# PyH2: Using PyMTL3 to Create Productive and Open-Source Hardware Testing Methodologies

**Shunning Jiang, Yanghui Ou, Peitian Pan, Kaishuo Cheng, Yixiao Zhang, and Christopher Batten**
Cornell University

*Editor's notes:*
This article proposes a new model testing and verification methodology, PyH2, using property-based random testing in Python. PyH2 leverages the whole Python ecosystem to build test benches and models.
*—Sherief Reda, Brown University*
*—Leon Stock, IBM*
*—Pierre-Emmanuel Gaillardon, University of Utah*

**As Dennard scaling** is over and Moore's law continues to slow down, modern system-on-chip (SoC) architectures have been moving toward heterogeneous compositions of general-purpose and specialized computing fabrics. This heterogeneity complicates the already challenging task of SoC design and verification. Building an open-source hardware community to amortize the nonrecurring engineering effort of developing highly parametrized and thoroughly verified hardware blocks is a promising solution to the heterogeneity challenge. However, the widespread adoption of open-source hardware has been obstructed by the scarcity of such high quality blocks. We argue that a key missing piece in the open-source hardware ecosystem is comprehensive, productive, and open-source verification methodologies that reduce the effort required to create thoroughly tested hardware blocks. Compared to closed-source hardware, verification of open-source hardware faces several significant challenges.

First, closed-source hardware is usually owned and maintained by companies with dedicated verification teams. These verification engineers usually have many years of experience in constraint-based random testing using a universal verification methodology (UVM) with commercial SystemVerilog simulators. However, open-source hardware teams usually follow an agile test-driven design approach stemming from the open-source software community, where the designer is also responsible for creating the corresponding tests. Moreover, the steep learning curve, in conjunction with very limited support in existing open-source tools, makes the UVM-based approach rarely used by open-source hardware teams. We argue that the open-source hardware community is in critical need of an alternative route for testing open-source hardware, instead of simply duplicating closed-source hardware testing frameworks.

Second, unlike closed-source hardware's development cycle where most engineers focus on a specific design instance for the next generation product, open-source hardware blocks

*Digital Object Identifier 10.1109/MDAT.2020.3024144*
*Date of publication: 14 September 2020; date of current version: 8 April 2021.*

usually exist in the form of design *generators* to maximize reuse across the community [1]. However, design generators are significantly more difficult to verify than design instances due to the combinatorial complexity in the multidimensional generator parameter space. There is a critical need to create an open-source framework that systematically and productively tests design generators and automatically simplifies both failing test cases and failing design instances to facilitate debugging.

Third, performing random testing can be difficult in important hardware domains. There has been a major surge in open-source RISC-V processor implementations. However, due to limited human resources, most of these implementations only include a few directed tests, randomly generated short assembly sequences, and/or very large scale system-level tests (e.g., booting Linux). There is a critical need to create an automated random testing framework to improve the fidelity of open-source processor implementations.

Fourth, many open-source hardware blocks are designed to improve reusability by exposing well-encapsulated timing-insensitive hand-shake interfaces that can provide an object-oriented view of the hardware block (e.g., a hardware reorder buffer exposes three object-oriented "method" interfaces: allocate, update, and remove). However, it is very hard to perform random testing to test the behavior of concurrent hardware data structures that have multiple interfaces accepting "transactions" in the same cycle. Converting a random transaction sequence into cycle-by-cycle test vectors using traditional testing approaches requires a cycle-accurate golden model. Manually creating multitransaction test-vectors only works for directed testing. One possible solution is to execute only one random transaction in each cycle, yet the inability to stress intracycle concurrent behavior harms the quality of the tests. There is critical need to create a novel testing approach for object-oriented hardware using concurrent intracycle transactions.

To address these challenges, we introduce PyH2,[1] our vision for a productive and open-source testing methodology for open-source hardware, which is significantly different from state-of-the-art closed-source hardware testing. Leveraging open-source software, PyH2 attempts to solve the open-source hardware testing challenge by holistically using property-based testing (PBT) in Python to significantly reduce designer effort in creating high-quality tests. The advantage of PBT over constraint-based random testing is as follows.

- PBT does not draw all of the random data beforehand, making it possible to leverage runtime information to guide the random data generation.
- PBT can automatically shrink the failing test case to a minimal failing case once a bug is discovered.

Compared to BlueCheck [2], a prior PBT framework for hardware, the key distinctions are as follows.

- PyH2 enables using a high-level behavioral specification written in Python as the reference model instead of requiring the reference model to be synthesizable.
- The random byte-stream internal representation of `hypothesis` provides more sophisticated auto-shrinking, while BlueCheck simply removes transactions along with *ad hoc* iterative deepening.
- PyH2 can auto-shrink not only the transactions but also the design itself by unifying the design parameter space and the test-case space.

We see coverage-guided mutational fuzzing (e.g., RFUZZ [3]) as complementary to PBT. PBT can be used to quickly find bugs with moderate complexity, while RFUZZ can be used to very slowly find potentially more complex bugs. Overall, PyH2 is able to combine the advantages of complete-random testing (CRT) and iterative-deepened testing (IDT) to identify a failing test case quickly and then provide a minimal failing case to facilitate debugging.

PyH2 is supported by the whole Python ecosystem, among which three main packages form the foundation of PyH2 (PyMTL3, `pytest`, and `hypothesis`). PyH2 users can use over 100,000 open-source Python libraries to build test benches and golden models. PyH2 leverages PyMTL3 [4], [5] to build Python test benches to drive register-transfer-level (RTL) simulations with PyMTL3 models and/or external SystemVerilog models leveraging PyMTL3's Verilator cosimulation support. PyH2 adopts `pytest`, a mature full-featured Python testing tool, to collect, organize, parametrize, instantiate, and refactor test cases for

[1]Python's hypothesis for hardware.

testing open-source hardware. PyH2 also exploits pytest plugins to evaluate hardware-specific testing metrics. For example, PyH2 tracks the line coverage of behavioral logic blocks of PyMTL3 models during simulation using coverage.py, a line coverage tool for normal Python code. The key component of PyH2 is hypothesis, a PBT framework to test Python programs by intelligently generating random test cases and rapidly auto-shrinking failing test cases.

PyH2 is realized by a collection of PyH2 frameworks which are discussed in depth in the rest of this article: PyH2G (PyH2 for RTL design generators), PyH2P (PyH2 for processors), and PyH2O (PyH2 for object-oriented hardware).

## Background

This section briefly introduces PyMTL3, pytest, and hypothesis, the three key Python libraries that form the foundation of PyH2.

### PyMTL3

PyMTL3 is an open-source Python-based hardware modeling, generation, simulation, and verification framework. PyMTL3 supports multilevel modeling for RTL, cycle-level, and functional-level models. To provide productive, flexible, and extensible workflows, PyMTL3 is designed to be strictly modular. Specifically, PyMTL3 separates the PyMTL3 embedded domain-specific language 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.

PyMTL3 aims at creating an evolving ecosystem with its modern software architecture and high interoperability with other open-source tools. PyMTL3 emphasizes performing simulation in the Python runtime and automatic Verilator black-box import for cosimulation. Driving the simulation from Python test benches to test both PyMTL3 designs and external SystemVerilog modules enables PyMTL3 to combine the familiarity of Verilog/SystemVerilog with the productivity features of Python. Tools that take the opposite approach (e.g., cocotb) embed Python in a Verilog simulator and drive the simulation from the Verilog runtime, but this complicates the ability to leverage the full power of Python. RTL designs built in PyMTL3 can be translated to SystemVerilog accepted by commercial EDA tools, or Yosys-compatible Verilog accepted by OpenROAD, a state-of-the-art open-source EDA flow [6].

### PyTest

pytest is a mature full-featured tool for testing Python programs. Using pytest, the programmer can create small tests with little effort and also parametrize numerous complex tests with compositions of pytest decorators succinctly as shown in Figure 1a. pytest also provides lightweight command line options to print out different kinds of error messages varying from a list of characters indicating

```python
def gcd( a, b ):
  while b > 0: # bug: while b > 10
    a, b = b, a
    return a
# Create two tricky directed test cases
@pytest.mark.parametrize(
   "a,  b,  ref", [
  [ 12, 18, 6 ],
  [ 65, 33, 1 ]
)
def test_directed( a, b, ref ):
  assert gcd( a, b ) == ref
```

(a)

| Desired Property | CRT | IDT | PBT |
|---|---|---|---|
| Small number of test cases to find bug | ✓ | X | ✓ |
| Small number of transactions in bug trace | X | ✓ | ✓ |
| Simple transactions in bug trace | X | ✓ | ✓ |

(b)

```python
import math, random, hypothesis

def test_complete_random():
  for _ in range( 100 ):
    a = random.randint(1, 128)
    b = random.randint(1, 128)
    assert gcd( a, b ) == math.gcd( a, b )

def test_iterative_deepened():
  for a in range( 1, 128 ):
    for b in range( 1, 128 ):
      assert gcd( a, b ) == math.gcd( a, b )

@hypothesis.given(
  a = hypothesis.strategies.integers(1, 128),
  b = hypothesis.strategies.integers(1, 128),
)
def test_property_based( a, b ):
  assert gcd( a, b ) == math.gcd( a, b )
```

(c)

**Figure 1. Background on testing methodologies. (a) Parametrizing directed tests using a pytest decorator. (b) Comparison of different testing techniques. (c) Code for testing a greatest common divisor function using CRT, IDT, and PBT.**

whether each test fails, to per-test full stack traces. pytest has hundreds of plugins, such as pytest-cov that leverages coverage.py to track line coverage.

### CRT, IDT, and hypothesis PBT

Traditional testing methodologies usually use a mix of CRT and IDT. As shown in Figure 1b, CRT can detect errors quickly because it randomly samples the input space, but can produce very complicated failing test cases which are difficult to debug. IDT finds bugs more slowly because it gradually samples the input space, but can produce simple counterexamples. PBT, first popularized by QuickCheck [7], is a high-level, black-box testing technique where one only defines *properties* of the program under test and uses *search strategies* to create randomized inputs. The original QuickCheck paper also discussed the integration with Lava [8] to test circuits. Properties are essentially partial specifications of the program under test and are more compact and easier to write and understand than full system specifications. Users can make full use of the host language when writing properties and thus can accurately describe the intended behavior. Most PBT tools support *shrinking*, a mechanism to simplify failing test cases into a minimal reproducible counterexample. With these features, PBT can achieve the benefits of both CRT and IDT.

Hypothesis [9] is a state-of-the-art Python PBT library that includes built-in search strategies for different data types and supports integrated auto-shrinking of failing test cases. All hypothesis strategies are built on top of a unified random byte-stream representation, and each strategy internally repurposes random bytes to produce the target random value. Search strategies in hypothesis are integrated with methods that describe how to simplify certain types of data, which makes shrinking effective. Users can compose built-in search strategies for any user-defined data type and shrinking will work out-of-the-box.

Complicated stateful systems can also be tested with RuleBasedStateMachine in hypothesis. The user inherits from the RuleBasedStateMachine class to add variables, a prologue, and an epilogue to create a new test class. The user needs to define rules and their preconditions and invariants, which describes conditional state transitions. For stateful testing, usually the user creates Python assertions inside the rule to compare against a golden reference model. Hypothesis repeatedly instantatiates the test class and executes a sequence of rules on the state machine.

Figure 1c shows examples of testing the greatest common divisor function using CRT, IDT, and hypothesis PBT against math.gcd. The CRT test (lines 16–20) includes 100 random samples. The IDT test (lines 22–25) iteratively tries all possible values for *a* and *b* from 1 to 128. We use the @hypothesis.given decorator to transform a normal function test_property_based that accepts arguments, into a randomized PBT test. Consider a bug where line 3 in Figure 1a is changed to while b > 10. CRT can find the bug quickly, but the failing test case involves relatively large numbers. IDT finds the bug in exactly 11 test cases [i.e., gcd(1,11)]. PBT can find the bug quickly with large numbers, but then auto-shrink the inputs to a minimal counterexample [i.e., gcd(2,1)].

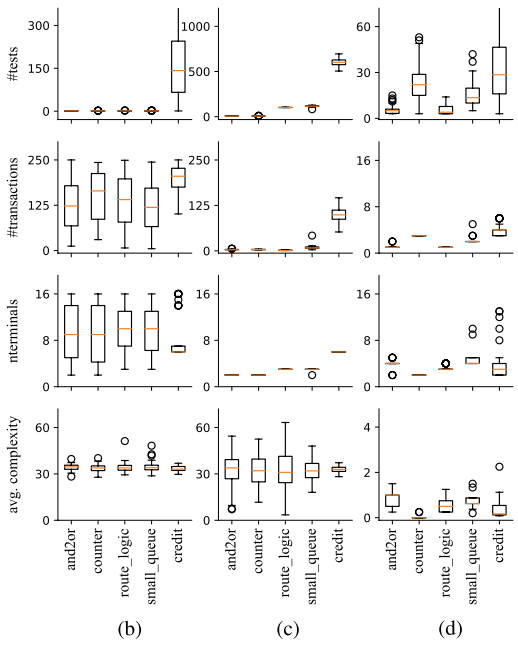
## PyH2G: PyH2 for RTL design generators

PyH2G is a PyH2 framework to productively and effectively test *RTL design generators*. We envision that future open-source SoC designs are heavily based on chip generators which are composed of numerous highly parametrized RTL design generators. Unfortunately, verifying design generators is significantly more challenging than verifying design instances due to the *combinatorial explosion* in the multidimensional generator parameter space. Traditional testing techniques such as CRT and IDT face new challenges when testing design generators. CRT can find a bug quickly with a few test cases but often leads to a complicated failing test case with numerous transactions *and a complex design*, which makes it more difficult to debug. IDT can produce a simple failing case with a small design instance, but may take a very long time to detect the error due to the iterative deepening required for the generator parameters.

In response to these challenges, PyH2G uses PBT to obtain the benefits of both CRT and IDT. Specifically, PyH2G creates composite search strategies in hypothesis to interpret part of the generated random byte stream as the design parameters and the rest as the test case (see lines 3–4 of Figure 2a). Unifying the design parameter space and the test case space allows hypothesis to simultaneously shrink the design parameters (i.e., reducing the complexity of the generated design instance), the length of the input transaction sequence, and the complexity of each transaction to a minimal failing test case.

```
from hypothesis import given, strategies as st
# pkt_strat is the search strategy for packet
@given( nterminals = st.integers(2, 16),
        test_pkts  = st.lists( pkt_strat() ) )
def test_ring_pyh2g( nterminals, test_pkts ):
    dut = RingNetwork( nterminals )
    th  = TestHarness( dut, test_pkts )
    run_sim( th )
```

(a)




**#tests**: number of test cases needed to find the bug
**#transactions**: number of transactions in the final failing case
**#nterminals**: size of the ring network in the final failing case
**avg. complexity**: average complexity of all packets in the test case calculated based on the value of each packet field

| Bug name | Description |
|---|---|
| and2or | mistakenly put an AND instead of an OR |
| counter | wrong enable logic in a counter |
| route_logic | partially wrong routing logic in the router |
| small_queue | size of the router's input buffer $<$ max credit |
| credit | incorrect credit update logic in the router |

**Figure 2. PyOCN RingNet generator case study. (a) PyH2G example. (b) CRT. (c) IDT. (d) PyH2G.**

### Case study: on-chip network generator

We quantitatively evaluated CRT, IDT, and PyH2G using the PyOCN [10] ring network generator against four real-world bugs. PyOCN is a multitopology, modular, and highly parametrized on-chip network generator built in PyMTL3. Figure 2a illustrates an example of a PyH2G test that uses search strategies to configure the ring network and generate the test packets. When a test case fails, hypothesis can simultaneously shrink the design instance and the packet sequence. We ran 50 trials for each bug, and the results are shown as box-and-whisker plots in Figure 2b–d. Overall, PyH2G detects errors quickly with a small number of test cases and produces a simple failing test case that has a short sequence of transactions and a simple design. PyH2G also significantly reduces the transaction complexity. PyH2G sometimes runs slightly more test cases than CRT because hypothesis will first generate explicit examples to stress-test the boundary conditions before exploring values randomly. However, this also help PyH2G discover the credit bug more quickly than CRT.

## PyH2P: PyH2 for processors

PyH2P is a PyH2 framework to automatically generate random assembly instruction sequences to test processors, which makes the case for effective domain-specific random testing methodologies. Different from existing work, PyH2P is able to automatically shrink a failed long program to a minimal instruction sequence with a minimal set of architectural registers and memory addresses. It is possible to combine auto-shrinking with other sophisticated random program generators [11] by carefully using PyH2P random strategies. PyH2P can also leverage Symbolic-QED [12] by applying QED transformations to generated random programs and performing bounded model checking to accelerate bug discovery.

PyH2P creates composite hypothesis strategies to generate random assembly programs for effective auto-shrinking. Specifically, PyH2P creates a hierarchy of strategies for arithmetic, memory, and branch instruction strategies using substrategies for architectural registers, memory addresses, and immediate values. PyH2P currently implements a block-based mechanism which first instantiates a control-flow template of branches, and then fills random instructions between branches. PyH2P ensures that each generated assembly program has well-defined behavior across the test and reference models. For arithmetic instructions, PyH2P constrains the range of the immediate value strategy to avoid overflow. For memory instructions, PyH2P constrains the range of the memory address strategy to avoid unaligned and out-of-bound memory accesses. For branch instructions, PyH2P first generates a sequence of branch instructions and their corresponding labels, and then randomly shuffles them to form the control-flow template. This eliminates the possibility of branch

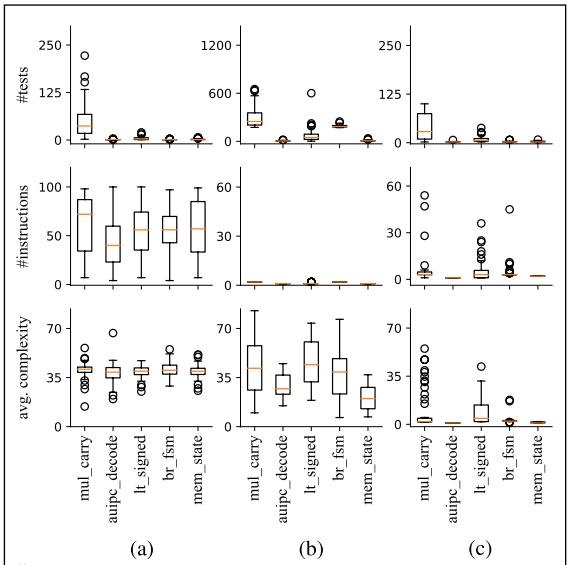
out-of-range errors. Additionally, a set of registers are dedicated to loop bounds and loop variables to avoid infinite loops.

### Case study: PicoRV32 processor

We demonstrate the effectiveness of PyH2P using PicoRV32, an open-source, area-optimized RV32IMC processor implemented in Verilog. We leverage PyMTL3's Verilator support to drive the cosimulation using a PyMTL3 testbench. The imported processor is connected to a PyMTL3 cycle-level test memory which stores the assembly program generated by PyH2P. After executing the program, we extract and compare the value of PicoRV32 architectural registers and the test memory against an instruction set simulator written in PyMTL3.

We inject five directed bugs into the Verilog code, and ran 50 trials for each methodology and bug combination. The results are shown as box-and-whisker plots in Figure 3a–c. CRT generally requires a small number of tests (less than to discover a bug, but the failing cases usually include more than 50 complex instructions. IDT significantly reduces the number of instructions in the failing test case, but needs significantly more cases to find the failing case. Note that IDT generates instructions of similar complexity to CRT because we have to generate random immediate values to avoid prohibitively long runtimes to find these bugs. PyH2P is able to discover the failing test case using a similar number of trials to CRT and can shrink it to a minimal case with similar length to the cases found by IDT. Moreover, PyH2P is able to shrink the immediate value so that the average instruction complexity is significantly reduced.

Figure 3d–g shows the failing cases for the mul_carry bug discovered by each methodology. This bug can only be triggered by specific operands. Figure 3d is the example found by CRT with 41 instructions, seven unique architectural registers, and large immediate values. Figure 3e shows the example found by IDT which uses only one register but a large random immediate value. Figure 3f and g includes two minimal failing cases from different PyH2P trials, which are significantly simpler.




**#tests**: number of assembly programs to discover the bug
**#transactions**: number of instructions in the failing program
**avg. complexity**: average complexity of all instructions calculated based on the registers and immediate values used

| Bug name | Description |
|---|---|
| mul_carry | carry bits of carry-save adders shifted by 1 |
| auipc_decode | a typo in the decode logic of auipc |
| lt_signed | a signed comparison performed w/o casting |
| br_fsm | incorrect state transition for branches |
| mem_state | the status of current load is ignored |

```
1 bge   x5, x14, 0x60
2 auipc x8, 0x100
3 beq   x14, x27, 0x2C
4 xori  x23, x14, 0x6C0A
5 sltu  x23, x14, x5
6 lui   x27, 0xBD6E
7 srli  x27, x21, 0xB1C2
8 sltiu x14, x21, 0x3307
9 bltu  x14, x27, 0x64
10 sltiu x21, x5, 0x2FCF
11 // 30 insts omitted
12 mul   x14, x27, x27
```
(d)

```
1 lui   x5, 0xC349
2 mul   x5, x5, x5
```
(e)

```
1 lui   x1, 0x6
2 mul   x1, x1, x1
```
(f)

```
1 lui   x1, 0x0
2 // PC=0x204 here
3 auipc x1, 0x0
4 // 0x204 * 0x204
5 mul   x1, x1, x1
```
(g)

**Figure 3. PicoRV32 processor case study. (a) CRT. (b) IDT. (c) PyH2P. (d) CRT example. (e) IDT example. (f) PyH2P example 1. (g) PyH2P example 2.**

## PyH2O: PyH2 for object-oriented hardware

PyH2O is a PyH2 framework that enables using method calls to test RTL hardware components with object-oriented latency-insensitive interfaces. The key contribution of PyH2O is a novel testing methodology for concurrent hardware data structures that are difficult to thoroughly test using traditional approaches. PyH2O proposes a novel simulation mechanism called *auto-ticking*, which has been implemented as a new PyMTL3 simulation pass. With merely "transaction-accurate" Python data structures as reference models, PyH2O uses the rule-based stateful testing features in hypothesis to perform a sequence of random method calls on both

the reference model and the auto-ticking simulator of the RTL model, and then checks if the outcomes match for each method call.

PyH2O is based on method-based interfaces which are decoupled handshake interfaces with four ports: 1) enable; 2) ready; 3) arguments; and 4) return value. Essentially, setting the enable signal high after making sure the ready signal is high is equivalent to calling the corresponding ready method, checking if it returns true, and then calling the actual method. Converting an RTL method interface to a Python method involves an adapter that provides a method and a ready method to the user and sets/modifies the signals inside the adapter. PyH2O leverages Python reflection to automatically wrap the RTL method interfaces with a generated top-level PyMTL3 wrapper with Python methods.

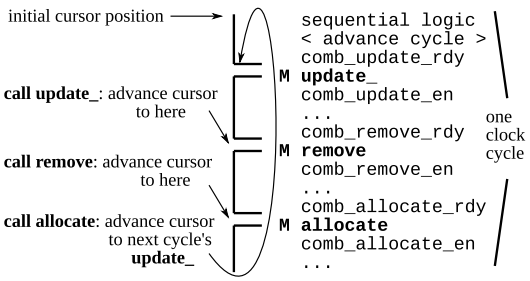
PyH2O applies the AutoTickSimPass to create an auto-ticking simulator for the wrapped model. Conceptually, auto-ticking is more fine-grained than the classical delta cycle approach. Auto-ticking divides the combinational logic into multiple parts based on logic related to the method interfaces. When the user calls the enhanced top-level method, not only the method but also all the logic between this method and the next method is executed. If the executed method is the last method of the cycle, the simulator advances to the first method of the next cycle. If the user skips a method in this cycle and calls another method later in the cycle or a previous method that is already skipped/called in the current cycle, the simulator ignores the in-between methods and executes all the logic until it reaches the called method. Unlike trivial one-method-per-cycle testing, this auto-ticking scheme is able to execute multiple methods in the same cycle if they are called in a specific order.

## Case study: reorder buffer data structure

Figure 4a shows an RTL reorder buffer implementation which exposes the three methods called interfaces. allocate is ready if the buffer is not full. It returns the entry index and advances the tail pointer. update_ is ready if the buffer has valid elements. It takes an index/value pair to update the buffer. remove is ready if the buffer head is valid and already updated, and returns the index/value pair. Note that remove and allocate can occur in the same cycle even if the reorder buffer is full, because the implementation combinationally

```
class ReorderBuffer( Component ):
  def construct( s, TData, num_entries ):
    TIndex  = mk_bits( clog2( num_entries ) )
    TROBMsg = mk_bitstruct( 'ROBMsg', {
      'index': TIndex, 'value': TData,
    })
    s.allocate = CalleeIfcRTL( RetType=TIndex )
    s.update_  = CalleeIfcRTL( MsgType=TROBMsg )
    s.remove   = CalleeIfcRTL( RetType=TROBMsg )
    s.head = Wire( TIndex )
    ...
    @update_ff
    def upff_head_pointer():
      if   s.reset:
        s.head <<= 0
      elif s.alloc.en & s.remove.en:
        s.head <<= s.head + 1
      elif not s.alloc.en & s.remove.en:
        s.head <<= s.head + 1 # "head + 0" bug
```

(a)




(b)

```
state = ReorderBuffer_PyH2O()
state.allocate()
state.update(msg=ROBMsg(Bits2(2),Bits16(63527)))
state.allocate()
state.update(msg=ROBMsg(Bits2(3),Bits16(2091)))
state.allocate()
state.allocate()
state.update(msg=ROBMsg(Bits2(1),Bits16(62455)))
state.update(msg=ROBMsg(Bits2(0),Bits16(38580)))
state.remove()
state.alloc()
state.remove() # error: ref ready, dut not ready
```

(c)

```
state.allocate()
state.update(msg=ROBMsg(Bits2(0),Bits16(0)))
state.allocate()
state.remove()
state.update(msg=ROBMsg(Bits2(1),Bits16(0)))
state.remove() # error: ref ready, dut not ready
```

(d)

**Figure 4. PyH2O reorder buffer case study. (a) PyMTL3 reorder buffer code snippet. (b) Auto-tick execution schedule for reorder buffer. (c) First falsifying example found by PyH2O. (d) Minimized failing case after auto-shrinking.**

factors whether remove is called into allocate's ready signal. Figure 4b shows the execution schedule generated by the AutoTickSimPass. The auto-ticking simulator guarantees that a sequence of three method calls in the order of update_ < remove < allocate will occur in the same cycle.

To show the effectiveness of PyH2O, we replace head+1 with head+0 in line 19 of Figure 4a. This subtle bug needs at least six transactions in a specific order to trigger because it requires six transactions to allocate, update and remove two entries, but must not remove the first one and allocate the second one in the same cycle. After trying several sequences with varying length from 5 to 19, PyH2O discovers a 11-transaction failing case as shown in Figure 4c. After auto-shrinking, PyH2O successfully finds one of the minimum failing case as shown in Figure 4d.

**This article has** introduced PyH2, which leverages PyMTL3, pytest, and hypothesis to create a novel open-source hardware testing methodology. We believe PyH2 is an important first step toward addressing four key challenges in open-source hardware testing as follows.

- PyH2 is more accessible to open-source hardware designers compared to complex closed-source hardware testing methodologies.
- PyH2G is well-suited for testing not just design instances but also design generators which are critical to the success of the open-source hardware ecosystem.
- PyH2P can improve the random testing of open-source processor implementations compared to the more limited directed and random testing currently used in many open-source projects.
- PyH2O can more effectively test object-oriented hardware data structures.

We have open-sourced PyMTL3 and PyH2 at https://github.com/pymtl/pymtl3. ■

## Acknowledgments

Shunning Jiang and Yanghui Ou contributed equally to this work. This work was supported in part by NSF CRI under Award 1512937, in part by DARPA POSH under Award FA8650-18-2-7852, 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 cosponsored by DARPA, as well as equipment, tool, and/or physical IP donations from Intel, Xilinx, Synopsys, Cadence, and ARM. We acknowledge and thank Derek Lockhart for his initial thoughts on combining PyMTL with hypothesis, and Cheng Tan for his contributions to PyH2G. We would like to acknowledge and thank David MacIver and Zac Hatfield-Dodds for their work on the Hypothesis framework and thoughtful discussions on how to leverage Hypothesis for hardware. 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., "Avoiding game over: Bringing design to the next level," in *Proc. 49th Annu. Design Autom. Conf. (DAC)*, Jun. 2012, pp. 623–629.

[2] M. Naylor and S. Moore, "A generic synthesisable test bench," in *Proc. ACM/IEEE Int. Conf. Formal Methods Models for Codesign (MEMOCODE)*, Sep. 2015, pp. 128–137.

[3] K. Laeufer et al., "RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2018, pp. 1–8.

[4] S. Jiang, B. Ilbeyi, and C. Batten, "Mamba: Closing the performance gap in productive hardware development frameworks," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018, pp. 1–6.

[5] S. Jiang et al., "PyMTL3: A Python framework for open-source hardware modeling, generation, simulation, and verification," *IEEE Micro*, vol. 40, no. 4, pp. 58–66, Jul. 2020.

[6] T. Ajayi et al., "Toward an open-source digital flow: First learnings from the OpenROAD project," in *Proc. 56th Annu. Design Autom. Conf. (DAC)*, Jun. 2019, pp. 1–4.

[7] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs," in *Proc. Int. Conf. Funct. Program. (ICFP)*, Sep. 2000, pp. 268–279.

[8] P. Bjesse et al., "Lava: Hardware design in Haskell," in *Proc. Int. Conf. Funct. Program. (ICFP)*, Sep. 1998, pp. 174–84.

[9] D. MacIver et al., "Hypothesis: A new approach to property-based testing," *J. Open Source Softw.*, vol. 4, no. 43, p. 1891, Nov. 2019.

[10] C. Tan et al., "PyOCN: A unified framework for modeling, testing, and evaluating on-chip networks," in *Proc. Int. Conf. Comput. Design (ICCD)*, Nov. 2019, pp. 437–445.

[11] F. Corno et al., "Fully automatic test program generation for microprocessor cores," in *Proc. Design Autom. Test Eur. (DATE)*, Mar. 2003, pp. 1006–1011.

[12] F. Corno et al., "Fully automatic test program generation for microprocessor cores," in *Proc. Design Autom. Test Eur. (DATE)*, Mar. 2018, pp. 55–60.

**Shunning Jiang** is currently pursuing a PhD in electrical and computer engineering with Cornell University, Ithaca, NY. Jiang has a BS in computer science from Zhiyuan College, Shanghai Jiao Tong University, Shanghai, China (2015). He is a student member of IEEE.

**Yanghui Ou** is currently pursuing a PhD in electrical and computer engineering with Cornell University, Ithaca, NY. Ou has a BEng in electrical and computer engineering from the Hong Kong University of Science and Technology, Hong Kong (2018). He is a student member of IEEE.

**Peitian Pan** is currently pursuing a PhD in electrical and computer engineering with Cornell University, Ithaca, NY. Pan has a BS in computer science from Shanghai Jiao Tong University, Shanghai, China (2018). He is a student member of IEEE.

**Kaishuo Cheng** is currently a junior undergraduate student in computer science with Cornell University, Ithaca, NY.

**Yixiao Zhang** has a BS and an MEng in electrical and computer engineering from Cornell University, Ithaca, NY (2018 and 2019, respectively).

**Christopher Batten** is currently an Associate Professor of Electrical and Computer Engineering with Cornell University, Ithaca, NY. Batten has a BS in electrical engineering from the University of Virginia, Charlottesville, VA (1999), an MPhil in engineering from the University of Cambridge, Cambridge, U.K. (2000), and a PhD in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, MA (2010). He is a member of IEEE.

■ Direct questions and comments about this article to Christopher Batten, School of Electrical and Computer Engineering, Cornell University, Ithaca, NY 14853 USA; cbatten@cornell.edu.