Teaching Agile Hardware Design with Chisel

Scott Beamer
Computer Science and Engineering
University of California, Santa Cruz
Santa Cruz, CA, USA
sbeamer@ucsc.edu

***Abstract*—Agile hardware design techniques take the best of software engineering methods and apply them to improve hardware design productivity. Agile approaches not only reduce the time to solution, but they can also produce solutions which are better tailored to their target problems. Chisel provides the perfect vehicle to teach these techniques as it allows for the creation of reusable hardware generators. In this work, we outline our experiences creating an agile hardware design course using Chisel, and the lessons learned from teaching it four times. All of the course materials are available as open source.**

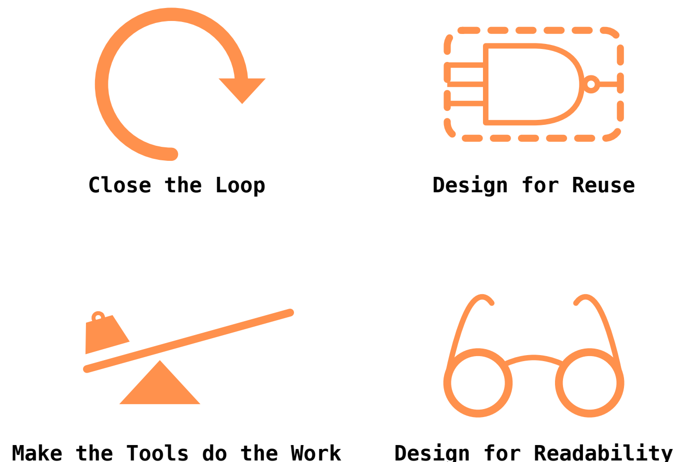

Fig. 1. Recurring themes in the course

I. Introduction

As the development of leading chip manufacturing technologies slows and skyrockets in cost, hardware specialization is essential to deliver efficiency improvements to enable compelling applications [11]. As hardware designs have grown in size and complexity, so too has their design and verification costs. Fortunately, agile hardware design techniques can greatly reduce these design costs through productivity enhancements and even improve the quality of the result. Agile development methods have revolutionized software development [4], so naturally their thoughtful application to hardware design has brought tremendous benefit [3], [16], [19].

In this work, we describe a course we created to teach agile hardware design, and we emphasize the incremental development aspects of agile rather than some of its trademark software development activities (e.g. sprints). By focusing on getting a minimum viable design running through the toolflow as soon as possible, practitioners can better guide their design effort to where it is needed for functionality or performance.

A waterfall development practice (in contrast to agile) can often be the right decision based on the dynamics of the engineering scenario. If the system is hard to modify once constructed, it is logical to want to carefully plan and simulate in advance and methodically move through development steps. For example, if building a dam, once the concrete is poured, it is infeasible to move the whole dam over by a foot. However, software is trivial to recompile and re-execute, so small tweaks are always possible, so iterative development in the agile style is possible and even preferable. Unfortunately, many existing hardware design methods and tools are more like pouring concrete than software development, so waterfall practices prevail. The concrete metaphor is especially apt for writing Verilog. If a Verilog component provides exactly the right functionality and it has been convincingly verified, it can be endlessly reused and will stand the test of time. However, if modifications are required, it can be brittle, especially with regard to the confidence in its verification. A more flexible hardware design language is needed to enable a more nimble agile hardware development process.

The most productive hardware component to develop is one that is successfully reused rather than developed from scratch. To successfully reuse a component, it must provide exactly the right functionality, and it must have performance and resource usage commensurate with the desired application. Mismatches in any of these aspects, even if subtle, can require such fundamental modifications to the component that reuse is more laborious than clean-slate design. Thus, achieving the productivity benefits of reuse requires the reused component closely align with the target application. Instead of a single static design, a *hardware generator* can produce a design instance according to given parameters to better match a desired use case. These flexible generators are not powered by high-level synthesis, but instead deliberate design and metaprogramming by the generators' creators.

We teach our agile hardware design course in Chisel [2] because it helps on two fronts: support for incremental development and enabling the creation of hardware generators. Chisel and its tools are able to leverage the best of software engineering (object-oriented programming, functional programming, strong type system) to enable productive incremental hardware development. Chisel is also great for metaprogramming for creating hardware generators, which

would otherwise require bespoke polyglot solutions in which one language emits another (e.g. Perl emitting Verilog [31]).

Our course has now been taught four times and has become officially part of the course catalog as “CSE 228A: Agile Hardware Design” at the University of California, Santa Cruz. It even satisfies an elective course requirement. A point of emphasis for creating this course was to make it accessible to a wide range of students. Proficiency at agile hardware design requires strong software development skills in addition to traditional hardware design skills. Building a hardware generator via metaprogramming is no trivial feat, as the the creator must think of both language and hardware considerations simultaneously. Few students have an opportunity to develop both skills, so we designed the course to be inclusive of both types of students. In other words, make hardware designers embrace software development practices, and encourage software engineers to apply their skills to hardware design.

In the remainder of this work, we provide various details on the course, its content, enabling technologies, and lessons learned over its four offerings. We hope this will benefit not only hardware design instructors, but also the agile hardware design community to see how their cutting-edge practices are being translated into the classroom.

II. Course Content

Our course is one of the first academic courses on agile hardware design. We designed the course using the backward design technique [17] since we were creating a whole new course. We identified our key learning outcomes, and worked backwards to identify the various concepts, assignments, and activities that would best lead student learning towards them. We have also embraced the agile approach, by revising and extending the course with each offering. We leverage the experiences of students each term to identify and implement these extensions to the course, whether they be new functionalities, content, or even Chisel development best practices.

The primary learning objectives of the course include both a concrete learning goal and a meta learning goal. The concrete goal is to empower students to be able to go from a hardware idea to a working hardware design via agile development techniques. This includes a variety of concrete skills such as: creating hardware generators, design considerations for parameterization, incremental optimization, and using Chisel.

The meta learning goal is to view hardware design as a programming opportunity. This includes abstract skills such as: embracing the agile philosophy of continuous revision, recognizing opportunities for automation, gaining a deeper understanding of the key factors of a problem being solved by metaprogramming a generator, and experiences from starting a project to its release. The skills will be useful beyond Chisel development and even hardware design, especially when recognizing opportunities to apply agile techniques.

Four themes reoccur throughout the course that espouse the course’s philosophy and main lessons (Fig. 1):

Close the Loop Improve through iterative revision, and not an overly ambitious initial design. Strive to get the minimum viable design running through the tools, and then add on functionality and optimize. You can better design and optimize a system once you see how things fit together to best allocate your efforts.

Design for Reuse Reusing well-made components improves productivity and reduces errors. Increase the flexibility of your components to make them more reusable and encapsulate that flexibility within a hardware generator.

Make Tools Do the Work Look for opportunities to make the tools do the work instead of humans (e.g. logic optimization, register retiming, integration). Understand the capabilities and limitations of the tools and adjust your design methodology accordingly.

Design for Readability Appreciate that code will be read many more times than it is written. Thus, it is worth revising code thinking about its readability as a primary concern. Consider how to make the functionality and purpose readily apparent.

The course is about agile hardware design, but naturally, it must also cover Chisel, and by extension, even some Scala. We arrange the course content to teach all three intermixed, incrementally covering each as needed. At times, code aspects are introduced as boilerplate, and then a few lectures later the language mechanisms that enable them are revealed. For example, in order to demonstrate the course theme of closing the loop, the second lecture demonstrates a simple Chisel module being elaborated to Verilog and tested in simulation, but is only able to shallowly explain most of the code. Such an integration requires covering a bit of Scala, Chisel, Verilog, and ChiselTest.

There are two lectures that demonstrate incremental development in practice with in-depth case studies. These lectures give examples of a development roadmap that takes small steps at a time to reach an impressive final result. The emphasis is on identifying the simplest version to get started, building the infrastructure around it, and incrementally optimizing it and adding functionality. One case study designs a FIFO queue (Fig. 2). It starts by making a single-entry queue and builds testing infrastructure for it. Next it revises it to allow for a parameterized number of entries, but it is implemented with a shift register, so it can have bubbles. An improved design removes those bubbles by using a priority encoder. That solution is made more scalable by using a circular buffer. Finally, the design is made to be able to support more parameters to increase flexibility. Along the way, each design is evaluated in terms or functionality and performance, and deliberate choices are made to improve it. The final queue design closely resembles the Queue in Chisel’s standard library, and the case study demonstrates how such an impressive module can be incrementally developed.

III. Course Structure

To support different learning styles and to create an engaging experience, the course engages in a variety of activ-

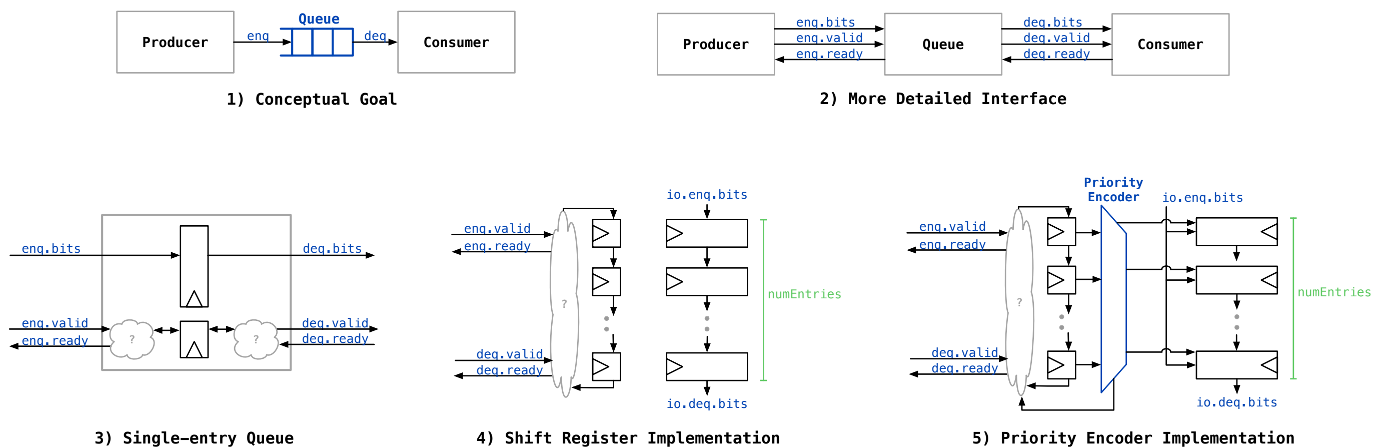

Fig. 2. Figures from the FIFO queue design case study. Using an iterative approach, the design is progressively extended and optimized. The final form uses a circular buffer (not shown).

ities: lectures, labs, homework, and a project. The course is built around the project. The course starts highly structured (lectures, labs, homework) to cover the needed foundational skills to prepare students for the project. As the project starts midway through the term, the course becomes more open-ended, with lecture time slots dedicated to project meetings, guest lectures, and fewer lab and homework assignments. The second half of the course best supports the students as they complete their projects and experience the primary learning objective of the course. Ideally, concepts are introduced in lectures, first coded in labs, substantially utilized in homework coding assignments, and finally applied in the project.

Building the course around the project not only focuses on the main learning outcomes, but its pedagogical strength can be appreciated using Bloom's Taxonomy [18]. Over the term, the course activities shift from the lowest categories in the taxonomy to the highest. At the end of the course, the project primarily exercises the *create* category and since it makes up a significant portion of the course, students kinesthetic engagement with the material is memorable. The overall project process including the presentation and meetings also engages the *evaluate* and *analyze* categories. The preceding lab and homework assignments engage with the *apply* and *understand* categories, and the lectures engage *remember*. However, the course's code-first nature allows students to engage on higher categories more of the time than a comparable course.

A. Lecture

Lectures are the primary instructional component of the course, as there are no assigned readings. Embracing the agile approach, the lectures are centered around incremental and live coding. Each lecture's concepts are interwoven between examples and code demos, all within the same Jupyter notebooks (Fig. 3). Fitting code examples within a slide places a much needed size constraint on the code length, and that modest size is still sufficient to cover most concepts.

The live code enables a playful spirit while exploring the code [28]. It is easy to make small tweaks, and immediately see the outcome. When lecturing in this manner, once a rapport is established with students, there can be frequent informal questions as to what will happen when certain changes are made. Although the instructor can often predict the outcome, it is best to simply make the change and execute, as that demonstrates what we hope students will try.

The first offering of the course occurred during fully remote instruction due to the COVID-19 pandemic. From recent experiences at that time, it was apparent that fully remote lecture was even less engaging than a traditional in-person lecture, which is already a disappointingly passive instructional technique. Since students would be watching the lectures live on their computers, we decided to take advantage of that situation. Instead of trying to compete for their attention, we empowered them to explore by giving them access to the same Jupyter notebooks the lecture was being given from[1]. That enables them to experiment with the code in the lecture slides and see immediate results.

When the instructor inevitably made the occasional mistake, most students found this interesting rather than distracting. The process of the instructor demonstrating the process of debugging what went wrong and then fixing it, made the course relatable and more representative of what their experience might be like. The live coding modality is also ideal to demonstrate the course theme of designing code for readability. The instructor will often show multiple ways to implement the same functionality, and lead discussion and debate about which is more robust or more clear.

During the structured portion of the course, there are 3 65-minute lectures per week. As the course shifts to later open-ended portion for the project, lecture time is often used for project meetings, project presentations, and guest lecturers. Lectures during this part of the course are not as critical for students starting the project and include topics like physical design and open source.

[1]https://github.com/agile-hw/lectures

Scala Values Are References to Chisel Objects

- Our generators are simply instantiating Chisel objects and connecting them together
  - Scala program allows us to control which objects & connections
- The connect operator (`:=`) assigns output of right hand side to input of left hand side
- Can use Scala references to name intermediate results

```
In [ ]: class MyXOR extends Module {
            val io = IO(new Bundle {
                val a   = Input(Bool())
                val b   = Input(Bool())
                val c   = Output(Bool())
            })
            val myGate = io.a ^ io.b
            io.c := myGate
        }
        printVerilog(new MyXOR)
```

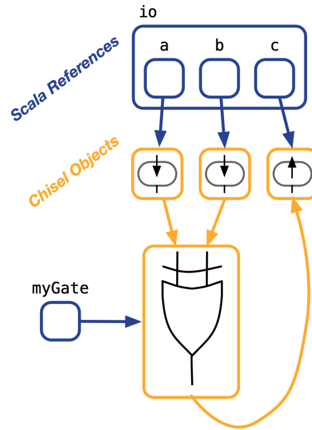

Fig. 3. Example lecture slide that mixes diagrams and live code. This is from the third lecture, and it hopes to clarify the distinction between Scala references and what they are referring to.

B. Labs

Labs are the first course activity for students to try out new concepts after lecture. They are in Jupyter notebooks[2], and the emphasis is on making the smallest possible snippets to try out a feature. With sufficient code provided (scaffolding), the code needed to complete a lab task is typically only a few lines (Fig. 4). Such a constrained example allows students to focus on a specific feature or API without having to worry about the context. The labs can be completed in the browser (thanks to Binder [5]) and autograded promptly to give swift feedback. There is one lab per week during the structured portion of the course, and they can often be completed in under an hour.

C. Homework

Homework assignments follow the labs, and they provide a much more substantial coding opportunity. The assignments are crafted with specific learning outcomes in mind, including demonstrating the iterative revision process. A task in a homework assignment often extends the result of a prior assignment. For example, a task in the first assignment is to create a polynomial evaluator, and the second assignment revises that evaluator to be parameterized and more flexible. An early assignment may require only dozens of lines of code to complete, while a later assignment may require a few hundred lines, especially when counting test cases. Students are provided with the boilerplate to write tests as well as a few example test cases, but these are typically insufficient. Although the homework assignments are autograded, there is sufficient ambiguity as to what causes a graded test case to fail that students are rightly motivated to write their own test cases. With the testing infrastructure in place, the effort to write an additional test case can be made quite minimal.

[2] https://github.com/agile-hw/labs

D. Project

The project is the culmination of the course and executing it well is the primary desired learning outcome. The goal is to make students comfortable with having an idea, designing both the end product and a roadmap to get there via incremental development, and actually building the thing using reasonable language features. Each project produces a unique hardware generator. The preceding activities (lecture, labs, homework) are designed to prepare students for the project.

The project is done in pairs over 6 weeks, overlapping with the tail end of the structured portion of the course. Since each project is unique, there is some variance in project size and scope. As a guideline, we suggest the size of the final project codebase may be $1.5 - 2\times$ the size of the last homework assignment. Although that may not seem large, for the project,

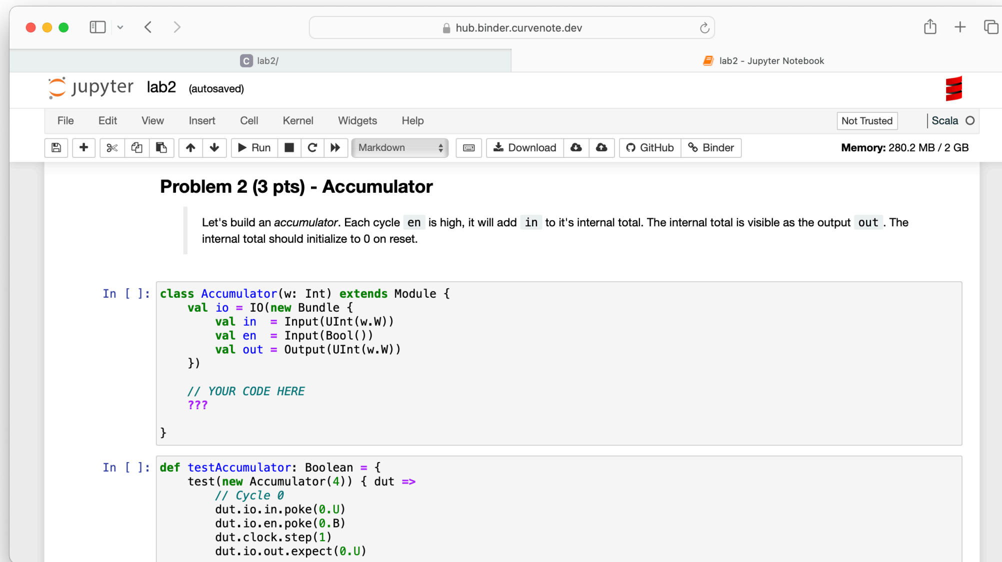

Fig. 4. Example lab running in the cloud (via Binder)

the students produce everything, including the code, its design, and the overall direction. The homework assignments have clear directions and helpful skeleton code. When discussing projects, the staff not only advises students on reasonable scope and useful resources, but also seek to identify project topics that are useful if only partially completed to avoid the stress of an all-or-nothing situation.

Students first meet with the instructional staff to pick a project topic including receiving suggestions. Next they prepare a 1-page proposal. They are given feedback on the proposal during class time, and students find it helpful to listen to the conversations with other groups. To better foster these inter-group interactions, we schedule the discussion order such that similar projects are back-to-back. To ensure continuous progress on the project, they have a checkpoint where they must submit a preliminary version of their project in which some functionality is demonstrated. At the end of the term, students present projects to their classmates. They are given feedback immediately following these presentations, which gives them opportunities to fix any shortcomings before their final submission the following week. Although not required, so far every project team has chosen to release their project open source on GitHub.

IV. Course Infrastructure

The course is powered by an array of software infrastructure, both to enable to hardware design as well as course functions such as autograding. In this section, we overview the various open-source tools we make use of. Beyond adding new functionality, each term, a significant amount of effort is required to modify the infrastructure to adopt updates in the supporting infrastructure as well as find compatible versions. A perennial challenge is also supporting the diversity of computing environments students have on their laptops, so when possible, we try to simplify what they need to install or even completely eliminate it.

At the core, we use Chisel 3.6 with Scala 2.13.10. We held off on using a newer version of Chisel since we extensively use ChiselTest [9]. Much of the development in the course focuses on RTL design, so we use ChiselTest to drive much of the evaluation (and simulation). It is used in lecture, labs, homework, and the project. ChiselTest greatly simplifies much of the course actions, and when running it on treadle [33], it is quite easy to install and operate. The Chisel language is being continuously developed and improved, so perhaps by the next offering, the core features will sufficiently replace ChiselTest so we can migrate to the newest Chisel version.

The Jupyter notebooks power the lectures and labs, and we enjoy Scala support within them thanks to Almond 0.13.14 [1], reusing an integration originally pioneered by the Chisel Bootcamp [7]. Almond not only enables Scala to run in Jupyter, but it also brings Ammonite, a user-friendly Scala shell. Ammonite allows for small snippets of Scala to run directly without being part of a larger program. For lectures, we use the RISE plugin [27] to make Jupyter notebooks into compelling presentations, and we use the splitcell extension [32] to create 2-column layouts. The entire stack to run Chisel within Jupyter is the most complicated installation of the course infrastructure, so we are grateful we can use Binder [5] to provide these notebooks in the cloud for free. Thus, users can interact with the lectures and labs by using only their browser, and the complexity of installing the infrastructure is only required if they choose to work locally. Most students use Binder.

We are indebted to the work of others for providing the

automation to autograde assignments. To collect student code submissions and execute the autograders, we use Gradescope [12], a commercial service our university subscribes to. For labs, we use nbgrader which conveniently allows one to specify problems, solutions, and autograde Jupyter notebooks [22]. Although nbgrader was designed for Python, it works without issue with the Scala environment provided by Almond. For homework assignments, students work on full-fledged Chisel projects powered by sbt. To autograde homework, we use autograding support for Chisel on Gradescope pioneered by dinocpu [20], [21].

We autograde both the lab and homework assignments. The autograders execute within a virtual machine (VM) within Gradescope that is triggered when a student submits, so they can see the autograder results before the deadline. Students typically will revise and improve their submission until they achieve a perfect score. The autograder output is deliberately terse, so when something doesn't pass, students are incentivized to write more tests on their own instead of brute forcing options until it passes. Although the executions for grading the labs use nbgrader and the homework assignments use sbt, under the hood, both use tests from ChiselTest to check for specific functionalities.

Syntax errors are a common challenge for anyone learning a new programming language. For this reason, we strongly encourage students to use a development environment that provides code completion and on-the-fly syntax warnings, such as an IDE like IntelliJ [15]. Most students chose VSCode due to prior familiarity, but with Scala Metals [29], they are able to enjoy the needed IDE-like functionalities.

V. Evaluation

In this section we discuss both how students and the course itself are evaluated. The points for students come from grading the labs, homework assignments, and the project. The homework assignments and labs are autograded, with partial credit possible if only some of the test cases pass. The autograder is unable to run if students do not properly submit their files or there is a compile-time error, but students are quickly notified of such an issue and they fix their submission. The project is the majority of the grade, and it is graded over a long time period including the proposal, initial checkpoint, code review, presentation, and final submission. The project grading rubric not only considers the scope and technical success of the generator the project produces, but also how they used an incremental process to develop it, testing, automation, documentation, and code quality. There was a deliberate decision to not use exams to evaluate students but to instead grade the course on coding and the project as that best evaluates how students engage with this content.

Since this is a new course, it is important to demonstrate its utility. The most compelling evidence is the strength of the students' projects. These projects not only are impressive technically, but nearly every student in the course makes a great project. The instructor also teaches a conventional logic design course in Verilog with waterfall techniques, and there is a stark productivity difference between the final projects for the two courses. The students in the agile course also take the departmental course survey, in which the course scores especially high, even for an elective course.

VI. Related Work

There are a variety of existing educational offerings for Chisel. *Digital Design with Chisel*, a textbook by Martin Schoeberl, is an introduction to digital design using Chisel [30]. Designed for lower-division undergraduates, it teaches hardware design from the beginning, with no expectation of prior Verilog experience. This book is helpful for our course, and is thus a recommended book. However, our course assumes more digital design experience for students coming into the course in order for us to cover more advanced topics and fully embrace agile design.

Chisel was initially created at UC Berkeley, so naturally, many of the first educational artifacts were created there too. The chisel-tutorial repository was the first, and it is a mix of small problems with comments and READMEs to guide its users as to what needs to be done [8]. It is not self-standing, and users also needed to consult the Chisel documentation to be instructed on the actual Chisel features and APIs. The Chisel Bootcamp was a big step forward [7]. It is a set of Jupyter notebooks that include both instructional material and coding problems to work on. We reuse the Chisel-Jupyter integration it pioneered. The bootcamp is designed to be done in 1-2 days, while our course provides a much more thorough introduction to Chisel, Scala, and agile hardware design.

To learn Scala, there are a variety of choices. Historically, much of the initial offerings were produced by Scala's designer, Martin Odersky. He co-authored *Programming in Scala* [24], now in its fifth edition, as well as co-created a 5-course Scala sequence on Coursera [23]. For learning Scala from a book, many have appreciated Li Haoyi's book *Hands-on Scala Programming* [13]. All of these materials are great, but fortunately, that level of Scala knowledge is not required to be a proficient Chisel user. Our course covers a sufficient amount of Scala that any small knowledge gaps can often be solved with a quick internet search.

VII. Discussion

A natural question from students in the course is: "Where in industry can I use Chisel?" Although Chisel is not widely used in industry, we stand by our decision to teach the course in Chisel. Primarily, we view Chisel as the best vehicle to teach and experiment with agile hardware design techniques and to make hardware generators. The meta skills they learn from the course for how to incrementally design and look for ways to leverage software engineering techniques to automate and improve hardware design will be invaluable in whichever language they use in their future careers [25]. After their experiences in the course, students are also more open to experimenting with other emerging hardware design languages.

Over the four course offerings, the course has evolved and improved. In addition to revising and refining assignments and

lectures each term, there have been some notable additions. First, formal verification in Chisel [10] has been added as a regular guest lecture complemented with a lab assignment. Most recently, we have added code reviews, as a great way for students to get more experience revising code and thinking about readability. The combination of intermediate assignments and deadlines for the project have been tweaked to keep students on track and give them more time overall for the project. Early course offerings had homework assignments that frequently included cryptographic cyphers, which are a good application of Chisel, but they proved to add an unnecessary conceptual burden for students not well versed in cryptography. Thus, we have replaced many of those assignments with tasks that require far less prior knowledge.

The primary challenge for the course was content creation. Since it was a new course, all of the lectures, assignments, and their associated code had to be written by the staff. Creating all of this material on-the-fly during the first offering was a demanding undertaking. In subsequent offerings, significant effort was expended revising and extending the course. By open-sourcing all of the course's materials, we hope to lower the burden for others to make similar courses.

Each course offering has been unique due to the guest lectures and the projects students choose to create. Some notable guest lectures included other hardware description languages (Pyrope [26], XLS [34]), core Chisel developers, BOOM (large-scale usage of Chisel) [6], and other open-source hardware design tools [14]. The student projects are the highlight of the course, and some interesting ones include: systolic matrix multiplier, game boy audio decoder, memory controller, Smith-Waterman sequence aligner, and GPS.

The course is centered around an agile philosophy of using incremental development to accomplish big feats. Using such an agile philosophy does not mean there is no plan initially, instead it emphasizes a willingness to re-evaluate and adjust along the way.

This course is indebted to the open-source hardware community, and thus we are keen to give back. All of the course materials[3] (lecture, labs, homework assignments) are released open-source on GitHub[4], and lecture recordings are available on YouTube[5].

Acknowledgements

We would like to thank UC Santa Cruz for providing the opportunity for this course, both for its initial pilot and for its eventual inclusion in the catalog as a regular course. We also are grateful for the teaching assistants (TAs) who have helped administer this course: Jason Vranek, Amogh Lonkar, and Yuanpeng Liao. This course has benefited greatly from generous guest lecturers, including Kevin Laeufer who created the formal verification lecture. Finally, we are thankful for the students who took the leap by taking this new course and enriched it with their enthusiasm, questions, and feedback.

[3]https://classes.soe.ucsc.edu/cse228a/Winter24/

[4]https://github.com/agile-hw

[5]https://youtube.com/playlist?list=PLfrN7RIcMe6g2LBRJLTHTdhyj5s8ag0Rg&si=jGQ6rAdqpEOz0xj3

References

[1] Almond, a scala kernel for jupyter. https://almond.sh, 2015.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, et al. Chisel: constructing hardware in a scala embedded language. *Design Automation Conference (DAC)*, pages 1216–1225, 2012.

[3] Yungang Bao and Trevor E Carlson. Agile and open-source hardware. *IEEE Micro*, 40(4):6–9, 2020.

[4] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, et al. The agile manifesto, 2001.

[5] Binder. https://mybinder.org, 2017.

[6] Christopher Celio, Krste Asanovic, and David Patterson. The berkeley out-of-order machine (BOOM): An industry- competitive, synthesizable, parameterized RISC-V processor. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167*, 2015.

[7] Chisel bootcamp. https://github.com/freechipsproject/chisel-bootcamp, 2017.

[8] Chisel tutorial. https://github.com/ucb-bar/chisel-tutorial, 2012.

[9] chiseltest. https://github.com/ucb-bar/chiseltest, 2018.

[10] Andrew Dobis, Kevin Laeufer, Hans Jakob Damsgaard, Tjark Petersen, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. Verification of chisel hardware designs with chiselverify. *Microprocessors and Microsystems*, 96:104737, 2023.

[11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[12] Gradescope. https://www.gradescope.com, 2014.

[13] Li Haoyi. *Hands-on Scala Programming*. 2020.

[14] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin DF Wong. Opentimer v2: A new parallel incremental timing analysis engine. *IEEE transactions on computer-aided design of integrated circuits and systems*, 40(4):776–789, 2020.

[15] Intellij idea. https://www.jetbrains.com/idea/.

[16] Lizy Kurian John. Agile hardware design. *IEEE Micro*, 40(04):4–5, 2020.

[17] Michael S Kirkpatrick, Mohamed Aboutabl, David Bernstein, and Sharon Simmons. Backward design: An integrated approach to a systems curriculum. In *SIGCSE*, pages 30–35, 2015.

[18] David R Krathwohl. A revision of bloom's taxonomy: An overview. *Theory into practice*, 41(4):212–218, 2002.

[19] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, et al. An agile approach to building risc-v microprocessors. *IEEE Micro*, 36(2):8–20, 2016.

[20] Davis in-order (dino) cpu. https://github.com/jlpteaching/dinocpu, 2019.

[21] Jason Lowe-Power and Christopher Nitta. The davis in-order (dino) cpu: A teaching-focused risc-v cpu design. In *Proceedings of the Workshop on Computer Architecture Education*, pages 1–8, 2019.

[22] nbgrader. https://github.com/jupyter/nbgrader, 2014.

[23] Functional programming in scala specialization, Coursera. https://www.coursera.org/specializations/scala, 2024.

[24] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2008.

[25] Luca Pezzarossa and Martin Schoeberl. Transitioning to chisel in university education: Experiences and lessons learned. In *Nordic Circuits and Systems Conference (NorCAS)*, pages 1–7. IEEE, 2023.

[26] Pyrope, a modern hdl with a live flow. https://github.com/masc-ucsc/pyrope_artifacts, 2018.

[27] RISE. https://github.com/damianavila/RISE, 2013.

[28] Marc J Rubin. The effectiveness of live-coding to teach introductory programming. In *SIGCSE*, pages 651–656, 2013.

[29] Scala metals. https://scalameta.org/metals/, 2016.

[30] Martin Schoeberl. *Digital Design with Chisel*. Kindle Direct Publishing, 2019.

[31] Ofer Shacham, Omid Azizi, Megan Wachs, et al. Rethinking digital design: Why design must change. *IEEE micro*, 30(6):9–24, 2010.

[32] Jupyter split cells extension. https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/nbextensions/splitcell/readme.html, 2015.

[33] treadle. https://github.com/chipsalliance/treadle, 2018.

[34] Xls: Accelerated hw synthesis. https://google.github.io/xls/, 2020.