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

Shunning Jiang, Christopher Torng, Christopher Batten

Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University
Outline

- Introduction
- PyMTL features
- PyMTL use cases
The Traditional Flow

- HDL: hardware description language
- DUT: design under test
- TB: test bench
- synth: synthesis

Traditional hardware description language
- Example: Verilog

✓ Fast edit-debug-sim loop
✓ Single language for design and testbench

X Difficult to parameterize
X Require specific ways to build powerful testbench
Hardware Preprocessing Frameworks (HPF)

Traditional hardware description language
- Example: Verilog
  ✓ Fast edit-debug-sim loop
  ✓ Single language for design and testbench
  ✗ Difficult to parameterize
  ✗ Require specific ways to build powerful testbench

Hardware preprocessing framework (HPF)
- Example: Genesis2
  ✓ Better parametrization with insignificant coding style change
  ✗ Multiple languages create “semantic gap”
  ✗ Still not easy to build powerful testbench
Hardware Generation Frameworks (HGF)

Traditional hardware description language
- Example: Verilog
  ✓ Fast edit-debug-sim loop
  ✓ Single language for design and testbench
  ✗ Difficult to parameterize
  ✗ Require specific ways to build powerful testbench

Hardware preprocessing framework (HPF)
- Example: Genesis2
  ✓ Better parametrization with insignificant coding style change
  ✗ Multiple languages create “semantic gap”
  ✗ Still not easy to build powerful testbench

Hardware generation framework (HGF)
- Example: Chisel
  ✓ Powerful parametrization
  ✓ Single language for design
  ✗ Slower edit-debug-sim loop
  ✗ Yet still difficult to build powerful testbench (can only generate simple testbench)
PyMTL is an Hardware Generation and Simulation framework

- Powerful parametrization
- Single language for design and testbench
- Use host language for verification
- Easy to create highly parameterized generators
PyMTL framework

PyMTL Specifications (Python)
- Test & Sim Harnesses
- Model
- Config

Elaboration → Model Instance

PyMTL "Kernel" (Python)

PyMTL Passes (Python)
- Simulation Pass → Simulatable Model
- Translation Pass
- Analysis Pass
- Transform Pass
- Analysis Output
- New Model
Outline

- Introduction
- **PyMTL features**
- PyMTL use cases
Eight features that make PyMTL productive

- Multi-level modeling
- Method-based interfaces
- Highly parametrized static elaboration
- Analysis and transform passes
- Pure-Python simulation
- Property-based random testing
- Python/SystemVerilog integration
- Fast simulation speed
Multi-level modeling

- Functional-level modeling: quickly building reference model and testbench
- Cycle-level modeling: design space exploration
- Register-transfer-level modeling: generating hardware

Example: Accelerator designers only want to implement the accelerator in RTL. How about cache and processor to do end-to-end testing?
Highly parametrized static elaboration

PyMTL embeds the DSL into Python, so the hardware designs can use full Python’s expressive power to construct hardware.

```python
class Register( Model ):
    def __init__(s, nbits):
        type = Bits( nbits )
        s.in_ = InPort( type )
        s.out = OutPort( type )

    def seq_logic():
        s.out.next = s.in_

    def comb_logic():
        s.out.value = s.in_[s.sel]

class MuxReg( Model ):
    def __init__(s, nbits=8, nports=4):
        s.in_ = [ InPort( nbits ) for x in range( nports ) ]
        s.sel = InPort( bw( nports ) )
        s.out = OutPort( nbits )

    def connect():
        for i in range( nports ):
            s.connect( s.sel, s.mux.sel )
            s.connect( s.in_[i], s.mux.in_[i] )
            s.connect( s.mux.out, s.reg_.in_ )
            s.connect( s.reg_.out, s.out )
```
PyMTL passes

PyMTL analysis/transform passes systematically traverse through the design and/or transform the module hierarchy by mutating the internal data structures.

```python
# 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 ret
```

```python
# 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 >
```
Property-based random testing

Since the simulation is just executing a piece of Python code, we can leverage random testing frameworks that test Python software for testing hardware.

```python
import hypothesis
from hypothesis import strategies as st

@hypothesis.given(
    x = st.integers( 2, 100 ),
    y = st.integers( 2, 100 ),
    src_delay = st.integers( 0, 20 ),
    sink_delay = st.integers( 0, 20 ),
    test_msgs = st.data() )
def test_dut_hypothesis( x, y, src_delay, sink_delay, test_msgs ):
    ...
    hypothesis.assume( x + y <= 200 )
    hypothesis.event( "Testing x=%d, y=%d" % (x, y) )
    # compose_test_msg is another function that draws random numbers
    # from hypothesis strategies.
    msgs = test_msgs.draw( st.lists( compose_test_msg( x, y ),
                                       min_size=1, max_size=32 ) )
    run_dut_test( DUT(), msgs, x, y, src_delay, sink_delay, msgs )
```
PyMTL/SystemVerilog integration

- PyMTL can import SystemVerilog and co-simulate it with the same Python test harness.
- PyMTL can also compose multiple PyMTL/SystemVerilog designs and translate the larger design into SystemVerilog.

```python
# 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,
        )
```
Fast pure-Python simulation

With Mamba techniques, the next version of PyMTL gets an order of magnitude of speedup when simulating in a pure-Python environment.

- Design the framework from the ground up with a just-in-time compiler in mind
- Enhance the just-in-time compiler to recognize critical hardware constructs

<table>
<thead>
<tr>
<th></th>
<th>PyMTL</th>
<th>MyHDL</th>
<th>PyRTL</th>
<th>Migen</th>
<th>IVerilog</th>
<th>CVS</th>
<th>Mamba</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divider</td>
<td>118K CPS</td>
<td>0.8×</td>
<td>2.2×</td>
<td>0.03×</td>
<td>0.6×</td>
<td>9.3×</td>
<td>20×</td>
</tr>
<tr>
<td>1-core</td>
<td>20K CPS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1×</td>
<td>15×</td>
<td>16×</td>
</tr>
<tr>
<td>32-core</td>
<td>360 CPS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.8×</td>
<td>25×</td>
<td>12×</td>
</tr>
</tbody>
</table>
Outline

- Introduction
- PyMTL features
- PyMTL use cases
  - PyMTL in teaching: 400+ students across 2 universities
  - PyMTL in research: four ISCA/MICRO papers use PyMTL
  - PyMTL in silicon prototyping: three tape-outs, two of which completely use PyMTL
PyMTL in Silicon Prototyping: BRGTC1 (2016)

- Fabricated in IBM 130nm
- 2mm x 2mm die, 1.2M transistor
PyMTL in Silicon Prototyping: BRGTC2 (2018)

- Fabricated in TSMC 28nm
- 1mm x 1.25mm die, 6.7M transistor
- Quad-core in-order RV32IMAF

- Advertisement: our open-source modular VLSI build system used in this tapeout
  https://github.com/cornell-brg/alloy-asic
We expect a new release in 2019.

PyMTL: [https://github.com/cornell-brg/pymtl](https://github.com/cornell-brg/pymtl)

Modular ASIC Build system: [https://github.com/cornell-brg/alloy-asic](https://github.com/cornell-brg/alloy-asic)