PyMTL3
A Python Framework for Hardware Modeling, Generation, Simulation, and Verification

https://pymtl.github.io

Christopher Batten
Electrical and Computer Engineering
Cornell University
Vertically Integrated Research Methodology

Our research group focuses on **accelerator-centric system-on-chip design** across the computing stack including applications, programming frameworks, compiler optimizations, runtime systems, instruction set design, microarchitecture design, and VLSI implementation.

**Key Metrics:** Cycle Count, Cycle Time, Area, Energy

Experimenting with full-chip layout, FPGA prototypes, and test chips is a key part of our research methodology.
Projects in the Batten Research Group

Computer Architecture
HPCA'21/23, ISCA'20, MICRO'22/18/17
- Integrated Rack-Scale Acceleration for Computational Pangenomics
- Ephemeral Vector Arch Using Processing-in-SRAM
- big.TINY Arch for Dynamic Task-Level Parallelism

Digital VLSI & Circuits
ISCAS'20, NOCS'20, VLSI'19, TCAS-I'18
- Accelerator-Centric Prototypes in TSMC16nm, GF12nm
- Bit-Serial/Bit-Parallel Bit-Line Computing with SRAM
- Chip-Level Silicon Photonic Interconnection Networks

Themes
- Power
- Performance
- Programmability
- Post-CMOS

Vertically Driven Research Approach
- Spans Entire Computing Stack
- FPGA Prototypes/Emulation
- ASIC Test Chip Tapeouts

Themes
- Power
- Performance
- Programmability
- Post-CMOS

Electronic Design Automation
TCAD'22, DAC'21/18, IEEE D&T'21, IEEE Micro'20, ICCD'19
- Productive Hardware Modeling, Generation, Simulation, Testing
- New On-Chip Network Logical and Physical Design Generators
- HLS Methodologies for Dynamic Task-Level Parallelism

Electronic Design Automation
- Manycore + Xcel Chiplet
- App Specific Chiplets
- Optics Chiplet

class Incr(Component):
    def construct(s, n):
        s.in_ = InPort(n)
        s.out = OutPort(n)
    @update
    def logic():
        s.out @= s.in_ + 1
Multi-Level Modeling Methodologies

- Applications
- Algorithms
- Compilers
- Instruction Set Architecture
- Microarchitecture
- VLSI
- Transistors

Functional-Level Modeling
- Behavior

Cycle-Level Modeling
- Behavior
- Cycle-Approximate
- Analytical Area, Energy, Timing

Register-Transfer-Level Modeling
- Behavior
- Cycle-Accurate Timing
- Gate-Level Area, Energy, Timing
Multi-Level Modeling Methodologies

**Multi-Level Modeling Challenge**

FL, CL, RTL modeling use very different languages, patterns, tools, and methodologies.

**SystemC** is a good example of a unified multi-level modeling framework.

**Is SystemC the best we can do in terms of productive multi-level modeling?**

**Functional-Level Modeling**

- Algorithm/ISA Development
- MATLAB/Python, C++ ISA Sim

**Cycle-Level Modeling**

- Design-Space Exploration
- C++ Simulation Framework
- SW-Focused Object-Oriented
- gem5, SESC, McPAT

**Register-Transfer-Level Modeling**

- Prototyping & AET Validation
- Verilog, VHDL Languages
- HW-Focused Concurrent Structural
- EDA Toolflow
Traditional RTL Design Methodologies

HDL
Hardware Description Language

- HDL (Verilog)
- RTL
- Sim
- TB
- synth
- FPGA/ASIC

HPF
Hardware Preprocessing Framework

- Mixed (Verilog+Perl)
- RTL
- gen
- TB
- FPG/ASIC

- RTL
- gen
- TB
- Sim
- FPG/ASIC

Example: Genesis2

HGF
Hardware Generation Framework

- Host Language (Scala)
- RTL
- gen
- TB
- sim
- FPG/ASIC

- RTL
- gen
- TB
- Sim
- FPG/ASIC

Example: Chisel

- ✔ Fast edit-sim-debug loop
- ✔ Single language for structural, behavioral, + TB
- ✔ Easier to create highly parameterized generators
- ✔ Single language for structural + behavioral
- ✔ Easier to create highly parameterized generators

- ✗ Slower edit-sim-debug loop
- ✗ Multiple languages create "semantic gap"
- ✗ Cannot use power of host language for verification

Is Chisel the best we can do in terms of a productive RTL design methodology?
PyMTL

Python-based hardware generation, simulation, and verification framework which enables productive multi-level modeling and RTL design.
PyMTL3: A Python Framework for Hardware Modeling, Generation, Simulation, and Verification

PyMTL3 Motivation

PyMTL3 Framework
[IEEE Micro’20, DAC’21]

PyMTL3 in Practice

PyMTL3 JIT
[DAC’18]

PyMTL3 Testing
[IEEE Design&Test’21]

PyMTL3 Gradual Typing
[LATTE’23]
PyMTL3: A Python Framework for Hardware Modeling, Generation, Simulation, and Verification

PyMTL3 Motivation

PyMTL3 Framework
[IEEE Micro’20, DAC’21]

PyMTL3 in Practice

PyMTL3 JIT
[DAC’18]

PyMTL3 Testing
[IEEE Design&Test’21]

PyMTL3 Gradual Typing
[LATTE’23]
PyMTL

- **PyMTL2**: [https://github.com/cornell-brg/pymtl](https://github.com/cornell-brg/pymtl)
  - released in 2014
  - extensive experience using framework in research & teaching

- **PyMTL3**: [https://github.com/pymtl/pymtl3](https://github.com/pymtl/pymtl3)
  - official release in May 2020
  - adoption of new Python3 features
  - significant rewrite to improve productivity & performance
  - cleaner syntax for FL, CL, and RTL modeling
  - completely new Verilog translation support
  - first-class support for method-based interfaces
### The PyMTL3 Framework

**PyMTL3 DSL (Python)**
- Model
- Config

**PyMTL3 In-Memory Intermediate Representation (Python)**
- Elaboration
- Model Instance

**PyMTL3 Passes (Python)**
- Simulation Pass
- Translation Pass
- Analysis Pass
- Transform Pass

**Test & Sim Harnesses**

**Simulatable Model**

**Verilog**

**Analysis Output**

**New Model**
PyMTL3 High-Level Modeling

```python
class QueueFL( Component ):  
    def construct( s, maxsize ):  
        s.q = deque( maxlen=maxsize )  
    
    @non_blocking(  
        lambda s: len(s.q) < s.q.maxlen  
    )  
    def enq( s, value ):  
        s.q.appendleft( value )  

    @non_blocking(  
        lambda s: len(s.q) > 0  
    )  
    def deq( s ):  
        return s.q.pop()
```

▶ FL/CL components can use method-based interfaces

▶ Structural composition via connecting methods

```python
class DoubleQueueFL( Component ):  
    def construct( s ):  
        s.enq = CalleeIfcCL()  
        s.deq = CalleeIfcCL()  
        s.q1 = QueueFL(2)  
        s.q2 = QueueFL(2)  
        connect( s.enq, s.q1.enq )  
        connect( s.q2.deq, s.deq )  

    @update  
    def upA():  
        if s.q1.deq.rdy() and s.q2.enq.rdy():  
            s.q2.enq( s.q1.deq() )
```

Christopher Batten  
Spring 2023 @ NVIDIA
**PyMTL3 Low-Level Modeling**

```python
from pymtl3 import *

class RegIncrRTL( Component ):
    def construct( s, nbits ):
        s.in_ = InPort( nbits )
        s.out = OutPort( nbits )
        s.tmp = Wire( nbits )

    @update_ff
    def seq_logic():
        s.tmp <<= s.in_

    @update
    def comb_logic():
        s.out @= s.tmp + 1
```

- Hardware modules are Python classes derived from `Component`
- `construct` method for constructing (elaborating) hardware
- ports and wires for signals
- `update` blocks for modeling combinational and sequential logic
SystemVerilog RTLIR/Translation Framework

- RTLIR simplifies RTL analysis passes and translation
- Translation framework simplifies implementing new translation passes
Translation+import enables easily testing translated SystemVerilog
Also acts like a JIT compiler for improved RTL simulation speed
Can also import external SystemVerilog IP for co-simulation
Translating to *Readable* SystemVerilog

```python
class StepUnit( Component ):
    def construct( s ):
        s.word_in = InPort ( 16 )
        s.sum1_in = InPort ( 32 )
        s.sum2_in = InPort ( 32 )
        s.sum1_out = OutPort( 32 )
        s.sum2_out = OutPort( 32 )

    @update
def up_step():
        temp1 = b32(s.word_in) + s.sum1_in
        s.sum1_out @= temp1 & b32(0xffff)
        temp2 = s.sum1_out + s.sum2_in
        s.sum2_out @= temp2 & b32(0xffff)
```

```verilog
module StepUnit
    (input logic [0:0] clk,
     input logic [0:0] reset,
     input logic [31:0] sum1_in,
     output logic [31:0] sum1_out,
     input logic [31:0] sum2_in,
     output logic [31:0] sum2_out,
     input logic [15:0] word_in)
);
// Temporary wire definitions
logic [31:0] __up_step$temp1;
logic [31:0] __up_step$temp2;

// PYMTL SOURCE:
// ...

always_comb begin : up_step
    __up_step$temp1 = {{16{1'b0}},word_in} + sum1_in;
    sum1_out = __up_step$temp1 & 32'd65535;
    __up_step$temp2 = sum1_out + sum2_in;
    sum2_out = __up_step$temp2 & 32'd65535;
end

endmodule
```

- **Readable signal names**
- **Generates useful comments**
- **Simple type inference for temporary variables**
What is PyMTL3 for and not (currently) for?

- **PyMTL3 is for ...**
  - Taking an accelerator design from concept to implementation
  - Construction of highly-parameterizable CL models
  - Construction of highly-parameterizable RTL design generators
  - Rapid design, testing, and exploration of hardware mechanisms
  - Interfacing models with other C++ or Verilog frameworks

- **PyMTL3 is not (currently) for ...**
  - Python high-level synthesis
  - Many-core simulations with hundreds of cores
  - Full-system simulation with real OS support
  - Users needing a complex OOO processor model “out of the box”
PyMTL3: A Python Framework for Hardware Modeling, Generation, Simulation, and Verification

PyMTL3 Motivation

PyMTL3 Framework
[IEEE Micro’20, DAC’21]

PyMTL3 in Practice

PyMTL3 JIT
[DAC’18]

PyMTL3 Testing
[IEEE Design&Test’21]

PyMTL3 Gradual Typing
[LATTE’23]
PyMTL has been used in many chip tapeouts

TSMC 180nm, 28nm, 16nm; SkyWater 130nm
GF 130nm, 12nm; Intel 16

- Simple RISC-V cores
- Coarse-grain reconfigurable arrays
- Clustered manycore architectures
- Mesh on-chip networks
- Crossbar interconnects
BRG Test Chip #1 (2016)

RISC processor, 16KB SRAM, HLS-generated accelerator
2x2mm, 1.2M-trans, IBM 130nm
95% done using PyMTL2
Four RISC-V RV32IMAF cores with “smart” sharing of L1$/LLFU
1x1.2mm, 6.7M-trans, TSMC 28nm
95% done using PyMTL2
BRG Test Chip #5 (2022)

- Three undergraduates → MEng
- $2 \times 2.5\text{mm}$ in TSMC 180nm
- RISC-V RV32IM micro-controller
- 16KB of instruction SRAM, 16KB of data SRAM
- SPI interface for config, SPI master, GP I/O
- 100% done using PyMTL3 (including chip bring-up)
Celerity SoC: BNN Xcel for DARPA CRAFT (2017)

Target Workload: High-Performance Embedded Computing

- 5 × 5mm in TSMC 16 nm FFC
- 385 million transistors
- 511 RISC-V cores
  - 5 Linux-capable Rocket cores
  - 496-core tiled manycore
  - 10-core low-voltage array
- 1 BNN accelerator
- 1 synthesizable PLL
- 1 synthesizable LDO Vreg
- 3 clock domains
- 672-pin flip chip BGA pkg
- 9-months from PDK access to tape-out

[HotChips’17, IEEE Micro’18, VLSI’19, SSCL’19]
Cifery SoC: TinyCore Cluster for DARPA POSH (2021)

- 4 × 4mm in GF 12 nm
- 450 million transistors
- 4 Linux-capable Ariane cores
- 1 Embedded FPGA
- 3 TinyCore clusters
  - 6 RISC-V RV32IMAF cores
  - 4KB private L1 data cache
  - Pairs share icache, MDU, FPU
  - Software-centric coherence
- Mesh-based on-chip network

[Fig. 1. SoC Architecture]
[Fig. 2. eFPGA Architecture]
[Fig. 3. Lab Evaluation Setup]
[Fig. 4. Maximum Operating Frequency vs. Supply Voltage]
[Fig. 5. Comparison to the State of the Art]
[Fig. 6. Performance and Efficiency Gains of Offloaded Benchmarks]
HammerBlade SoC: CGRA for DARPA SDH (2022)

▶ Elastic latency-insensitive interfaces simplify compilation & MC integration
▶ 32-bit fxp/fp add, subtract, multiply, madd, accumulator
▶ copy0, copy1, sll, srl, and, or, xor, eq, ne, gt, geq, lt, leq
▶ phi and branch for control flow
▶ concurrent routing bypass paths
PyMTL3 for Undergraduate and Graduate Courses

Computer Arch Course
Labs use PyMTL for verification, PyMTL or Verilog for RTL design

Chip Design Course
Labs use PyMTL for verification, PyMTL or Verilog for RTL design, standard ASIC flow

First Teaching Tapeout in 10+ years!
Four student projects
All use PyMTL for testing
Two use PyMTL for design
<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>% python3 -m venv pymtl3</code></td>
<td>&gt;&gt;&gt; from pymtl3.examples.ex00_quickstart <code>import FullAdder</code></td>
</tr>
<tr>
<td><code>% source pymtl3/bin/activate</code></td>
<td>&gt;&gt;&gt; import inspect</td>
</tr>
<tr>
<td><code>% pip install pymtl3</code></td>
<td>&gt;&gt;&gt; print(inspect.getsource(FullAdder))</td>
</tr>
<tr>
<td><code>% python</code></td>
<td>&gt;&gt;&gt; fa = FullAdder()</td>
</tr>
<tr>
<td>&gt;&gt;&gt; from pymtl3 import *</td>
<td>&gt;&gt;&gt; fa.apply( DefaultPassGroup(textwave=True) )</td>
</tr>
<tr>
<td>&gt;&gt;&gt; a = Bits8(6)</td>
<td>&gt;&gt;&gt; fa.sim_reset()</td>
</tr>
<tr>
<td>&gt;&gt;&gt; b = Bits8(3)</td>
<td>&gt;&gt;&gt; fa.a @= 0</td>
</tr>
<tr>
<td>&gt;&gt;&gt; a &lt;&lt; 4</td>
<td>&gt;&gt;&gt; fa.b @= 1</td>
</tr>
<tr>
<td>&gt;&gt;&gt; c = (a &lt;&lt; 4)</td>
<td>&gt;&gt;&gt; fa.cin @= 0</td>
</tr>
<tr>
<td>&gt;&gt;&gt; c</td>
<td>&gt;&gt;&gt; fa.sim_tick()</td>
</tr>
<tr>
<td>&gt;&gt;&gt; c[4:8]</td>
<td>&gt;&gt;&gt; fa.a @= 1</td>
</tr>
<tr>
<td>&gt;&gt;&gt; fa.print_textwave()</td>
<td>&gt;&gt;&gt; fa.b @= 0</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;&gt; fa.cin @= 1</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;&gt; fa.sim_tick()</td>
</tr>
</tbody>
</table>
PyMTL3: A Python Framework for Hardware Modeling, Generation, Simulation, and Verification

PyMTL3 Motivation

PyMTL3 Framework
[IEEE Micro’20, DAC’21]

PyMTL3 in Practice

PyMTL3 JIT
[DAC’18]

PyMTL3 Testing
[IEEE Design&Test’21]

PyMTL3 Gradual Typing
[LATTE’23]
Evaluating HDLs, HGFs, and HGSFs

- Apple-to-apple comparison of simulator performance
- 64-bit radix-four integer iterative divider
- All implementations use same control/datapath split with the same level of detail
- Modeling and simulation frameworks:
  - Verilog: Commercial verilog simulator, Icarus, Verilator
  - HGF: Chisel
  - HGSFs: PyMTL, MyHDL, PyRTL, Migen
Productivity/Performance Gap

- Higher is better
- Log scale (gap is larger than it seems)
- Commercial Verilog simulator is $20 \times$ faster than Icarus
- Verilator requires C++ testbench, only works with synthesizable code, takes significant time to compile, but is $200 \times$ faster than Icarus
Productivity/Performance Gap

Chisel (HGF) generates Verilog and uses Verilog simulator
Using CPython interpreter, Python-based HGSFs are much slower than commercial Verilog simulators; even slower than Icarus!
Using PyPy JIT compiler, Python-based HGSFs achieve \( \approx 10 \times \) speedup, but still significantly slower than commercial Verilog simulator.

---

Christopher Batten  Spring 2023 @ NVIDIA
Hybrid C/C++ co-simulation improves performance but:
- only works for a synthesizable subset
- may require designer to simultaneously work with C/C++ and Python
PyMTL3 achieves impressive simulation performance by co-optimizing the framework and JIT.
## PyMTL3 Performance

<table>
<thead>
<tr>
<th>Technique</th>
<th>Divider</th>
<th>1-Core</th>
<th>16-core</th>
<th>32-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-Driven</td>
<td>24K CPS</td>
<td>6.6K CPS</td>
<td>155 CPS</td>
<td>66 CPS</td>
</tr>
<tr>
<td><strong>JIT-Aware HGSF</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ Static Scheduling</td>
<td>13×</td>
<td>2.6×</td>
<td>1×</td>
<td>1.1×</td>
</tr>
<tr>
<td>+ Schedule Unrolling</td>
<td>16×</td>
<td>24×</td>
<td>0.4×</td>
<td>0.2×</td>
</tr>
<tr>
<td>+ Heuristic Toposort</td>
<td>18×</td>
<td>26×</td>
<td>0.5×</td>
<td>0.3×</td>
</tr>
<tr>
<td>+ Trace Breaking</td>
<td>19×</td>
<td>34×</td>
<td>2×</td>
<td>1.5×</td>
</tr>
<tr>
<td>+ Consolidation</td>
<td>27×</td>
<td>34×</td>
<td>47×</td>
<td>42×</td>
</tr>
<tr>
<td><strong>HGSF-Aware JIT</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ RPython Constructs</td>
<td>96×</td>
<td>48×</td>
<td>62×</td>
<td>61×</td>
</tr>
<tr>
<td>+ Huge Loop Support</td>
<td>96×</td>
<td>49×</td>
<td>65×</td>
<td>67×</td>
</tr>
</tbody>
</table>

- RISC-V RV32IM five-stage pipelined cores
- Only models cores, no interconnect nor caches
PyMTL3: A Python Framework for Hardware Modeling, Generation, Simulation, and Verification

PyMTL3 Motivation

PyMTL3 Framework
[IEEE Micro’20, DAC’21]

PyMTL3 in Practice

PyMTL3 JIT
[DAC’18]

PyMTL3 Testing
[IEEE Design&Test’21]

PyMTL3 Gradual Typing
[LATTE’23]
Testing RTL Design Generators is Challenging

Testing a specific ring network instance requires a number of different test cases

```
  test_ring_1pkt_2x2_0_chnl
  test_ring_2pkt_2x2_0_chnl
  test_ring_self_2x2_0_chnl
  test_ring_clockwise_2x2_0_chnl
  test_ring_aclockwise_2x2_0_chnl
  test_ring_neighbor_2x2_0_chnl
  test_ring_tornado_2x2_0_chnl
  test_ring_backpressure_2x2_0_chnl
```

Ideal testing technique:
1. Detect error quickly with **small number of test cases**
2. The failing test case has **minimal number of transactions**
3. The bug trace has **simplest transactions**
4. The failing test case has the **simplest design**

A design generator can have many parameters: topology, routing, flow control, channel latency
Software Testing Techniques

- Complete Random Testing (CRT)
  - Randomly generate input data
  - Detects error quickly
  - Debug complicated test case

- Iterative Deepened Testing (IDT)
  - Gradually increase input complexity
  - Finds bug with simple input
  - Takes many test cases to find bug

- Property-Based Testing (PBT)
  - Search strategies, auto shrinking
  - Detects error quickly
  - Produces minimal failing test case
  - Increasingly state-of-the-art in software testing

```python
def gcd( a, b ):
    while b > 0:
        a, b = b, a % b
    return a

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

def test_idt():
    for a_max in range( 1, 128 ):
        for b_max 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_pbt( a, b ):
    assert gcd( a, b ) == math.gcd( a, b )
```
PyH2 Creatively Adopts PBT for SW to Test HW

- PyH2 combines **PyMTL3**, a unified hardware modeling framework, with **Hypothesis**, a PBT framework for Python software and creates a property-based testing framework for hardware.

- PyH2 leverages PBT to explore not just the input values for an RTL design but to also **explore the parameter values** used to configure an RTL design generator.

<table>
<thead>
<tr>
<th></th>
<th>CRT</th>
<th>IDT</th>
<th>PyH2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small number of test cases to find bug</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Small number transactions in bug trace</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Simple transactions in bug trace</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Simple design instance for bug trace</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
PyH2 Example: GCD Unit Generator

- GCD unit w/ or w/o input FIFO, parameterized by FIFO size, bitwidth of input

- Complete Random Testing
  - Randomly pick size of input FIFO and bitwidth of data, randomly generate a sequence of transactions

- Iterative Deepened Testing
  - Gradually increase size of input FIFO, bitwidth, and range of input value
Results of Applying PyH2 to GCD Unit Generator

- **Four directed bugs**
  - q-rd.ptr: read pointer of input FIFO does not increment when a message is dequeued (need 2+ entry FIFO to observe bug)
  - q-wr.ptr: write pointer of input FIFO does not wrap around when FIFO is full (need 2+ entry FIFO to observe bug)
  - gcd-idle: not check valid signal in IDLE
  - gcd-done: not check ready signal in DONE
  - 200 trials each

- **100 randomly injected bugs**
  - Each random bug has two trials
  - Randomly mutate expression in source code
Case Study #1: Results of Applying PyH2 to GCD Unit Generator

PyH2 requires few tests (like CRT) but also produces easy to debug failing test cases (like IDT).
Failing Test Case Shrinking Example

Original Failing Test Case

```
test case #0
- nbits = 4
- qsize = 0
- ntrans = 1
- seq = [TestVector(a=1, b=1)]

shrinking...
- nbits = 4
- qsize = 2
- ntrans = 1
- seq = [TestVector(a=2, b=2)]

shrinking...
- nbits = 4
- qsize = 2
- ntrans = 1
- seq = [TestVector(a=1, b=1)]

Falsifying example: _run_hypothesis(nbits=4, qsize=2, src_intv=0, sink_intv=0, seq=data(...))
```

Minimized Failing Test Case

```
Draw 1: [TestVector(a=1, b=1), TestVector(a=2, b=2)]

bug found with 3 test cases
- ntrans = 2
- nbits = 4
- qsize = 2
- seq = [TestVector(a=1, b=1), TestVector(a=2, b=2)]

avg_value = 1.5
```
PyMTL3: A Python Framework for Hardware Modeling, Generation, Simulation, and Verification

PyMTL3 Motivation

PyMTL3 Framework

[IEEE Micro’20, DAC’21]

PyMTL3 in Practice

PyMTL3 JIT

[DAC’18]

PyMTL3 Testing

[IEEE Design&Test’21]

PyMTL3 Gradual Typing

[LATTE’23]
<table>
<thead>
<tr>
<th>Statically Typed HDLs</th>
<th>Gradually Typed HDLs</th>
<th>Dynamically Typed HDLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Static correctness guarantees on generators</td>
<td>✓ Static correctness guarantees on generators</td>
<td>✓ High testing &amp; verification productivity</td>
</tr>
<tr>
<td>✓ Fast simulation</td>
<td>✓ High testing &amp; verification productivity</td>
<td>✗ No static correctness guarantees</td>
</tr>
<tr>
<td>✗ Limited testing &amp; verification productivity</td>
<td>✓ Disciplined mixed-type component composition</td>
<td>✗ Slow simulation</td>
</tr>
<tr>
<td>✓ Simulation perf optimizations</td>
<td>✓ Simulation perf optimizations</td>
<td>✓ Simulation perf optimizations</td>
</tr>
</tbody>
</table>
Gradually Typed HDLs Enable Statically Type Checking Hardware Generators

- Leverage Python3 standard type annotation syntax to annotate bitwidths
- Translate the bitwidth equivalence invariant into integer constraints
- Use SMT solvers to prove or disprove the invariant

```python
T_Adder = TypeVar("T_Adder", bound=Bits)
class Adder(Component, Generic[T_Adder]):
    def __init__(s, Width: Type[T_Adder]) -> None:
    def construct(s, Width: Type[T_Adder]) -> None:
        n = get_nbits(Width)
        # s.a and s.b have type Signal[T_Adder]
        s.a = InPort(Width)
        s.b = InPort(Width)
        # s.out has type Signal[Bits]
        s.out = OutPort(mk_bits(n+1))
        # s.fa has type List[FullAdder]
        s.fa = [FullAdder() for _ in range(n)]
        # s.carry has type Signal[Bits]
        s.carry = Wire(mk_bits(n+1))
        # s.sum has type Signal[T_Adder]
        s.sum = Wire(Width)

@update
def upblk() -> None:
    # concat arg1: Signal[Bits1]
    # concat arg2: Signal[T_Adder]
    # concat(s.carry[n], s.sum): Signal[Bits]
    s.out @= concat(s.carry[n], s.sum)
```
Gradually Typed HDLs Enable Safe Mixed-Type Component Composition

- Statically typed components expect well-typed inputs
- Errors propagate past the origin given ill-typed inputs
- During elaboration: each generator checks the given parameters against annotations
- During simulation: each signal assignment checks the given values against its type

A Mixed-Typed Component Composition with Statically Typed DUT (divider) and Dynamically Typed Test Bench
PyMTL3: A Python Framework for Hardware Modeling, Generation, Simulation, and Verification

PyMTL3 Motivation

PyMTL3 Framework
[IEEE Micro’20, DAC’21]

PyMTL3 in Practice

PyMTL3 JIT
[DAC’18]

PyMTL3 Testing
[IEEE Design&Test’21]

PyMTL3 Gradual Typing
[LATTE’23]
PyMTL3 Publications


PyMTL3 Developers

- **Shunning Jiang**: Lead researcher and developer for PyMTL3
- **Peitian Pan**: Leading work on translation & gradually-typed HDL
- **Yanghui Ou**: Leading work on property-based random testing
- **Tuan Ta, Moyang Wang, Khalid Al-Hawaj, Shady Agwal, Lin Cheng**
PyMTL3 Project Sponsors

Funding partially provided by the National Science Foundation through NSF CRI Award #1512937 and NSF SHF Award #1527065.

Funding partially provided by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA.

Funding partially provided by the Defense Advanced Research Projects Agency through a DARPA POSH Award #FA8650-18-2-7852.

Funding partially provided by an unrestricted industry gift from the Xilinx University Program
PyMTL3: A Python Framework for Hardware Modeling, Generation, Simulation, and Verification

PyMTL3 Motivation

PyMTL3 Framework
[IEEE Micro’20, DAC’21]

PyMTL3 in Practice

PyMTL3 JIT
[DAC’18]

PyMTL3 Testing
[IEEE Design&Test’21]

PyMTL3 Gradual Typing
[LATTE’23]
This work was supported in part by NSF XPS Award #1337240, NSF CRI Award #1512937, NSF SHF Award #1527065, AFOSR YIP Award #FA9550-15-1-0194, DARPA Young Faculty Award #N66001-12-1-4239, DARPA POSH Award #FA8650-18-2-7852, a Xilinx University Program industry gift, and the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, and equipment, tool, and/or physical IP donations from Intel, NVIDIA, Synopsys, and ARM.

Thanks to Derek Lockhart, Ji Kim, Shreesha Srinath, Berkin Ilbeyi, Yixiao Zhang, Jacob Glueck, Aaron Wisner, Gary Zibrat, Christopher Torng, Cheng Tan, Raymond Yang, Kaishuo Cheng, Jack Weber, Carl Friedrich Bolz, David Maclver, and Zac Hatfield-Dodds for their help designing, developing, testing, and using PyMTL2 and 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.