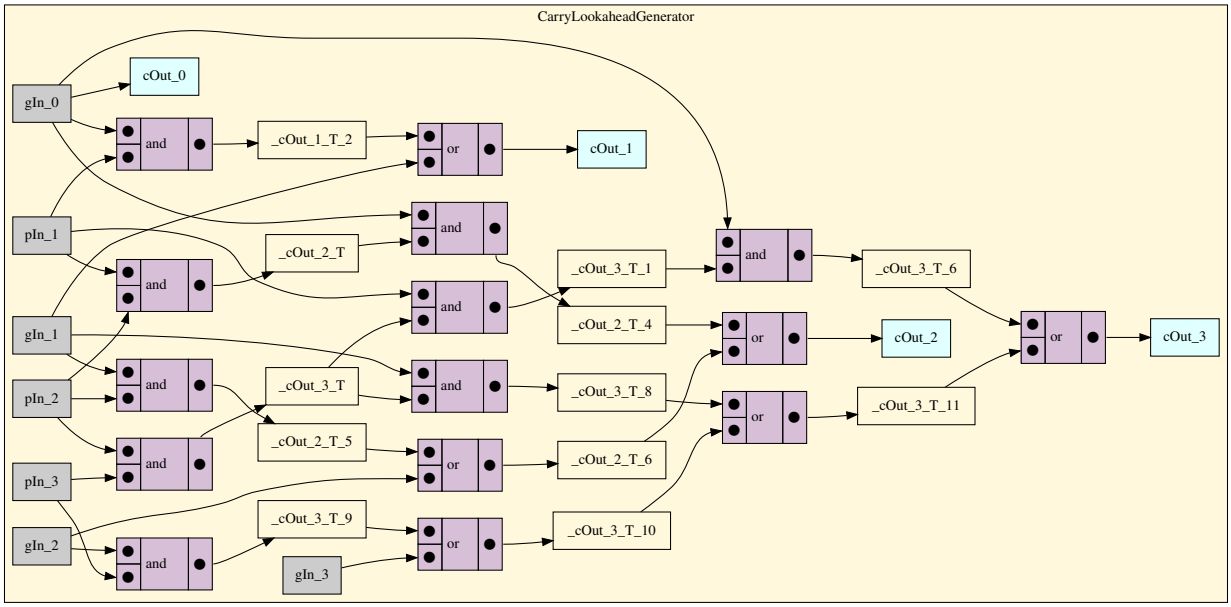


Figure 4: The generated circuit component that performs the carry lookahead operation.

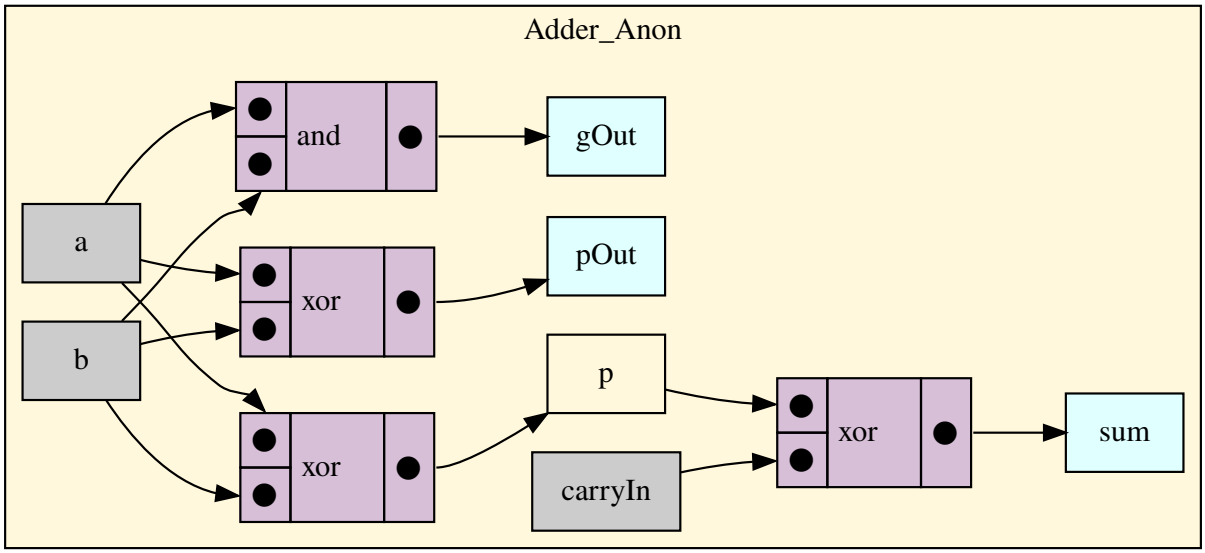


Figure 5: The internals of the one bit adder generated for the carry lookahead design.