## 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>Presentation: PyMTL/Pydgin Tutorial Overview</td>
</tr>
<tr>
<td>9:00am – 9:10am</td>
<td>Presentation: Introduction to Pydgin</td>
</tr>
<tr>
<td>9:10am – 10:00am</td>
<td>Hands-On: Adding a GCD Instruction using Pydgin</td>
</tr>
<tr>
<td>10:00am – 10:10am</td>
<td>Presentation: Introduction to PyMTL</td>
</tr>
<tr>
<td>10:10am – 11:00am</td>
<td>Hands-On: 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>Presentation: Multi-Level Modeling with PyMTL</td>
</tr>
<tr>
<td>11:40am – 12:30pm</td>
<td>Hands-On: FL, CL, RTL Modeling of a GCD Unit</td>
</tr>
</tbody>
</table>
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():
    # BitStructs are subclasses of Bits, we can slice them
    addr, data = s.in_[34:66], s.in_[0:32]

    # ... but it's usually more convenient to use fields!
    addr, data = s.in_.addr, s.in_.data
```
PyMTL 102: Complex Datatypes

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

```
A
(req_msg, req_val, req_rdy)

32

B
(resp_msg, resp_val, resp_rdy)

16
```

**Example Diagram:**

![Diagram of GCD request and response messages with fields and data widths]
**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
**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 3.1: Create a BitStruct for the GCD request**

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

class GcdUnitReqMsg( BitStructDefinition ):

    def __init__( s ):
        s.a = BitField( 16 )
        s.b = BitField( 16 )

    def __str__( s ):
        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() )
s.resp = OutValRdyBundle ( Bits(16) )

# 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 3.2: Build an FL model for the GCD unit ★

```python
% cd ~/pymtl-tut/build
% gedit ../gcd/GcdUnitFL.py

31 @s.tick_fl
32 def block():
33
34     # Use adapter to pop value from request queue
35     req_msg = s.req_q.popleft()
36
37     # Use gcd function from Python's standard library
38     result = gcd( req_msg.a, req_msg.b )
39
40     # Use adapter to append result to response queue
41     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

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 3.4: 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

```python
67  # Handle delay to model the gcd unit latency
68  69  if s.counter > 0:
70      s.counter -= 1
71    elif s.counter == 0:
72        s.resp_q.enq( s.result )
73
74  # If we have a new message and the output queue is 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      ncycles = 1
19      while b:
20          ncycles += 1
21          a, b = b, a%b
22      return (a, ncycles)
23
24  25

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

../gcd_soln/GcdUnitCL_test.py::test[basic_0x0] ()

```python
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: > ()0ff > 00ff
```
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.
% 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
188     # Transitions out of CALC state
189     if ( curr_state == s.STATE_CALC ):
190         if ( not s.is_a_lt_b and s.is_b_zero ):
191             next_state = s.STATE_DONE
192
194     # Transitions out of DONE state

% py.test ../gcd/GcdUnitRTL_test.py -k basic_0x0 -v
```
../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
% cd ~/pymtl-tut/build
% py.test ../gcd/GcdUnitRTL_test.py --test-verilog -sv
% gedit GcdUnitRTL_*.v

module GcdUnitRTL_0x791afe0d4d8c
(input wire [ 0:0] clk,
input wire [31:0] req_msg,
output wire [ 0:0] req_rdy,
input wire [ 0:0] req_val,
input wire [ 0:0] reset,
output wire [15:0] resp_msg,
input wire [ 0:0] resp_rdy,
output wire [ 0:0] resp_val
);

wire [ 0:0] ctrl$b_reg_en;
wire [ 0:0] ctrl$a_reg_en;
wire [ 0:0] ctrl$req_rdy;

GcdUnitCtrlRTL_0x791afe0d4d8c ctrl
(is_b_zero ( ctrl$is_b_zero ),
resp_rdy ( ctrl$resp_rdy ),
clk ( ctrl$clk ),
is_a_lt_b ( ctrl$is_a_lt_b ),
req_val ( ctrl$req_val ),
reset ( ctrl$reset ),
a_mux_sel ( ctrl$a_mux_sel ),
resp_val ( ctrl$resp_val ),
b_mux_sel ( ctrl$b_mux_sel ),
reg_en ( ctrl$b_reg_en ),
a_reg_en ( ctrl$a_reg_en ),
req_rdy ( ctrl$req_rdy );

wire [ 0:0] ctrl$reset;
wire [ 1:0] ctrl$a_mux_sel;
wire [ 0:0] ctrl$resp_val;
wire [ 0:0] ctrl$b_mux_sel;

wire [ 1:0] dpath$a_mux_sel;
★ 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
PyMTL In Practice: Matrix Vector Accelerator

- In the PyMTL paper [MICRO’14], we discuss how multi-level modeling in PyMTL can facilitate the design of coprocessors.

- Selecting FL/CL/RTL models for the cache/processor/accelerator allows designers to tradeoff simulation speed and accuracy.

- PyMTL-generated Verilog passed into Synopsys toolflow for area/energy/timing estimates.
In the XLOOPS paper (published in MICRO’14), PyMTL was combined with gem5 to evaluate an architecture for loop acceleration.

- gem5 provided access to complex out-of-order processor and memory system models (red).
- PyMTL was used to quickly build and iterate on a CL model for the loop acceleration unit (blue).

We are currently experimenting with accelerators generated using high-level synthesis.

We can import the HLS-generated Verilog into PyMTL, and then use PyMTL to verify these accelerators and compose accelerators using various interconnects.

We can also include our own accelerators written in PyMTL using FL, CL, and RTL modeling.

We then use PyMTL+gem5 integration to experiment with tightly integrated general-purpose processors with accelerators.
PyMTL Next Steps and More Resources

Next Steps:

► See the detailed tutorial on the Cornell ECE5745 website:
  http://www.csl.cornell.edu/courses/ece5745/handouts/ece5745-tut-pymtl.pdf

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

► Writing Pythonic PyMTL Models and Tests
► Writing Verilog Translatable PyMTL RTL
► Importing Verilog Components into PyMTL
► **Coming Soon**: Embedding PyMTL Models into gem5

Become a contributor! We’d love your PyMTL hacks and models!

► https://github.com/cornell-brg/pymtl
► https://github.com/cornell-brg/pydgin
Thank you for coming!


Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers

[ MICRO 2014 ]
https://github.com/cornell-brg/pymtl

[ ISPASS 2015 ]
https://github.com/cornell-brg/pydgin