An In-Depth Analysis of Pythonic Paradigms for Rapid Hardware Prototyping and Instrumentation

Isra aljrah[1], Ghaith Alomari[2]*, Maymoona Aljarrah[3], Anas Aljarah[4], Bilal Aljarah[5]

[1] The Department of Mathematics and Statistics, Jordan University of Science and Technology, ORCID: 0009-0005-0998-1532

[2] The Department of Mathematics and Computer Science, Chicago State University, galomari@csu.edu, ORCID: 0000-0002-5196-7049

[3] The Department of Mathematical Sciences, University Kebangsaan Malaysia, ORCID: 0009-0006-4208-9666

[4] The Department of Mathematical Sciences, University Kebangsaan Malaysia, ORCID: 0000-0002-9033-6928

[5] The Department of Electrical Power Engineering, Yarmouk University, ORCID: 0009-0009-9484-221x

Abstract: This in-depth examination review about pythonic paradigms for quick hardware prototyping and instrumentation to solve the widening gap between the fluidity of software development and the more conventional methodology used in hardware design. The proliferation of specialized hardware and accelerators has resulted in an increased necessity for development strategies that are flexible. Reconfigurable hardware, such as FPGAs and coarse-grain reconfigurable arrays, enables incremental changes in early design stages, which is a key component of agile software development. This article goes into modern agile development methods, emphasizing the integration of services, repositories, and specialized tools to meet particular users' requirements. Continuous improvement in areas such as software and hardware design, testing, analysis, and evaluation, as well as collaboration with current development tools, are all supported by an ecosystem that has been successfully integrated.

Keywords: Rapid Hardware Prototyping; Instrumentation; Pythonic Paradigms; Reconfigurable Hardware.

1. INTRODUCTION

A mismatch has emerged in the development flow between the more top-down waterfall technique of development that predominates in traditional hardware design and the faster cycles predicted by software due to the recent growth in specialized gadgets and accelerators to boost performance and energy efficiency. The recent surge in specialized hardware and accelerators to boost performance and energy efficiency is the cause of this discrepancy [1]. "Field-programmable gate arrays (FPGAs)" and coarse-grain reconfigurable arrays are two examples of reconfigurable hardware that can be used. These devices are particularly noteworthy because they further facilitate agile development by enabling the gradual improvement and modification of early roll-out designs [2]. However, based on what we've observed, most modern agile development models heavily rely on a varied ecosystem of tools, documents, and services that can be customized to meet the unique demands of the user and integrated seamlessly into the workflow. This ecosystem allows for continuous improvement of the entire process of development when it is integrated effectively [3]. This involves not only the hardware and software design but also test, analysis, and evaluation in addition to the way it integrates with the development tools already in use.

2. DIFFERENT PYTHON BASED HADRWARE ARCHITECTURE

a. PyRTL

Python's use in the production of hardware, particularly for quick prototyping and architectural tradeoff evaluation, has seen a substantial uptick in popularity over the past few years. The hardware development process has been significantly sped up, and its efficiency has been significantly improved as a direct result of this transition [4]. Python supports hardware prototyping and instrumentation, specifically through the framework known as PyRTL [3]. The shift toward more integrated hardware acceleration logic, which includes components with both programmable and fixed functionality, has made it more difficult to understand systems in their entirety. In light of this, a paper investigates the difficulties associated with hardware instrumentation and analysis as well as the potential solutions to those difficulties [3].

"Gate-Level Information Flow Tracking": Information flow analysis plays a critical role when assessing the security features of hardware designs. The scope of information flow analysis can now be brought down to the hardware level with the help of GLIFT, an example tool for "Gate-Level Information Flow Tracking. It monitors the "taintedness" of values and looks for unwanted flows in order to maintain the confidentiality and authenticity of the data. PyRTL's instrumentation architecture enables the smooth implementation of GLIFT, in which additional "shadow" hardware is introduced to track "taintedness" and calculate the taintedness of wire outcomes. This is made possible by the use of GLIFT's taint tracking capabilities. The negligible influence on performance and the condensed code (72 lines) indicate how straightforward it is to construct such instruments [5].

PyRTL's Instrumentation: API is the lynchpin that makes it possible to make hardware instrumentation available and efficient. This application programming interface simplifies the process of gathering often-required information and making hardware modifications. The Instrumentation API is comprised of a number of crucial components, including the following:

Iterator for Topological Sorting: An iterator is made available by the instrumentation framework of PyRTL. This iterator ensures that a logical action is only returned after its preceding actions, which maintains the flow of data. This ensures that the activities are completed correctly, reducing the need for extra verification.

Net Connections: The framework provides the function net_connections, which tracks which wires are linked to nets and which networks use the wire. This helps improve performance and avoid confusion by keeping track of which wires are connected to which nets. The circuit can be traversed and transformed more effectively with the use of this information.

Wire and Logic Replacement: Changing a piece of hardware's logical and electrical components can be difficult and time-consuming. PyRTL's application programming interface (API) streamlines this procedure by offering wire_transform and net_transform methods, which enable element substitution in a more organized fashion.

Python, using frameworks such as PyRTL, has substantially contributed to the increased productivity and adaptability of the hardware prototyping and instrumentation industries. The comparison in Table 1 demonstrates how proficient Python is in hardware design [5].

Comparison of PyRTL + Verilog Designs

Design	Area (sq. µm)	Delay (ns)	Gate Count	Lines of Code
PyRTL Design 1	1250	3.2	8500	1600
Verilog Design 1	1270	3.3	8600	1750
PyRTL Design 2	980	2.8	7200	1400
Verilog Design 2	1000	2.9	7300	1600
PyRTL Design 3	1350	3.5	9200	1850
Verilog Design 3	1380	3.6	9300	2000

The table provides a comparative examination of hardware designs that have been implemented in both PyRTL and Verilog. The table focuses on key metrics such as area, latency, gate count, and lines of code. The table below highlights the significance of determining the effectiveness of Pythonic hardware designs. Notably, the findings indicate that PyRTL designs typically display delay and gate counts that are comparable to their Verilog counterparts. This suggests that Python's function in hardware prototyping is not affected by worries of inefficiency.

b. PyRTL-Matrix

Creating plans for the apparatus Pythonically provides an option that is more approachable than the high learning curves associated with standard Hardware Description Languages (HDLs) like Verilog. PyRTL is a high-level hardware description language (HDL) written in Python. It emerges as a powerful and easy-to-use tool in this setting because it places an emphasis on simplicity, usefulness, clarity, and flexibility in hardware design [6]. PyRTL is unique among its contemporaries in that it places a larger focus on clarity and abstractions over-optimization. This enables it to be used for the analysis and design of hardware. PyRTL gives its users a high-level comprehension of the process of creating hardware as well as its analysis and testing by introducing an intermediate structure supported by an extensive toolchain [7].

The most crucial stage of the procedure is called the inference phase, and it is comprised of "matrix multiplication operations" involving the input and weighted matrix vectors in each layer of a neural network. To make the development of such matrix operations easier and their implementation simpler, the PyRTLMatrix class was created. PyRTLMatrix makes the application more user-friendly by abstracting the hardware that is required to represent matrix operations [8]. A PyRTLMatrix instance is essentially a wire bus that routes data through logical computations that mimic matrix operations. In addition to previously developed PyRTLMatrix routines, this solution includes basic PyRTL multiplication methods. More comprehensible function designs are produced by using an object-oriented approach to hardware design. The decorator pattern is also utilized by the PyRTLMatrix class to ensure that every element has a bitwidth that is consistent with every other element. While this approach might expedite the testing procedure and provide precise outcomes, it sacrifices other advantages in the process. It is possible for users to feel as though they are not fully integrated into the hardware implementation, which is akin to a more software-like approach. However, low-level control and comprehension may become more challenging as a result of this abstraction [9].

Training parameters are shared by the software networks in PyTorch and the hardware networks in PyRTL, as Figure 1 [10] illustrates. The computer network in PyTorch and its hardware equivalent in PyRTL make up the two fundamental parts of this hardware-software pipeline. Smooth translation between the two networks is made possible by this arrangement, which is fully Python-based. Unlike the typical challenges associated with HDLs, the PyRTL framework presents a viable route towards hardware design and analysis. One can compare this to the traditional HDLs. It offers hardware designers an approachable starting point by emphasizing simplicity, abstraction, and clarity, opening up new avenues for the development and implementation of hardware systems [6].

Figure 1: Conversion of PyTorch's training parameters into those of PyRTL [6].

c. PyRTL-Matrix

PyMTL3 is a framework with the goal of providing hardware designers with a platform that is flexible, modular, and extendable [11]. This platform will give hardware designers the ability to customize their workflow by selecting different "flow steps." In addition, it makes very minor adjustments to the preexisting code to cater to the ever-changing requirements of RTL designers and computer architects. PyMTL3 takes its cues for its design from LLVM, which has a rigorously modular architecture. This architecture separates the frontend integrated "domain-specific language (DSL), intermediate representation (IR), and passes". The embedded DSL in PyMTL3 reveals modeling primitives, allowing designers to describe hardware, build test benches, and define parameters. PyMTL3 is the component that is accountable for the development of the hardware model as well as the generation of an "in-memory intermediate representation (IMIR)". The IMIR includes application programming interfaces (APIs), allowing users to retrieve and

edit the metadata stored for the full hierarchical model [12]. PyMTL3's IMIR provides a model-level perspective of the complete design hierarchy, in contrast to other intermediate hardware representations, such as FIRRTL and CoreIR, which concentrate their efforts primarily on the modeling of circuits. This not only offers RTL circuits but concurrent and functional-level techniques as well as update blocks that are able to include arbitrary Python code.

The PyMTL3 framework allows designers to use passes, which are predetermined programs that engage in conversation with the IMIR. These passes are divided into three categories: analysis, instrumentation, and transformation passes. Each of these passes serves a particular function:

Analysis Passes: These passes generate valuable outputs by performing in-depth analysis on the PyMTL3 IMIR model. They give valuable insight into the plan without impacting the framework of the model in any way.

Instrumentation Passes: Instrument passes offer new functions to the model, expanding its capabilities without altering the hierarchical structure of the model. These passes give designers new tools that they can use to better understand and manipulate the design.

Transform Passes: The model hierarchy is changed via transform passes. They enable structural changes to the design by adding, removing, or replacing sections of the model [11].

d. MyHDL

MyHDL is a language designed to make hardware creation accessible to beginners by utilizing the Python framework to accomplish HDL needs. (40) MyHDL is the product of MyHDL. With specific integrated libraries, it is simple to convert MyHDL code to Verilog or VHDL, and you may use constructs to quickly validate the designs. While this language's HDL description is similar to Verilog's, the verification procedure for this language is simpler. With MyHDL, waveform viewing is also available. Lightweight, interactive threads that communicate with one another are used in MyHDL to represent hardware. Specifically, generators—modules that wait for a certain signal to perform an action and communicate with one another through generator functions—are at the core of the structure of the MyHDL description language. Furthermore, generator operations enable the state of the functions that are now in use to be preserved and resumed as needed. They can therefore be used as extremely light threads. This approach will make it feasible to send control-related data to the specialized real-time simulator. Finally, by converting MyHDL's code into Verilog, other HDL simulators can co-simulate with it. MyHDL is responsible for enabling this feature [6].

e. PHDL

A Python framework called Python Hardware Description Language (PHDL) [6] aims to increase the virtualization of hardware design and hence increase the amount of awareness of the jobs that designers perform. Python Hardware Description Language is referred to as PHDL. The two main parts of the PHDL framework are framework subclasses and component libraries, which contain pre-made descriptions of low-level components. There are two components available for users. The three main categories of items that designers use while building components and systems are connectors, elements, and connections. Connectors are figurative representations of single wires as well as specific wire groups. Meta-components are in charge of determining which component is most suited for a given design, whereas vanilla components carry out the logic. A PHDL component can be one of these two types. Connection objects manage the process of joining two or more connectors. While the designers build connections as functions, the PHDL offers templates that the designers can utilize to develop components and circuits as classes. Finally, but just as importantly, PHDL allows hardware modules to be parameterized and integrated with existing libraries via a wrapper.

f. ArchHDL

ArchHDL [13] is a high-level description language (HDL) for RTL modeling that is based on C++ and places an emphasis on providing user-friendly module descriptions as well as extensible testbench specifications. The major goal of ArchHDL is to increase the speed at which hardware simulation can be performed while also making it easier to access hardware implementation. Code that was written with the ArchHDL library, which provides capabilities such as non-blocking allocations and all Verilog constructs, can be simply compiled with a C++ compiler, parallelized with OpenMP, and simulated with the hardware design running the resulting binary. Designers can do all of this with ease. Because of these considerations, it is possible to significantly outpace the Synopsys VCS simulator in terms of

simulation speed. ArchHDL has some restrictions, despite the fact that it shares all of C++'s advantages. As a result of the necessity for ArchHDL to be converted into Verilog, the ArchHDL framework only supports the data types used in Verilog. In instance, the construction of some ArchHDL types of data directly originates from C++ ones, such as the wire and integer types, which come from the C++ integer. This necessitates the inclusion of various limits (for example, arrays can only be a maximum of bi-dimensional), as a result of which some ArchHDL data categories can only be implemented as bi-dimensional.

3. DIFFERENT STUDIES

"Traditional HDLs" have a steep learning curve that might take a very long period, even for seasoned engineers. However, HLS approaches have the potential to increase the abstraction level and split standard C/C++ functions into logic components. Nevertheless, there is currently no clear-cut route that can be taken to create an ideal design without prior hardware engineering expertise. By enabling programmers to explicitly express their hardware designs utilizing a language they have become familiar with, PyRTL aims to provide an option that sits midway between these two extremes. This wasn't the first attempt, as was to be anticipated, to address this requirement. One Scala project that tries to achieve similar goals to PyRTL is called Chisel [14]. In this sense, Chisel is akin to PyRTL as an elaborate-through-execution hardware development language. OpenRISC is one of the many great research initiatives that makes use of the powerful tool chisel. In addition to labeled cable hierarchies and signed types, it offers a well-constructed control framework. While Chisel has concentrated on creating a comprehensive toolchain, PyRTL has concentrated on creating a comprehensive toolchain that is useful for educational initiatives. It also provides a reasonably well-defined intermediate structure that makes hardware analysis routine development quick and simple. Haskell-based ClaSH [15] is a domain-specific language for embedded hardware descriptions. ClaSH offers a technique that works with both synchronous and combinational sequential circuits, and it makes it possible to convert these high-level specifications into low-level representations that Verilog HDL may be used to synthesis. PyRTL and this functionality are similar. Unlike PyRTL, which is built on Python, one of the most widely used languages that supports imperative and functional style programming, the user of ClaSH has to deal with the challenges specific to functional programming. Based on Python is PyRTL. Furthermore, designs must be statically typed, much like VHDL. More specifically, variable types in PyRTL can be deduced while a Python program is running, eliminating the requirement for explicit definitions in the source code. This facilitates the design of reusable structures. Python is the engine behind two hardware design programs: PyMTL [17] and MyHDL [16]. Generators and decorators serve as the foundation for MyHDL, an embedded language that allows higher-level modeling as well as asynchronous logic with semantics akin to those of Verilog. Still, only a structural "convertible subset" of the language's syntax might be automatically translated into hardware, just like with conventional HDLs. One benefit of the language is this. It is possible to simulate and model at many different stages of the design process with PyMTL. The Python Abstract Syntax Tree parsing process facilitates the automatic translation of executable software specifications into hardware, much like My HDL does. Still, there are some restrictions on this technique. Unlike these other methods, PyRTL always works inside a single clock domain and does not incorporate unsynthesized hardware foundations (i.e., composable set data structures-based bottom-up development). This is of tremendous assistance to less frequent users, for whom it is still not clear why some programs can be generated while other scripts cannot. It also has a hardware instrumentation framework that provides quick and easy ways to walk through accelerator functionality and enhance it in PyRTL. The SysPy project [18]adopts a novel perspective on the problem of hardware realization, making it more approachable. The authors offered a "glue software" method to bridge the gap between programming expressions and their corresponding hardware implementations, utilizing pre-made VHDL components and programmable microcontroller soft IP cores. Moreover, utilizing pre-existing HDL elements that may be exported from frameworks that already exist is another objective of PHDL [5].

To keep up with the ever-increasing need for higher performance, there has recently been a remarkable growth in artificial intelligence acceleration. For the purpose of converting high-level code into hardware, high-level synthesis, or HLS, technologies give programmers the ability to abstract away the complexity of hardware design. For example, Richmond et al., [17] demonstrate how the power of higher-order functions can be included in the electronics design using C/C++. It does this by employing a syntax that is comparable to that of modern software languages in order to provide a higher-level abstraction of hardware design. These issues include low-level programming, registers, and scheduling. Using traditional HDLs to design hardware is associated with a wide variety of challenges, some of which are listed above. Similarly, SPA-TIAL [19]is a domain-specific language and compiler that enables increased

productivity through software ideas such as nested loops while enabling access to explicitly access to lower-level notions such as memory hierarchy and memory transfers. Other HLS tools, such as TABLA[20], are explicitly geared towards machine learning and operate as accelerator generators for machine learning algorithms. These tools are also available. However, in contrast to TABLA, our work concentrates more on a design flow as opposed to automatically producing hardware based on a stated learning model. This is because TABLA was developed to automatically generate hardware. Traditional HDLs, like Verilog, are notoriously challenging to understand and apply, particularly for software engineers; they also lack the modularity necessary to reuse components. As a result, other tools have been developed that enable developers to build hardware using interfaces more akin to those used in software.

4. CONCLUSION

In the end, this review describes pythonic paradigms have brought in a new era of quick hardware prototyping and instrumentation. This era has been made possible by the rise of the Internet of Things. The historically difficult and time-consuming process of hardware design has been greatly simplified because to Python's adaptability, which is shown by frameworks such as PyRTL, and PyMTL3. Python gives engineers and software developers the ability to solve complex hardware difficulties in an effective manner. It does this by providing a seamless integration of the development of both hardware and software. Python's ability to make hardware design more approachable, efficient, and agile positions it to play a major role in determining the direction that the future of hardware creation will take. The need for specialized hardware is expected to continue to climb.

5. COPYRIGHT FORMS AND REPRINT ORDERS

You must submit the Copyright Form per Step 7 of the CPS author kit’s web page. THIS FORM MUST BE SUBMITTED IN ORDER TO PUBLISH YOUR PAPER.

Please see Step 9 for ordering reprints of your paper. Reprints may be ordered using the form provided as <reprint.doc> or <reprint.pdf>.

Acknowledgment

The preferred spelling of the word “acknowledgment” in America is without an “e” after the “g”. Avoid the stilted expression, “One of us (R.B.G.) thanks” Instead, try “R.B.G. thanks”. Put applicable sponsor acknowledgments here; DO NOT place them on the first page of your paper or as a footnote.

6. REFERENCES

[1] Fraternali, F., et al., Quantifying the impact of variability and heterogeneity on the energy efficiency for a next-generation ultra-green supercomputer. IEEE Transactions on Parallel and Distributed Systems, 2017. 29(7): p. 1575-1588.

[2] Batten, S.J.C.T.C. An open-source python-based hardware generation, simulation, and verification framework. in Proceedings of the Workshop on Open-Source EDA Technology (WOSET’18). 2018.

[3] Dangwal, D., G. Tzimpragos, and T. Sherwood, Agile hardware development and instrumentation with PyRTL. IEEE Micro, 2020. 40(4): p. 76-84.

[4] Bahr, R., et al. Creating an agile hardware design flow. in 2020 57th ACM/IEEE Design Automation Conference (DAC). 2020. IEEE.

[5] Clow, J., et al. A pythonic approach for rapid hardware prototyping and instrumentation. in 2017 27th International Conference on Field Programmable Logic and Applications (FPL). 2017. IEEE.

[6] Sozzo, E.D., et al., Pushing the level of abstraction of digital system design: A survey on how to program FPGAs. ACM Computing Surveys, 2022. 55(5): p. 1-48.

[7] Mirza, D., D. Dangwal, and T. Sherwood. PyRTL in Early Undergraduate Research. in Proceedings of the Workshop on Computer Architecture Education. 2019.

[8] Aboye, D., et al. PyRTLMatrix: An object-oriented hardware design pattern for prototyping ML accelerators. in 2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2). 2019. IEEE.

[9] Hanif, M.A., et al., Energy-efficient design of advanced machine learning hardware. Machine Learning in VLSI Computer-Aided Design, 2019: p. 647-678.

[10] Patel, D., et al. Automatic Transpiler that Efficiently Converts Digital Circuits to a Neural Network Representation. in 2022 International Joint Conference on Neural Networks (IJCNN). 2022. IEEE.
[11] Jiang, S., et al., PyMTL3: A Python framework for open-source hardware modeling, generation, simulation, and verification. IEEE Micro, 2020. 40(4): p. 58-66.
[12] Koren, I. and R. Klamma. The exploitation of openapi documentation for the generation of web frontends. in Companion Proceedings of the The Web Conference 2018. 2018.
[13] Sato, S. and K. Kise. ArchHDL: a new hardware description language for high-speed architectural evaluation. in 2013 IEEE 7th International Symposium on Embedded Multicore Socs. 2013. IEEE.
[14] Bachrach, J., et al. Chisel: constructing hardware in a scala embedded language. in Proceedings of the 49th Annual Design Automation Conference. 2012.
[15] Bjesse, P., et al., Lava: hardware design in Haskell. ACM SIGPLAN Notices, 1998. 34(1): p. 174-184.
[16] Chen, Y.-H., et al., Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE journal of solid-state circuits, 2016. 52(1): p. 127-138.
[17] Chung, E., et al., Serving dnns in real time at datacenter scale with project brainwave. iEEE Micro, 2018. 38(2): p. 8-20.
[18] Logaras, E. and E.S. Manolakos. SysPy: using Python for processor-centric SoC design. in 2010 17th IEEE International Conference on Electronics, Circuits and Systems. 2010. IEEE.
[19] Koeplinger, D., et al. Spatial: A language and compiler for application accelerators. in Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2018.
[20] Mahajan, D., et al. Tabla: A unified template-based framework for accelerating statistical machine learning. in 2016 IEEE International
[21] Symposium on High Performance Computer Architecture (HPCA). 2016. IEEE.
[22] Ghaith,A.*,Anas,J. Efficiency of Using the Diffie-Hellman Key in Cryptography for Internet Security. 2021
[23] Talafha, M. ,Alkouri, A., Alqaraleh, S, Zureigat, H .,Aljarrah, A. Complex hesitant fuzzy sets and its applications in multiple attributes decision-making problems. 2020
[24] Razak ,S., Oqla m., Anas ,A., Abd ULazeez ,A. Complex Fuzzy Parameterized Soft Set.2020
[25] Aljrah I,Alomari G &Aljarah A,aljrrah M,Aljarah B. Enhancing Chip Design Performance with Machine Learning and PyRTL ,2023:2147-6799(Accepted)
[26] Ahmed ,S., Anas ,A., Sek Sok, K., Zaidi ,I. Robust estimation and outlier detection on panel data: an application to environmental science.2017.

DOI: https://doi.org/10.15379/ijmst.v10i3.2735