## PyMTL/Pydgin Tutorial Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30am – 8:50am</td>
<td>Virtual Machine Installation and Setup</td>
</tr>
<tr>
<td>8:50am – 9:00am</td>
<td><strong>Presentation:</strong> PyMTL/Pydgin Tutorial Overview</td>
</tr>
<tr>
<td>9:00am – 9:10am</td>
<td><strong>Presentation:</strong> Introduction to Pydgin</td>
</tr>
<tr>
<td>9:10am – 10:00am</td>
<td><strong>Hands-On:</strong> Adding a GCD Instruction using Pydgin</td>
</tr>
<tr>
<td>10:00am – 10:10am</td>
<td><strong>Presentation:</strong> Introduction to PyMTL</td>
</tr>
<tr>
<td>10:10am – 11:00am</td>
<td><strong>Hands-On:</strong> PyMTL Basics with Max/RegIncr</td>
</tr>
<tr>
<td>11:00am – 11:30am</td>
<td>Coffee Break</td>
</tr>
<tr>
<td>11:30am – 11:40am</td>
<td><strong>Presentation:</strong> Multi-Level Modeling with PyMTL</td>
</tr>
<tr>
<td>11:40am – 12:30pm</td>
<td><strong>Hands-On:</strong> FL, CL, RTL Modeling of a GCD Unit</td>
</tr>
</tbody>
</table>

ISCA 2015

PyMTL/Pydgin Tutorial: Python Frameworks for Highly Productive Computer Architecture Research
**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>Relational Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;</code></td>
<td><code>&gt;&gt;</code> shift right</td>
<td><code>==</code> equal</td>
<td><code>!=</code> not equal</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td><code>&lt;&lt;</code> shift left</td>
<td><code>&gt;</code> greater than</td>
</tr>
<tr>
<td><code>^</code></td>
<td></td>
<td><code>&lt;</code> less than</td>
<td><code>&lt;=</code> less than or equals</td>
</tr>
<tr>
<td><code>~~</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>~</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Arith. Operators

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `%` modulo

### Slice Operators

- `[x]` get/set bit `x`
- `[x:y]` get/set bits `x` upto `y`

### Shift Operators

- `>>` shift right
- `<<` shift left

### Relational Operators

- `==` equal
- `!=` not equal
- `>` greater than
- `>=` greater than or equals
- `<` less than
- `<=` less than or equals

### Other Functions

<table>
<thead>
<tr>
<th>Reduction Operators</th>
<th>Other Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reduce_and</code> reduce via AND</td>
<td><code>concat</code> concatenate</td>
</tr>
<tr>
<td><code>reduce_or</code> reduce via OR</td>
<td><code>sext</code> sign-extension</td>
</tr>
<tr>
<td><code>reduce_xor</code> reduce via XOR</td>
<td><code>zext</code> zero-extension</td>
</tr>
</tbody>
</table>
% cd \midtilde
% ipython

```py
>>> 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 )
```

```
>>> c = concat( a, b )
>>> c
Bits( 16, 0x0503 )
>>> c[0:8]
Bits( 8, 0x03 )
>>> c[8:16]
Bits( 8, 0x05 )
>>> exit()
```
★ Task 2.2: Interactively simulate a max unit ★

```python
% cd ~/pymtl-tut/maxunit
% ipython

>>> 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
★ Task 2.3: Write a registered incrementer model ★

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

```python
class RegIncrRTL(Model):
    def __init__(self, dtype):
        s.in_ = InPort(dtype)
        s.out = OutPort(dtype)
        s.tmp = Wire(dtype)  #
    @s.tick_rtl  #
    def seq_logic():  # add these lines  # to implement the
        s.tmp.next = s.in_  # registered
    @s.combinational  # incrementer
    def comb_logic():  # behavior
        s.out.value = s.tmp + 1

% py.test ../regincr/RegIncrRTL_test.py --verbose
```

ISCA 2015
PyMTL/Pydgin Tutorial: Python Frameworks for Highly Productive Computer Architecture Research
★ 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.
Translation as Part of PyMTL SimJIT-RTL

TranslationTool
(SimJIT-RTL)

Verilog
Source

Verilator

RTL C++
Source

Translation

C Interface
Source

LLVM/GCC

Translation
Cache

C Shared
Library

Wrapper
Gen

PyMTL
RTL Model
Instance

PyMTL
CFFI Model
Instance
The **Translation Tool** 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
  - 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**
★ Task 2.5: Translate the RegIncr model into Verilog ★

```python
% cd ~/pymtl-tut/build
% gedit ../regincr/RegIncrRTL_test.py

20 def test_simple( dtype, test_verilog ):
    model = RegIncr( dtype )
    if test_verilog:
        model = TranslationTool( model ) # add these two lines to enable testing translation
    model.elaborate()

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

```verilog
4 // dtype: 8
5 // dump-vcd: False
6 `default_nettype none
7 module RegIncrRTL_0x7a355c5a216e72a4
8 (
9   input wire [ 0:0] clk,
10   input wire [ 7:0] in_,
11   output reg  [ 7:0] out,
12   input wire [ 0:0] reset
13 );

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

// logic for seq_logic()
27 always @ (posedge clk) begin
28   tmp <= in_;
29 end

// PYMTL SOURCE:
31 // @s.tick_rtl
32 // def seq_logic():
33 // s.tmp.next = s.in_

// logic for comb_logic()
38 always @ (*) begin
39   out = (tmp+1);
40 end

// PYMTL SOURCE:
43 endmodule // RegIncrRTL_0x7a355c5a216e72a4
44 `default_nettype wire
```
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
...

% 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-incr-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
11    def __init__( s, dtype ):
12        s.in_ = InPort( dtype )
13        s.out = OutPort( dtype )
14
15        s.incrs = [RegIncr( dtype ) for _ in range( 2 )]
16
17        s.connect( s.in_, s.incrs[0].in_ )
18
19        # TASK 2.8: Comment out the Exception and implement the
20        # structural composition below.
21
% 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.
% cd ~/pymtl-tut/build
% gedit ../regincr/RegIncrParamPipeline.py

class RegIncrParamPipeline( Model ):  
    def __init__( s, dtype, nstages ):  
        s.in_ = InPort( dtype )  
        s.out = OutPort( dtype )  

        s.incrs = [RegIncr( dtype ) for _ in range( nstages )]
    
        assert len( s.incrs ) > 0

        s.connect( s.in_, s.incrs[0].in_ )
        for i in range( nstages - 1 ): pass  
        #-------------------------------------------------------  
        # 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 parametrize to verify more pipeline depths!

@pytest.mark.parametrize( 'nstages', [1,2,5,10] )
def test_simple( test_verilog, nstages ):
    
    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