## PyMTL Tutorial Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:45 – 2:00pm</td>
<td>Virtual Machine Installation and Setup</td>
</tr>
<tr>
<td>2:00 – 2:15pm</td>
<td>Presentation: PyMTL Overview</td>
</tr>
<tr>
<td>2:15 – 2:55pm</td>
<td>Part 1: PyMTL Basics</td>
</tr>
<tr>
<td>2:55 – 3:30pm</td>
<td>Part 2: Multi-Level Modeling with PyMTL</td>
</tr>
<tr>
<td>3:30 – 4:00pm</td>
<td>Coffee Break</td>
</tr>
<tr>
<td>4:00 – 4:45pm</td>
<td>Part 3: Processor Modeling with PyMTL</td>
</tr>
<tr>
<td>4:45 – 5:30pm</td>
<td>Part 4: Multi-Level Composition in PyMTL</td>
</tr>
</tbody>
</table>
PyMTL

Python-based hardware generation, simulation, and verification framework which enables productive multi-level modeling and VLSI design.

Python

Functional-Level

Cycle-Level

RTL

Multi-Level Simulation

Test Bench

SystemVerilog

generate

cycle-simulate

synthesize

FPGA

ASIC

prototype

bring-up
Fletcher’s Checksum Algorithm

- Traditional modular sum checksum algorithm
  - Accumulates input words and discards any overflow bits
  - Produces the same checksum if the input data is rearranged

- Fletcher’s checksum algorithm
  - Accumulates input words, also accumulates the cumulative sum values
  - Produces a different checksum if the input data is rearranged

```c
uint16_t
sum32( uint16_t* w, int n ) {
    uint16_t sum = 0;
    for( int i = 0; i < n; i++ )
        sum = sum + w[i];
    return sum;
}

uint32_t
fletcher32( uint16_t* w, int n ) {
    uint32_t sum1 = 0;
    uint32_t sum2 = 0;
    for( int i = 0; i < n; i++ ){
        sum1 = ( sum1 + w[i] ) % 65536;
        sum2 = ( sum2 + sum1 ) % 65536;
    }
    return ( sum2 << 16 ) | sum1;
}
```
★ Task 2.1: Implementing a Checksum FL Model ★

```python
% cd $HOME/pymtl-tut/examples/build
% geany ../ex02_cksum/ChecksumFL.py

1 def checksum( words):
2
3   sum1 = b32(0)
4   sum2 = b32(0)
5   for word in words:
6       sum1 = ( sum1 + word ) & 0xffffffff
7       sum2 = ( sum2 + sum1 ) & 0xffffffff
8
9   return ( sum2 << 16 ) | sum1

% pytest ../ex02_cksum/test/ChecksumFL_test.py -sv
% geany ../ex02_cksum/test/ChecksumFL_test.py

1 def test_order( s):
2   words0 = [ b16(x) for x in [ 1, 2, 3, 4, 5, 6, 7, 8 ] ]
3   words1 = [ b16(x) for x in [ 1, 2, 3, 4, 8, 7, 6, 5 ] ]
4   assert s.cksum_func( words0 ) != s.cksum_func( words1 )

% pytest ../ex02_cksum/test/ChecksumFL_test.py -sv -k order
```
class ChecksumCL( Component ):

def construct( s ):
    s.recv = NonBlockingCalleeIfc( Bits128 )
    s.send = NonBlockingCallerIfc( Bits32 )

    s.in_q = PipeQueueCL( num_entries=1 )
    s.connect( s.recv, s.in_q.enq )

    @s.update
def up():
        if s.in_q.deq.rdy() and s.send.rdy():
            bits = s.in_q.deq()
            words = b128_to_words( bits )
            result = checksum( words )
            s.send( result )
class ChecksumCL( Component ):

def construct( s, nstages=1 ):
    s.recv = NonBlockingCalleeIfc( Bits128 )
    s.send = NonBlockingCallerIfc( Bits32 )
    s.in_q = PipeQueueCL( num_entries=2 )
    s.connect( s.recv, s.in_q.enq )
    s.pipe = DelayPipeDeqCL( delay=nstages-1 )

@s.update
def upA():
    if s.pipe.enq.rdy() and s.in_q.deq.rdy():
        bits = s.in_q.deq()
        words = b128_to_words( bits )
        result = checksum( words )
        s.pipe.enq( result )

@s.update
def upB():
    if s.send.rdy() and s.pipe.deq.rdy():
        s.send( s.pipe.deq() )
Reusing Tests Across Multiple Modeling Levels

```python
1 class ChecksumFL_Tests( object ):
2     def cksum_func( s, words ):
3         return checksum( words )
4     
5 def test_simple( s ):
6     words = [ b16(x) for x in [ 1, 2, 3, 4, 5, 6, 7, 8 ] ]
7     assert s.cksum_func( words ) == b32( 0x00780024 )
8     
9 def checksum_cl( words ):
10    dut = WrappedChecksumCL()
11    dut.apply( SimulationPass )
12    dut.recv( words_to_b128( words ) )
13    dut.tick()
14    return dut.give()

15 from .ChecksumFL_test import ChecksumFL_Tests as BaseTests

16 class ChecksumCL_Tests( BaseTests ):
17     def cksum_func( s, words ):
18         return checksum_cl( words )
```
% cd $HOME/pymtl-tut/examples/build
% geany ../ex02_cksum/ChecksumCL.py

class ChecksumCL( Component ):
    def construct( self ):
        self.recv = NonBlockingCalleeIfc( Bits128 )
        self.send = NonBlockingCallerIfc( Bits32 )

        self.in_q = PipeQueueCL( num_entries=2 )
        self.connect( self.recv, self.in_q.enq )

    @self.update
    def up():
        if self.in_q.deq.rdy() and self.send.rdy():
            bits = self.in_q.deq()
            words = b128_to_words( bits )
            result = checksum( words )
            self.send( result )

% pytest ../ex02_cksum/test/ChecksumCL_test.py -v
% pytest ../ex02_cksum/test/ChecksumCL_test.py -sv -k pipeline
Checksum CL Line Traces

<table>
<thead>
<tr>
<th>source</th>
<th>recv</th>
<th>send</th>
<th>sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: .</td>
<td>&gt; .</td>
<td>()</td>
<td>&gt; .</td>
</tr>
<tr>
<td>1: 00080007...00020001</td>
<td>&gt; 00080007...00020001()</td>
<td>&gt; .</td>
<td></td>
</tr>
<tr>
<td>2: 00010002...00070008</td>
<td>&gt; 00010002...00070008(00780024)</td>
<td>&gt; 00780024</td>
<td></td>
</tr>
<tr>
<td>3: 00080007...00020001</td>
<td>&gt; 00080007...00020001(00cc0024)</td>
<td>&gt; 00cc0024</td>
<td></td>
</tr>
<tr>
<td>4: 00010002...00070008</td>
<td>&gt; 00010002...00070008(00780024)</td>
<td>&gt; 00780024</td>
<td></td>
</tr>
<tr>
<td>5: &gt;</td>
<td>()00cc0024</td>
<td>&gt; 00cc0024</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>source</th>
<th>recv</th>
<th>send</th>
<th>sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: .</td>
<td>&gt; .</td>
<td>([ ]))</td>
<td>&gt;</td>
</tr>
<tr>
<td>1: 00080007...00020001</td>
<td>&gt; 00080007...00020001([ ]))</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>2: 00010002...00070008</td>
<td>&gt; 00010002...00070008([* ]))</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>3: 00080007...00020001</td>
<td>&gt; 00080007...00020001([** ]))</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>4: 00010002...00070008</td>
<td>&gt; 00010002...00070008([*** ]))</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>5: &gt;</td>
<td>([****])</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>11: &gt;</td>
<td>([ ***])00780024</td>
<td>&gt; 00780024</td>
<td></td>
</tr>
<tr>
<td>12: &gt;</td>
<td>([ **])00cc0024</td>
<td>&gt; 00cc0024</td>
<td></td>
</tr>
<tr>
<td>13: &gt;</td>
<td>([ *])00780024</td>
<td>&gt; 00780024</td>
<td></td>
</tr>
<tr>
<td>14: &gt;</td>
<td>([ ])00cc0024</td>
<td>&gt; 00cc0024</td>
<td></td>
</tr>
</tbody>
</table>
Checksum Step RTL Implementation

```python
1  class StepUnit( Component ):
2    def construct( s ):
3
4    s.word_in = InPort( Bits16 )
5    s.sum1_in = InPort( Bits32 )
6    s.sum2_in = InPort( Bits32 )
7    s.sum1_out = OutPort( Bits32 )
8    s.sum2_out = OutPort( Bits32 )

9  @s.update
10  def up_step():
11    temp1 = b32(s.word_in) + s.sum1_in
12    s.sum1_out = temp1 & b32(0xffff)

14    temp2 = s.sum1_out + s.sum2_in
16    s.sum2_out = temp2 & b32(0xffff)
```
class ChecksumRTL( Component ):

def construct( s ):
    s.recv = RecvIfcRTL( Bits128 )
    s.send = SendIfcRTL( Bits32 )
Checksum RTL Implementation

```python
class ChecksumRTL( Component ):
    def construct( s ):
        s.recv = RecvIfcRTL( Bits128 )
        s.send = SendIfcRTL( Bits32 )

        # Register input and decompose input message into 8 words
        s.in_q = PipeQueueRTL( Bits128, num_entries=1 )
        s.connect( s.recv, s.in_q.enq )

        s.words = [ Wire( Bits16 ) for _ in range( 8 ) ]
        for i in range( 8 ):
            s.connect( s.words[i], s.in_q.deq.msg[i*16:(i+1)*16] )

        # Connect step units
        s.steps = [ StepUnit() for _ in range( 8 ) ]
        for i in range( 8 ):
            s.connect( s.steps[i].word_in, s.words[i] )
            if i == 0:
                s.connect( s.steps[i].sum1_in, b32(0) )
                s.connect( s.steps[i].sum2_in, b32(0) )
            else:
                s.connect( s.steps[i].sum1_in, s.steps[i-1].sum1_out )
                s.connect( s.steps[i].sum2_in, s.steps[i-1].sum2_out )
```

% cd $HOME/pymtl-tut/examples/build
% geany ../ex02_cksum/ChecksumRTL.py

1 class StepUnit( Component ):
2   def construct( s ):
3       s.word_in = InPort ( Bits16 )
4       s.sum1_in = InPort ( Bits32 )
5       s.sum2_in = InPort ( Bits32 )
6       s.sum1_out = OutPort( Bits32 )
7       s.sum2_out = OutPort( Bits32 )

8   @s.update
9       def up_step():
10          temp1 = b32(s.word_in) + s.sum1_in
11          s.sum1_out = temp1 & b32(0xffffffff)
12          temp2 = s.sum1_out + s.sum2_in
13          s.sum2_out = temp2 & b32(0xffffffff)

% pytest ../ex02_cksum/test/ChecksumRTL_test.py -k step_unit
% pytest ../ex02_cksum/test/ChecksumRTL_test.py -sv -k test_srcsink_pipeline --dump-vcd
% gtkwave ChecksumRTL.test_srcsink_pipeline.vcd &
- RTLIR simplifies RTL analysis passes and translation
- Translation framework simplifies implementing new translation passes
Translation+import enables easily testing translated SystemVerilog
Also acts like a JIT compiler for improved RTL simulation speed
Can also import external SystemVerilog IP for co-simulation
Translating to **Readable** SystemVerilog

```python
class StepUnit( Component ):
    def construct( s ):
        s.word_in = InPort( Bits16 )
        s.sum1_in = InPort( Bits32 )
        s.sum2_in = InPort( Bits32 )
        s.sum1_out = OutPort( Bits32 )
        s.sum2_out = OutPort( Bits32 )

    @s.update
    def up_step():
        temp1 = b32(s.word_in) + s.sum1_in
        s.sum1_out = temp1 & b32(0xffff)

        temp2 = s.sum1_out + s.sum2_in
        s.sum2_out = temp2 & b32(0xffff)
```

- **Readable signal names**
- **Generates useful comments**
- **Simple type inference for temporary variables**
**Task 2.4: Translate RTL to SystemVerilog and Test**

- Run translate script and look at the generated SystemVerilog

  ```
  % cd $HOME/pymtl-tut/examples/build
  % ../ex02_cksum/cksum-translate
  % geany ChecksumRTL.sv
  ```

- Add code to apply translation, import, simulation pass for testing

  ```
  % geany ../ex02_cksum/test/ChecksumVRTL_test.py
  
  1 th.apply( TranslationPass() )
  2 th = ImportPass()( th )
  3 th.apply( SimulationPass )
  ```

- Then run all tests using translated SystemVerilog

  ```
  % pytest ../ex02_cksum/test/ChecksumVRTL_test.py -v
  ```
**Recent History of Prototypes at Cornell University**

**DCS (2014)**
- TSMC 65nm
- 1mm x 2.2mm

**BRGTC1 (2016)**
- IBM 130nm
- 2mm x 2mm

**Celerity (2017)**
- TSMC 16nm FinFET
- 5mm x 5mm

**BRGTC2 (2018)**
- TSMC 28nm
- 1mm x 1.25mm

**PCOSYNC (2018)**
- IBM 180nm
- 2mm x 1mm

---

**Why Prototype?**

**Research Ideas**

- Smart Sharing Architectures
- Interconnection Networks for Manycores
- Python-Based Hardware Modeling
- High-Level Synthesis
- Synthesizable Analog IP
- Scalable Baseband Synchronization
- Integrated Voltage Regulation
Recent History of Prototypes at Cornell University

Why Prototype?

Chip-Based Startups

- Graphcore
- Nervana
- Cerebras
- Wave Computing
- Horizon Robotics
- Cambricon
- DeePhi
- Esperanto
- SambaNova
- Eyeriss
- Tenstorrent
- Mythic
- ThinkForce
- Groq
- Lightmatter
https://github.com/cornell-brg/alloy-asic

--- The ASIC Design Kit (ADK) ---
**Problem**: Bad portability across techs/libs  
**Solution**: Standard interface to ASIC design kit

--- Modularizing the ASIC Flow ---
**Problem**: Rigid, static ASIC flows  
**Solution**: Modularize steps, assemble flows

--- Design-Space Exploration ---
**Problem**: Rewriting scripts for each design  
**Solution**: Fast parameterization

--- Helping Everyone Build Chips ---
**Problem**: High barrier of entry for building chips  
**Solution**: Open-source our flow as a reference
The ASIC Design Kit (ADK)

**Problem:** Bad portability across technologies and libraries

**BRGTC1**
- stdcells – ARM / IBM
- routing tech kit – ARM
- iocells – IBM
- pdk – IBM

**BRGTC2**
- stdcells – ARM
  - 30-nm channel length, 35-nm channel length, LVT, SVT, HVT
  - corners
- routing tech kit – ARM
- iocells and bondpads – TSMC
- pdk – TSMC
- sealring – MOSIS
The ASIC Design Kit (ADK)

**Solution**: Standardize the interface to physical backend files

**Standard cell library**
- stdcells.db
- stdcells.lef
- stdcells.lib
- stdcells.mwlib
- stdcells.v

**Routing technology kit**
- rtk-max.tluplus
- rtk-min.tluplus
- rtk-typical.captable
- rtk-tech.lef
- rtk-tech.tf
- rtk-tluplus.map
- rtk-stream-out.map

**IO cell library**
- iocells.db
- iocells.lef
- iocells.lib
- iocells.mwlib
- iocells.v

**ADK-specific variables**
- adk.tcl
The ASIC Design Kit (ADK)

**Problem:** Bad portability across techs/libs
**Solution:** Standard interface to ASIC design kit

---

Modularizing the ASIC Flow

**Problem:** Rigid, static ASIC flows
**Solution:** Modularize steps, assemble flows

---

Design-Space Exploration

**Problem:** Rewriting scripts for each design
**Solution:** Fast parameterization

---

Helping Everyone Build Chips

**Problem:** High barrier of entry for building chips
**Solution:** Open-source our flow as a reference

---

Overview Basics Multi-Level Modeling **Alloy ASIC** Processor Modeling Multi-Level Composition
Modularizing the ASIC Flow

**Problem:** Rigid, static ASIC flows

**Typical ASIC Flows**

- Flows are automated for exact sequences of steps
  - Want to add/remove a step? Modify the build system. Copies..
  - Once the flow is set up, you don’t want to touch it anymore

- Adding new steps between existing steps is troublesome
  - Steps downstream magically reach upstream — hardcoding
  - In general, the overhead to add new steps is high

- Difficult to support different configurations of the flow
  - E.g., chip flow vs. block flow
  - How to add new steps before or after
  - Each new chip ends up with a dedicated non-reusable flow
Modularizing the ASIC Flow

Solution: Modularize the ASIC flow!

A Modular ASIC Flow

- Use the build system to mix, match, and assemble steps together
  - Create modular steps that know how to run/clean themselves
  - The build system can also check prerequisites and outputs before and after execution to make sure each step can run
- Assemble the ASIC flow as a graph
  - Can target architecture papers by assembling a minimal graph
  - Can target VLSI papers by assembling a medium graph with more steps (e.g., need dedicated floorplan)
  - Can target a chip by assembling a full-featured tapeout graph
https://github.com/cornell-brg/alloy-asic

___ The ASIC Design Kit (ADK) ___

**Problem:** Bad portability across techs/libs  
**Solution:** Standard interface to ASIC design kit

---

PDK  
IP  
Vendors


___ Modularizing the ASIC Flow ___

**Problem:** Rigid, static ASIC flows  
**Solution:** Modularize steps, assemble flows

---


___ Design-Space Exploration ___

**Problem:** Rewriting scripts for each design  
**Solution:** Fast parameterization

---

Chip A  
Chip B  
Chip C

pnr.tcl  
pnr.tcl  
pnr.tcl


___ Helping Everyone Build Chips ___

**Problem:** High barrier of entry for building chips  
**Solution:** Open-source our flow as a reference

---

Chip A  
Chip B  
Chip C

Batten Research Group  
PyMTL Tutorial @ ISCA’19
The Open-Source Flow in this Tutorial

- **RTL**
  - .v
  - Yosys Synthesis
    - .lib
    - Reports
      - .v
      - Graywolf Placement
        - .par, .lef
        - Reports
          - .def
          - Qrouter Routing
            - .def
            - Reports
              - Klayout
  - Standard Cell Library / Routing Tech Library
    - .lib
    - .par, .lef
    - Klayout
    - .def
    - Klayout
★ Task 2.5: Use Alloy ASIC and Open EDA Toolflow ★

% cp ChecksumRTL.sv $HOME/alloy-asic/designs/ChecksumRTL/rtl
% cd $HOME/alloy-asic
% mkdir build-cksum
% cd build-cksum
% ../configure.py --design ChecksumRTL
% make info
Design name    -- ChecksumRTL
Clock period   -- 2.0
ADK            -- freepdk-45nm

% make open-yosys-synthesis
% make open-graywolf-place
% make open-qrouter-route
% klayout

▶ Ignore the Klayout errors
▶ See next slide for how to use Klayout to view place-and-routed design
Using Klayout to View Place-and-Routed Design

- Choose *File → Import → DEF/LEF* from menu
- Click on ... button on the right to open file browser for DEF
- Select `/home/pymtl/alloy-asic/build-cksum`
  `/5-open-qrouter-route/design.routed.def`
- Click *Open*
- Click on + button on the right to open file browser for LEF
- Select `/home/pymtl/alloy-asic/build-cksum`
  `/5-open-qrouter-route/inputs/adk/stdcells.lef`
- Click *Open*
- Click *OK*
- Choose *Display → Full Hierarchy* from menu
- Choose *File → Load Layer Properties* from menu
- Select `/home/pymtl/alloy-asic/utils/cksum.lyp`
- Click *Open*
Post-Place-and-Route Layout for ChecksumRTL

- Pure-PyMTL RTL design
- FreePDK45 Process
- Nangate standard cell library
- Yosys, graywolf, Qrouter, klayout
- 1.1 V, 625 MHz, 3400 µm²
The Need for Productive Multi-Level Verification

How can we productively verify complex generated designs across abstraction levels?

```python
class RingNet( Component ):
    def construct( s, sz=4, ... many more parameters ... ):
        s.in_ = [ RecvIfc( MsgType ) for _ in range(sz) ]
        s.out = [ SendIfc( MsgType ) for _ in range(sz) ]
        s.routers = [ RingRouter() for _ in range(sz) ]
        s.channels_E = [ Queue(MsgType,2) for _ in range(sz) ]
        s.channels_W = [ Queue(MsgType,2) for _ in range(sz) ]
        for i in range( nrouters ) :
            i_n = (i+1) % nrouters
            s.connect_pairs( s.routers[i].router_id, i,
                            s.routers[i].out[DIR_C], s.out[i],
                            s.routers[i].in_[DIR_C], s.in_[i] )
            s.connect_pairs( s.routers[i].out[DIR_E], s.channels_E[i].enq,
                            s.channels_E[i].deq, s.routers[i_n].in_[DIR_W],
                            s.routers[i].in_[DIR_E], s.channels_W[i].deq,
                            s.channels_W[i].enq, s.routers[i_n].out[DIR_W] )
```
## Constraint-Based vs Property-Based Random Testing

### Constraint-Based Testing
- Statically generate test vectors that satisfy constraints
- Strong support in SystemVerilog with associated libraries (UVM)
- De facto standard in commercial HW design
- Rarely used in open-source HW design
- No open-source tool support (e.g., no open-source Verilog simulator supports UVM)

### Property-Based Testing
- Dynamically generate test vectors that satisfy constraints
- Auto-shrink failures to generate minimal failing test cases
- First popularized with the Haskell QuickCheck library
- Emerging methodology in open-source SW design
- Significant open-source tool support across many different languages
PBRT in Python using Hypothesis

```python
from hypothesis import given
from hypothesis.strategies import
given, text
@given(text())
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```

- Customizable random value generation strategies
- Stateless and stateful testing frameworks
- Auto-shrinking
- Excellent documentation

https://hypothesis.readthedocs.io
Using Stateless Hypothesis in PyMTL

# FL golden reference model
```python
1  # FL golden reference model
2  def incr_8bit(x):
3      return b8(x) + b8(1)
```

# CL design under test
```python
1  # CL design under test
2  class Incr8bitCL(Component):
3      def construct(s):
4          s.in_ = InPort(Bits8)
5          s.out = OutPort(Bits8)
6  @s.update
7      def up():
8          tmp = s.in_ + b8(1)
9          s.out = tmp & b8(0xef)
10     def incr_8bit_cl(x):
11        incr = Incr8bitCL()
12        incr.apply(SimpleSim)
13        incr.in_ = x
14        incr.tick()
15        return incr.out
```

def test_simple():
```python
1  def test_simple():
2      assert incr_8bit_cl(b8(2)) == b8(3)
3  @given(x=pst.bits(8))
4      def test_hypothesis(x):
5          print("x =", x)
6          assert incr_8bit_cl(x) == incr_8bit(x)
7  % pytest example_test.py
```

===== test session starts ======
... incr_8bit_test.py::test_hypothesis
x = 00
x = 88
... 19 other tries ...
x = 0e
Falsifying example:
```python
  test_hypothesis(x=Bits8(0x0f))
```
★ Task 2.6: Using Hypothesis to Test Bits To Words ★

% cd $HOME/pymtl-tut/examples/build
% geany ../ex02_cksum/test/utils_test.py

1   @hypothesis.given( words=st.lists( pst.bits(16), min_size=8, max_size=8 ) )
2   def test_hypothesis( words ):
3       print(words)
4       assert b128_to_words( words_to_b128( words ) ) == words

% pytest ../ex02_cksum/test/utils_test.py -sv -k hypothesis

[b16(0x0000), b16(0x0000), b16(0x0000), ... b16(0x0000), b16(0x0000)]
[b16(0x7167), b16(0x3445), b16(0xe989), ... b16(0x3381), b16(0x7343)]
[b16(0x2527), b16(0x4481), b16(0xb065), ... b16(0x2c00), b16(0xf9dc)]
[b16(0xe737), b16(0xe327), b16(0xe66c1), ... b16(0x1c11), b16(0xcee1)]
[b16(0x7a51), b16(0x3c92), b16(0xe5c9), ... b16(0x7eb2), b16(0x2f9d)]
[b16(0x4c4e), b16(0x4479), b16(0xdbab), ... b16(0x562e), b16(0x6855)]
[b16(0xaf2a), b16(0xeb02), b16(0x6b01), ... b16(0x7045), b16(0x7619)]
[b16(0xe36a), b16(0xf005), b16(0x9e20), ... b16(0x7009), b16(0xb9cc)]
[b16(0x3d65), b16(0xd4a8), b16(0xb846), ... b16(0xc808), b16(0x27fd)]
[b16(0x81f5), b16(0xca95), b16(0xd084), ... b16(0xda94), b16(0x7980)]
**Task 2.7: Using Hypothesis to Test Checksum CL**

```python
% cd $HOME/pymtl-tut/examples/build
% geany ../ex02_cksum/test/ChecksumCL_test.py

1   from .ChecksumFL_test import ChecksumFL_Tests as BaseTests
2
3   class ChecksumCL_Tests( BaseTests ):
4       def cksum_func( s, words ):
5           return checksum_cl( words )
6
7       @hypothesis.settings( deadline=None )
8       @hypothesis.given( words=st.lists( pm_st.bits(16), min_size=8, max_size=8 ) )
9       def test_hypothesis( s, words ):
10          print( [ int(x) for x in words ] )
11          assert s.cksum_func( words ) == checksum( words )

% pytest ../ex02_cksum/test/ChecksumCL_test.py \
   -k test_hypothesis --tb=short
```

- Inject a bug into ChecksumCL by forcing `word[5]` to be zero
- Rerun the Hypothesis test and observe shrinking
Example of Hypothesis Shrinking

words = [0, 0, 0, 0, 0, 0, 0, 0]
words = [31783, 16263, 46089, 31500, 14977, 36071, 57387, 44378]
words = [3925, 42511, 26674, 57232, 26245, 61711, 20014, 43124]
...
words = [3925, 42511, 26674, 144, 26245, 61711, 20014, 43124]
words = [3925, 15, 26674, 0, 26245, 61711, 20014, 116]
words = [3840, 15, 50, 0, 26245, 61711, 20014, 0]
words = [3840, 15, 0, 0, 0, 61711, 0, 0]
words = [3840, 15, 0, 0, 0, 61711, 0, 0]
words = [3840, 15, 0, 0, 0, 61711, 0, 0]
words = [3840, 15, 0, 0, 0, 61711, 0, 0]
words = [3840, 15, 0, 0, 0, 61711, 0, 0]
words = [3840, 15, 0, 0, 0, 61711, 0, 0]
words = [3840, 15, 0, 0, 0, 61711, 0, 0]
words = [3840, 15, 0, 0, 0, 61711, 0, 0]
words = [3840, 15, 0, 0, 0, 61711, 0, 0]

-------------------------- Hypothesis ---------------------------
Falsifying example:
test_hypothesis(words= [
  Bits16(0x0000), Bits16(0x0000), Bits16(0x0000), Bits16(0x0000), Bits16(0x0000), Bits16(0x0000), Bits16(0x0000), Bits16(0x0000) ] )
Using Stateful Hypothesis in PyMTL

The key to leveraging state-of-the-art SW testing frameworks for HW testing is to make HW look like SW.
Using Stateful Hypothesis in PyMTL

Stateful PBRT in PyMTL will automatically compare a QueueFL reference component to a wrapped QueueRTL component by randomly calling methods with random data in a well-structured way.

Hypothesis + Python’s powerful reflection features means we can create a property-based random test in just four lines of code!

```python
def test_state_machine():
    run_test_state_machine(RTL2FLWrapper(QueueRTL(Bits16, 4)), QueueFL(Bits16, 4))
```

Example Random Test Sequence

```
enq(data=59488)
deq() -> 59488
enq(data=8637)
deq() -> 8637
enq(data=18754)
enq(data=37951)
deq() -> 18754
enq(data=23589)
enq(data=17602)
deq() -> 37951
enq(data=43506)
enq(data=48999)
deq() -> 23589
enq(data=62684)
```

```
enq(data=37367)
deq() -> 36011
enq(data=33990)
enq(data=227)
deq() -> 43506
enq(data=50664)
deq() -> 227
enq(data=45118)
enq(data=64506)
deq() -> 48999
```

Batten Research Group

PyMTL Tutorial @ ISCA’19
Auto-Generating Minimal Counter-Examples

```python
1 class QueueRTL( Component ):
2     def construct( s, DataType, maxsize ):
3         ...
4
4     @s.update
5     def set_enq_rdy():
6         s.enq.rdy = s.num < maxsize
7
7     @s.update
8     def set_deq_rdy():
9         s.deq.rdy = s.num > 0
10
10    @s.update_on_edge
11     def update_head():
12         if s.reset:
13             s.head = 0
14         elif s.deq.en:
15             s.head = index_type( s.head + 1 )
```

Trying example:
```
enq(data=9352)
enq(data=27368)
deq() -> 9352
enq(data=31419)
enq(data=46760)
enq(data=9228)
==== error ==== 
deq() -> 9228
```

 Injected Bug: Do not update head pointer

Falsifying example:
```
enq(data=0)
deq() -> 0
enq(data=1)
==== error ==== 
deq() -> 0
```

Hypothesis will automatically derive a minimal failing test case by “shrinking” the sequence of random method calls and input data