Novel Hardware Design Variation through Feature-Oriented Programming

Justin Deters
Advisor: Ron Cytron, ACM Member Number: 9693611, Category: Graduate
j.deters@wustl.edu
Washington University in St. Louis
St. Louis, Missouri, USA

1 PROBLEM AND MOTIVATION

Unlike software programmers, hardware designers must still contend with the relatively low-level constructs of **hardware design languages** (HDLs) such as Verilog or VHDL. This results in monolithic hardware characterizations where many features are hard coded. Effort has been made to alleviate this through **hardware generation languages** which are embedded in software languages. One example, Chisel [1], is a domain-specific language embedded in Scala. These software language based solutions make significant progress towards algorithmic generation of hardware.

Even with hardware generation languages, exploring design spaces to optimize hardware characterizations remains difficult. In this paper, we focus on the potential optimizations of an adder component. Normally, in Chisel, Verilog, or VHDL, such a component would be automatically generated when the + operator is used. The hardware compiler will either optimize for space or performance, depending on what the designer requests. Hardware compilers and synthesis tools offered by companies are often opaque. Thus, hardware designers do not have access to the exact hardware constructs that are being generated. However, a designer may want to optimize both power and performance around a fixed point.

Instead of designers starting from scratch to build optimized hardware components, our approach borrows even more from the software languages domain by using Feature-Oriented Programming (FOP). We take a base design and *compose* different features together creating a system that can quickly be rearranged, and can explore this optimization efficiently. As a stepping stone to larger hardware constructs, we use an adder as a proof of concept for this technique[1].

2 BACKGROUND AND RELATED WORK

Feature-Oriented Programming in Software. [2–4, 7] have shown the efficacy of FOP for creating novel mixtures of features in the context of the CORBA [8] *Event Channel*. The feature-oriented version was 1.4–3 times smaller than the monolithic version, depending on the configuration [7]. Furthermore, the feature-oriented version was able to perform approximately 131,000 events versus approximately 1,600 events per second of the monolithic version. In order to feature-orient their design, this body of work uses Aspect-Oriented Programming (AOP) [6] to optionally "weave" features into the codebase.

Aspect-Oriented Programming. AOP allows programmers to implement concerns that do not cleanly fit into software modules (such as logging). Aspects contain the necessary information to implement a concern *across* modules in a system. They are "woven" into the code to produce the desired final system. Capturing features inside of aspects gives end users a choice of what features they would like in their particular implementation of a system.

Classic Adder Designs. In this work, we use elements of two well known adder designs: ripple-carry adders and carry-lookahead adders. Ripple-carry adders work much like simple pen-and-paper addition where the carry moves from one digit to the next. Due to this, these adders have time complexity of $\theta(n)$. Carry-lookahead adders have circuitry that allows the carry to bypass the lower digit,s making it a faster circuit. Instead, these adders have time complexity of $\theta(log(n))$ at the expense of more space taken by the extra circuitry.

3 APPROACH AND UNIQUENESS

In order to feature-orient hardware designs, we take a cue from [2–4, 7] and use AOP. Normally, the two types of carries discussed in Section 2 would be hard coded into the hardware design. Instead, our approach captures each type of carry in an aspect that then can be selectively applied to the design. Thus, we can quickly rearrange modules in the adder and apply carries to them, allowing easy exploration of the optimization space.

To demonstrate this, we have created an adder that contains elements from the two adders discussed earlier. Our hybrid designs are n bits long. Each n bits can be divided into m **sub-adders** that are each n/m bits in length. The sub-adders use carry-lookahead *internally*, and ripple-carry connects the sub-adders *externally*. We denote these hybrid adders as $S(n, m)$ adders where $S(16, 1)$ is a classic carry-lookahead adder and $S(16, 16)$ is a classic ripple-carry adder. For example, Figure 1 shows an $S(16, 2)$ adder. The length of the adder is $n = 16$, there are $m = 2$ sub-adders, and then each sub-adder is of length $n/m = 8$.

We use Chisel [1] to implement our hybrid adder. Chisel also includes an aspect library [5] that allows new functionality to be inserted into hardware modules. Figure 2 shows the tool flow used in our experiments. The design enters the tool flow on the left *fully configured*. The Chisel toolchain takes care of generating the circuit in the design and applying the aspects. Verilog of the configured adder is emitted on the other end. Typically, the adder would be generated and optimized at the `Vivado` stage of the toolchain. With our method, we pull the generation of the adder and its optimization into the language itself at the `Desgin.scala` stage.

[1]The code for our adder can be found at github.com/jdeters/ChiselAdder

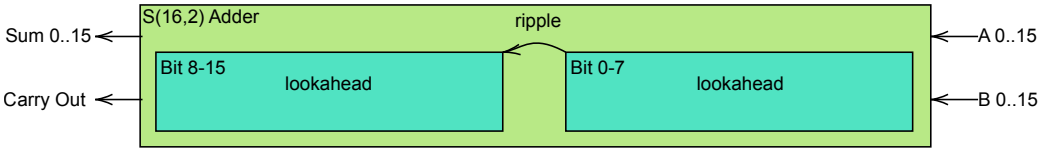

Figure 1: An $S(16, 2)$ Adder. This adder is 16 bits long and has 2 sub-adders each of length 8.

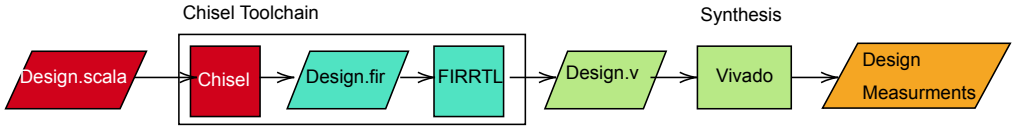

Figure 2: The tool chain we are using for hardware generation.

4 RESULTS AND CONTRIBUTIONS

Using aspects to apply the carry feature made building the hybrid design relatively easy. In total, it only took us a few hours to create our hybrid design. This included designing, coding, and testing. Figure 3 shows the breakdown of the code for our hybrid design. This relatively small codebase generates tens of thousands of lines of hardware code for higher bit adders.

In our hybrid design experiments we examined an $S(128, m)$ adder where $m \in \{2, 4, 8, 16, 32\}$. Figures 4 and 5 show the space utilization and maximum delay of our hybrid designs. The blue and green lines show the performance of each of the classic adder designs, whereas the red and orange lines show the performance of our hybrid design.

Our experiments show that by utilizing a feature-oriented approach, we can quickly rearrange and regenerate the adder characterization to explore the optimization space that would normally be out of the reach of hardware designers. While we recognize that this is a small example, it motivates how hardware characterization and optimization could be aided by feature-oriented design. In continuing work, we have already applied this concept to larger hardware constructs within a RISC-V chip characterization.

Type	Lines of Code
Base	112
Aspect	45
Carry-Lookahead Generator	37
Total	194

Figure 3: The number of lines of code to generate the hybrid designs.

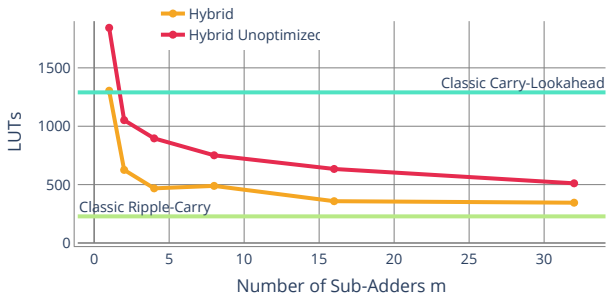

Figure 4: The number of LUTs used in the hybrid design.

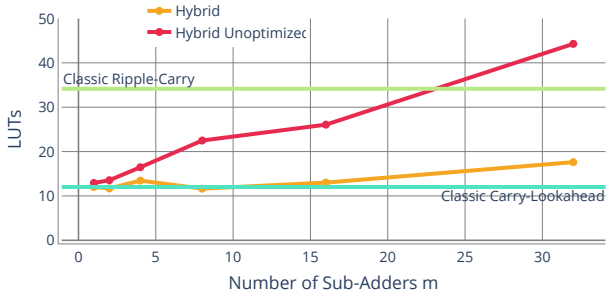

Figure 5: The maximum delay for the hybrid design.

REFERENCES

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[2] Frank Hunleth. 2002. *Building Customizable Middleware using Aspect-Oriented Programming*. Master's thesis. Washington University in Saint Louis.

[3] Frank Hunleth, Ron Cytron, and Christopher Gill. 2001. Building Customizable Middleware using Aspect Oriented Programming. In *The OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*. ACM, Tampa Bay, FL. www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/ASoC.html.

[4] Frank Hunleth and Ron K. Cytron. 2002. Footprint and feature management using aspect-oriented programming techniques. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems* (Berlin, Germany). ACM Press, 38–45. https://doi.org/doi.acm.org/10.1145/513829.513838

[5] Adam Izraelevitz. 2019. *Unlocking Design Reuse with Hardware Compiler Frameworks*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-168.html

[6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242.

[7] Ravi Pratap M., Ron K. Cytron, David Sharp, and Edward Pla. 2003. Transport Layer Abstractions in Event Channels for Embedded Systems. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems* (San Diego, California). 144–152.

[8] Object Management Group 2001. *Event Service Specification Version 1.1* (OMG Document formal/01-03-01 ed.). Object Management Group.