A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation

John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan and Timothy Sherwood
University of California, Santa Barbara, CA, 93106 USA
Email: {jclow, gtzimpragos, deeksha, sguo, jmcmahan, sherwood}@cs.ucsb.edu

***Abstract*—We introduce PyRTL, a Python embedded hardware design language that helps concisely and precisely describe digital hardware structures. Rather than attempt to infer a good design via HLS, PyRTL provides a wrapper over a well-defined “core” set of primitives in a way that empowers digital hardware design teaching and research. The proposed system takes advantage of the programming language features of Python to allow interesting design patterns to be expressed succinctly, and encourage the rapid generation of tooling and transforms over a custom intermediate representation. We describe PyRTL as a language, its core semantics, the transform generation interface, and explore its application to several different design patterns and analysis tools. Also, we demonstrate the integration of PyRTL-generated hardware overlays into Xilinx PYNQ platform. The resulting system provides an almost “pure” pythonic experience for the prototyping and evaluation of FPGA-based SoCs.**

I. Introduction

From Systems-on-Chip to warehouse sized computing, system engineers are increasingly turning to custom and reconfigurable hardware blocks to provide performance and energy efficiency where it counts most. Traditional hardware description languages, such as Verilog and VHDL, were developed more than three decades ago and are still the dominant HDLs. In recent years, High-Level-Synthesis (HLS) approaches are also gaining ground, leading to the market’s expansion to a usership beyond traditional RTL coders to include “non-experts”. However, the tremendous amount of work focusing on how good designs can be inferred via these methods proves that deep hardware understanding is still required, in both cases, for the development of reasonably optimized solutions.

In this paper, we introduce PyRTL. PyRTL is an open-source Python-based HDL aiming to enable the rapid prototyping of complex digital hardware. PyRTL provides a collection of classes for pythonic RTL design, simulation, tracing, and testing, suitable for teaching and research. Simplicity, usability, clarity and extensibility, rather than performance or optimization, are the overarching goals.

At a high level, PyRTL builds the hardware structure that the user explicitly defines. In contrast with HLS approaches, PyRTL does not promise to take “random” high-level code and turn it into hardware. However, our system aims at (a) lowering the barrier of entry to digital design (for both students and software engineers), (b) promoting the co-design of hardware transforms and analysis with digital designs (through a simple core and translation interface), and (c) ultimately allowing complex hardware design patterns to be expressed in a way that promotes reuse beyond just hardware blocks.

To achieve these goals, PyRTL intentionally restricts users to a set of reasonable digital design practices. PyRTL’s small and well-defined internal core structure makes it easy to add new functionality that works across every design, including logic transforms, static analysis, and optimizations. Features such as elaboration-through-execution (e.g. introspection), design and simulation without leaving Python, and the option to export to, or import from, common HDLs (Verilog-in via Yosys [1] and BLIF-in, Verilog-out) are also supported. More information about PyRTL’s high level flow can be found in Figure 1.

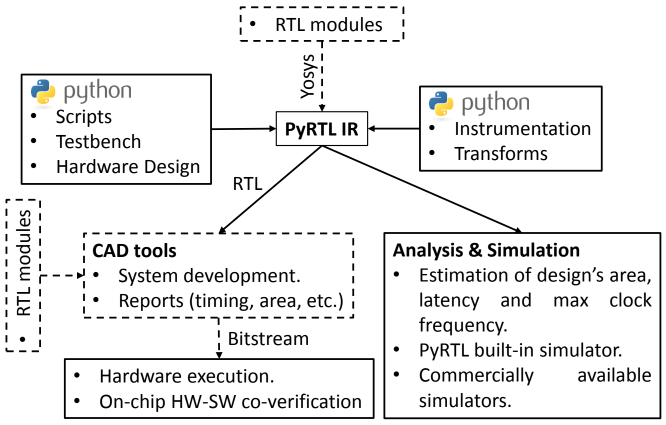

Fig. 1. Overview of PyRTL’s flow.

The paper first gives an overview of the related work, highlighting the similarities and differences between existing projects and our approach. Section III describes the PyRTL language and its main features. In Section IV, PyRTL’s intermediate representation is presented. An instrumentation API along with a set of possible transforms over PyRTL’s core are discussed in Section V. The performance of our tool is quantified in Section VI. The on-board deployment and SoC integration of PyRTL-generated designs are presented in this section as well. Finally, concluding remarks are given in Section VII. Overall, the main contributions of this paper are the following:

- A Python embedded hardware design language that allows the use of dynamic typing, introspection, comprehensions, and other Python features to capture reoccurring design patterns and facilitate extensive reusability.

- A concise intermediate representation that simplifies the co-design of hardware transforms and analysis with digital designs.
- An instrumentation API that supports transforms through an interface similar to software-like binary instrumentation toolkits. The applicability and efficiency of these methods are also demonstrated across a variety of sample designs.
- The integration of PyRTL-generated hardware overlays into Xilinx PYNQ (and therefore Zynq) for quick and easy SoC development and evaluation via Python.

II. Related Work

Traditional HDLs have a very long learning curve, even for experienced engineers. On the other hand, HLS approaches promise to raise the level of abstraction and compile regular C/C++ functions into logic elements. However, there is still no clear way as to how to come up with an optimized design without prior hardware engineering experience. PyRTL's goal is to provide an alternative between these two extremes by enabling programmers to directly specify their hardware designs in a language they already know. As expected, this is not the first attempt to cover this need.

Chisel [2] is a Scala-based project with similar goals to PyRTL. Chisel is (like PyRTL) an elaborate-through-execution hardware design language. With support for signed types, named hierarchies of wires, and a well-designed control structure, Chisel is a powerful tool used in some great research projects, including OpenRISC. Unlike Chisel, PyRTL has concentrated on a complete tool chain, which is useful for instructional projects, and provides a clearly defined and relatively easy to manipulate intermediate structure, which allows rapid prototyping of hardware analysis routines.

ClaSH [3] is a hardware description embedded DSL in Haskell. In a similar way to PyRTL, ClaSH provides an approach suitable for both combinational and synchronous sequential circuits, and allows the transformation of these high-level descriptions to low-level synthesizable Verilog HDL. Unlike PyRTL, which is based on Python (one of the most popular languages with support for both functional and imperative style programming), in ClaSH the user has to deal with the challenges that come with functional programming. Moreover, designs have to be statically typed (like VHDL). To be more specific, in PyRTL variable types do not have to be explicitly declared in source code, but rather they can be inferred during Python program execution; this facilitates the development of more reusable structures.

MyHDL [4] and PyMTL [5] are both Python-based hardware design tools. MyHDL is built around generators and decorators; the semantics of this embedded language are close to Verilog and allow asynchronous logic and higher-level modeling. Much like traditional HDLs, though, only a structural "convertible subset" of the language can be automatically synthesized into real hardware. PyMTL allows simulation and modeling at multiple levels of the design process. Like MyHDL, parsing of the Python AST allows executable software descriptions to be (under certain restrictions) automatically converted into implementable hardware. In contrast with these approaches, PyRTL always uses a single clock domain and does not contain any unsynthesizable hardware primitives (i.e., bottom-up design based on the use of composable set of data structures), which is of great help to less-experienced users, for whom it can be very unclear why certain code can be synthesized and certain code cannot. Moreover, PyRTL introduces a hardware instrumentation framework that provides methods of walking and augmenting accelerator functionality concisely and efficiently (e.g. insertion of counters, probes, and other arbitrary analysis logic).

The SysPy project [6] looks at the hardware realization problem from a different perspective. To bridge the gap between software expressions and hardware implementations, the authors proposed a "glue software" solution between ready-to-use VHDL components and programmable processor soft IP cores. PHDL [7] is also aiming to enable the utilization of available HDL components, importable from pre-existing libraries. Although these approaches are interesting, we do not consider them PyRTL's counterparts, as their focus is different.

III. PyRTL Overview

PyRTL is a new hardware design language based on Python. The main motivation behind PyRTL's design is to help the user concisely and precisely describe a digital hardware structure through a set of Python classes. To achieve simplicity and clarity, PyRTL intentionally restricts users to a set of reasonable digital design practices. For instance, clock and reset signals are implicit, block memories are synchronous by default, there are no undriven or high-impedance states, and no unregistered feedbacks are allowed. That way, any design expressed as valid code always corresponds to synthesizable hardware. Moreover, Python's dynamic and object-oriented nature allows the user to write introspective containers and build hardware using common software abstractions.

A. PyRTL Datatypes & Operators

The primary data structure users interact with is `WireVector`. To allow users to build on their existing experience with Python, we have designed multi-bit `WireVectors` to act much as a list of individual wires. This allows for `WireVectors` to use the various functionality already built-in to Python, such as getting the length of a wire through the built-in `len()` function, creating iterators and comprehensions over `WireVectors`, and retrieving wire subsets through both indexing and slicing.

```
wire2 = wire1[0]   # indexing least significant bit
wire3 = wire1[-1]  # indexing most significant bit
wire4 = wire1[:3]  # slicing bits 0 through 2
```

Rather than require that every `WireVector` in the system has a length (i.e. bitwidth) set explicitly, in PyRTL `WireVectors` can get their length in one of the following three ways: (a) the length can be set by the user at the time of declaration, (b) the length can be inferred from the producing operation when `WireVector` is created, or (c) it can be inferred when the `WireVector` is assigned to.

In PyRTL, we have two different operators that perform similar but distinct operations: the assignment operator $=$ and the connection operator $<<=$. The assignment operator is executed by the Python runtime, binding a name on the left hand side to an object on the right hand side; no hardware is created. In contrast, the connection operator adds a net to the circuit to directionaly connect the two `WireVectors` together; the left hand side becomes driven by the right hand side. Having both operators allows traditional HDL connections along with dynamic changes in names (for iterative and recursive structures).

```
def kogge_stone(a, b, cin=0):
    a, b = libutils.match_bitwidth(a, b)

    prop_orig = a ^ b
    prop_bits = [i for i in prop_orig]
    gen_bits = [i for i in a & b]
    prop_dist = 1

    while prop_dist < len(a):  # creation of the carry calculation
        for i in reversed(range(prop_dist, len(a))):
            prop_old = prop_bits[i]
            gen_bits[i] = gen_bits[i] | (prop_old & gen_bits[i - prop_dist])
            if i >= prop_dist * 2:
                prop_bits[i] = prop_old & prop_bits[i - prop_dist]
        prop_dist *= 2

    gen_bits.insert(0, pyrtl.as_wires(cin))
    return pyrtl.concat(*reversed(gen_bits)) ^ prop_orig
```

Fig. 2. Kogge Stone adder utilizing list-like properties of `WireVector`, as well as Python lists of `WireVectors`.

In the Kogge-Stone [8] example shown in Figure 2, two lists of `WireVectors` are used to iteratively store the *propagate* and *generate* variables, allowing the adder to be described as multiple rounds of operations. Specifically, in the example, we first populate two lists of `WireVectors`: the original lists of *generate* and *propagate* bits. Then, for each power of 2, (denoted as *prop_dist*) from 1 to the bit width of the longer input, we create new *generate* and *propagate* bits from the old ones. The *generate* bits are updated by `OR`-ing them with the result of the `AND` of the *propagate* bit of the same index and the *generate* bit that has an index *prop_dist* less than itself. The *propagate* bit is updated by `AND`-ing it with the *propagate* bit with index $(i - prop\_dist)$, where i is the current loop index. Implicit in this is that all `WireVectors` with index less than *prop_dist* are not updated. After we create the final *generate* and *propagate* signals, we shift the *generate* wire list over by one to add the carry in to it. Finally, we `XOR` the *generate* bits with the original *propagate* bits to create the result of the addition. Note that while the ability to operate on a list of `WireVectors` made it easy to implement the concept of "updating" some of the `WireVectors`, we are actually updating only the Python binding of the name, without "updating" the `WireVectors` themselves. The logic described in Figure 2 is fully combinational, and as the function executes it wires together this complex adder structure.

As memories are complex but critical elements of hardware designs, they also have their own construct. In PyRTL, block memories appear as an array of registers. They are written as registers by default (e.g., `mem[index] <<= 5`) and values become available in the following clock cycle. The `Register` class acts just like a `WireVector`, meaning they can be used in arbitrary expressions.

B. *Instrospection and Hardware Comprehensions*

In traditional hardware design languages, creating reusable code relies on reusable or extensible modules — blocks that correspond to a block of hardware. Capturing design patterns or behaviors that don't belong to a particular single block is difficult in these languages, but the flexibility of Python and PyRTL allow us to effectively abstract structured behavior in addition to traditional blocks. This tends to rely on introspection, where some code introspects and examines runtime values in objects. We use this feature to implement a set of `Pipeline` classes that abstract the common, structured behavior of pipelines, such as inserting pipeline registers and stopping on stalls. A designer instantiates an object of the class and uses it as a container for the circuit signals. The object tracks which stage is "active:" newly created logic and wires belong to that stage. Calling a `next_stage()` function "finishes" the logic for the stage and advances the active stage. When the designer attempts to use a wire and access an attribute of the object, it searches for the name in the current and all previous pipelines stages. If the wire was created in a previous stage, it automatically inserts pipeline registers to bring it to the current stage. The overhead of tracking which values need to be buffered, instantiating pipeline registers, and inserting stall logic (on user-defined conditions) is handled by the `Pipeline` class, allowing the user code to consist solely of the actual logic of each stage, delineated by calls to `next_stage()`.

Besides making the code cleaner, this feature allows the pipeline to be tested independently of the application logic. It also helps users develop structures that are easy to reuse and extend: for example, we added support for forwarding with a class extension allowing wires in different stages to be declared as sources and destinations of a forward, circumventing the normal buffer-creation process for just those wires; it required minimal code modifications. We use these classes in the implementation of a five-stage pipelined MIPS processor.

Complex interwoven structures are especially cumbersome to specify in many hardware design languages. Python's comprehension syntax is particularly useful in such cases. An implementation of AES decryption, for example, which performs data unscrambling operations on 8-bit data blocks, can concisely be performed using comprehensions in just a few lines of code, as seen in Figure 3 (Bhargav et al.'s implementation [9] gives us an intuition about its VHDL equivalent). The 128-bit input vector is partitioned into 8-bit slices and stored as a list. The elements of this list are then scrambled using `pyrtl.concat()` to return one single `WireVector` *out_vector*.

```
def inv_shift_rows(in_vector):
    a = [in_vector[offset - 8:offset] for offset 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
```

Fig. 3. Concise code using comprehensions to implement the data unscrambling operation of Advanced Encryption Standard (AES).

IV. PyRTL IR

The goal of PyRTL's intermediate representation is to provide a complete set of operations and structures for the description and manipulation of hardware, without complicating factors. Users that want to use PyRTL to define a hardware design do not have to bother with this representation at all. However, knowledge of how our compiler works can be useful for more complex operations, such as efficient hardware instrumentation.

A. Circuit Logic & Interconnection

PyRTL provides two built-in data structures, `Block` and `WireVectors`, to describe hardware in a bottom-up way. A `Block` is a container composed of primitive operations and stores both basic logic elements and references to their interconnection. Each logic element is stored as a `LogicNet`, a 4-tuple of:

- A primitive operation represented with a single character.
- A set of any additional parameters that the operation may require (such as the indexes used by the 's' operator described below). These parameters cannot change at runtime.
- A tuple of arguments listing the `WireVectors` connected as inputs for the particular operation.
- A tuple of destinations, which list the `WireVectors` driven as outputs for this particular operation.

Regarding interconnection, `WireVectors` represent a bundle of wires that act much like a Python list of 1-bit nets. Our IR contains five different types of `WireVectors`: Basic, Input, Output, Const, and Register. Basic `WireVectors` connect two or more different logic elements together. Input and Output `WireVectors` represent dynamic input and output signals of the circuit. The Const wires represent fixed values in the circuit. Register `WireVectors` store the value from its source for the next cycle.

B. Basic Operations

The complete list of primitive operations is shown below. All of the properties described are checked as the circuit is constructed to certify that the working model of hardware is always valid. This ensures that user-defined transforms never violate the IR semantics and create invalid hardware states.

- Logical and arithmetic operations have their standard definition, each taking exactly two arguments and performing the arithmetic or logical operation specified. This includes `AND`, `OR`, `XOR`, and unsigned addition, subtraction, and multiplication ($\& \mid \wedge + - *$). All inputs must be the same bitwidth. Logical operations produce `WireVectors` with as many bits as the input, while $+$ and $-$ produce $(n+1)$ bits, and $*$ produces $2n$ bits.
- In addition, basic comparison operations are supplied. The = op checks to see if the bits of the vectors are equal, while $<$ and $>$ do unsigned arithmetic comparisons. All comparisons generate a `WireVector` of length 1.
- The `w` operator is simply a buffer with no logic function, while `n` is an inverting buffer (NOT gate).
- The `x` operator is a MUX, which takes a single select bit (`WireVector` of length 1) and two other `WireVectors` of arbitrary but equal length. If the value of the first argument (select bit) is 0, it selects the second argument; if it is 1, it selects the third argument.
- The `c` operator is the concat operator and combines any number of `WireVectors` (a_0, a_1,..., a_n) into a single new `WireVector` with a_0 in the MSB and a_n in the LSB position.
- The `s` operator is the selection operator and chooses, based on the constant parameters specified, a subset of the logic bits from a wire vector to select. Repeats are accepted.
- The `r` operator is a register. On positive clock edges, it simply copies the value from the input to the output of the register.
- The `m` operator is a single memory read port, which supports asynchronous or synchronous reads (acting like combinational logic). Multiple read (and write) ports are possible on the same memory. The extra parameters field holds a tuple containing two references: a memory identifier, and a reference to the memory instance containing this port. Each read port additionally has one address (an argument) and one data (a destination).
- The `@` operator is a memory write port, supporting synchronous writes (writes are positive edge triggered). As with read ports, each `@` defines only one write. The parameters are the same as the read port: the mem id, and the memory instance. Writes have three input arguments: address, data, and write-enable. Written value changes are not applied until the following cycle.

A summary of PyRTL's "core" set of primitives is shown in Table I.

Primitives	Number of inputs	Number of outputs	Output length (bits)
{and, or, xor}	2	1	max(len(inputs))
{add, sub}	2	1	max(len(inputs)) + 1)
mult	2	1	2*max(len(inputs))
{lt, gt, equal}	2	1	1
{not, wire}	1	1	len(input))
mux	3	1	max(len(inputs))
concat	n	1	sum(len(inputs))
bitselect	1 (+tuple)	1	tuple length
register	1	1	len(input)
memread	1	1	datawidth
memwrite	3	0	-

TABLE I
PyRTL's core set of primitives.

V. Instrumentation & Transforms

Binary instrumentation is a method commonly used by software engineers to uncover performance bottlenecks and identify bugs, race conditions, and how information flows through their system. Given its usefulness, one may ask how to easily extend the idea of instrumentation to reconfigurable logic accelerators. In our experience, Verilog is a poor language on which to develop an instrumentation platform. Doing even simple modifications to the netlist involves fragile custom scripts or lots of manual work. In addition, there is no common representation for some commonly used objects, such as block memories; instead, these are often described in vendor specific formats. For these reasons, we opted to use our custom intermediate representation to simplify the development and use of instrumentation tools and support transforms. In contrast with other HDLs, PyRTL is actually built to enable direct designer manipulation of the IR. Although this type of manipulation can affect the original design's performance, it opens the door to the rapid evaluation of research questions, structured optimizations and transforms, static analysis, and deep debugging.

A. Instrumentation API

Despite the simplicity of PyRTL's IR, there are still some complexities involved in making tools that modify the hardware efficiently. Naive implementations of such functionality often have bad performance when scaling, or bugs appear in corner cases. To facilitate the creation of instrumentation tools, we provide a set of easy to use API calls to get commonly needed information, as well as do common modifications to the hardware block.

1) Net Connections: According to the PyRTL IR description provided above, `LogicNets` contain information about a set of logic elements and the wires that are connected to them. The wires, though, do not store any information regarding which nets they are connected to. To alleviate potential pain points due to this limitation, we provide net connections. This function returns a dictionary that, for each wire, notes which net is its source and which nets use the wire. With both this information and the information contained in the nets themselves, an instrument is able to efficiently traverse and transform the circuit.

2) Wire and Logic Replacement: All of the instrumentation tools involve modifications to the original hardware design. These actions can range from replacing a logic operation to adding in extra instrumentation to an existing circuit design. Adding new logic and wires is trivial; however, modifying existing logic can become complicated. To address this issue, our framework provides two API functions to facilitate the replacement of existing hardware: `wire_transform()` and `net_transform()`. Both of these functions take as input a function that maps a single wire or net to new wires and nets (with which the original will be replaced). This allows the user to focus on the changes that need to be done instead of their integration to the rest of system.

3) Data-flow Respecting Iterator: While some forms of instrumentation depend only on an unordered list of circuit elements, for many others a well-defined data-flow preserving iteration order can be extremely useful. Without such an ordering, many transformations would require extra checks in order to verify whether required related logic elements were already created, and generate them if not present. For example, in the case of a Gate Level Information Flow Tracking (GLIFT) analysis [10], such a data-flow respecting iterator guarantees that the tainted wires for a gate's input operations already exist and can be referenced when adding the GLIFT tracking for each logic operation. Our framework supports this feature as a default iterator on a `Block`.

B. Transforms

PyRTL supports transforms explicitly through an interface similar to binary instrumentation toolkits [11], [12], [13] popular in the software world. The back end of the transform interface takes care of the hard question of how to replace wires and nets and stitch the circuit back together. We note that the majority of transforms can be categorized into two major types: (a) operation transforms, and (b) connection transforms.

1) Operation Transformer: The operation-transformer is a higher order function which takes as input a block to be transformed and a procedure (the transform function), which will be called on every `LogicNet` in the system. The transform function takes as argument a reference to a specific `LogicNet`, which it will modify as needed (typically by adding to it). In addition, the transform function must return a Boolean, which will let the operation-transformer know if the original `LogicNet` should be kept in the `Block` . (The identity transform function is simply `lambda x: True`.) In this way, it is easy to augment or modify designs on an operation-by-operation basis. Error coding, information flow analysis, and cryptographic analysis all follow this pattern.

NAND Synthesis Example:
As an example of the operation transformer, consider the problem of lowering a design to a subset of functionality. For example, consider the problem of lowering `and`/`or`/`not` gates down to a set of `nand` gates as a simple transformation, shown in Figure 4.

```
def nand_synth_op(net):
    if net.op in '~nrwcsm@':
        return True # aka, keep original net
    def arg(num):
        return net.args[num]
    dest = net.dests[0]
    if net.op == '&':
        dest <<= ~(arg(0).nand(arg(1)))
    elif net.op == '|':
        dest <<= (~arg(0)).nand(~arg(1))
    elif net.op == '^':
        temp_0 = arg(0).nand(arg(1))
        dest <<= temp_0.nand(arg(0)).nand(temp_0.nand(arg(1)))
    return False
```

Fig. 4. Reduction to NAND gates using operation transforms

2) Connection Transformer: Like the operation-transformer, the connection-transformer is a higher order function which takes both a block and a transform function as input. In PyRTL, a `WireVector` is really a net; it should have exactly one driver but may have many consumers. Replacing a specific connection is thus a bit more involved. The transform function takes as input a `WireVector`, and returns a pair of `WireVectors`, `(src,dest)`, which corresponds to a new consumer and producer for that connection "slot". Thus the identity for this transform is a function `lambda x:(x,x)`, which says the original wire should connect the same thing it used to. However, one can use this transform to replace a connection with arbitrary logic. This pattern comes up in debugging, re-factoring, and error modeling [14].

Modeling Transient Errors Example:
Using the transform tools, we can easily treat our hardware designs as both code and data to perform analysis and experimentation. For example, one such question would be: does the behavior of a specific design change when in the face of transient errors? Such a question can be used to determine where error correction can be most judiciously applied [15], or to understand the effect of transient errors on application-level metrics of performance [16]. We can write the transform very simply on a PyRTL design as shown in Figure 5. For example, if we want to quantify the effect of some fraction of potentially vulnerable wires (0.001 in the example), we simply need to `XOR` those `WireVectors` with a source of run-time transient noise (the transient function generates such a `WireVector` on demand). While this gives a very simple example, it should be easy to see how this could be trivially extended to support stuck-at faults or just errors on specific types of connections based on other models.

```
rate = 0.001  # fraction of vulnerable wires
def transient_error(wire):
    clone = transform.clone_wire
    if random.random() < rate:
        new_src, new_dst = clone(wire), clone(wire)
        new_dst <<= new_src ^ transient()
        return new_src, new_dst
    return wire, wire
transform.wire_transform(transient_error)
```

Fig. 5. Inserting the possibility of transient errors into a randomly chosen set of connections in a PyRTL design for testing and analysis.

VI. Performance Analysis & SoC Integration

PyRTL's goal is to provide the community with a programming language and interface able to empower digital hardware design teaching and research. Therefore, simplicity, usability, clarity and extensibility are our system's main overarching goals.

A. PyRTL's Performance

PyRTL has been used so far for the design of both simple and complex hardware systems. The latest example is the Zarf architecture for recursive functions [17], recently presented by McMahan et al. Among other sample designs developed with PyRTL are: a 32-bit five-stage MIPS pipeline, the AES block cipher algorithm for decryption, and a 32 bit Wallace tree multiplier.

Table II shows how our system performs in terms of simulation time when compared to existing popular solutions, such as Icarus [18] and Mentor Graphics ModelSim, that use traditional HDLs. Our simulator first generates C code as well as an interface to Python to pass results back to it. The compilation of the generated code directly updates the machine state, removing the overhead of calculating the wire evaluation order at run time (walking the circuit graph at every step).

Design	PyRTL	Icarus Verilog	ModelSim
MIPS Pipeline	2.641s	1.327s	1.990s
AES	2.695s	55.065s	7.644s
Wallace Tree Multiplier	0.835s	0.654s	0.521s

TABLE II
Simulation time for PyRTL designs and comparable hand-coded Verilog.

Summarizing these examples, we also observe a significant reduction in terms of code size when compared to traditional RTL implementations. For example, we implemented AES block (128 byte key) using PyRTL in 303 lines of code (LOC), whereas Strombergson's [19] (it supports both 128 and 256 byte keys) and Usselmann's [20] Verilog implementations are 2729 (PyRTL code is 9x denser) and 1405 (PyRTL code is 4.6x denser) LOC long, respectively. Regarding MIPS pipeline, PyRTL implementation is 655 LOC, when 936 and 2419 LOCs where required for its Verilog (PyRTL code is 1.42x denser) [21] and VHDL (PyRTL code is 3.7x denser) [22] implementations.

B. SoC Integration

As a proof-of-concept, we are also using the Xilinx PYNQ board to demonstrate the quick and easy SoC integration and on-board evaluation of PyRTL-generated designs.

PYNQ is an open-source project that provides an easy software interface and framework for Zynq designs. Developers can use Python (in Jupyter Notebook) to load hardware libraries and overlays on the programmable logic in order to speed up software running on the board's ARM processor. Although PYNQ makes it easier for embedded systems designers to exploit the benefits of hardware acceleration in their applications, it does not provide any new way of creating these overlays. Considering that one of this paper's goals is to demonstrate the prototyping and evaluation of FPGA-based SoCs using Python, the "under-test" accelerated hardware designs were developed in PyRTL.

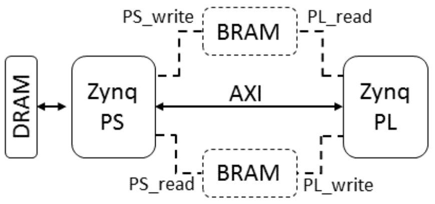

Fig. 6. System architecture.

Figure 6 shows an overview of our system architecture. To be more specific, an ARM Cortex-9 processor is responsible for running software, and the loading and control of the hardware core. Moreover, the use of two dual-ported BRAMs allows the data transfer between the Programmable System (Zynq PS) and Programmable Logic (Zynq PL) parts (where our PyRTL-generated hardware cores are mapped). As can be seen, AXI-Lite is also used for the transfer of control signals to our hardware core. Generally, AXI protocol can be considered as an alternative to the BRAM interface for the data transfer between the PS and PL parts. Considering that PyRTL exports Verilog code, the designer should use common practices to wrap and package the generated hardware IPs before starting the system development.

The described solution is actually applicable to any Zynq platform. However, PYNQ allows the use of Python for the programming of the board's on-chip processor. This feature simplifies our hardware's verification as we do not have to leave Python to use hardware libraries and overlays. More specifically, we can just use the same Python validation code for both software simulation and on-board hardware evaluation. Taking advantage of Python's multi-threading feature, the user can run a Python golden reference model of our design in "parallel" with its hardware equivalent and make cross-comparisons "locally". Besides speed, this approach helps the user verify the communication between PS and PL, especially in cases where the on-chip processor is offloading part of its workload to the FPGA.

VII. Conclusion

Python has easily become one of the most popular programming languages world wide. PyRTL is an open-source Python-based HDL aiming to enable rapid prototyping, instrumentation and analysis of complex digital hardware systems. Opening the problem of digital design both to a broader community of "non-experts" and to students earlier in their engineering training has the potential to help and empower designers as the traditionally hard line between hardware and software continues to blur. With this approach, interesting design patterns can be expressed concisely, and it encourages the rapid co-generation of tooling and transforms on the intermediate representation. We describe the underlying mechanisms, the opportunities for generalization, and techniques for instrumenting such designs in PyRTL. The integration of PyRTL-generated hardware overlays into Xilinx PYNQ platfrom also shows the potential to design FPGA-based embedded systems and SoC validation and test environments using Python. The full system is open source and available for install via `pip`.

References

[1] C. Wolf, "Yosys manual."

[2] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1216–1225.

[3] C. Baaij, "Clash: From haskell to hardware," Master's thesis, University of Twente, 2009.

[4] J. Decaluwe, "Myhdl: a python-based hardware description language," *Linux journal*, vol. 2004, no. 127, p. 5, 2004.

[5] D. Lockhart, G. Zibrat, and C. Batten, "Pymtl: A unified framework for vertically integrated computer architecture research," in *47th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014, pp. 280–292.

[6] E. Logaras and E. S. Manolakos, "Syspy: using python for processor-centric soc design," in *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on*. IEEE, 2010, pp. 762–765.

[7] A. Mashtizadeh, "Phdl: A python hardware design framework," Ph.D. dissertation, Massachusetts Institute of Technology, 2007.

[8] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE transactions on computers*, vol. 100, no. 8, pp. 786–793, 1973.

[9] S. Bhargav, L. Chen, A. Majumdar, and S. Ramudit, "FPGA-based 128-bit AES decryption," 2008. [Online]. Available: https://http://www.cs.columbia.edu/ sedwards/classes/2008/4840/reports/AES.pdf

[10] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," *SIGPLAN Not.*, vol. 44, no. 3, pp. 109–120, Mar. 2009. [Online]. Available: http://doi.acm.org/10.1145/1508284.1508258

[11] A. Srivastava and A. Eustace, *ATOM: A system for building customized program analysis tools*. ACM, 1994, vol. 29, no. 6.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[13] N. Nethercote, R. Walsh, and J. Fitzhardinge, "Building workload characterization tools with valgrind," 2006. [Online]. Available: http://valgrind.org/docs/iiswc2006.pdf

[14] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin, "Crashtest: A fast high-fidelity fpga-based resiliency analysis framework," in *Computer Design, 2008. ICCD 2008. IEEE International Conference on*. IEEE, 2008, pp. 363–370.

[15] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003, p. 29.

[16] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain¡ t¿: A first-order type for uncertain data," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 51–66, 2014.

[17] J. McMahan, M. Christensen, L. Nichols, J. Roesch, S.-Y. Guo, B. Hardekopf, and T. Sherwood, "An architecture supporting formal and compositional binary analysis," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16, 2017.

[18] S. Williams and M. Baxter, "Icarus verilog: Open-source verilog more than a year later," *Linux J.*, vol. 2002, no. 99, pp. 3–, Jul. 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=513581.513584

[19] J. Strombergson, "Verilog implementation of the symmetric block cipher AES," 2017. [Online]. Available: https://github.com/secworks/aes

[20] R. Usselmann, "AES (Rijndael) IP Core," 2016. [Online]. Available: https://opencores.org/project,aes_core

[21] J. Mahler, "MIPS CPU implemented in Verilog," 2016. [Online]. Available: https://github.com/jmahler/mips-cpu

[22] E. Lujan, "VHDL Implementation of a basic Pipeline MIPS processor," 2016. [Online]. Available: https://opencores.org/project,vhdl-pipeline-mips