PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification

Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten
Cornell University

***Abstract*—In this article, we present PyMTL3, a Python framework for open-source hardware modeling, generation, simulation, and verification. In addition to compelling benefits from using the Python language, PyMTL3 is designed to provide flexible, modular, and extensible workflows for both hardware designers and computer architects. PyMTL3 supports a seamless multilevel modeling environment and carefully designed modular software architecture using a sophisticated in-memory intermediate representation and a collection of passes that analyze, instrument, and transform PyMTL3 hardware models. We believe PyMTL3 can play an important role in jump-starting the open-source hardware ecosystem.**

Due to the breakdown of transistor scaling and the slowdown of Moore's law, there has been an increasing trend toward energy-efficient system-on-chip (SoC) design using heterogeneous architectures with a mix of general-purpose and specialized computing engines. Heterogeneous SoCs emphasize both flexible parameterization of a single design block and versatile composition of numerous different design blocks, which have imposed significant challenges to state-of-the-art hardware modeling and

Digital Object Identifier 10.1109/MM.2020.2997638
Date of publication 25 May 2020; date of current version 30 June 2020.

0272-1732 © 2020 IEEE

Authorized licensed use limited to: Cornell University Library. Downloaded on July 03,2020 at 00:53:45 UTC from IEEE Xplore. Restrictions apply.

verification methodologies. Developing, open-sourcing, and collaborating on *hardware generators* is a compelling solution to increase the reuse of highly parametrized and thoroughly tested hardware blocks in the community. However, the general lack of high-quality open-source hardware designs and hardware verification methodologies have been a major concern that limits the widespread adoption of open-source hardware.

To respond to these challenges, the open-source hardware community is augmenting or even replacing traditional domain-specific hardware description languages (HDLs) with productive hardware development frameworks empowered by high-level general-purpose programming languages such as C++, Scala, Perl, and Python. Hardware preprocessing frameworks (e.g., Genesis2)[1] intermingle a high-level language for macro-processing and a low-level HDL for logic modeling, which enables more powerful parametrization, yet creates an abrupt semantic gap in the hardware description. Hardware generation frameworks completely embed parametrization and logic description in a unified high-level "host" language,[2] but still generates and simulates low-level HDL code. This requires test benches to be written in the low-level HDL, which creates a modeling/simulation language gap that may require the designer to frequently cross language boundaries during iterative development. All these challenges have inspired completely unified hardware generation and simulation frameworks where parametrization, static elaboration, test benches, behavioral modeling, *and* a simulation engine are all embedded in a general-purpose high-level language.[3,4] High-level synthesis (HLS) is an alternative approach that seeks to automatically synthesize software-oriented programs written in C++ into low-level HDL implementations.[5] We see HLS as complementary to the emerging trend toward hardware generation and simulation frameworks, since any realistic SoC will require a mix of blocks well-suited to HLS (e.g., well-structured data-processing blocks, low-performance control blocks) and blocks that require designers to control more hardware details (e.g., processors, memory hierarchies, networks-on-chip, and complex accelerators). Our previous work presents a detailed comparison of contemporary approaches.[6]

To further improve the productivity of both hardware designers and computer architects, we have built PyMTL3, an open-source Python-based hardware modeling, generation, simulation, and verification framework.

At the same time, computer architects are using open-source cycle-level (CL) modeling methodologies such as SystemC and Cascade[7] to facilitate rapid design-space exploration of large SoCs before creating RTL implementations. When moving from CL to RTL, the ability to support seamless *multilevel modeling* (i.e., mix and match RTL models with CL models) provides significant productivity benefits. For each individual design block, the CL model can serve as the golden reference model, which means all the unit tests can be reused to test the RTL model. Moreover, in a development flow with continuous integration, gradually replacing existing CL blocks with newly developed RTL blocks in a large design while maintaining the integration tests, end-to-end tests, and performance regressions significantly reduces the integration effort and steadily improves the performance accuracy of the overall model.

To further improve the productivity of both hardware designers and computer architects, we have built PyMTL3, an open-source Python-based hardware modeling, generation, simulation, and verification framework. PyMTL3 supports seamless multilevel modeling across register-transfer level (RTL), CL, and functional level (FL) to enable simulating critical models in RTL with noncritical CL/FL behavioral models. Note that PyMTL3 supports generic multilevel modeling, while previous work on architecture description languages is domain-specific and mostly focuses on processor modeling.[8] PyMTL3's predecessor, PyMTL2,[4] has been extensively used in several undergraduate and graduate courses, many research papers, and three chip tape-outs in IBM 130 nm, TSMC 28 nm, and TSMC 16 nm. The design philosophy of PyMTL3 incorporates two important takeaways from PyMTL2: 1) *modularity* of the framework is

Authorized licensed use limited to: Cornell University Library. Downloaded on July 03,2020 at 00:53:45 UTC from IEEE Xplore. Restrictions apply.

the key to creating a vibrant and evolving open-source hardware development ecosystem; and 2) *interoperability* with other open-source tools is the key to achieving widespread adoption. To provide flexible, modular, and extensible workflows, PyMTL3 is designed to have a strictly modular software architecture. Specifically, PyMTL3 separates the PyMTL3 embedded domain-specific language (DSL) that constructs PyMTL3 models, the PyMTL3 in-memory intermediate representation (IMIR) that systematically stores hardware models and exposes APIs to query/mutate the elaborated model, and PyMTL3 passes that are well-organized programs to analyze, instrument, and transform the PyMTL3 IMIR using APIs. While maintaining the key modeling features of PyMTL2, PyMTL3 also includes unified modular ordering constraints (UMOC) for seamless multilevel modeling, a new parameter configuration system, first-class method-based interfaces, polymorphic interface connections, and faster simulation performance using the PyPy just-in-time compiler. PyMTL3 leverages the latest Python 3 features where PyMTL2 only works on Python 2.

PyMTL3 is an ideal framework to jump-start the open-source hardware ecosystem for three major reasons.

- *PyMTL3 is embedded in Python.* Python is currently the most popular programming language for its high productivity. Python has been evolving for nearly three decades, supported by a large *open-source* community with over 100 000 third-party libraries. PyMTL3 users can use these third-party libraries to build test benches, golden reference models, and passes. For example, PyMTL3 analysis passes can leverage matplotlib and graphviz to visualize characteristics of hardware designs. Open-source hardware built in PyMTL3 can also directly reuse Python's package-management system pip for distribution. For example, installing PyOCN[9] (an open-source on-chip network generator built with PyMTL3) involves a single command (pip install pymtl3-net), during which pymtl3 and other dependencies are automatically installed.
- *PyMTL3 emphasizes interoperability with other open-source hardware tools.* A significant amount of open-source hardware is written in Verilog or SystemVerilog. Verilator is currently the fastest and most capable open-source simulator for synthesizable Verilog and SystemVerilog. Unfortunately, Verilator requires driving these simulations with low-level C++. PyMTL3 passes can automatically use Verilator to import Verilog and SystemVerilog models into PyMTL3 for black-box co-simulation. This enables PyMTL3 to combine the familiarity of Verilog/SystemVerilog with the productivity of Python. PyMTL3 passes can also support black-box co-simulation with SystemC, translate RTL models to Yosys-compatible or Verilator-compatible SystemVerilog, and generate GTKWave-compatible waveforms. We have also implemented a FIRRTL[10] backend that generates PyMTL3 models.
- *PyMTL3 promotes agile and test-driven design methodologies.* PyMTL3 adopts pytest, a mature full-featured Python testing tool to collect, manage, parametrize, and refactor tests. PyMTL3 also includes the PyH2 framework that repurposes hypothesis, a property-based testing (PBT) framework for Python software, to test hardware generators (PyH2G), processors (PyH2P), and hardware data structures (PyH2O). Currently, there is no standard verification methodology for open-source hardware. Open-source simulators (e.g., Verilator and Icarus Verilog) have limited support for industry standard verification methodologies (e.g., UVM). cocotb embeds Python in a Verilog simulator, which can limit the use of Python features. PyMTL3 takes the opposite approach by embedding Verilog in Python using Verilator, which unleashes the full potential of the Python runtime. Additionally, cocotb only targets building test benches, while PyMTL3 is a full-fledged modeling framework. Combining the familiarity of Verilog/SystemVerilog with the productivity features of Python, PyMTL3 realizes the agile hardware manifesto.[11]

PyMTL3 WORKFLOW

Figure 1(a) illustrates an example PyMTL3 workflow. The designer starts from developing an FL design-under-test (DUT) and test bench

Authorized licensed use limited to: Cornell University Library. Downloaded on July 03,2020 at 00:53:45 UTC from IEEE Xplore. Restrictions apply.

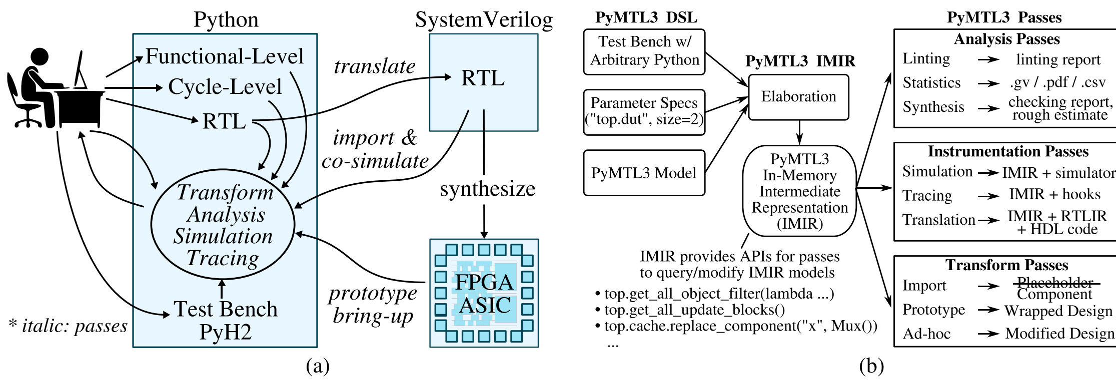

Figure 1. PyMTL3 Overview. (a) PyMTL3 Workflow. (b) PyMTL3 Framework.

(TB) completely in Python. Then, the DUT is manually refined to a CL and/or RTL model. The designer simulates and evaluates the DUT/TB composition, and debugs the FL/CL/RTL DUT leveraging various tracing output. The designer can also leverage the built-in PyH2 PBT framework to find minimal failing test cases. Meanwhile, the designer uses the existing analysis tools or creates new ones to assist iterative refinement. The designer may temporarily transform the hardware model to replace modules or add new logic without modifying the original design. After iterating in the pure-Python environment, the designer invokes translation backends to generate SystemVerilog code and import it back to PyMTL3 for co-simulation with the same TB. Finally, the designer can push the translated SystemVerilog code through an FPGA/ASIC toolflow, and use a prototype proxy that PyMTL3 generates based on the original DUT to test the FPGA/ASIC prototype using the same TB. Designers who only write SystemVerilog code can still benefit from most of the productive workflow steps through PyMTL3's SystemVerilog import. Computer architects may iterate more in CL modeling and only implement RTL for critical parts.

PyMTL3 FRAMEWORK

The goal of PyMTL3 is to create a flexible, modular, and extensible framework that not only allows the designers to select "flow steps" to form their own suitable workflow, but also accommodates the ever-growing wishlist of RTL designers and computer architects with lightweight changes to the existing codebase. To this end, we take inspiration from LLVM and design PyMTL3 to be a strictly modular framework that separates front-end embedded DSL, intermediate representation (IR), and passes. Figure 1(b) shows the software architecture of PyMTL3. The PyMTL3 embedded DSL exposes the modeling primitives to the designer for describing hardware, creating test benches, and configuring parameters. PyMTL3 is responsible for elaborating the hardware model and creating an IMIR that exposes APIs to query/modify the stored metadata of the whole hierarchical model. Compared to existing hardware IRs (e.g., FIRRTL,[10] CoreIR[12]) that focus on representing *circuits*, PyMTL3 IMIR provides a *model-level* view of the whole design hierarchy for not only the RTL circuits, but also CL/FL methods and update blocks which can sometimes include arbitrary Python code. While any Python program could invoke IMIR APIs, PyMTL3 passes are *systematic* programs that interact with PyMTL3 IMIR. PyMTL3 passes are generally categorized into *analysis*, *instrumentation*, and *transform* passes. *Analysis* passes simply analyze the PyMTL3 IMIR model and generate useful outputs. *Instrumentation* passes enhance the model with additional functionalities without modifying the model hierarchy. *Transform* passes mutate the model hierarchy by adding/removing/replacing part of the model.

PyMTL3 EMBEDDED DSL

Lines 1–28 of Figure 2 show the PyMTL3 implementation of a registered incrementer unit and a parametrized N-stage registered incrementer using PyMTL3 embedded DSL primitives. The rest of this section focuses on the distinctive

Authorized licensed use limited to: Cornell University Library. Downloaded on July 03,2020 at 00:53:45 UTC from IEEE Xplore. Restrictions apply.

```
 1 # Creating hardware using PyMTL3 embedded DSL
 2 class RegIncr( Component ):
 3   def construct( s, Type, inc=1 ):
 4     s.in_ = InPort ( Type )
 5     s.out = OutPort( Type )
 6
 7     s.tmp = Wire( Type )
 8     @update_ff
 9     def seq_reg():
10       s.tmp <<= s.in_
11
12     @update
13     def comb_out():
14       s.out @= s.tmp + inc
15
16 class RegIncrNstage( Component ):
17   def construct( s, Type=Bits32, N=1 ):
18     s.in_ = InPort ( Type )
19     s.out = OutPort( Type )
20
21     s.rs = [ RegIncr( Type ) for _ in range(N) ]
22
23     connect( s.rs[0].in_,  s.in_ )
24     connect( s.rs[-1].out, s.out )
25
26     for i in range(N-1):
27       # //= is syntactic sugar for connect
28       s.rs[i].out //= s.rs[i+1].in_
29
30 # Parametrization using PyMTL3 embedded DSL
31 dut = RegIncrNstage( Bits16, 3 )
32 dut.set_param( "top.rs[0].construct", inc=5  )
33 dut.set_param( "top.rs[2].construct", inc=13 )
34
35 # Static elaboration to create PyMTL3 IMIR
36 dut.elaborate()
37
38 # Apply PyMTL3 passes on the IMIR model
39 dut.apply( RefactoringAnalysisPass() )
40 dut.apply( CheckInferedLatchPass() )
41 dut.apply( SimulationPass() )
```

Figure 2. PyMTL3 code example.

modeling features in PyMTL3 that are not found in existing frameworks (including PyMTL2).

Unified Multilevel Scheduling

PyMTL3 provides three sets of primitives for FL, CL, and RTL modeling. FL/CL update blocks communicate through methods, and RTL update blocks communicate through signals. PyMTL3 deploys a novel scheme, UMOC, to schedule FL/CL/RTL update blocks together under the same abstraction. The intracycle ordering of RTL update blocks is implicitly inferred from the signals that each block reads or writes. The intracycle ordering of CL/FL update blocks is deduced from local explicit ordering constraints between method and/or update blocks, and the information of the methods each update block calls. The user can simply set explicit ordering constraints in each component. The simulation passes will handle all the ordering constraints globally. UMOC eliminates the need to manually schedule CL update blocks to model the desired behavior and is the key mechanism in PyMTL3 to support seamless multilevel modeling.

Highly Parametrized Static Elaboration

Python's object-oriented programming and dynamic typing features enable PyMTL3 users to intuitively parametrize hardware components, as opposed to using low-level HDL's limited parametrization constructs and static typing. The users can use parameters of arbitrary types and instantiate different models or update blocks based on value or type. Moreover, PyMTL3 provides a powerful parameter configuration system to solve the common pitfall of parametrizing a hierarchical design. Usually the designer must pass the same parameter from the top-level design through the entire hierarchy. In PyMTL3, the designer can instead specify the parameter at the top-level component using a string with wildcard selection. PyMTL3 will resolve simple regular expressions and distribute the parameters accordingly. Lines 32–33 of Figure 2 show how the individual RegIncr components in the array are configured. In practice, this system can significantly reduce the chance of misconfiguration in a complex SoC composed by many hardware generators.

Polymorphic Interface Connections

PyMTL3 interfaces are bundles of value ports or method ports. By default, connecting two interfaces involves recursively connecting nested interfaces and port pairs with the same name. However, the designer may want to insert an adapter between two incompatible interfaces. In highly parametrized PyMTL3 design generators, manually inserting such adapters is tedious and error-prone due to the verbose type introspection code that checks for matching interface pairs and duplicated code across different components that instantiate the same interface pair. For example, composing any FL/CL/RTL components often involves inspecting the interface type and inserting the corresponding cross-level adapters. To solve this problem, PyMTL3 allows the interface designer to provide a customized connect method in the interface class to centralize type introspection and adapter insertion code. When connecting two interfaces, PyMTL3 *automatically* invokes the

Authorized licensed use limited to: Cornell University Library. Downloaded on July 03,2020 at 00:53:45 UTC from IEEE Xplore. Restrictions apply.

Analysis Passes
- Linting
  - `CheckSignalNamePass`
  - `CheckUnusedSignalPass`
- Statistics
  - `RefactoringAnalysisPass`
  - `DumpUDGPass`
- Pre-synthesis
  - `CheckInferredLatchPass`
  - `CheckClockGatingPass`
  - `AreaEstimationPass`

Instrumentation Passes
- Simulation
  - `EventDrivenPass`
  - `StaticSchedulingPass`
  - `MambaUnrollTickPass`
  - `MambaHeuToposortPass`
  - `MambaTraceBreakingPass`
- Tracing
  - `VcdGenerationPass`
  - `TextWavePass`
  - `VerilogTBGenPass`
- Translation
  - `RTLIRGenPass`
  - `TranslateSVPass`
  - `TranslateYosysPass`

Transform Passes
- Import
  - `VerilogImportPass`
  - `SystemCImportPass`
- Prototype Proxy
  - `ProtoProxyPass`
- Ad-Hoc Transform
  - `AddDebugSignalPass`
  - `SwapHardenedIPPass`

Figure 3. PyMTL3 example passes.

customized connect and falls back to by-name connection if no match is found.

High-Level User-Defined Data Types

Inspired by Python's dataclass, PyMTL3 supports arbitrarily arrayed/nested user-defined data types for both native-Python simulation and HDL generation. PyMTL3 provides Pythonic dataclass-like APIs to declare new data types. The simulation passes can determine the sensitivity of subfields to correctly schedule the simulation. The translation passes can directly generate nested SystemVerilog struct types, or recursively map subfields to slices of a flattened signal (for Verilog).

PyH2: Property-Based Random Testing

PyMTL3 includes PyH2, a property-based random testing framework for hardware generators, processors, and hardware data structures. PyMTL3 provides carefully implemented hypothesis composite search strategies to generate random Bits and user-defined type objects. One key advantage of PyH2 over traditional random testing and iterative-deepened testing is that PyH2 first samples the test-case space and design-parameter space to quickly find a failing test case and then automatically shrinks the failing case and the design parameters. The result is a minimal failing case with minimal design parameters (e.g., shrinking a 50-transaction case for an eight-node network to a 10-transaction case for a four-node network).

PyMTL3 PASSES

PyMTL3 passes are grouped into multiple categories (see Figure 3). Many passes leverage open-source Python libraries and reuse/target open-source hardware tools. The Python language significantly simplifies not only the implementation of passes, but also how designers can configure the passes (e.g., configure a linting pass with a Python lambda function). The designer can skip unneeded passes and only apply a subset of passes as shown in lines 39–41 of Figure 2. While this article introduces some example passes, there are numerous ongoing efforts to implement additional passes, illustrating the modularity and extensibility of the PyMTL3 software architecture.

Linting Passes

Linting passes are analysis passes that check the coding style of PyMTL3 designs. The CheckSignalNamePass queries all of the signal names to enforce a naming convention defined by a given lambda function. The CheckUnusedSignalPass queries all of the signals, all of the update block read/write information, and all of the connections to report signals that are declared but never used.

Statistics Passes

Statistics passes are analysis passes that extract and/or visualize characteristics of the design. RefactoringAnalysisPass gives insights

Authorized licensed use limited to: Cornell University Library. Downloaded on July 03,2020 at 00:53:45 UTC from IEEE Xplore. Restrictions apply.

into code refactoring by using matplotlib to create a scatter plot of the total input/output bit-width of each module and a histogram plot of all the update block lengths. DumpUDGPass leverages graphviz to visualize the directed graph of all update blocks as vertices and all dependencies as edges.

Presynthesis Passes

Presynthesis passes attempt to address RTL synthesis related issues. The CheckInferredLatchPass reports potential inferred latches by querying the AST of combinational update blocks to check if each signal written in the block has valid assignments in all conditional branches. The CheckClockGatingPass reports all signals that are inferred to flip-flops, but nonblocking assignments are not included in an if statement block. Early-stage estimation passes give rough estimates of the hardware based on annotated area/power/timing (automatic annotation is work-in-progress) without invoking external tools. The AreaEstimationPass reports the aggregated area from the annotated area estimates of all leaf components in a structurally composed design.

Simulation Passes

PyMTL3 provides a platform for simulation mechanism research. Simulation passes are instrumentation passes that add a tick function to the top-level component for the user to simulate the whole design cycle-by-cycle. Each simulation pass implements different modeling semantics and/or creates a different simulator for different simulation performance. The EventDrivenPass can schedule pure-RTL models with cyclic dependencies between update blocks and throw exceptions for actual combinational loops. The pass queries the read/write information of all update blocks and constructs sensitivity information to decide the dependent blocks of each update block. The added tick function maintains an event queue to trigger update blocks. The StaticSchedulingPass can only schedule models without cyclic dependencies even though they may not be actual combinational loops. However, removing the event queue leads to higher simulation performance when the toggle rate is high. The pass constructs a direct acyclic graph and applies topological sort to compute a linear execution schedule for every cycle. The added tick function simply iterates over the static schedule. Our previous work on Mamba[6] proposed several novel scheduling techniques that boost the simulation performance under the PyPy just-in-time compiler in a pure-Python environment. The techniques are implemented as additional simulation passes.

Tracing Passes

PyMTL3 provides many tracing options to debug or visualize the execution. Tracing passes are instrumentation passes that add corresponding tracing hook functions to the per-cycle execution schedule. The classic VcdGenerationPass adds a callback function before the simulated rising clock edge to record the value changes in the VCD format compatible with GTKWave, an open-source waveform viewer. Inspired by PyRTL, the TextWavePass horizontally visualizes per-cycle value changes of every signal using ASCII text sequences. VerilogTBGenPass captures the cycle-by-cycle value change of the interface signals of a marked component, and generates a Verilog TB with assertions for use in pure-Verilog four-state RTL or gate-level simulation.

Translation Passes

PyMTL3 RTL designs can be translated into HDL code that is compatible with open-source/commercial FPGA/ASIC synthesis tools. Translation passes are instrumentation passes that attach the translated source file to the design. The RTLIRGenPass first lowers the RTL design from IMIR into RTLIR, a low-level hardware IR provided by PyMTL3. Then, the translation backend pass turns the RTLIR into corresponding HDL source code. Currently PyMTL3 has a synthesizable SystemVerilog backend and a synthesizable Yosys-compatible SystemVerilog backend. To streamline the process of adding a new backend, PyMTL3 ships a carefully designed translation framework that provides a code generator template to be specialized by the target HDL backend with the mapping from RTLIR primitives to HDL source code. A backend can also inherit from an existing backend to maximize code reuse. For example, the Yosys-SystemVerilog backend inherits most code generation

Authorized licensed use limited to: Cornell University Library. Downloaded on July 03,2020 at 00:53:45 UTC from IEEE Xplore. Restrictions apply.

functions from the regular SystemVerilog backend and only adds several hundred lines of code to override the interface/struct-specific functions.

Import Passes

PyMTL3 provides import passes to integrate external IPs with PyMTL3 designs/testbenches using black-box import (simulation only) or white-box import (creating a new PyMTL3 component with internal constructs). Co-simulating existing IPs in Python significantly facilitates verification. Import passes are transform passes that create PyMTL3 components on-the-fly and replace the original placeholders so that the external IPs are integrated seamlessly with rest of the design hierarchy. SystemVerilog and SystemC IPs are imported as black-box modules backed by external C++ shared libraries. The user needs to specify interfaces and source files in the placeholder. Specifically, the VerilogImportPass leverages Verilator to generate a C++ simulator for all specified SystemVerilog files, generates a C interface wrapper, and links the C++ simulator against the wrapper to produce a C++ shared library. Similarly, the SystemCImportPass directly creates a C++ shared library by compiling a generated C++ interface wrapper with the SystemC code and the SystemC kernel library. Then, the placeholder is replaced by a generated PyMTL3 wrapper component that communicates with the shared library through Python's C foreign function interface.

This article discusses PyMTL3, our attempt to jump-start the open-source hardware ecosystem. PyMTL3 takes advantage of the existing Python ecosystem, emphasizes interoperability with other open-source tools, and provides strong support for agile test-driven design.

Prototype Proxy Passes

After pushing the RTL model through an FPGA/ASIC flow, PyMTL3 provides prototype proxy passes that integrate the real prototype with the same Python test bench, which can significantly improve the prototype testing productivity compared to an ad-hoc flow. The proxy passes extensively use Python reflection and IMIR APIs to generate wrapper components that wrap around the prototype. The PyMTL3 TB can send data to the wrapped prototype over the same interface as the original RTL model, as the wrapper components will serialize/deserialize the data and communicate with the system device.

Ad-Hoc Transform Passes

Motivated by real-world situations, PyMTL3 provides many ad-hoc transform passes to help avoid making significantly modifications (that may be reverted eventually) to the codebase. These passes creatively exploit the add, delete, and replace APIs to mutate the design hierarchy *in situ* and open up many opportunities for productive verification and rapid prototyping that would be challenging in other frameworks. Leveraging Python's dynamic typing feature, the AddDebugSignalPass pulls a signal from deep in the hierarchy to expose it at the top level for debugging. For example, the pass takes a signal's hierarchical name top.chip.tiles[1].core.dpath.mult.en, iteratively inserts a debug_en port to the multiplier, the datapath, the core, the tile, the chip, and the top, and connects all the added port together. The user can then apply translation passes to generate HDL code with the additional ports. SwapHardenedIPPass searches for instances of marked PyMTL3 behavioral models and swaps them with placeholders that import hardened Verilog models. Co-simulating the design with real hardened models improves the fidelity of the tests.

CONCLUSION

This article discusses PyMTL3, our attempt to jump-start the open-source hardware ecosystem. PyMTL3 takes advantage of the existing Python ecosystem, emphasizes interoperability with other open-source tools, and provides strong support for agile test-driven design. Moreover, the flexible, modular, and extensible software architecture enables the PyMTL3 framework itself to evolve alongside the open-source hardware ecosystem. PyMTL3 has been open-sourced at https://github.com/pymtl.

Authorized licensed use limited to: Cornell University Library. Downloaded on July 03,2020 at 00:53:45 UTC from IEEE Xplore. Restrictions apply.

ACKNOWLEDGMENTS

This work was supported in part by NSF CRI Award #1512937, DARPA POSH Award #FA8650-18-2-7852, 7853, a research gift from Xilinx, Inc., and the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, as well as equipment, tool, and/or physical IP donations from Intel, Xilinx, Synopsys, Cadence, and ARM. The authors would like to thank D. Lockhart for his valuable feedback and his work on PyMTL2, as well as Cheng Tan, Berkin Ilbeyi, Khalid Al-Hawaj, Lin Cheng, Yixiao Zhang, Raymond Yang, Kaishuo Cheng, and Jack Weber for their contributions to PyMTL3. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

REFERENCES

1. O. Shacham *et al.*, "Rethinking digital design: Why design must change," *IEEE Micro*, vol. 30, no. 6, pp. 9–24, Nov./Dec. 2010.
2. J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Proc. Des. Autom. Conf.*, Jun. 2012, pp. 1212–1221.
3. J. Clow *et al.*, "A pythonic approach for rapid hardware prototyping and instrumentation," in *Proc. 27th Int. Conf. Field Programmable Logic Appl.*, Sep. 2017, pp. 1–7.
4. D. Lockhart *et al.*, "PyMTL: A unified framework for vertically integrated computer architecture research," in *Proc. 47th Annu. Int. Symp. Microarchit.*, Dec. 2014, pp. 280–292.
5. A. Canis *et al.*, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc. 19th Int. Symp. Field Programmable Gate Arrays*, Feb. 2011, pp. 33–36.
6. S. Jiang *et al.*, "Mamba: Closing the performance gap in productive hardware development frameworks," in *Proc. 55th Des. Autom. Conf.*, Jun. 2018, pp. 1–6.
7. J. P. Grossman, B. Towles, J. A. Bank, and D. E. Shaw, "The role of Cascade, a cycle-based simulation infrastructure, in designing the Anton special-purpose supercomputers," in *Proc. 50th Des. Autom. Conf.*, Jun. 2013, pp. 1–9.
8. A. Chattopadhyay *et al.*, "LISA: A uniform ADL for embedded processor modeling, implementation, and software toolsuite generation," in *Processor Description Languages*. New York, NY, USA: Elsevier, 2008, pp. 95–132.
9. C. Tan *et al.*, "PyOCN: A unified framework for modeling, testing, and evaluating on-chip networks," in *Proc. 37th Int. Conf. Comput. Des.*, Nov. 2019, pp. 437–445.
10. A. Izraelevitz *et al.*, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 2017, pp. 209–216.
11. Y. Lee *et al.*, "An agile approach to building RISC-V microprocessors," *IEEE Micro*, vol. 36, no. 2, pp. 8–20, Mar./Apr. 2016.
12. C. Mattarei *et al.*, "CoSA: Integrated verification for agile hardware design," in *Proc. Int. Conf. Formal Methods Comput. Aided Des.*, Oct. 2018, pp. 1–5.

Shunning Jiang is currently working toward the Ph.D. degree in electrical and computer engineering with Cornell University. Jiang received the B.S. degree in computer science from Zhiyuan College, Shanghai Jiao Tong University, in 2015. He is a student member of IEEE. Contact him at sj634@cornell.edu.

Peitian Pan is currently working toward the Ph.D. degree in electrical and computer engineering with Cornell University. Pan received the B.S. degree in computer science from Shanghai Jiao Tong University, in 2018. He is a student member of IEEE. Contact him at pp482@cornell.edu.

Yanghui Ou is currently working toward the Ph.D. degree in electrical and computer engineering with Cornell University. Ou received the B.Eng. degree in electrical and computer engineering from Hong Kong University of Science and Technology, in 2018. He is a student member of IEEE. Contact him at yo96@cornell.edu.

Christopher Batten is currently an Associate Professor in electrical and computer engineering with Cornell University. Batten received the B.S. degree in electrical engineering from the University of Virginia in 1999, the M.Phil. degree in engineering from the University of Cambridge in 2000, and the Ph.D. degree in electrical engineering and computer science from the Massachusetts Institute of Technology in 2010. He is a member of IEEE. Contact him at cbatten@cornell.edu.

Authorized licensed use limited to: Cornell University Library. Downloaded on July 03,2020 at 00:53:45 UTC from IEEE Xplore. Restrictions apply.