PyMTL Tutorial

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

Batten Research Group

Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University

DARPA POSH Kick-Off @ Princeton, October 2018
Presentation: PyMTL Introduction

Hands-On: Max/RegIncr

Presentation: Multi-Level Modeling

Hands-On: GCD Unit
Multi-Level Modeling Methodologies

- Applications
- Algorithms
- Compilers
- Instruction Set Architecture
- Microarchitecture
- VLSI
- Transistors
Multi-Level Modeling Methodologies

Functional-Level Modeling
– Behavior

- Applications
- Algorithms
- Compilers
- Instruction Set Architecture
- Microarchitecture
- VLSI
- Transistors
Multi-Level Modeling Methodologies

**Functional-Level Modeling**
- Behavior

**Cycle-Level Modeling**
- Behavior
- Cycle-Approximate
- Analytical Area, Energy, Timing
Multi-Level Modeling Methodologies

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

Applications

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

Algorithms

Instruction Set Architecture

Compilers

Microarchitecture

VLSI

Transistors
Multi-Level Modeling Methodologies

Applications
Algorithms
Compilers
Instruction Set Architecture
Microarchitecture
VLSI
Transistors

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
Multi-Level Modeling Methodologies

Multi-Level Modeling Challenge
FL, CL, RTL modeling use very different languages, patterns, tools, and methodologies

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
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

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
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
VLSI Design Methodologies

HDL
Hardware Description Language

Fast edit-sim-debug loop
Single language for structural, behavioral, + TB
Difficult to create highly parameterized generators
VLSI Design Methodologies

**HDL (Hardware Description Language)**
- HDL (Verilog)
- DUT
- Sim
- TB
- fast edit-sim-debug loop
- single language for structural, behavioral, + TB
- difficult to create highly parameterized generators

**HPF (Hardware Preprocessing Framework)**
- Mixed (Verilog+Perl)
- DUT
- Sim
- TB
- slow edit-sim-debug loop
- multiple languages create "semantic gap"
- easier to create highly parameterized generators

Example: Genesis2

- HDL synthesis
- HPF generation
- FPGA/ASIC synthesis
- DUT
- Sim
- TB
VLSI Design Methodologies

HDL
Hardware Description Language

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

HPF
Hardware Preprocessing Framework

- Mixed (Verilog+Perl)
- DUT
- gen
- TB
- gen
- FPGA/ASIC

HGF
Hardware Generation Framework

- Host Language (Scala)
- DUT
- gen
- TB
- gen
- FPGA/ASIC

Examples:
- Genesis2
- Chisel

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

Slower edit-sim-debug loop
Multiple languages create "semantic gap"
Easier to create highly parameterized generators

Slower edit-sim-debug loop
Single language for structural + behavioral
Easier to create highly parameterized generators

Cannot use power of host language for verification
Productive Multi-Level Modeling and VLSI Design

Multi-Level Modeling
SystemC

VLSI Design
Chisel
**Productive Multi-Level Modeling and VLSI Design**

Multi-Level Modeling
SystemC

HGSF
Hardware Generation and Simulation Framework

VLSI Design
Chisel

Host Language (Python)

DUT
Sim
TB
cosim

tb

HDL (Verilog)

dut

FPGA/ASIC

DUT'
synthetic

Princeton 2018
PyMTL: Open-Source Python-Based Hardware Generation, Simulation, and Verification
Productive Multi-Level Modeling and VLSI Design

Multi-Level Modeling
SystemC

VLSI Design
Chisel

HGSF
Hardware Generation and Simulation Framework

- Single framework for ML modeling & VLSI design
- Fast edit-sim-debug loop
- Single language for structural, behavioral, + TB
- Easy to create highly parameterized generators
- Use power of host language for verification
PyMTL is a Python-based hardware generation and simulation framework for SoC design which enables productive multi-level modeling and VLSI implementation.
The PyMTL Framework

PyMTL Specifications (Python)
The PyMTL Framework

PyMTL Specifications
(Python)

Model
Config

PyMTL "Kernel"
(Python)

Elaboration

Model Instance
The PyMTL Framework

PyMTL Specifications (Python)

- Model
- Config

Test & Sim Harnesses

Elaboration

Model Instance

PyMTL "Kernel" (Python)

Simulation Pass

Translation Pass

Python

Simulatable Model

Verilog

PyMTL Passes (Python)
The PyMTL Framework

PyMTL Specifications (Python)

- Model
- Config
- Test & Sim Harnesses

PyMTL "Kernel" (Python)

- Elaboration

PyMTL Passes (Python)

- Simulation Pass
- Translation Pass
- Analysis Pass
- Transform Pass

Simulatable Model
Verilog
Analysis Output
New Model
```python
from pymtl import *

class RegIncrRTL( Model ):

def __init__( s, dtype ):
    s.in_ = InPort( dtype )
    s.out = OutPort( dtype )
    s.tmp = Wire( dtype )

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

@s.combinational
def comb_logic():
    s.out.value = s.tmp + 1
```
```python
class GcdUnitFL(Model):
    def __init__(self):
        # Interface
        s.req = InValRdyBundle(GcdUnitReqMsg())
        s.resp = OutValRdyBundle(Bits(16))

        # Adapters (e.g., TLM Transactors)
        s.req_q = InValRdyQueueAdapter(s.req)
        s.resp_q = OutValRdyQueueAdapter(s.resp)

        # Concurrent block
        @s.tick_fl
        def block():
            req_msg = s.req_q.popleft()
            result = gcd(req_msg.a, req_msg.b)
            s.resp_q.append(result)
```

But isn’t Python too slow?
Performance/Productivity Gap

Python is growing in popularity in many domains of scientific and high-performance computing. How do they close this gap?

- Python-Wrapped C/C++ Libraries
  (NumPy, CVXOPT, NLPy, pythonoCC, gem5)

- Numerical Just-In-Time Compilers
  (Numba, Parakeet)

- Just-In-Time Compiled Interpreters
  (PyPy, Pyston)

- Selective Embedded Just-In-Time Specialization
  (SEJITS)
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.
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
Mamba is \(20\times\) faster than PyMTLv2, \(4.5\times\) faster than PyMTLv2 with hybrid co-simulation, comparable to commercial simulators.

Mamba uses careful co-optimization of the framework and the JIT.
PyMTL ASIC Tapeouts

**BRGTC1 in 2016**
RISC processor, 16KB SRAM
HLS-generated accelerator
2x2mm, 1.2M-trans, IBM 130nm

**BRGTC2 in 2018**
4xRV32IMAF cores with “smart” sharing L1$/LLFU, PLL
1x1.2mm, 6.7M-trans, TSMC 28nm
PyMTL and Open-Source Hardware

▶ State-of-the-art in open-source HDL simulators
  ▶ *Icarus Verilog*: Verilog interpreter-based simulator
  ▶ *Verilator*: Verilog AOT-compiled simulator
  ▶ *GHDL*: VHDL AOT-compiled simulator
  ▶ No open-source simulator supports modern verification environments

▶ PyMTL as an open-source design, simulation, verification environment
  ▶ Open-source hardware developers can use Verilog RTL for design and Python, a well-known general-purpose language, for verification
  ▶ PyMTL for FL design enables creating high-level golden reference models
  ▶ PyMTL for RTL design enables creating highly parameterized hardware components which is critical for encouraging reuse in an open-source ecosystem
PyMTL and Open-Source Hardware

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

DARPA POSH Open-Source Hardware Program
PyMTL used as a powerful open-source generator for both design and verification

Graduate ASIC Design Course
Labs use PyMTL for verification, PyMTL or Verilog for RTL design, standard ASIC flow
Hands-On: PyMTL Basics with Max/RegIncr

- Task 2.1: Experiment with Bits
- Task 2.2: Interactively simulate a max unit
- Task 2.3: Write a registered incrementer (RegIncr) model
- Task 2.4: Test the RegIncr model
- Task 2.5: Translate the RegIncr model into Verilog
- Task 2.6: Simulate the RegIncr model with line tracing
- Task 2.7: Simulate the RegIncr model with VCD dumping
- Task 2.8: Compose a pipeline with two RegIncr models
- Task 2.9: Compose a pipeline with N RegIncr models
- Task 2.10: Parameterize test to verify multiple Ns
Hands-On: PyMTL Basics with Max/RegIncr

- Task 2.1: Experiment with Bits
- Task 2.2: Interactively simulate a max unit
- Task 2.3: Write a registered incrementer (RegIncr) model
- Task 2.4: Test the RegIncr model
- Task 2.5: Translate the RegIncr model into Verilog
- Task 2.6: Simulate the RegIncr model with line tracing
- Task 2.7: Simulate the RegIncr model with VCD dumping
- Task 2.8: Compose a pipeline with two RegIncr models
- Task 2.9: Compose a pipeline with N RegIncr models
- Task 2.10: Parameterize test to verify multiple Ns
### Bits Class for Fixed-Bitwidth Values

#### PyMTL Bits Operators

<table>
<thead>
<tr>
<th>Logical Operators</th>
<th>Shift Operators</th>
<th>Relational Operators</th>
<th>Arith. Operators</th>
<th>Slice Operators</th>
<th>Other Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>&gt;&gt;</td>
<td>==</td>
<td>+</td>
<td>[x]</td>
<td>concat</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>!=</td>
<td>-</td>
<td>[x:y]</td>
</tr>
<tr>
<td>l</td>
<td>&lt;&lt;</td>
<td>&gt;</td>
<td>*</td>
<td>get/set bit x</td>
<td>zext</td>
</tr>
<tr>
<td>~</td>
<td></td>
<td>&gt;=</td>
<td>/</td>
<td>get/set bits</td>
<td></td>
</tr>
<tr>
<td>~~~</td>
<td></td>
<td>&lt;</td>
<td>%</td>
<td>x upto y</td>
<td></td>
</tr>
<tr>
<td>~</td>
<td></td>
<td>&lt;=</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Logical Operators**: \& (bitwise AND), | (bitwise OR), ^ (bitwise XOR), ~ (bitwise NOT)
- **Shift Operators**: >> (shift right), << (shift left)
- **Relational Operators**: == (equal), != (not equal), > (greater than), >= (greater than or equals), < (less than), <= (less than or equals)
- **Arith. Operators**: + (addition), - (subtraction), * (multiplication), / (division), % (modulo)
- **Slice Operators**: [x] (get/set bit x), [x:y] (get/set bits x upto y)
- **Other Functions**: concat, sext, zext

---

**Reduction Operators**

- reduce_and (reduce via AND)
- reduce_or (reduce via OR)
- reduce_xor (reduce via XOR)
Task 2.1: Experiment with Bits

% cd ~
% ipython

```python
>>> from pymtl import *
>>> a = Bits( 8, 5 )
>>> b = Bits( 8, 3 )
>>> a + b
Bits( 8, 0x08 )
>>> a - b
Bits( 8, 0x02 )
>>> a | b
Bits( 8, 0x07 )
>>> a & b
Bits( 8, 0x01 )
```

```python
>>> c = concat( a, b )
>>> c
Bits( 16, 0x0503 )
>>> c[0:8]
Bits( 8, 0x03 )
>>> c[8:16]
Bits( 8, 0x5 )
>>> exit()
```
Task 2.2: Interactively simulate a max unit

% cd ~/pymtl-tut/maxunit
% ipython

```python
>>> from pymtl import *
>>> from MaxUnitFL import MaxUnitFL
>>> model = MaxUnitFL( nbits=8, nports=3 )
>>> model.elaborate()
>>> sim = SimulationTool(model)
>>> sim.reset()
>>> model.in_[0].value = 2
>>> model.in_[1].value = 5
>>> model.in_[2].value = 3
>>> sim.cycle()
>>> model.out
Bits( 8, 0x05 )
>>> exit()
```
**Hands-On: PyMTL Basics with Max/RegIncr**

- Task 2.1: Experiment with Bits
- Task 2.2: Interactively simulate a max unit
- Task 2.3: Write a registered incrementer (RegIncr) model
- Task 2.4: Test the RegIncr model
- Task 2.5: Translate the RegIncr model into Verilog
- Task 2.6: Simulate the RegIncr model with line tracing
- Task 2.7: Simulate the RegIncr model with VCD dumping
- Task 2.8: Compose a pipeline with two RegIncr models
- Task 2.9: Compose a pipeline with N RegIncr models
- Task 2.10: Parameterize test to verify multiple Ns
% cd ~/pymtl-tut/build
% gedit ../regincr/RegIncrRTL.py

```python
class RegIncrRTL( Model ):

def __init__( s, dtype ):
    s.in_ = InPort ( dtype )
    s.out = OutPort( dtype )
    s.tmp = Wire ( dtype )

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

@s.combinational
def comb_logic():
    s.out.value = s.tmp + 1

% py.test ../regincr/RegIncrRTL_test.py --verbose
```
★ Task 2.4: Test the RegIncr model ★

% cd ~/pymtl-tut/build
% py.test ../regincr/RegIncrRTL_test.py --verbose

========== test session starts ==========
platform darwin -- Python 2.7.5 -- pytest-2.6.4
plugins: xdist
collected 2 items

../regincr/RegIncrRTL_test.py::test_simple[8] PASSED
../regincr/RegIncrRTL_test.py::test_random[8] PASSED

========== 2 passed in 0.36 seconds =========
Translating PyMTL RTL into Verilog

- PyMTL models written at the register-transfer level of abstraction can be translated into Verilog source using the TranslationTool.

- Generated Verilog can be used with commercial EDA toolflows to characterize area, energy, and timing.

Diagram showing the process from FL Model to Verilog RTL Model with steps indicated as Manual or Automated.
Translation as Part of PyMTL SimJIT-RTL

TranslationTool
(SimJIT-RTL)

Verilog Source

Verilator

RTL C++ Source

C Interface Source

LLVM/GCC

C Shared Library

Wrapper Gen

PyMTL CFFI Model Instance

PyMTL RTL Model Instance
Translation as Part of PyMTL SimJIT-RTL

TranslationTool
(SimJIT-RTL)

Verilog Source

Verilator

RTL C++ Source

Translation Cache

C Shared Library

PyMTL RTL Model Instance

Translation

C Interface Source

LLVM/GCC

Wrapper Gen

PyMTL CFFI Model Instance

Princeton 2018 PyMTL: Open-Source Python-Based Hardware Generation, Simulation, and Verification 25 / 75
PyMTL to Verilog Translation Limitations

The **TranslationTool** has limitations on what it can translate:

- **Static elaboration** can use arbitrary Python (connections $\Rightarrow$ connectivity graph $\Rightarrow$ structural Verilog)
- **Concurrent logic blocks** must abide by language restrictions
PyMTL to Verilog Translation Limitations

The **TranslationTool** has limitations on what it can translate:

- **Static elaboration** can use arbitrary Python
  (connections ⇒ connectivity graph ⇒ structural Verilog)

- **Concurrent logic blocks** must abide by language restrictions
  - Data must be communicated in/out/between blocks using **signals**
    (InPorts, OutPorts, and Wires)
  - Signals may only contain **bit-specific value types**
    (Bits/BitStructs)
  - Only pre-defined, **translatable operators/functions** may be used
    (no user-defined operators or functions)
  - Any variables that don’t refer to signals must be **integer constants**
% cd ~/pymtl-tut/build
% gedit ../regincre/RegIncrRTL_test.py

20   def test_simple( dtype, test_verilog ):  
21  
22       # instantiate the model and elaborate it  
23  
24       model = RegIncr( dtype )  
25  
26  if test_verilog:  
27        # add these two lines to 
28        model = TranslationTool( model )  
29        # enable testing translation  
30  
31       model.elaborate()  

% py.test ../regincre/RegIncrRTL_test.py -v --test-verilog
% gedit ./RegIncrRTL_*.v
Example Verilog Generated from Translation

```verilog
module RegIncrRTL_0x7a355c5a216e72a4 (
    input wire [0:0] clk,
    input wire [7:0] in_,
    output reg [7:0] out,
    input wire [0:0] reset
);

// register declarations
reg [7:0] tmp;

// PYMTL SOURCE:
// @s.tick_rtl
// def seq_logic():
//     s.tmp.next = s.in_
always @ (posedge clk) begin
    tmp <= in_;
end

// PYMTL SOURCE:
// @s.combinational
// def comb_logic():
//     s.out.value = s.tmp + 1
always @ (*) begin
    out = (tmp+1);
end

endmodule // RegIncrRTL_0x7a355c5a216e72a4
```
Hands-On: PyMTL Basics with Max/RegIncr

- Task 2.1: Experiment with Bits
- Task 2.2: Interactively simulate a max unit
- Task 2.3: Write a registered incrementer (RegIncr) model
- Task 2.4: Test the RegIncr model
- Task 2.5: Translate the RegIncr model into Verilog
- Task 2.6: Simulate the RegIncr model with line tracing
- Task 2.7: Simulate the RegIncr model with VCD dumping
- Task 2.8: Compose a pipeline with two RegIncr models
- Task 2.9: Compose a pipeline with N RegIncr models
- Task 2.10: Parameterize test to verify multiple Ns
Unit Tests vs. Simulators

**Unit Tests**: `ModelName_tests.py`
- Tests that verify the simulation behavior of a model isolation
- Test functions are executed by the `py.test` testing framework
- Unit tests should always be written before simulator scripts!

**Simulators**: `model-name-sim.py`
- Simulators are meant for model evaluation and stats collection
- Simulation scripts take commandline arguments for configuration
- Used for experimentation and (design space) exploration!
★ Task 2.6: Simulate RegIncr with line tracing ★

% cd ~/pymtl-tut/build
% python ../regincr/reg-incr-sim.py 10
% python ../regincr/reg-incr-sim.py 10 --trace
% python ../regincr/reg-incr-sim.py 20 --trace

0: 04e5f14d (00000000) 00000000
1: 7839d4fc (04e5f14d) 04e5f14e
2: 996ab63d (7839d4fc) 7839d4fd
3: 6d146dfc (996ab63d) 996ab63e
4: 9cb87fec (6d146dfc) 6d146dfd
5: ba43a338 (9cb87fec) 9cb87fed
6: a0c394ff (ba43a338) ba43a339
7: f72041ee (a0c394ff) a0c39500
...
Line Tracing vs. VCD Dumping

► Line Tracing

△ Shows a single cycle per line and uses text characters to indicate state and how data moves through a system

△ Provides a way to visualize the high-level behavior of a system (e.g., pipeline diagrams, transaction diagrams)

△ Enables quickly debugging high-level functionality and performance bugs at the commandline

△ Can be used for FL, CL, and RTL models

► VCD Dumping

△ Captures the bit-level activity of every signal on every cycle

△ Requires a separate waveform viewer to visualize the signals

△ Provides a much more detailed view of a design

△ Mostly used for RTL models
✿ **Task 2.7: Simulate RegIncr with VCD dumping** ✿

% cd ~/pymtl-tut/build
% python ../regincr/reg-increment-sim.py 10 --dump-vcd
% gtkwave ./reg-incr-rtl-10.vcd
Hands-On: PyMTL Basics with Max/RegIncr

- Task 2.1: Experiment with Bits
- Task 2.2: Interactively simulate a max unit
- Task 2.3: Write a registered incrementer (RegIncr) model
- Task 2.4: Test the RegIncr model
- Task 2.5: Translate the RegIncr model into Verilog
- Task 2.6: Simulate the RegIncr model with line tracing
- Task 2.7: Simulate the RegIncr model with VCD dumping
- Task 2.8: Compose a pipeline with two RegIncr models
- Task 2.9: Compose a pipeline with N RegIncr models
- Task 2.10: Parameterize test to verify multiple Ns
Structural Composition in PyMTL

- In PyMTL, more complex designs can be created by hierarchically composing models using structural composition.
- Models are structurally composed by connecting their ports using `s.connect()` or `s.connect_pairs()` statements.
- Data is communicated between PyMTL models using `InPorts` and `OutPorts`, not via method calls!
% cd ~/pymtl-tut/build
% gedit ../regincr/RegIncrPipeline.py

9  class RegIncrPipeline( Model ):
10   def __init__( s, dtype ):
11     s.in_ = InPort ( dtype )
12     s.out = OutPort( dtype )
13
14     s.incrs = [RegIncr( dtype ) for _ in range( 2 )]
15
16     s.connect( s.in_, s.incrs[0].in_ )
17     #-------------------------------------------------------
18     # TASK 2.8: Comment out the Exception and implement the
19     # structural composition below.
20     #-------------------------------------------------------

% py.test ../regincr/RegIncrPipeline_test.py -sv
Line Tracing from Pipelined RegIncr

../regincr/RegIncrPipeline_test.py::test_simple

0: 04 (00 00) 00
1: 06 (05 02) 02
2: 02 (07 06) 06
3: 0f (03 08) 08
4: 08 (10 04) 04
5: 00 (09 11) 11
6: 0a (01 0a) 0a
7: 0a (0b 02) 02
8: 0a (0b 0c) 0c

PASSED
Parameterizing Models in PyMTL

- Static elaboration code (everything inside `__init__` that is not in a decorated function) can use the full expressiveness of Python

- Static elaboration code constructs a connectivity graph of components, is always Verilog translatable (as long as leaf modules are translatable)

- Enables the creation of powerful and highly-parameterizable hardware generators
★ Task 2.9: Compose a pipeline with N RegIncrs ★

% cd ~/pymtl-tut/build
% gedit ../regincr/RegIncrParamPipeline.py

```python
9  class RegIncrParamPipeline( Model ):
10
11  def __init__( s, dtype, nstages ):
12      s.in_ = InPort( dtype )
13      s.out = OutPort( dtype )
14
15  s.incrs = [RegIncr( dtype ) for _ in range( nstages )]
16
17  assert len( s.incrs ) > 0
18
19  s.connect( s.in_, s.incrs[0].in_ )
20  for i in range( nstages - 1 ): pass
21  #-------------------------------------------------------
22  # TASK 2.9: Comment out the Exception and implement the

% py.test ../regincr/RegIncrParamPipeline_test.py -sv
```
Parameterizing Tests in PyMTL

- We leverage the opensource `py.test` package to drive test collection and execution in PyMTL

- Significantly simplifies process of writing unit tests, and enables functionality such as parallel/distributed test execution and coverage reporting via plugins

- More importantly, `py.test` has powerful facilities for writing extensive and highly parameterizable unit tests

- One example: the `@pytest.mark.parametrize` decorator
% cd ~/pymtl-tut/build
% gedit ../regincr/RegIncrParamPipeline_test.py

#---------------------------------------------------------------
# TASK 2.10: Change parametrizize to verify more pipeline depths!
#---------------------------------------------------------------
@pytest.mark.parametrize( 'nstages', [1,2,5,10] )
def test_simple( test_verilog, nstages ):
    # instantiate the model and elaborate it
    model = RegIncrParamPipeline( dtype = 8, nstages = nstages )

% py.test ../regincr/RegIncrParamPipeline_test.py -sv
Line Tracing from Pipelined RegIncr

../regincr/RegIncrParamPipeline_test.py::test_simple[5]

0: 04 (00 00 00 00 00) 00
1: 06 (05 02 02 02 02) 02
2: 02 (07 06 03 03 03) 03
3: 0f (03 08 07 04 04) 04
4: 08 (10 04 09 08 05) 05
5: 00 (09 11 05 0a 09) 09
6: 0a (01 0a 12 06 0b) 0b
7: 0e (0b 02 0b 13 07) 07
8: 10 (0f 0c 03 0c 14) 14
9: 0c (11 10 0d 04 0d) 0d
...

PASSED
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
Multi-Level Modeling in PyMTL

- FL modeling allows for the rapid creation of a working model. Designers can quickly experiment with interfaces and protocols.
- This design is *manually refined* into a PyMTL CL model that includes timing, which is useful for rapid design space exploration.
- Promising architectures can again be *manually refined* into a PyMTL RTL implementation to accurately model resources.
Multi-Level Modeling in PyMTL

- Verilog generated from PyMTL RTL can be passed to an EDA toolflow for accurate area, energy, and timing estimates.
- Throughout this process, the same PyMTL test harnesses can used to verify each model!
- Requires good design, the use of latency-insensitive interfaces helps considerably.
def sorter_network( input_list ):
    return sorted( input_list )

class SorterNetworkFL( Model ):
    def __init__( s, nbits, nports ):
        s.in_ = InPort[ nports ]( nbits )
        s.out = OutPort[ nports ]( nbits )

    @s.tick_fl
    def logic():
        for i, v in enumerate( sorted( s.in_ ) ):
            s.out[i].next = v
CL Model in PyMTL

```python
def sorter_network( input_list ):
    return sorted( input_list )

class SorterNetworkCL( Model ):
    def __init__( s, nbits, nports ):
        s.in_ = InPort [nports]( nbits )
        s.out = OutPort[nports]( nbits )

@s.tick_cl
def logic():
    # behavioral logic + timing delays
```

$[3, 1, 2, 0] \rightarrow f(x) \rightarrow [0, 1, 2, 3]$
def sorter_network( input_list ):
    return sorted( input_list )

class SorterNetworkRTL( Model ):
    def __init__( s, nbits, nports ):
        s.in_ = InPort[ nports ]( nbits )
        s.out = OutPort[ nports ]( nbits )

@s.tick_rtl
def seq_logic():
    # sequential logic

@s.combinational
def comb_logic():
    # combinational logic
PyMTL 102: The GCD Unit

- Computes the greatest-common divisor of two numbers.
- Uses a latency insensitive input protocol to accept messages only when sender has data available and GCD unit is ready.
- Uses a latency insensitive output protocol to send results only when result is done and receiver is ready.
PortBundles are used to simplify the handling of multi-signal interfaces, such as ValRdy:

```python
s.req = InValRdyBundle( dtype )
s.resp = OutValRdyBundle( dtype )
s.child = ChildModel( dtype )

# connecting bundled request ports individually
s.connect( s.req.msg, s.child.req.msg )
s.connect( s.req.val, s.child.req.val )
s.connect( s.req.rdy, s.child.req.rdy )

# connecting bundled response ports in bulk
s.connect( s.resp, s.child.resp )
```
BitStructs are used to simplify communicating and interacting with complex packages of data:

```python
# MemReqMsg(addr_nbits, data_nbits) is a BitStruct datatype:
# +-----------------------------+
# | type | addr   | len | data   |
# +-----------------------------+
dtype = MemReqMsg(32, 32)
s.in_ = InPort(dtype)

@s.tick
def logic():
    addr, data = s.in_[34:66], s.in_[0:32]
    addr, data = s.in_.addr, s.in_.data
```

# BitStructs are subclasses of Bits, we can slice them
# ... but it's usually more convenient to use fields!

The GCD request message can be implemented as a BitStruct that has two fields, one for each operand:

```
A (16 bits)   B (16 bits)
```

```
req_msg
req_val
req_rdy
```

```
resp_msg
resp_val
resp_rdy
```

```
GCD
in_q
out_q
```
Hands-On: FL, CL, RTL Modeling of a GCD Unit

- Task 3.1: Create a BitStruct for the GCD request
- Task 3.2: Build an FL model for the GCD unit
- Task 3.3: Create a latency insensitive test
- Task 3.4: Add timing to the GCD CL model
- Task 3.5: Fix the bug in the GCD RTL model
- Task 3.6: Verify generated Verilog GCD RTL
- Task 3.7: Experiment with the GCD simulator
- Task 3.8: Importing hand-written GCD Verilog
**Hands-On: FL, CL, RTL Modeling of a GCD Unit**

- **Task 3.1:** Create a BitStruct for the GCD request
- **Task 3.2:** Build an FL model for the GCD unit
- **Task 3.3:** Create a latency insensitive test
- **Task 3.4:** Add timing to the GCD CL model
- **Task 3.5:** Fix the bug in the GCD RTL model
- **Task 3.6:** Verify generated Verilog GCD RTL
- **Task 3.7:** Experiment with the GCD simulator
- **Task 3.8:** Importing hand-written GCD Verilog
★ Task 3.1: Create a BitStruct for the GCD request ★

% cd ~/pymtl-tut/build
% gedit ../gcd/GcdUnitMsg.py

12 #----------------------------------------------------------
13 # TASK 3.1: Comment out the Exception below.
14 # Implement GcdUnitMsg code shown on the slides.
15 #----------------------------------------------------------
16 class GcdUnitReqMsg( BitStructDefinition ):
17     
18     def __init__( s ):
19         s.a = BitField( 16 )
20         s.b = BitField( 16 )
21     
22     def __str__( s ):
23         return "{{}}:{{}}".format( s.a, s.b )

% py.test ../gcd/GcdUnitMsg_test.py -vs
Implementing latency insensitive communication protocols can be complex to implement and a challenge to debug.

PyMTL provides **Interface Adapters** which abstract away the complexities of ValRdy, and expose simplified method interfaces.
Implementing latency insensitive communication protocols can be complex to implement and a challenge to debug.

PyMTL provides Interface Adapters which abstract away the complexities of ValRdy, and expose simplified method interfaces.

```python
# Interface
s.req = InValRdyBundle ( GcdUnitReqMsg() )
# Adapters
s.resp = OutValRdyBundle ( Bits(16) )
```

```python
# Adapters
s.req_q = InValRdyQueueAdapter ( s.req )
s.resp_q = OutValRdyQueueAdapter ( s.resp )
```
**Hands-On: FL, CL, RTL Modeling of a GCD Unit**

- Task 3.1: Create a BitStruct for the GCD request
- Task 3.2: Build an FL model for the GCD unit
- Task 3.3: Create a latency insensitive test
- Task 3.4: Add timing to the GCD CL model
- Task 3.5: Fix the bug in the GCD RTL model
- Task 3.6: Verify generated Verilog GCD RTL
- Task 3.7: Experiment with the GCD simulator
- Task 3.8: Importing hand-written GCD Verilog
% cd ~/pymtl-tut/build
% gedit ../gcd/GcdUnitFL.py

```python
@s.tick_fl
def block():
    # Use adapter to pop value from request queue
    req_msg = s.req_q.popleft()

    # Use gcd function from Python's standard library
    result = gcd( req_msg.a, req_msg.b )

    # Use adapter to append result to response queue
    s.resp_q.append( result )

% py.test ../gcd/GcdUnitFL_test.py -v
```
To simplify testing of latency insensitive designs, PyMTL provides TestSources and TestSinks with ValRdy interfaces.

TestSources/TestSinks only transmit/accept data when the “design under test” is ready/valid.

Can be configured to insert random delays into valid/ready signals to verify latency insensitivity under various conditions.
 Task 3.3: Create a latency insensitive test ★

% cd ~/pymtl-tut/build
% gedit ../gcd/GcdUnitFL_simple_test.py

```python
22  class TestHarness (Model):
23
24      def __init__( s, src_msgs, sink_msgs ):
25
26          s.src  = TestSource (GcdUnitReqMsg(), src_msgs)
27          s.gcd  = GcdUnitFL ()
28          s.sink = TestSink (Bits(16), sink_msgs)
29
30          s.connect( s.src.out,  s.gcd.req  )
31          s.connect( s.gcd.resp, s.sink.in_ )

% py.test ../gcd/GcdUnitFL_simple_test.py -vs
```
PyMTL 102: Latency Insensitive FL Models

..gcd/GcdUnitFL_simple_test.py::test

2:  >     ()  > .
3: 000f:0005 > 000f:0005() >
4:  # > # (0005 > 0005
5:  # > # () >
6: 0003:0009 > 0003:0009() >
7:  # > # (0003 > 0003
8:  # > # () >
9: 001b:000f > 001b:000f() >
10: # > # (0003 > 0003
11: # > # () >
12: 0015:0031 > 0015:0031() >
13: . > . (0007 > 0007

PASSED
PyMTL 102: Latency Insensitive CL Models

- Cycle-level models add timing information to a functional model and can provide a cycle-approximate estimation of performance.

- Useful for rapid, initial exploration of an architectural design space.

- We’ll use a simple GCD algorithm to provide timing info.
Hands-On: FL, CL, RTL Modeling of a GCD Unit

- Task 3.1: Create a BitStruct for the GCD request
- Task 3.2: Build an FL model for the GCD unit
- Task 3.3: Create a latency insensitive test
- Task 3.4: Add timing to the GCD CL model
- Task 3.5: Fix the bug in the GCD RTL model
- Task 3.6: Verify generated Verilog GCD RTL
- Task 3.7: Experiment with the GCD simulator
- Task 3.8: Importing hand-written GCD Verilog
★ Task 3.4: Add timing to the GCD CL model ★

% cd ~/pymtl-tut/build
% py.test ../gcd/GcdUnitCL_test.py
% py.test ../gcd/GcdUnitCL_test.py -k basic_0x0 -sv
% gedit ../gcd/GcdUnitCL.py

67    # Handle delay to model the gcd unit latency
68
69    if s.counter > 0:
70        s.counter -= 1
71    if s.counter == 0:
72        s.resp_q.enq( s.result )
73
74    # If we have a new msg and output queue not full
75
76    elif not s.req_q.empty() and not s.resp_q.full():
77        req_msg = s.req_q.deq()
78        s.result,s.counter = gcd( req_msg.a, req_msg.b )

17    def gcd( a, b ):
18
19        ncycles = 1
20
21        while b:
22            ncycles += 1
23            a, b = b, a%b
24
25        return (a, ncycles)

% py.test ../gcd/GcdUnitCL_test.py -k basic_0x0 -sv
PyMTL 102: Latency Insensitive CL Models

PASSED

```python
../gcd_soln/GcdUnitCL_test.py::test[basic_0x0] ()
2: > () . > .
3: 000f:0005 > 000f:0005() >
4: 0003:0009 > 0003:0009() >
5: # > # ()0005 > 0005
6: 0000:0000 > 0000:0000() >
7: # > # ()0003 > 0003
8: 001b:000f > 001b:000f() >
9: # > # ()0000 > 0000
10: 0015:0031 > 0015:0031() >
11: # > # ()0003 > 0003
12: 0019:001e > 0019:001e() >
13: # > # ()0007 > 0007
14: 0013:001b > 0013:001b() >
15: # > # ()0005 > 0005
16: 0028:0028 > 0028:0028() >
17: # > # ()0001 > 0001
18: 00fa:00be > 00fa:00be() >
19: # > # ()0028 > 0028
20: 0005:00fa > 0005:00fa() >
21: # > # ()000a > 000a
22: ffff:00ff > ffff:00ff() >
23: . > . ()0005 > 0005
24: > () >
25: > ()00ff > 00ff
```

PASSED

```python
../gcd_soln/GcdUnitCL_test.py::test[basic_0x0] ()
2: > () . > .
3: 000f:0005 > 000f:0005() >
4: 0003:0009 > 0003:0009() >
5: # > # () >
6: 0000:0000 > 0000:0000() >
7: # > # ()0005 > 0005
8: 0000:0000 > 0000:0000() >
9: # > # () >
10: # > # ()0003 > 0003
11: 001b:000f > 001b:000f() >
12: # > # () >
13: 0015:0031 > 0015:0031() >
14: # > # () >
15: # > # () >
16: # > # () >
17: # > # ()0003 > 0003
18: 0019:001e > 0019:001e() >
19: # > # () >
20: # > # () >
21: # > # () >
22: # > # ()0007 > 0007
23: 0013:001b > 0013:001b() >
24: # > # () >
25: # > # () >
26: # > # () >
27: # > # ()0005 > 0005
28: 0028:0028 > 0028:0028() >
29: # > # () >
```
PyMTL 102: Latency Insensitive RTL Models

- RTL models allow us to accurately estimate executed cycles, cycle-time, area and energy when used with an EDA toolflow.

- Constructing is time consuming! PyMTL tries to make it more productive by providing a better design and testing environment.
Latency insensitive hardware generally separates logic into control and datapath (shown below).

Today, we won’t be writing RTL for GCD, but well be fixing a bug in the RTL implementation of the state machine.

Latency insensitive hardware generally separates logic into control and datapath (shown below).

Today, we won’t be writing RTL for GCD, but well be fixing a bug in the RTL implementation of the state machine.
★ Task 3.5: Fix the bug in the GCD RTL model ★

% cd ~/pymtl-tut/build
% py.test ../gcd/GcdUnitRTL_test.py -k basic_0x0 -v
% gedit ../gcd/GcdUnitRTL.py

```python
183    # Transitions out of IDLE state
184    if ( curr_state == s.STATE_IDLE ):
185        pass
186
187    # Transitions out of CALC state
188    if ( curr_state == s.STATE_CALC ):
189        if ( not s.is_a_lt_b and s.is_b_zero ):
190            next_state = s.STATE_DONE
191
192    # Transitions out of DONE state

% py.test ../gcd/GcdUnitRTL_test.py -k basic_0x0 -v
```
PyMTL 102: Latency Insensitive RTL Models

../gcd_soln/GcdUnitRTL_test.py::test[basic_0x0]

2: > (000f 0005 I ) > .
3: 000f:0005 > 000f:0005(000f 0005 I ) >
4: # > # (000f 0005 C-) >
5: # > # (000a 0005 C-) >
6: # > # (0005 0005 C-) >
7: # > # (0000 0005 Cs) >
8: # > # (0005 0000 C ) >
9: # > # (0005 0000 D )0005 > 0005
10: 0003:0009 > 0003:0009(0005 0000 I ) >
11: # > # (0003 0009 Cs) >
12: # > # (0009 0003 C-) >
13: # > # (0006 0003 C-) >
14: # > # (0003 0003 C-) >
15: # > # (0000 0003 Cs) >
Hands-On: FL, CL, RTL Modeling of a GCD Unit

- Task 3.1: Create a BitStruct for the GCD request
- Task 3.2: Build an FL model for the GCD unit
- Task 3.3: Create a latency insensitive test
- Task 3.4: Add timing to the GCD CL model
- Task 3.5: Fix the bug in the GCD RTL model
- Task 3.6: Verify generated Verilog GCD RTL
- Task 3.7: Experiment with the GCD simulator
- Task 3.8: Importing hand-written GCD Verilog
Task 3.6: Verify generated Verilog GCD RTL

% cd ~/pymtl-tut/build
% py.test ../gcd/GcdUnitRTL_test.py --test-verilog -sv
% gedit GcdUnitRTL_*.v
★ Task 3.7: Experiment with the GCD simulator ★

# Simulating both the CL and RTL models

% cd ~/pymtl-tut/build
% ../gcd/gcd-sim --stats --impl fl --input random
% ../gcd/gcd-sim --stats --impl cl --input random
% ../gcd/gcd-sim --stats --impl rtl --input random

# Experimenting with various datasets

% ../gcd/gcd-sim --impl rtl --input random --trace
% ../gcd/gcd-sim --impl rtl --input small --trace
% ../gcd/gcd-sim --impl rtl --input zeros --trace
**Hands-On:** FL, CL, RTL Modeling of a GCD Unit

- Task 3.1: Create a BitStruct for the GCD request
- Task 3.2: Build an FL model for the GCD unit
- Task 3.3: Create a latency insensitive test
- Task 3.4: Add timing to the GCD CL model
- Task 3.5: Fix the bug in the GCD RTL model
- Task 3.6: Verify generated Verilog GCD RTL
- Task 3.7: Experiment with the GCD simulator
- **Task 3.8: Importing hand-written GCD Verilog**
Task 3.8: Importing hand-written GCD Verilog

% cd ~/pymtl-tut/build
% gedit ../gcd/GcdUnitVerilog.py
% py.test ../gcd/GcdUnitVerilog_test.py -sv

```python
class GcdUnitVerilog( VerilogModel ):
    def __init__( s ):
        # Interface
        s.req = InValRdyBundle ( GcdUnitReqMsg() )
        s.resp = OutValRdyBundle ( Bits(16) )

        # Verilog ports
        s.set_ports({
            'clk' : s.clk,
            'reset' : s.reset,
            'req_val' : s.req.val,
            'req_rdy' : s.req.rdy,
            'req_msg' : s.req.msg,
            'resp_val' : s.resp.val,
            'resp_rdy' : s.resp.rdy,
            'resp_msg' : s.resp.msg,
        })
```
PyMTL Next Steps and More Resources

Next Steps:

- See the detailed tutorials on the Cornell ECE 5745 website:
  http://www.csl.cornell.edu/courses/ece5745/handouts.html

Check out the /docs directory in the PyMTL repo for guides on:

- Writing Pythonic PyMTL Models and Tests
- Writing Verilog Translatable PyMTL RTL

Check out Mamba, our ultra-fast new version of PyMTL:

- https://github.com/cornell-brg/mamba-dac2018