PyRTL in Early Undergraduate Research

Diba Mirza, Deeksha Dangwal, Timothy Sherwood
{dimirza,deeksha,sherwood}@cs.ucsb.edu
UC Santa Barbara, CA, 93106 USA

ABSTRACT

Undergraduate research experiences are a promising way to broaden participation in computer architecture research and have been shown to improve student learning, engagement, and retention. These outcomes can be more profound and lasting if students experience research early. However, there are many barriers to early research in computer architecture some of which include the gap between pedagogy and research, the lower emphasis on hardware design compared to software in first year courses, and the lack of online resources. We propose lowering these barriers through a methodical approach by involving undergraduates in early research and by creating freely available and innovative educational tools for designing hardware.

We present the experience of a team of undergraduate students with research over one academic year using a Python hardware description language, PyRTL. PyRTL was developed to enable early entry into digital design. Its overarching goals are simplicity, usability, clarity, and extensibility, a stark contrast to traditional languages like Verilog and VHDL that have a steep learning curve. Instead of introducing traditional languages early in the undergraduate curriculum, PyRTL takes the opposite approach, which is to build on what students already know well: a popular programming language (Python), software design patterns, and software engineering principles. The students conducted their research in the context of the Early Research Scholars Program (ERSP), a program designed to expand access to research among women and underrepresented minority students in their second year through a well designed support structure.

KEYWORDS

undergraduate research, hardware tools

ACM Reference Format:
Diba Mirza, Deeksha Dangwal, Timothy Sherwood. 2019. PyRTL in Early Undergraduate Research. In *Workshop on Computer Architecture Education (WCAE'19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3338698.3338890

1 INTRODUCTION

A recent article in *Computer Architecture Today* sheds light on the pipeline problem in computer architecture [5]. While there are many contributing factors to this problem, it is well understood that undergraduate research can improve student retention, increase interest in graduate schools, help to develop critical thinking, improve motivation and persistence, build confidence, help students clarify their career goals, and encourage them to apply to graduate school [2]. Research experiences can be particularly important for students who are underrepresented in STEM, increasing their retention in STEM fields [15, 16]. Research provides a connection to a broader academic community, it allows students to see faculty and graduate students not just as disseminators but also creators of knowledge, and helps students to connect the skills they learn in class to real life problems with tangible impact. As such it seems that undergraduate research in computer architecture can be an important point of leverage in addressing this larger pipeline problem.

While research experience has tremendous potential to be beneficial, there are additional barriers to engaging undergraduates in *early research* specifically in the domain of computer architecture. **First**, there is a lack of "hardware thinking" developed in the early curriculum (in both Computer Science and Electrical Engineering). Instead, courses have a much stronger emphasis on the fundamentals of programming and data structures than digital design. This in turn leads students to think almost exclusively about computing as a *sequential* set of operations which is in conflict with the inherently parallel nature of hardware. Whether this is the right or wrong way to approach computing is not a focus of our work, we simply note the structure we already see existing at most universities. **Second**, most undergraduate computer architecture courses concentrate on areas which, while fundamental to understanding how a computer works, are far from the open research problems and interesting applications redefining our discipline. Furthermore there is a lack of online resources for self-learning concepts and test their understanding, and as one gets closer to open research problems the problem is even more acute. Contrast this with data science and that discipline's use of Jupyter Notebooks both in the laboratory and classroom. A **third** issue is the difficulty of working with the complex ecosystem of toolchains required to develop, play with, test, and evolve a hardware design as compared to software. Student's early debugging skills which they develop from software design do not map cleanly over into the hardware world.

While these barriers are significant, with a combination of technical advances and careful mentoring we believe that meaningful undergraduate research experiences can be had as early as the second year. In this paper we describe our experiences attempting to overcome these barriers in the context of a focused undergraduate research experience supported by PyRTL. PyRTL is python-based hardware design framework designed to help both students and researchers concisely and precisely describe a digital hardware structure in Python. It gives developers a restricted set of blocks from which to build their digital designs (addition, multiplications, arrays, etc.) and naturally avoids many of the pitfalls and confusing boilerplate that dominates the student experience in more traditional hardware design approaches. For example, the clock and resets are implicit, block memories are synchronous by default, there are no "undriven" states, and no unregistered feedback is allowed. In the end everything that is described in PyRTL is synthesizable. While those restrictions can limit what can be described in PyRTL, those limitations were chosen to be in line with best practices today on hardware development and to free developers to treat hardware design more like a software problem. Similar to software it allows more advanced abstractions to unfold naturally from these primitives (e.g. building hardware recursively, writing introspective containers, the creation of hardware analysis and transform functions) which creates more friendly learning curve without sacrificing expressive power.

For early undergraduate research we find that PyRTL helps resolve some of the barriers by providing a gateway to hardware design in the more familiar environment of Python. Students are able to write and test normal Python programs in a way that they are already used to. PyRTL tries hard to avoid any dependence on advanced Python concepts such as decorators, comprehensions, and inheritance[1]. There certainly are both advantages and disadvantages to this approach (e.g. PyRTL does rely on both contexts and some degree of global state to present a reasonable abstraction to users), however being able to write hardware designs, synthesize them, and compute area and latency numbers, all without ever leaving a very understandable Python interface makes it possible to start with research questions rather than more traditional toy digital designs.

Although open and usable tools are important, tools alone are not sufficient to create a positive research experience for novices. Performing research challenges students to learn in ways they are not socialized to expect [8]. While *experts* may find open-ended problems with unpredictable outcomes an appealing aspect of research, to a novice research can be intimidating, foreign, and chaotic. Schultz points out that although some planned chaos is good in undergraduate research to challenge traditional thinking, too much unpredictability leads to ineffective, energy draining, and non-productive cycling [25]. As such, we describe more than just the technical tools we use, but how those tools connected (and in some cases failed to connect) with both our mentoring approach and the students academic preparation at this early stage. Our contributions are as follows:

- We present an experience report on the use of PyRTL for a year-long undergraduate research project involving four second year students in the context of the early research scholars program [9] [4].
- We present outcomes in terms of the students' success with conducting research, and changes to their learning in the cognitive and affective domains.
- We reflect on how PyRTL leveraged student's current knowledge to enable research, which aspects/concepts that the students found most difficult, and the ways in which this experience points to improvements and changes in both PyRTL and our approach to mentoring.

We begin with details of our early research program, follow that with a discussion of PyRTL specific to early research, present the outcomes of our effort, and finally conclude.

2 PROGRAM OVERVIEW

At most R1 universities undergraduates have to seek out research opportunities individually and they typically do so late in their studies. Those who get involved in research early can find it difficult to succeed due to the lack of domain-specific knowledge and skills, training in research, and inadequate support structures. The Early Research Scholars Program (ERSP) was designed to address these challenges in a systematic way [4] [9]. ERSP students complete a full year apprenticeship, in which they take an introductory computer science research course; observe a research group's activities; and then propose and complete an independent project with guidance from a team of mentors consisting of research mentors (faculty member and their graduate student mentor), the program coordinator and their graduate assistant. The program was originally created at UC San Diego and we have been involved in its recent adoption at UC Santa Barbara as part of a multi-institutional scaling effort.

[1]We use all of these concepts extensively as researchers when we build advanced hardware *with PyRTL*, but one is not required to understand these concepts *to use* PyRTL.

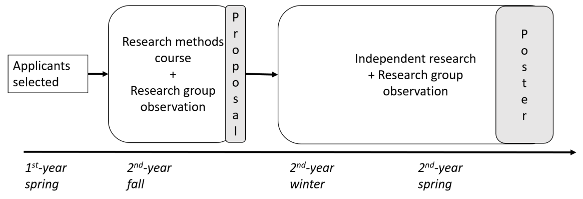
Applicants selected
Research methods course + Research group observation
Proposal
Independent research + Research group observation
Poster
1st-year spring
2nd-year fall
2nd-year winter
2nd-year spring

Figure 1: ERSP timeline of activities showing the main activities and deliverables by students over the course of the academic year [4]

We report on a team of four second year undergraduates who worked on a machine learning project in PyRTL in the context of ERSP at UCSB. The timeline and structure provided by ERSP is shown in Figure 1. Students applied to the program in the Spring quarter. Over the Summer, the research mentors worked on scoping a research project for the incoming students. In the Fall, students took a research methods course where they learned and practiced critical skills in the context of their research project. Students engaged in activities such as reading and summarizing research papers, conducting a literature search, and writing a research proposal. In the same quarter students had regular weekly meetings with their research mentors and observed the activities of the group in the ArchLab [1].

We refer the interested reader to previous publications for more details about the ERSP program [4] [9]. Here, we highlight some of the key strategies used for a successful implementation of the program. In describing these strategies we hope to give the reader an idea of the support and structure provided to the students. This helps us contextualize and interpret the outcomes presented later.

Scoping the research problem

Defining a well-scoped research problem is an important first step in setting up students for success. The challenge is to find a problem that is open-ended yet approachable to second year students. Preferably students should be able to carry out their investigation without the need for extensive engineering or advanced knowledge of architecture.

The research mentors brainstormed on a number of problems before honing in on one that related to the design and study of neural networks in hardware. The concepts that students needed to carry out this work were scoped to include a specific neural network architecture (fully connected networks), combinational and sequential circuits, and an understanding of the performance metrics of neural networks and hardware. Students also needed to learn two new tools to implement their design: PyRTL [11] and PyTorch [24], a Python-based scientific computing package targeted for machine learning applications. These tools would allow them to start with a "pure software" implementation of a neural network for an image classification application in PyTorch and translate it to a circuit design using the software abstractions provided by PyRTL. The students would later collect and analyze their data, all without ever leaving the familiar world of Python. The beauty of this research problem is that it is investigative, cutting edge, and yet approachable.

Mentoring strategies

Effective mentoring requires striking a fine balance between giving students structure and freedom. At the onset of the program, the mentors gave students concrete weekly goals to make sure they have tangible and regular accomplishments. They structured research meetings to facilitate discussion. As an example, the graduate student mentor asked each student to present a slide where they spoke to three prompts, each describing: a new accomplishment, a new learning outcome, and a new challenge they had faced that week. This format allowed the students to have many productive discussions during weekly meetings.

In the first quarter of the program, the graduate and faculty mentors focused on developing a positive rapport with the students and integrating them into the larger research group. The ArchLab weekly group meetings were so popular that other undergraduates in ERSP attended them. These students anecdotally told the coordinator of the program that they came to see the positive exchange between research mentors and the undergraduates during these meetings.

3 PYRTL

PyRTL is an embedded hardware design language with a small and mathematically well-defined set of composable "core" primitives wrapped in a user-friendly syntax. Designing hardware in a dynamic language in general, and in Python specifically, introduces new opportunities to early undergraduate researchers with a concise, broadly understandable, and familiar syntax. Unlike Verilog, anything expressible as valid code in PyRTL always corresponds to synthesizable hardware; PyRTL intentionally restricts users to a set of reasonable digital designs practices.

PyRTL treats hardware design like a software problem, allowing us to build recursive hardware, write introspective containers, and concentrate on building hardware using software abstractions. As future code examples will demonstrate, the use of list slicing, recursion, dynamic typing, introspection, and the creation and use of design patterns is the *natural* way to code in PyRTL. With just 17 primitives, PyRTL makes it simple to add new functionality that works across every design, including logic transforms, static analysis, and optimizations.

The goals of PyRTL are to—

- enable the rapid prototyping of complex digital hardware,
- make all hardware decisions explicit and concrete (rather than inferred, as is the case with HLS),

- lower the barrier of entry to digital design (for both students and software engineers),
- promote the co-design of hardware transforms and analysis with digital designs (through a simple core and translation interface), and ultimately,
- allow complex hardware design patterns to be expressed in a way that promotes reuse beyond just hardware blocks.

Instead of introducing an early undergraduate researcher to hardware design, a vastly different programming style, in an unfamiliar programming language, PyRTL tries to *build on top of* concepts that students have learnt in their introductory undergraduate classes. In the rest of this section, we present the features of PyRTL and how undergraduate students at different phases of their studies can leverage their current knowledge to use PyRTL.

Programming in PyRTL

At the end of their first year of undergraduate studies, students have learned and understood ***recursion***. As seen in Figure 2, with PyRTL we can treat hardware design of a ripple-carry adder much like a software problem. Building recursive hardware using PyRTL's software-like abstractions can be used to create simple and succinct hardware. Complex interwoven structures are particularly cumbersome to specify in many hardware design languages. As students' Python knowledge advances in their second year of undergraduate studies, they can begin to describe complex structures through ***hardware comprehension***. An implementation of `AES decryption`, for example, which performs data unscrambling operations on 8-bit data blocks, can concisely be described using comprehensions in just a few lines of code as seen in Figure 3. PyRTL leverages ***introspection*** for building structures that depend on runtime values within objects. For example, we can implement a `pipeline` as a class where methods of that class define the pipeline stages. A specific type of pipeline is then derived from this base class. When a design is ready, it can be simulated with PyRTL's ***built-in simulator***; the waveforms can be rendered directly on the terminal or as designs scale can be output to VCD. PyRTL also has a ***testing*** and general purpose ***instrumentation*** framework for the more advanced user. Undergraduates in their third or fourth year will be able to build ***transformation passes*** for optimization, synthesis, and more.

As a beginner-friendly hardware design language, PyRTL provides a library of commonly used hardware blocks, such as adders, multipliers, encryption primitives, generators, etc. and these can easily be imported from `pyrtl.rtllib` and `pyrtl.generators`. For early researchers, the provision of such functions can save design time and enables design reuse. PyRTL also provides tools for analyzing aspects of PyRTL designs such as estimating area, finding maximum frequency of a hardware block in Mhz, etc.

```
def one_bit_add(a, b, cin=0):
    return pyrtl.concat(*_one_bit_add_no_concat(a,
         b, cin))

def ripple_add(a, b, cin=0):
    if len(a) < len(b):# ensure b is shorter
      b, a = a, b
    cin = pyrtl.as_wires(cin)
    if len(a) == 1:
      return one_bit_add(a, b, cin)
    else:
      rcarry = one_bit_add(a[0], b[0], cin)
      if len(b) == 1:
        msbits = ripple_half_add(a[1:], rcarry[1])
      else:
        msbits = ripple_add(a[1:], b[1:], rcarry
            [1])
      return pyrtl.concat(msbits, rcarry[0])
```

Figure 2: PyRTL can be used to build hardware using recursion. One place recursion fits nicely is in the design of a ripple-carry adder. We first define what a `one_bit_add` looks like and then call the `one_bit_add` in the ripple-carry adder function recursively.

```
def inv_shift_rows(in_vector):
    a = [in_vector[offser - 0:offset] for offser
        in range(128, 0, -8)]
    out_vector = pyrtl.concat(a[0], a[13], a[10],
        a[7],a[4], a[1], a[14], a[11], a[8], a[5],
         a[2], a[15], a[12], a[9], a[6], a[3],)
    return out_vector
```

Figure 3: Hardware comprehension in PyRTL demonstrated through `inv_shift_rows` function from AES.

4 OUTCOMES

Project Outcomes

In their project, students studied how the choice of neural network hyperparameters in software affect energy and latency at the custom hardware level. This project involved the design of neural networks in software using PyTorch [24], the design of hardware blocks in PyRTL, and understanding how hyperparameter choices affect the energy and latency numbers in the hardware. In effect, this work comes up with rules of thumb to inform neural network design in software to build energy-efficient systems. When the students first arrived, they had not taken any digital design classes and did not have any experience in hardware design. They had previously taken software engineering classes and were very comfortable in object-oriented design. As a result, when the students were introduced to hardware design in

```
def __matmul__(self, other):
    ''' Performs the matrix multiplication
        operation.
    Is used with a @ b
    :param PyRTLMatrix a: the first matrix.
    :param PyRTLMatrix b: the second matrix.
    :return: a PyRTL Matrix that contains the dot
        product of the two PyRTL Matrices.'''
    assert (type(other) == PyRTLMatrix)
    assert (self.columns == other.rows)
    result = PyRTLMatrix(self.rows, other.columns)
    for i in range(self.rows):
      for j in range(other.columns):
        for k in range(self.columns):
          result[i, j] = mult.fused_multiply_adder
              (self[i, k], other[k, j], result[i,
              j])
            result[i, j] = result[i, j][:32]
    result.bits = len(result[0, 0])
    return result
```

Figure 4: The implementation of the matrix multiply function by the ERSP students.

PyRTL, they leveraged their knowledge of object-oriented programming practices and came up with a *hardware design pattern* to concisely instantiate machine learning primitives in hardware. They defined the hardware blocks in their PyRTLMatrix class, instantiated designs with varying parameters, and studied their effect on area and latency. They wrote a paper about their paper which has been accepted for publication at a workshop in the area of machine learning and embedded systems. The matrix multiplication function from their PyRTLMatrix class is shown in Figure 4.

Student Outcomes

We used surveys to collect student feedback on PyRTL's usability, its impact on their learning, and their perceptions about research. The survey contained a set of Likert-scale and free-form questions [3]. All four students responded. Since our sample size is very low, a quantitative analysis of the data is not meaningful. Instead, we discuss some of the interesting themes that emerged from a qualitative examination. All students reported knowing one or more programming language prior to the start of the project among them were C++, Python, Java, C#, and Swift. None of them had any prior experience with hardware, except for one student who took the introductory course in digital design in the first quarter of ERSP.

We asked students to describe computer architecture research and found their responses insightful. In Table 1, we present their responses in the voice of a single student and grouping responses that speak to the same theme. Students' responses describe architecture research as an integral part of computing, understand its significance, and see connections with software and performance. Our hypothesis is that they came to this understanding not only by working on their specific research project, but also by engaging in the structured activities that ERSP provided. As an example, one of the assignments that students completed in the research methods course was to conduct a literature search. As part of this, students read and summarized nine research papers: [11, 13, 14, 19–21, 23, 26, 28]. They discussed these papers with their research mentors during weekly meetings, which encouraged them to think about architecture research more broadly and connect their specific problem to the field.

Themes in student responses to the question: "What is Computer architecture research"?
Connection to computing and software: *"Approaching computing questions from a hardware level. The study of the interaction layer between hardware and software."*
Significance of architecture research: *"It's a really interesting, and very important, area of research due to the decline of Moore's law. Computer Architects have to come up with different ways to optimize computer performance since they can't rely on making transistors smaller anymore."*
The place of performance: *"Creating efficient hardware designs to make better computers, reducing hardware area and critical path."*

Table 1: Student description of architecture research

Later in the course, students used the results of their literature search to write a research proposal. In their proposal, students motivated the research problem, discussed their proposed solution in the context of related work, and described an evaluation plan. The learning goal of this activity was to integrate new information and develop additional clarity (or build awareness of the lack thereof) about their research problem. An excerpt from the introduction of their proposal is shown in Table 2. We note the reference to Moore's Law which also features in their description of architecture research in Table 1.

"With increasing demand for faster machines, computer architects have turned toward optimizing hardware design to further increase performance in terms of speed, energy efficiency, and computational accuracy. Due to the increased popularity of mobile platforms, power consumption and energy efficiency have become the top priority and design constraint within systems [23]. In recent decades, very-large-scale integration (VLSI) technology has reduced the size of transistors to nanometers. However, quantum physics effectively slows the progress of developing smaller transistors; therefore, speed of computers can no longer increase based on transistor size alone [20]. Additionally, as hardware shrinks, more energy is leaked than applied, leading to thermal runaway and hardware damage [23]. Therefore, researchers must look to other solutions to increase speed and reduce energy leakage. One way to do this is using tools that can learn from past behavior to improve future performance."

Table 2: An excerpt from the introduction of the research proposal that students wrote as part of ERSP's Research Methods course

Students speak to the usability of PyRTL and its impact on their learning as summarized by the responses in Table 3. All four students found significant value in PyRTL and report specific features that helped them understand hardware design. Two key PyRTL features that students identified as helpful to their learning are: (1) the use of software frameworks familiar to them, referring to PyRTL test cases, classes and github examples (2) PyRTL's high level abstractions of low level hardware concepts, referring to the `WireVector` class. These two features helped students understand hardware design by building on what they already knew. They also describe specific problems that helped them understand the difference between regular Python programs and PyRTL code (referring to the different types of *if* statements in Table 3). Identifying these differences motivated the need to "think in hardware", potentially priming students for later coursework in computer architecture.

Using PyRTL, students were able to successfully apply software design patterns to create new hardware designs in a familiar medium. However, in making this significant transition from software to hardware, some students felt a bit like *Alice in Wonderland*. At times, PyRTL code, although familiar, did not work as expected. It was during these times, when students' existing knowledge broke down, that they learned and struggled the most, as seen in their comments: "*I think I learned the most when I messed up*" and *"things not clicking intuitively"*.

Impact of PyRTL on student learning
Usability of PyRTL and knowledge transfer from software: *"The documentation was pretty thorough, and the python syntax made it a lot easier to work with. [Documentation] helped me see how to use PyRTL features in test cases. The examples available on github were also helpful."*
Understanding low level concepts via PyRTL's high level abstractions: *"It helped me understand how multiple wires (ie wirevectors) work together in a bitwise way. Wirevectors are interesting to think about and made me understand how things operate in hardware. Because at the very lowest level, computers are just a bunch of wires carrying information so the use of that class helped me visualize that."*
Understanding the difference between software and hardware design: *"The structure of pyrtl files (the design, then the sim step) really helped to convey the differences between hardware and software design. I think when I messed up I learned the most. For instance, learning more about floats and logical ifs. It helped me understand why hardware cannot have conditional logic and how to reuse pieces of hardware, such as the matrix. I think PyRTL has helped me understand digital design extremely well."*

Table 3: Themes in student feedback about the impact of PyRTL on their learning

We asked students to rate the level of difficulty and reward of learning specific concepts on a scale of 1 to 5. Their average ratings are summarized in Table 4. We interpret this data with some caution because of the small sample size. Our main takeaways from the data are: (1) In most cases students experienced a higher level of reward than difficulty. (2) They found crossing over from PyTorch to PyRTL to be the most rewarding learning experience, reporting an average difficulty rating of 4.25 out of 5. (3) As such, students did not find designing hardware in PyRTL easy, reporting an average difficulty rating of 3.25 out of 5. This is not surprising given that they had no prior experience and very limited conceptual understanding of digital design. The main difficulties students faced were debugging PyRTL code, understanding hardware concepts (which at times seemed indistinguishable from difficulties with PyRTL concepts), time management and working as a team. We summarize student responses that speak to each of these themes in Table 5. Students suggested improving PyRTL for use by novices by having clearer error messages, simpler examples in the documentation and tutorial videos that illustrate the hardware designer's thought process as they program in PyRTL.

Concept/skill	Difficulty	Reward
Implementing a feed-forward NN in PyTorch	2.5	3.25
Implementing a hardware design in PyRTL you		
(a) sketched on paper	4	4
(b) had NOT sketched on paper	4.25	3.75
Creating new hardware designs	3.25	3.25
Creating new algorithms	3.25	4
Crossing over from PyTorch to PyRTL	3.25	4.35
Understanding hardware concepts	3.25	4
Understanding concepts related to NNs	2.75	3.2
Making sense of data collected from PyRTL	3	3.75
Collecting performance metrics in PyRTL	2.75	3
Understanding hardware performance metrics	3	3.75

Table 4: Average level of difficulty and reward reported by students on a scale of 1 to 5

So, what did students find rewarding? Their responses in Table 5 provide some clues. We interpret these answers together with our observations of the students over the past academic year. Students had their *aha* moments when they were able to collect latency and area data in PyRTL. To explain the trends in the data, they had to revisit and understand their circuit design more deeply. Students also enjoyed the creative process and the social connections they had made as part of ERSP.

Finally, we summarize changes in student perceptions of their abilities and preparation for the future in Table 6. Students report increased self-efficacy in tackling new problems and just-in-time learning, resilience to failures, and seeing the value of collaboration when tackling hard problems. Students also report feeling better prepared for coursework and graduate studies. Finally, their comments suggest a better understanding of the research process. For example students said that while solutions to open problems are often not straightforward, unlike their experiences with tests and examinations, many solutions exist with their own relative merits. Students speak with a sense of realism about the time it takes to tackle new and challenging problems.

Types of difficulties faced by students
Debugging: *"Encountering errors generated by backend PyRTL code, which were mostly based on our misunderstanding of how hardware worked on the lowest level. Working through several levels of highly abstracted code to pinpoint source of error."*
Learning PyRTL and hardware concepts: *"Having zero prior knowledge in hardware design, things just weren't clicking intuitively . . . PyRTL had a bit of a learning curve . . . I got stuck trying to integrate floating point numbers within PyRTL"*
Team dynamics and time management: *"I would allocate a certain amount of time to work on our project, I would get stuck on something and end up not getting much done, then stress about having something to show by the end of the week which was not easy."*
Summary of most rewarding experiences
Collecting and interpreting data: *"The moment when we had collected data from a fully functioning model, and were able to correctly interpret it . . . seeing the designs actually come into fruition and getting an output."*
Creating new knowledge: *"I had gone into this project with almost no knowledge about any of the related fields. It was incredibly rewarding to analyze and develop new knowledge from something we had built."*
Building social connections: *"It would definitely be meeting all the awesome students, advisors, and professors. Everyone is so nice, helpful, and genuinely cared. That was really uplifting. I also got close to a lot of the other students, especially my group mates so it was nice to see us grow closer together as the year went on. This program has done so much for me to grow as a computer scientist. I credit that to all the amazing help our professors and advisors provided. "*

Table 5: Feedback on the difficulties and rewards of research

Impact of research on students' perceptions
Increased self-efficacy in tackling new problems: *"It's definitely made me feel more capable of diving into problems I've never experienced before and learning advanced skills on the fly . . . it has made me more confident in my abilities to tackle problems, simply because with research, there is no real wrong answer."*
Increased resilience to failure: *"If the solution you thought of does not work, it's not considered a failure but becomes a learning experience."*
Increased value for collaborations: *"I used to always think working independently is better, this research experience has made me realize that's not the case at all. You always need to work with other people, learn from each other, and feed on each other's ideas if you want to tackle difficult problems."*
Increased preparation for coursework: *"I think my research experience has helped me learn a lot of computer architecture concepts before I come across them in my courses. For instance, I just learned about floating points in hardware in CS 111 (Introduction to Computational Science) when I learned about it over winter break for research. A lot of the things we touched on in research have been occurring in all of my CS classes, not just hardware so there is a lot of connection."*
Increased preparation for graduate school: *"I'm planning to pursue a research project in my master's year, and this experience has definitely helped me prepare for the process. I hope to go in to machine learning in the future so working with NNs at such a low level has definitely helped solidify the foundational understanding of ML."*
Increased understanding of the research process: *"I think research has taught me that things take time. Several times during the project, we have hit stopping blocks. I think by working through them we got better but it still takes time . . . it has made me realize there are A LOT of difficult problems to solve, and there always would be . . . I think research is more about motivation than knowledge. There is not always one answer to a problem. We can usually understand what the answer to a problem should be, but sometimes it is outside our expectations. It's made me realize that there are a lot of problems that don't have straightforward answers or ways of getting those answers, which is a little discouraging but also very exciting."*

Table 6: Student feedback on the impact of research on their personal growth

5 RELATED WORK

A clear understanding of computer architecture concepts is necessary to begin early research in the area. In order to start gaining an in-depth understanding of hardware design for computer architecture research, it is important for students to program designs themselves. This has been transformative in computer architecture education, with the introduction of HDLs in the classroom [17, 27]. We stress that when introducing students to early research, we need the introduction of a productive, familiar, but expressive and feature-rich HDLs, like PyRTL. PyRTL's open-source community also makes seeking help and collaborating easy.

Verilog and VHDL are traditional HDLs with steep learning curves and require prior knowledge about how hardware must function, making them difficult to integrate into early research. While there are techniques such as High-Level Synthesis (HLS) which produce logic elements from high level code written in C/C++, these high level abstractions can be difficult to reason about and debug. Understanding what logic blocks some C/C++ code would synthesize requires a deep understanding of how these tools perform the synthesis. PyRTL tries to bridge this gap by providing a middle ground, a high level language with which to write hardware *directly*. PyRTL is not the first, nor will it be the last to try to bridge this gap. Chisel [7] is a Scala-based tool to design hardware. It uses many high level programming features of Scala to provide features like elaboration-through-execution, just like PyRTL. CλaSH [6] is written in Haskell and supports the generation of synthesizable Verilog code. Lava [10], also a Haskell-based HDL comes in variants and provides features like simulation, formal verification and generation of code for implementable circuits. But, the use of a functional programming language like Haskell comes with its own challenges, and the learning curve for early researchers might still be steep. MyHDL [12] is a Python-based embedded language which uses Python generators and decorators, and looks very similar to Verilog. It comes with a wide variety of features like synthesis, simulation, test bench creation, and optimization. PyMTL [18, 22], also Python-based, provides a vertically integrated tool for functional-level, cycle-level and register-transfer level modeling.

6 CONCLUSION

In this paper we have identified three key barriers to early undergraduate research: (1) lack of "hardware thinking", leading students to think almost exclusively about computing as a *sequential* set of operations, (2) the gap between pedagogy and research in undergraduate architecture curricula, and (3) complexity of toolchains to develop, play with, and test new hardware designs, and the misalignment of their early debugging skills when applied to the hardware world. We report on a case study of a structured year-long undergraduate research experience with a team of second-year students. The data collected from the students points to the presence of all three barriers. However, our initial results show that PyRTL lowers these barriers by building on what students already know and second year students can contribute to early research in architecture with the thoughtful mentoring and structure that was provided by ERSP.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1740352, 1717779, 1563935, and 1821415.

REFERENCES

[1] Archlab, UCSB. https://www.arch.cs.ucsb.edu/prof-sherwood.

[2] Council on undergraduate research fact sheet. http://www.cur.org/about_cur/fact_sheet/. Accessed: 2016-22-12.

[3] Survey instrument used in this paper. http://bit.ly/PyRTL-Student-Survey.

[4] Christine Alvarado. Expanding the pipeline by engaging undergraduates in research UC San Diego Early Research Scholars Program. https://cra.org/expanding-pipeline-engaging-undergraduates-research-uc-san-diego-early-research-scholars-program/. Accessed: 2017-10-02.

[5] Newsha Ardalani and Lena Olson. Tackling the pipeline problem in the architecture research community. https://www.sigarch.org/tackling-the-pipeline-problem-in-the-architecture-research-community/. Accessed: 2019-05-16.

[6] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. C? ash: Structural descriptions of synchronous hardware using haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pages 714–721. IEEE, 2010.

[7] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[8] L. Barker. Student and faculty perceptions of undergraduate research experiences in computing. *Trans. Comput. Educ.*, 9(1):5:1–5:28, March 2009.

[9] Michael Barrow, Shelby Thomas, and Christine Alvarado. Ersp: A structured cs research program for early-college students. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 148–153, New York, NY, USA, 2016. ACM.

[10] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. In *ACM SIGPLAN Notices*, volume 34, pages 174–184. ACM, 1998.

[11] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood. A pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7, Sep. 2017.

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

[13] E. Chung et al. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, Mar 2018.

[14] N. P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, June 2017.

[15] S. R. Gregerman, J. S. Lerner, W. von. Hippel, J. Jonides, and B. A. Nagda. Undergraduate student-faculty research partnerships affect student retention. *The Review of Higher Education*, 22(1):55–72, 1998.

[16] Carlos G. Gutierrez, Linda M. Tunstad, Anthony Fratiello, and Scott L. Nickolaisen. Undergraduate research participation increases minority retention and success in chemistry. In *Abstracts of Papers, 225th ACS National Meeting*, 2003.

[17] Daniel C Hyde. Using verilog hdl to teach computer architecture concepts. In *Proceedings of the 1998 workshop on Computer architecture education*, page 10. ACM, 1998.

[18] Shunning Jiang, Berkin Ilbeyi, and Christopher Batten. Mamba: closing the performance gap in productive hardware development frameworks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[19] D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206, Jan 2001.

[20] L. B. Kish. Moore's law and the energy requirement of computing versus performance. *IEE Proceedings - Circuits, Devices and Systems*, 151(2):190–194, April 2004.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[22] Derek Lockhart, Gary Zibrat, and Christopher Batten. Pymtl: A unified framework for vertically integrated computer architecture research. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 280–292. IEEE Computer Society, 2014.

[23] T. Mudge. Power: A first-class architectural design constraint. *Computer*, 34:52–58, 04 2001.

[24] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[25] Jeffery R Schultz. The transformational process of mentoring. *Council on Undergraduate Research Quarterly*, 17:72–73, 2001.

[26] D. Shin, J. Lee, J. Lee, J. Lee, and H. Yoo. Dnpu: An energy-efficient deep-learning processor with heterogeneous multi-core architecture. *IEEE Micro*, 38(5):85–93, Sep. 2018.

[27] Zvonko Vranesic and Stephen Brown. Use of hdls in teaching of computer hardware courses. In *Proceedings of the 2003 workshop on Computer architecture education: Held in conjunction with the 30th International Symposium on Computer Architecture*, page 16. ACM, 2003.

[28] Y. Wang, J. Lin, and Z. Wang. An energy-efficient architecture for binary weight convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(2):280–293, Feb 2018.