Cornell University
Computer Systems Laboratory

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

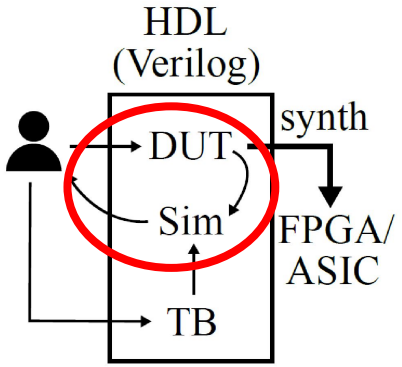
Shunning Jiang, Christopher Torng, Christopher Batten

Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University

Outline

- **Introduction**
- PyMTL features
- PyMTL use cases

The Traditional Flow



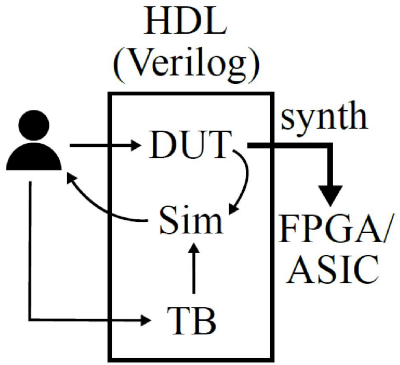
Traditional hardware description language
- Example: Verilog

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

X Difficult to parameterize
X Require specific ways to build powerful testbench

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

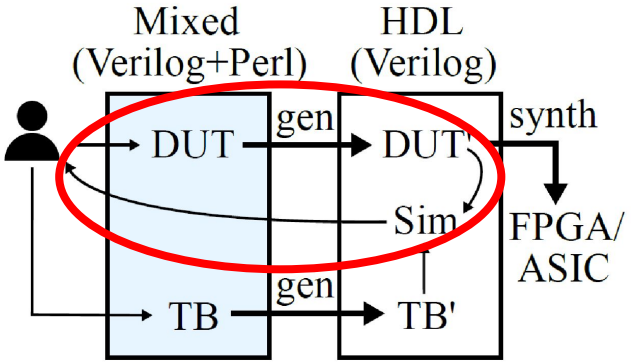
Hardware Preprocessing Frameworks (HPF)



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

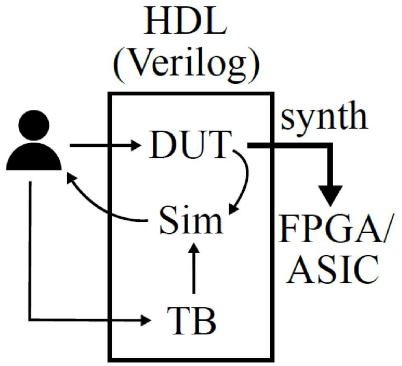


Hardware *preprocessing* framework (HPF)
- Example: Genesis2

✓ Better parametrization with insignificant coding style change

X Multiple languages create "semantic gap"
X Still not easy to build powerful testbench

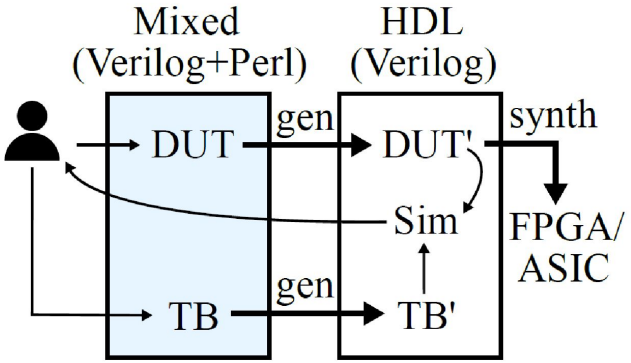
Hardware Generation Frameworks (HGF)



Traditional hardware description language
- Example: Verilog

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

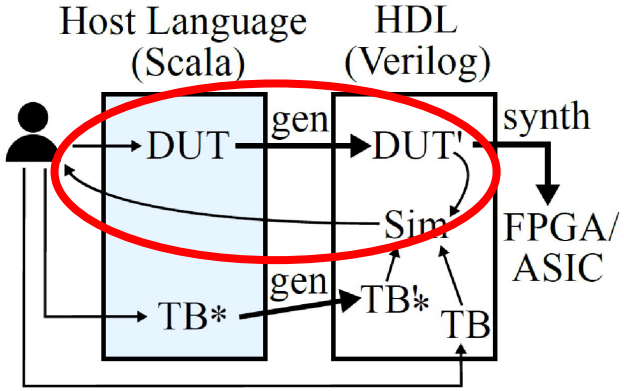
X Difficult to parameterize
X Require specific ways to build powerful testbench



Hardware *preprocessing* framework (HPF)
- Example: Genesis2

✓ Better parametrization with insignificant coding style change

X Multiple languages create "semantic gap"
X Still not easy to build powerful testbench

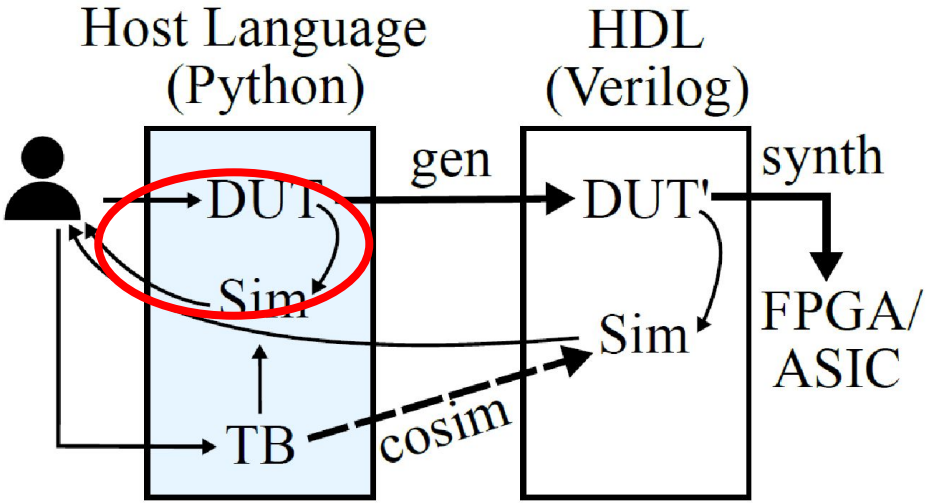


Hardware *generation* framework (HGF)
- Example: Chisel

✓ Powerful parametrization
✓ Single language for design

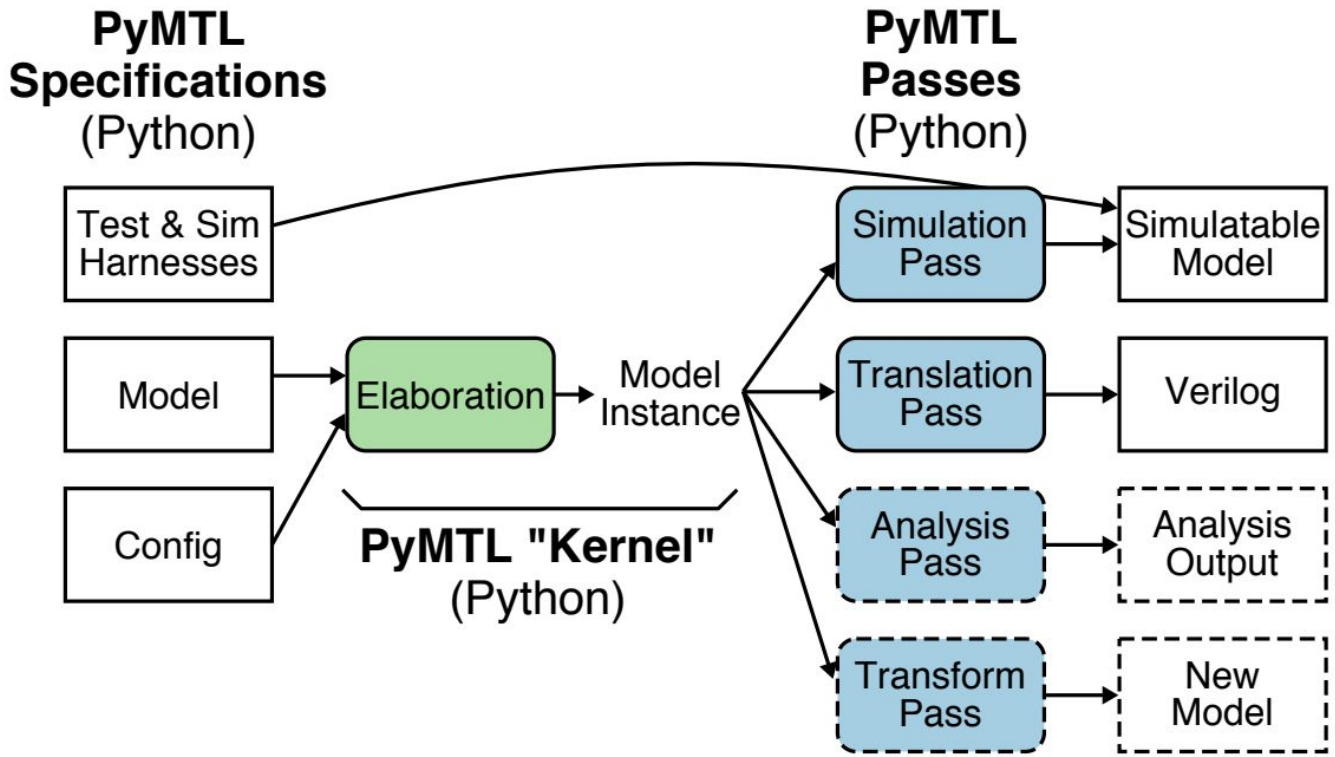
X Slower edit-debug-sim loop
X Yet still difficult to build powerful testbench (can only generate simple testbench)

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



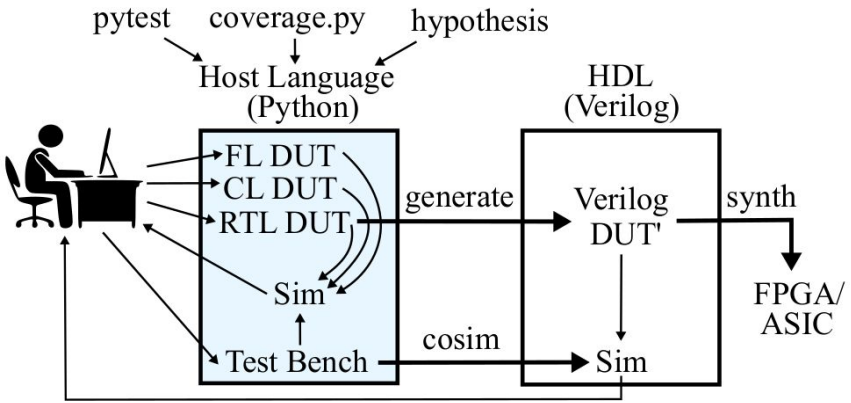
Outline

- 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 )

    @s.tick_rtl
    def seq_logic():
      s.out.next = s.in_
```

```python
class Mux( Model ):
  def __init__(s, nbits, nports):
    s.in_ = InPort[nports](nbits)
    s.sel = InPort (bw(nports))
    s.out = OutPort(nbits)

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

```python
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 )

    s.reg_ = Register( nbits )
    s.mux  = Mux     ( nbits )

    s.connect( s.sel, s.mux.sel )
    for i in range( nports ):
      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.

```
# 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
```

```
# 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.

- hypothesis

```
1   import hypothesis
2   from   hypothesis import strategies as st
3
4   @hypothesis.given(
5     x = st.integers( 2, 100 ),
6     y = st.integers( 2, 100 ),
7     src_delay  = st.integers( 0, 20 ),
8     sink_delay = st.integers( 0, 20 ),
9     test_msgs  = st.data() )
10  def test_dut_hypothesis( x, y, src_delay, sink_delay, test_msgs ):
11    ...
12    hypothesis.assume( x + y <= 200 )
13    hypothesis.event( "Testing x=%d, y=%d" % (x, y) )
14    # compose_test_msg is another function that draws random numbers
15    # from hypothesis strategies.
16    msgs = test_msgs.draw( st.lists( compose_test_msg( x, y ),
17                           min_size=1, max_size=32 ) )
18
19    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.

```
# 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**.

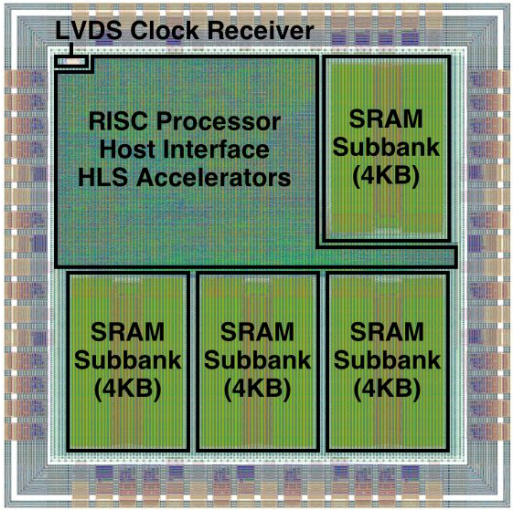
- Design the framework from the ground up with a just-in-time compiler in mind
- Enhance the just-in-time compiler to recognize critical hardware constructs

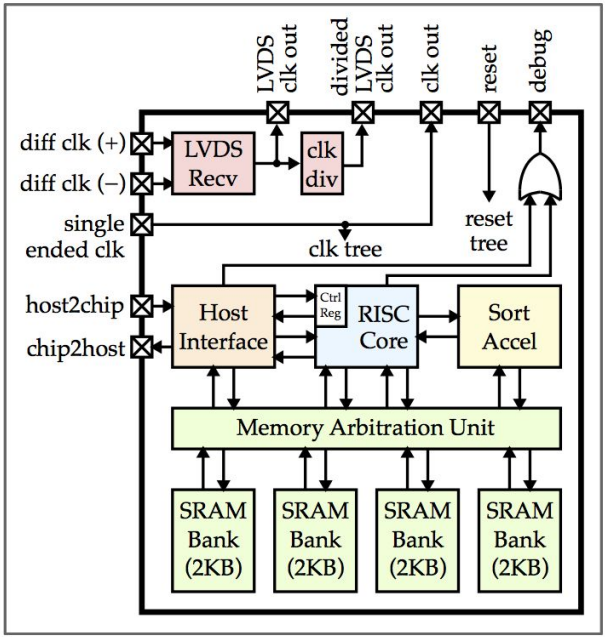
	PyMTL	MyHDL	PyRTL	Migen	IVerilog	CVS	Mamba
Divider	118K CPS	0.8×	2.2×	0.03×	0.6×	9.3×	20×
1-core	20K CPS	-	-	-	1×	15×	16×
32-core	360 CPS	-	-	-	1.8×	25×	12×

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**

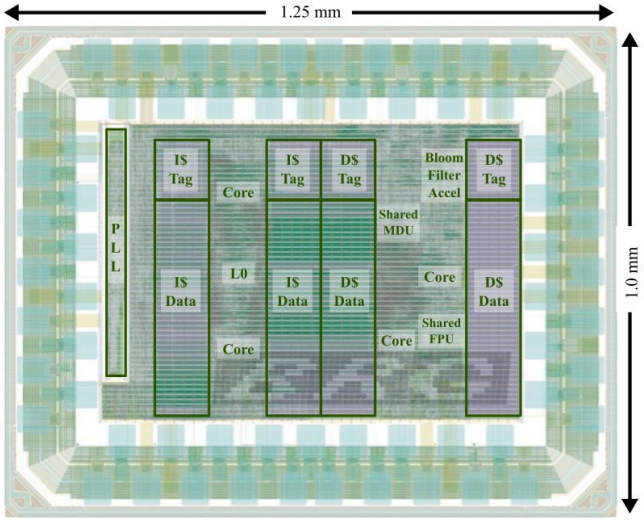
PyMTL in Silicon Prototyping: BRGTC1 (2016)



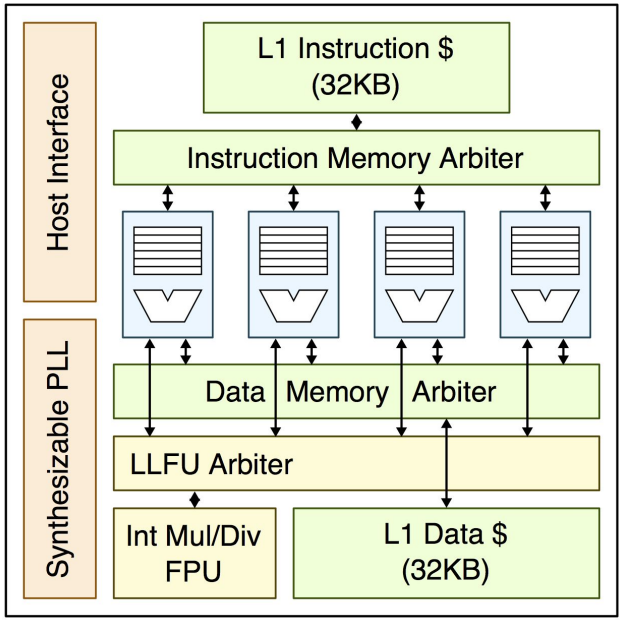


- Fabricated in IBM 130nm
- 2mm x 2mm die, 1.2M transistor

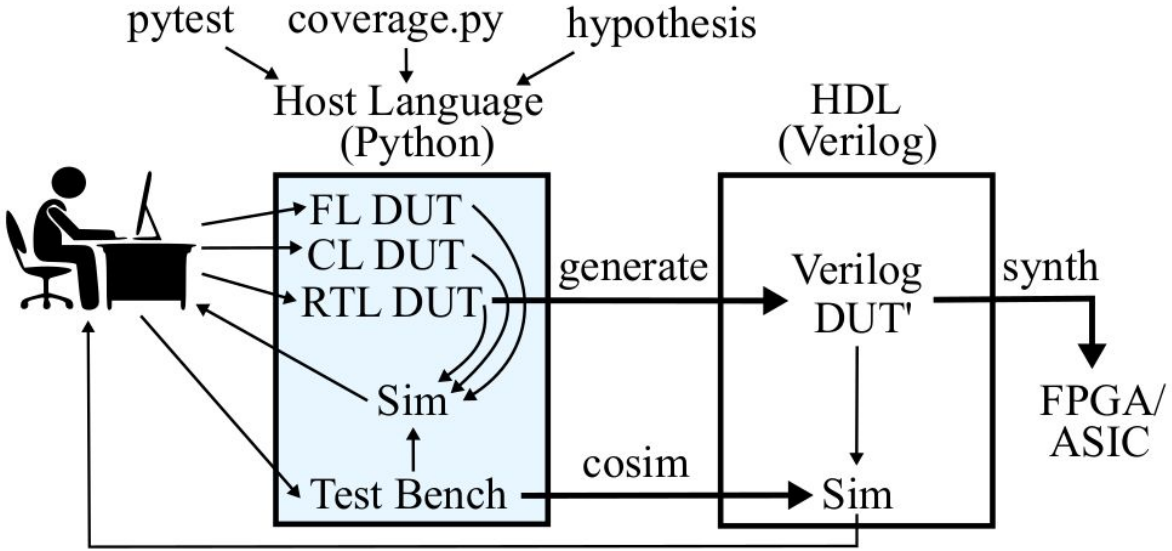
PyMTL in Silicon Prototyping: BRGTC2 (2018)



- 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



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
- Fast simulation speed

We expect a new release in 2019.

PyMTL: https://github.com/cornell-brg/pymtl

Modular ASIC Build system: https://github.com/cornell-brg/alloy-asic