An Open-Source Python-Based Hardware Generation, Simulation, and Verification Framework

Shunning Jiang Christopher Torng Christopher Batten

School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{ sj634, clt67, cbatten }@cornell.edu

ABSTRACT

We present an overview of previously published features and work in progress for PyMTL, an open-source Python-based hardware generation, simulation, and verification framework that brings compelling productivity benefits to hardware design and verification. PyMTL provides a natural environment for multi-level modeling using method-based interfaces, features highly parametrized static elaboration and analysis/transform passes, supports fast simulation and property-based random testing in pure Python environment, and includes seamless SystemVerilog integration.

1 INTRODUCTION

There have been multiple generations of open-source hardware generation frameworks that attempt to mitigate the increasing hardware design and verification complexity. These frameworks use a high-level general-purpose programming language to express a hardware-oriented declarative or procedural description and explicitly *generate* a low-level HDL implementation. Our previous work [17] has classified these framework into three major categories. *Hardware preprocessing frameworks* (HPFs) intermingle a high-level language for macro-processing and a low-level HDL for logic modeling. HPFs enable more powerful parametrization but create an abrupt semantic gap in the hardware description [16, 25]. *Hardware generation frameworks* (HGFs) completely embed parametrization and behavioral modeling in a unified high-level "host" language [4, 5, 8, 19, 21], but still generate a low-level HDL implementation for simulation. This limits the use of available host-language features, requires test benches be written in the low-level HDL, and 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* (HGSFs) where parametrization, static elaboration, test benches, behavioral modeling, *and* a simulation engine are all embedded in a general-purpose high-level language [3, 6, 12, 14, 15, 22].

Our previous work on PyMTL [17, 20] demonstrated the potential for a Python-based HGSF to improve the productivity of hardware development. The Python language provides a flexible dynamic type system, object-oriented programming paradigms, powerful reflection and introspection, lightweight syntax, and rich standard libraries. HGSFs that are built upon these productivity features enable a designer to write more succinct descriptions, to avoid crossing any language boundaries for development, testing, and evaluation, and to use the complete expressive power of the host language for verification, debugging, instrumentation, and profiling. A typical workflow using PyMTL is shown in Figure 1. The designer starts from developing a functional-level (FL) design-under-test (DUT) and test bench (TB) completely in Python. Then the DUT is iteratively refined to the cycle level (CL) and register-transfer level (RTL), along with verification and evaluation using Python-based simulation and the same TB. The designer can then translate a PyMTL RTL model to Verilog and use *the same TB* for co-simulation. Note that designers can also co-simulate existing SystemVerilog source code with a PyMTL test bench. The ability to simulate/co-simulate the design in the Python runtime environment drastically reduces the iterative development cycle, eliminates any semantic gap, and makes it feasible to adopt verification methodologies emerging in the open-source software community [11, 23]. Finally, the designer can push the translated DUT through an FPGA/ASIC toolflow. Section 2 gives an overview of key PyMTL features that enable this productive workflow.

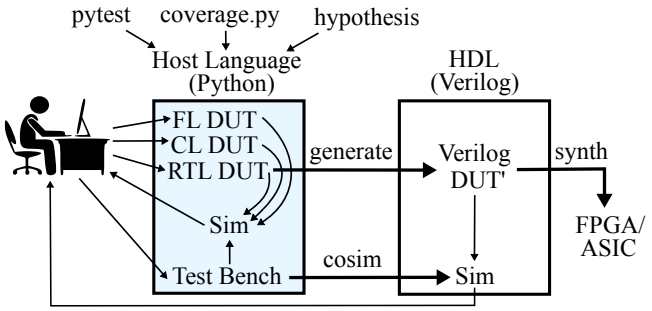


Figure 1: PyMTL's workflow – The designer iteratively refines the hardware within the host Python language, with the help from pytest, coverage.py, and hypothesis. The same test bench is later reused for co-simulating the generated Verilog. FL = functional level; CL = cycle level; RTL = register-transfer level; DUT = design under test; DUT' = generated DUT; Sim = simulation.

Section 3 discusses a variety of PyMTL use cases in the computer architecture community. PyMTL has been used by over 400 students for computer architecture course lab assignments. Multiple research papers at top conferences have used PyMTL for productive CL and RTL modeling [9, 10, 18, 26]. PyMTL has also been used in three chip tapeouts: BRGTC1 [29] in IBM 130 nm, Celerity [1, 13] in TSMC 16 nm, and BRGTC2 [28] in TSMC 28 nm.

2 OVERVIEW OF PYMTL FEATURES

In this section, we introduce the following key productivity features of PyMTL: multi-level modeling, method-based interfaces, highly parametrized static elaboration, analysis and transform passes, pure-Python simulation, property-based random testing, Python/SystemVerilog integration, and fast simulation speed.

Multi-Level Modeling – PyMTL provides a unified environment for modeling hardware at the functional level (FL), cycle level (CL), and register-transfer level (RTL) by providing mechanisms that ensure compatible communication at cross-level boundaries. This multi-level modeling approach systematically builds confidence in verifying a single RTL design-under-test (DUT). The designer is encouraged to first create straightforward FL models which can serve

```python
# FL implementation for calculating log2(N)
@s.tick_fl
def fl_algorithm():
  # put/get have blocking semantics
  s.out.put( math.log( s.in.get(), 2 ) )
```

```python
# CL implementation emulates a 3-cycle pipeline
s.pipe = Pipeline( latency = 3 )
@s.tick_cl
def cl_algo_pipelined():
  if s.out_q.enq_ready():
    if s.pipe.can_pop():  s.out_q.push( s.pipe.do_pop() )
    else:                 s.pipe.advance()

  if not s.in_q.deq_ready():
    s.pipe.do_push( math.log( s.in_q.deq(), 2 ) )
```

```python
# Part of RTL implementation
s.N   = Reg( Bits32 )
s.res = RegEn( Bits32 )
s.connect( s.res.out, s.out.msg )
...
@s.combinational
def rtl_combN():
  s.res.in_  = s.res.out + 1
  s.N.in_    = s.N.out >> 1
  if s.N.out == 0:  s.res.en = Bits1( 0 )
  else:             s.res.en = Bits1( 1 )
```

Figure 2: Example of PyMTL Multi-Level Modeling – The log2 function is implemented at different levels. Different decorators are used to mark FL/CL/RTL blocks.

as golden models for CL/RTL modeling, along with test benches (TB) which can also be reused for CL/RTL verification/simulation. The Python language enables rapid *algorithmic exploration* at the functional level. Then the designer refines the FL model into CL model for cycle-approximate *design-space exploration*. After the FL and CL models are implemented, verified, and evaluated, the designer can implement the actual hardware in RTL and reuse the same test bench that has validated the FL/CL models. Figure 2 shows an example of the same design implemented at different levels.

Seamless multi-level modeling in PyMTL also shines in composing multiple models at different levels together for faster design-space exploration. Sometimes the designer might be working under a tight time constraint, and wants to implement only the performance-critical DUT in RTL. To reduce the time spent to simulate the very first composition, the designer can implement critical components in RTL, and non-critical ones in CL based on rough performance estimates (such as a *cycle-level* cache with a single-cycle hit latency). Later, the CL components may be refined to RTL without any change to other RTL components.

Method-Based Interfaces (Work in Progress) – Method-based interfaces provide designers greater semantic meaning for inter-model communication by raising the level of abstraction at the interface. Currently, cross-layer (e.g., CL to FL/RTL) communications are handled by PyMTL adapters that provide FL/CL with methods to call and RTL with valid/ready handshake signals. However, under the hood they are all implemented using RTL signals and just wrapped with methods. These overheads slow down the simulation of FL/CL models, even though FL/CL models are supposed to be simpler and simulate faster than RTL. In addition, these adapters must be instantiated and managed *manually* by the designer, which adds extra complexity to the design effort. To address these challenges, we take inspiration from Bluespec's method-based interfaces [24] and SystemC's transaction-level modeling (TLM) [27]. We are working on *true method-based* interfaces for FL/CL modeling and *automatic interface coercion* between different levels as shown in Figure 3. When composing two models at the same level, CL and FL interfaces can be "connected" by passing a method pointer for FL/CL, and RTL interfaces are still connected via signals. When composing two models at different levels, PyMTL automatically inserts an

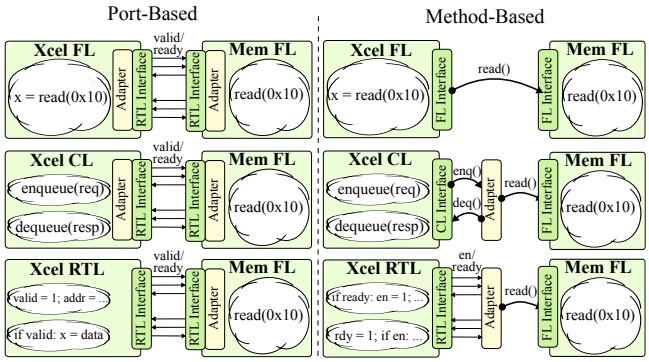


Figure 3: Port-Based Interfaces vs. Proposed Method-Based Interfaces

appropriate adapter that tries to preserve as many method calls as possible, instead of resorting to port-based connections.

Highly Parametrized Static Elaboration – Constructing a highly parametrized hardware generator is one of the key motivations behind modern productive hardware modeling frameworks. In PyMTL, the designer can leverage Python's object-oriented programming and dynamic typing features to intuitively parametrize PyMTL components, as opposed to using low-level HDL's limited parametrization constructs and static typing. Python's extensive support for polymorphism allows the designers to pass parameters of different types around and instantiate different models or logic blocks based on value or type. The static elaboration process executes valid Python code and can be inspected step by step.

Analysis and Transform Passes (Work in Progress) – PyMTL provides APIs to query, add, and remove certain components in the model hierarchy where the root node is the top-level model. Inspired by passes over intermediate representations (IR) in the software realm, the designers can write passes that call these APIs to analyze and transform the whole design. Previous work in Chisel [4] and PyRTL [12] advocate for adding another hardware IR level between the host language and a low-level HDL. We argue that PyMTL passes over the module hierarchy at Python level are more intuitive and productive. *Analysis passes* usually query a list of modules in the hierarchy and accumulate the obtained information for a grand goal. For example, we can query the total number of models that has at least two input ports, a list of ports that starts with a specific name, or even the average number of statements of all logic blocks. *Transform passes* modify the model hierachy. Increment-only transform passes add components to instrument the design without invoking APIs to remove components or connections. An example is to add a child module and bring a signal up to the top level by recursively going up the hierarchy to the top. Other passes involve both adding and removing components or connections. An example is to insert a wrapper component between a module and its child module, which requires removing the child module first, instantiating a new wrapper module, instantiating the same child module within the wrapper module, and establishing all the connections. Figure 4 shows code for these example passes.

Pure-Python Simulation – Unlike HGFs which translate the high-level hardware description to low-level HDL and use an HDL simulator, PyMTL's simulation kernel is built in Python. The designer can track the simulation cycle by cycle and line by line in Python code *at runtime* instead of relying solely on waveform-based debugging. Note that PyMTL can also dump waveforms.

```
# Analysis pass example:
# Get a list of processors with >=2 input ports
def count_pass( top ):
  ret = []
  for m in top.get_all_modules_filter(
    lambda m: len( m.get_input_ports() ) >= 2 ):
    if isinstance( m, AbstractProcessor ):
      ret.append( m )
  return m

# Increment-only transform pass example:
# Bring up state variable of every state machine to a top-level
# output port
def debug_port_pass( top ):
  for m in top.get_all_modules():
    if m.get_full_name().startswith("ctrl"):
      signal_type = m.state.get_type()
      port_name   = "debug_state" + mangle( m.get_full_name() )

      m_outport = m.add_output_port( port_name,
                                     OutPort( signal_type ) )
      m.add_connection( m_outport, m.state )

      while m.has_parent():
        p = m.get_parent()
        p_outport = p.add_output_port( port_name,
                                       OutPort( signal_type ) )
        p.add_connection( p_outport, m_outport )
        m, m_outport = p, p_outport

# Transform pass example:
# Wrap every ctrl with CtrlWrapper
def debug_port_pass( top ):
  for m in top.get_all_modules():
    if m.get_full_name().startswith("ctrl"):
      p = m.get_parent()
      ctrl = p.delete_component( "ctrl" )
      w = p.add_component( "ctrl_wrap", CtrlWrapper() )
      new_ctrl = w.add_component( "ctrl", m )
      ...
      < connect ports >
      ...
```

Figure 4: Example of Analysis and Transform Passes

Simulating in a pure Python runtime means the designer can leverage all the existing third-party Python packages for verification. This significantly helps bring the success of open-source software to open-source hardware. For example, machine learning accelerator designers can import packages like PyTorch and Tensorflow to generate input/reference datasets for test benches and reuse the algorithm implementation for FL/CL models. PyMTL also leverages existing open-source software testing/verification facilities. *py.test* testing framework is used for instantiating numerous tests from a single concise definition, and *coverage.py* tool is used for line-by-line code coverage.

Property-Based Random Testing (Work in Progress) – Constraint-based hardware verification frameworks such as UVM have been widely adopted in the chip-building industry. However, there is no simulator that supports UVM in the open-source hardware/EDA community. As the first step, PyMTL integrates *hypothesis*, a sophisticated property-based random testing framework which was originally designed for verifying Python software. Hypothesis automatically generates numerous random test cases according to a given "hypothesis strategy". After one test case fails, hypothesis will try to construct a minimal failed test case by "auto-shrinking". PyMTL will develop specialized strategies such as generating a random-length list of random packets for verifying PyMTL designs as well as to verify the PyMTL framework itself by generating random logic statements.

Python/SystemVerilog Integration – PyMTL supports importing SystemVerilog code for plug-and-play co-simulation and composition with other PyMTL models or imported SystemVerilog models. Combined with the ability to translate PyMTL RTL models into Verilog code, PyMTL becomes a holistic hardware composition and verification framework. Below are the three major advantages.

```
# By default PyMTL imports module DUT of DUT.v
# in the same folder as the python source file.
class DUT( VerilogModel ):
  def __init__( s ):
    s.in_ = InPort  ( Bits32 )
    s.out = OutPort ( Bits32 )

    # Connect top level ports of DUT
    # to corresponding PyMTL ports
    s.set_ports({
      'clk'   : s.clk,
      'reset' : s.reset,
      'in'    : s.in_,
      'out'   : s.out,
    })
```

Figure 5: PyMTL Source Code for SystemVerilog Import

	PyMTL	MyHDL	PyRTL	Migen	IVerilog	CVS	Mamba
Divider	118K CPS	0.8×	2.2×	0.03×	0.6×	9.3×	20×
1-core	20K CPS	-	-	-	1×	15×	16×
32-core	360 CPS	-	-	-	1.8×	25×	12×

Table 1: Simulation Performance Comparison – CPS = simulated cycle per second; CVS = commercial Verilog simulator.

First, Verilog code generated by PyMTL can be validated by co-simulating with the same PyMTL test bench from FL/CL/RTL development. This helps make sure there are no *code generation mismatches* and *simulation mismatches*. PyMTL generates Verilog for tagged components, calls Verilator (an open-source SystemVerilog simulator) to compile each generated Verilog file, compiles this C++ into a shared library and dynamically links it back to PyMTL program using CFFI (C Foreign Function Interface), and replaces the tagged PyMTL model with the Verilog model. Compared to hardware generation frameworks (HGF) where there is usually no way to test if the translated HDL matches the high-level code, PyMTL builds more confidence for RTL developers.

Second, PyMTL can help verification engineers even if they are not writing PyMTL RTL models. They can take hand-written SystemVerilog code, write a simple PyMTL wrapper as shown in Figure 5, and build a PyMTL test bench to drive the simulation. At a larger design scale, PyMTL can simulate a design composed of many FL, CL, RTL, and SystemVerilog models based on how detailed the designer models each component. This enables the designer to productively verify SystemVerilog models using supporting FL/CL PyMTL models in an end-to-end testing approach.

Third, PyMTL can act as a glue for composing multiple SystemVerilog models and PyMTL RTL models. The imported code of a child Verilog model is preserved when PyMTL translates the top-level module to Verilog. Structural composition of SystemVerilog models can take advantage of PyMTL's parametrization power to create a larger composition of wrapped SystemVerilog models.

Fast Simulation Speed – Adapted from our recent work [17], Table 1 shows an apples-to-apples simulation performance comparison of an iterative divider, a single RISC-V RV32IM [2] core hooked up to a two-port test memory, and 32 cores hooked up to a 64-port test-memory. PyMTL offers competitive simulation performance compared to other HGSFs, but commercial HDL simulators can still be orders of magnitude faster than HGSFs. Our work shows that a carefully designed HGSF can close the simulation-performance gap by deeply co-optimizing the HGSF and the underlying general-purpose JIT compiler *within the host high-level language*. With Mamba techniques, PyMTL's pure-Python simulation performance matches a commercial Verilog simulator and is 10X-20X faster than the original PyMTL.

3 PYMTL USE CASES

In this section, we discuss how PyMTL has already been employed for teaching in the classroom, for driving computer architecture research, and for building silicon prototypes.

3.1 PyMTL for Teaching

PyMTL has been used by over 400 students across two universities, including in a senior-level undergraduate computer architecture course at Cornell University (ECE 4750), in a similar course at Boston University (EC 513), and in a graduate-level ASIC design course at Cornell University (ECE 5745). The computer architecture courses involved multiple design labs (integer multiplier, simple RISC-V processor, set-associative blocking cache, and bus/ring network), culminating in a final lab composing all previous components to build a multi-core system. Students chose whether to design in PyMTL, in SystemVerilog, or with a mix, but they were required to test their designs using PyMTL. Overall, PyMTL accelerates students' learning curve. Developing and simulating the design in a pure-Python runtime environment accommodates students with more of a software background. PyMTL's SystemVerilog integration feature shortens the iterative development cycle for students with more of a hardware background.

3.2 PyMTL for Architecture Research

PyMTL has driven experiments for multiple computer architecture research projects that have been published at top conferences.

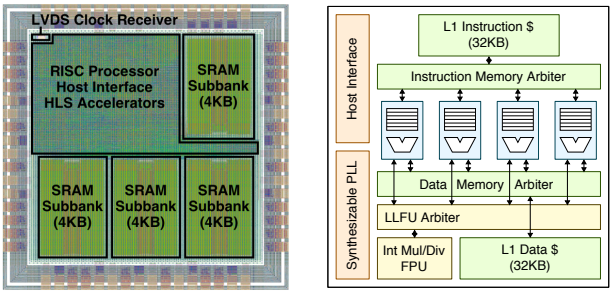
The LTA [18] and XLOOPS [26] papers both modeled novel accelerators as PyMTL CL models, which were then composed with general-purpose control processors in gem5 [7] using PyMTL/gem5 co-simulation support (co-simulation is implemented using the C Foreign Function Interface, CFFI). Gem5's popular and well-supported CPU models, memory system, and runtime features are directly reusable, enabling researchers to focus their efforts on exploring the accelerator design-space in PyMTL.

Similarly, DAE [10] and ParallelXL [9] both leveraged PyMTL, with particular emphasis on the framework's highly parametrized static elaboration features for RTL design. These two papers explored the design of architectural templates that efficiently generate tuned accelerators.

3.3 PyMTL for Silicon Prototyping

PyMTL designs have been taped out in advanced nodes including TSMC 16 nm, TSMC 28 nm, and IBM 130 nm. The associated projects have been published: Celerity [1, 13], BRGTC1 [29], and BRGTC2 [28].

Celerity is a 5×5 mm 385M-transistor chip in TSMC 16 nm designed and implemented by a large team of over 20 students and faculty from UC San Diego, University of Michigan, and Cornell as part of the DARPA Circuit Realization At Faster Timescales (CRAFT) program. PyMTL played a key role in the integration of a complex HLS-generated BNN (i.e., binarized neural network) with the broader system's general-purpose compute tier (i.e., five modified Chisel-generated RISC-V Rocket cores) as well as its massively parallel compute fabric (i.e., 496-core RISC-V tiled manycore processor). The BNN was wrapped with PyMTL-generated parametrized RoCC wrappers and adapters, and these wrappers were generic and were automatically generated using reflection. The wrapped BNN was translated back to Verilog for composition with the rest of the chip. The control blocks for the high-speed links between the BNN and the manycore were also designed and verified in PyMTL.



(a) Chip Diagram of BRGTC1 (b) Block Diagram of BRGTC2

Figure 6: BRGTC1 and BRGTC2

BRGTC1 (i.e., Batten Research Group Test Chip 1) was implemented nearly entirely using PyMTL. BRGTC1 marked the first exploration of the interaction between PyMTL RTL and HLS-generated Verilog models. The prototype is a small 2×2 mm 1.3M-transistor chip in IBM 130 nm, and it pairs a simple pipelined 32-bit RISC processor developed in PyMTL with an HLS-generated application-specific accelerator. Fabricated BRGTC1 chips have been post-silicon validated for functionality using assembly tests. Our BRGTC1 lab-bench setup re-uses the PyMTL test suite to drive signals through a "host" FPGA base board (i.e., Xilinx Zedboard) and out to an FMC-connected daughter card that contains the test chip.

BRGTC2 (i.e., Test Chip 2) was built with much more aggressive use of PyMTL, resulting in a 1×1.25 mm 6.7M-transistor chip in TSMC 28 nm. The chip contains four RISC-V RV32IMAF cores sharing a 32kB instruction cache, a 32kB data cache, and a single-precision floating point unit, along with microarchitectural mechanisms to mitigate the performance impact of resource sharing. The BRGTC2 project stressed PyMTL's features extensively. For example, multi-level modeling supported debugging efforts by enabling the team to swap in an FL cache to narrow the location of an RTL bug down to other components. A single PyMTL test suite is used to test all modeling abstractions, including the FL, CL, RTL, and even gate-level models. As another success, the floating-point unit was designed using a collection of Synopsys DesignWare components that were each Verilog-imported into PyMTL for composition and simulation. Overall, the PyMTL framework was a tremendous success as a productive hardware modeling framework for designing two non-trivial research test chips.

4 CONCLUSION

The PyMTL framework and the code used for Mamba paper have been open-sourced at `https://github.com/cornell-brg/pymtl` and `https://github.com/cornell-brg/mamba-dac2018`. We anticipate a new release of PyMTL in 2019.

ACKNOWLEDGMENTS

This work was supported in part by NSF CRI Award #1512937, NSF SHF Award #1527065, DARPA POSH Award #FA8650-18-2-7852, and a donation from Intel. The authors acknowledge and thank Derek Lockhart for his valuable feedback and his work on the original PyMTL framework. The authors would like to thank Ajay Joshi for using PyMTL for the computer architecture course at Boston University. The author also thank all the students who have provided feedback to PyMTL. U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation theron. 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] T. Ajayi et al. Celerity: An Open Source RISC-V Tiered Accelerator Fabric. *Symp. on High Performance Chips (Hot Chips)*, Aug 2017.

[2] K. Asanovic et al. Instruction Sets Should Be Free: The Case for RISC-V. Technical report, UCB/EECS-2014-146, Aug 2014.

[3] C. Baaij et al. Cλash: Structural Descriptions of Synchronous Hardware Using Haskell. *Euromicro Conf. on Digital System Design (DSD)*, Sep 2010.

[4] J. Bachrach et al. Chisel: Constructing Hardware in a Scala Embedded Language. *Design Automation Conf. (DAC)*, Jun 2012.

[5] S. Belloeil et al. Stratus: A Procedural Circuit Description Language Based Upon Python. *Int'l Conf. on Microelectronics (ICM)*, Dec 2007.

[6] P. Bellows et al. JHDL-An HDL for Reconfigurable Systems. *Symp. on FPGAs for Custom Computing Machines (FCCM)*, Apr 1998.

[7] N. Binkert et al. The gem5 Simulator. *SIGARCH Computer Architecture News (CAN)*, 39(2):1–7, Aug 2011.

[8] P. Bjesse et al. Lava: Hardware Design in Haskell. *Int'l Conf. on Functional Programming (ICFP)*, Sep 1998.

[9] T. Chen et al. An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2018.

[10] T. Chen et al. Efficient Data Supply for Hardware Accelerators with Prefetching and Access/Execute Decoupling. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2016.

[11] K. Claessen et al. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 46(4):53–64, Apr 2011.

[12] J. Clow et al. A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation. *Int'l Conf. on Field Programmable Logic (FPL)*, Sep 2017.

[13] S. Davidson et al. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro*, Mar 2018.

[14] J. Decaluwe. MyHDL: A Python-based Hardware Description Language. *Linux Journal*, Nov 2004.

[15] P. Haglund et al. Hardware Design with a Scripting Language. *Int'l Conf. on Field Programmable Logic (FPL)*, Sep 2003.

[16] J. Jennings et al. Verischemelog: Verilog Embedded in Scheme. *Conf. on Domain-Specific Languages (DSL)*, Oct 1999.

[17] S. Jiang et al. Mamba: closing the performance gap in productive hardware development frameworks. *Design Automation Conf. (DAC)*, Jun 2018.

[18] J. Kim et al. Using Intra-Core Loop-Task Accelerators to Improve the Productivity and Performance of Task-Based Parallel Programs. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2017.

[19] Y. Li et al. HML, A Novel Hardware Description Language and Its Translation to VHDL. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, 8(1):1–8, Dec 2000.

[20] D. Lockhart et al. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[21] A. Mashtizadeh. PHDL: A Python Hardware Design Framework. Master's thesis, EECS Department, MIT, May 2007.

[22] Migen. `https://m-labs.hk/gateware.html`.

[23] M. Naylor et al. A generic synthesisable test bench. *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, Sep 2015.

[24] N. Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High-Level Specifications. *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, Jun 2004.

[25] O. Shacham et al. Rethinking Digital Design: Why Design Must Change. *IEEE Micro*, 30(6):9–24, Nov/Dec 2010.

[26] S. Srinath et al. Architectural Specialization for Inter-Iteration Loop Dependence Patterns. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[27] systemc tlm. SystemC TLM (Transaction-level Modeling). Online Webpage, accessed Oct 1, 2014. `http://www.accellera.org/downloads/standards/systemc/tlm`.

[28] C. Torng et al. A New Era of Silicon Prototyping in Computer Architecture Research. *The RISC-V Day Workshop at the 51st Int'l Symp. on Microarchitecture*, Oct 2018.

[29] C. Torng et al. Experiences Using a Novel Python-Based Hardware Modeling Framework for Computer Architecture Test Chips. *HOTCHIPS Student Poster*, Aug 2016.