

PyMTL Tutorial Schedule

1:45 – 2:00pm Virtual Machine Installation and Setup

2:00 – 2:15pm *Presentation:* PyMTL Overview

2:15 – 2:55pm *Part 1:* PyMTL Basics

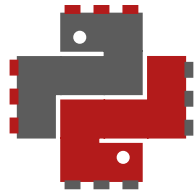
2:55 – 3:30pm *Part 2:* Multi-Level Modeling with PyMTL

3:30 – 4:00pm Coffee Break

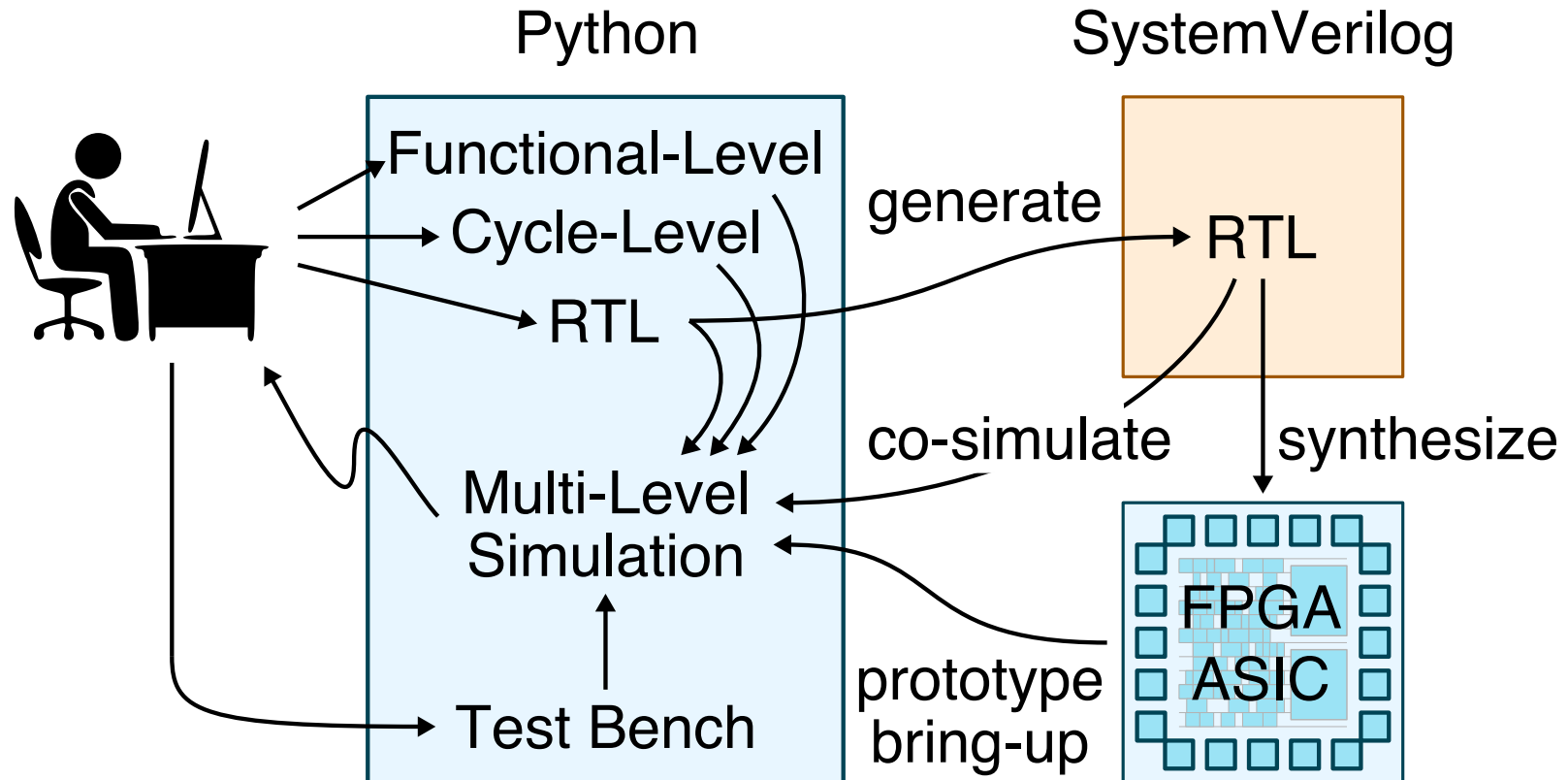
4:00 – 4:45pm *Part 3:* Processor Modeling with PyMTL

4:45 – 5:30pm *Part 4:* Multi-Level Composition in PyMTL

PyMTL



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



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

```
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 ★

```
% 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 ) & 0xffff
7         sum2 = ( sum2 + sum1 ) & 0xffff
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 )
```

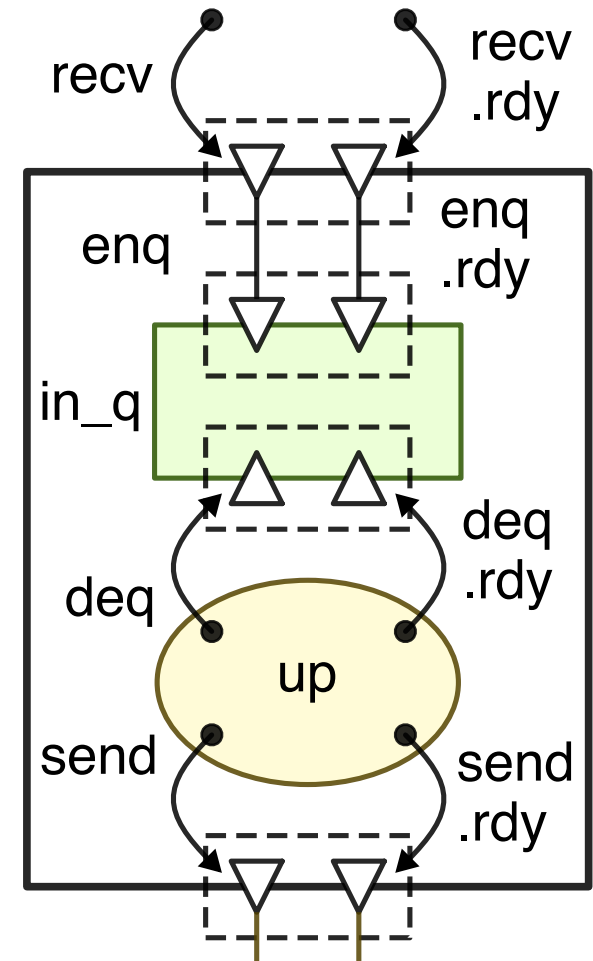
```
% pytest ../ex02_cksum/test/ChecksumFL_test.py -sv -k order
```

Checksum Single-Cycle CL Implementation

```

1 class ChecksumCL( Component ):
2     def construct( s ):
3         s.recv = NonBlockingCalleeIfc( Bits128 )
4         s.send = NonBlockingCallerIfc( Bits32 )
5
6         s.in_q = PipeQueueCL( num_entries=1 )
7         s.connect( s.recv, s.in_q.enq )
8
9         @s.update
10        def up():
11            if s.in_q.deq.rdy() and s.send.rdy():
12                bits = s.in_q.deq()
13                words = b128_to_words( bits )
14                result = checksum( words )
15                s.send( result )

```

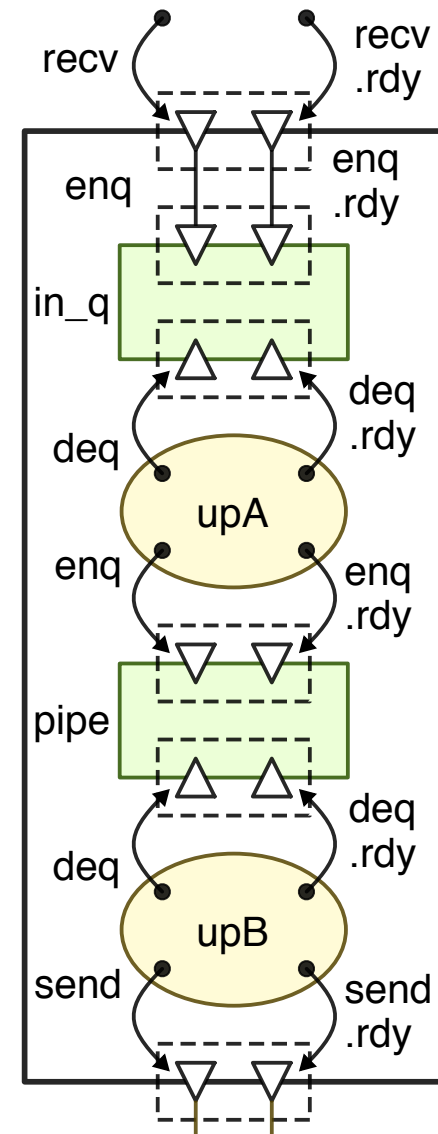


Checksum Pipelined CL Implementation

```

1 class ChecksumCL( Component ):
2     def construct( s, nstages=1 ):
3         s.recv = NonBlockingCalleeIfc( Bits128 )
4         s.send = NonBlockingCallerIfc( Bits32 )
5
6         s.in_q = PipeQueueCL( num_entries=2 )
7         s.connect( s.recv, s.in_q.enq )
8
9         s.pipe = DelayPipeDeqCL( delay=nstages-1 )
10
11     @s.update
12     def upA():
13         if s.pipe.enq.rdy() and s.in_q.deq.rdy():
14             bits = s.in_q.deq()
15             words = b128_to_words( bits )
16             result = checksum( words )
17             s.pipe.enq( result )
18
19     @s.update
20     def upB():
21         if s.send.rdy() and s.pipe.deq.rdy():
22             s.send( s.pipe.deq() )

```



Reusing Tests Across Multiple Modeling Levels

```
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
16 from .ChecksumFL_test import ChecksumFL_Tests as BaseTests
17
18 class ChecksumCL_Tests( BaseTests ):
19     def cksum_func( s, words ):
20         return checksum_cl( words )
```

★ Task 2.2: Implementing a Checksum CL Model ★

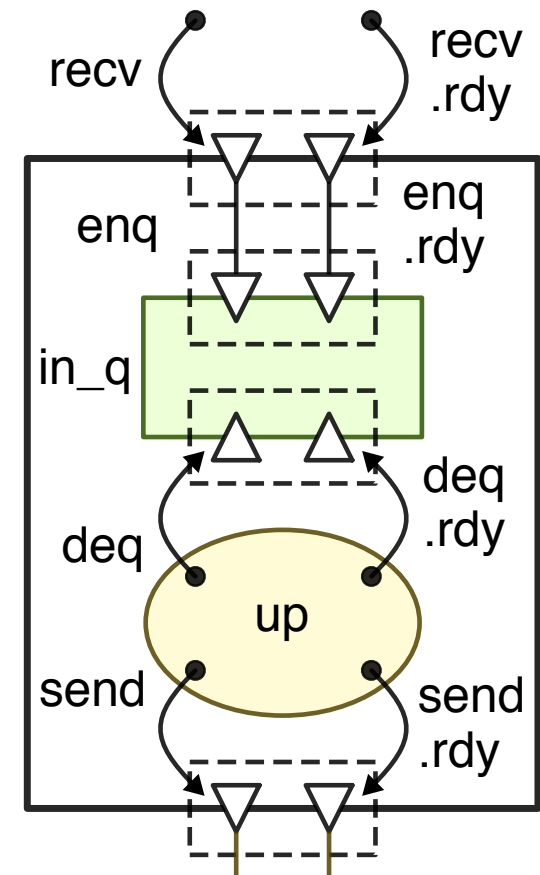
```

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

1 class ChecksumCL( Component ):
2     def construct( s ):
3         s.recv = NonBlockingCalleeIfc( Bits128 )
4         s.send = NonBlockingCallerIfc( Bits32 )
5
6         s.in_q = PipeQueueCL( num_entries=2 )
7         s.connect( s.recv, s.in_q.enq )
8
9         @s.update
10        def up():
11            if s.in_q.deq.rdy() and s.send.rdy():
12                bits = s.in_q.deq()
13                words = b128_to_words( bits )
14                result = checksum( words )
15                s.send( result )

% pytest ../ex02_cksum/test/ChecksumCL_test.py -v
% pytest ../ex02_cksum/test/ChecksumCL_test.py -sv -k pipeline

```



Checksum CL Line Traces

source	recv	send	sink
0: .	> .	().	> .
1: 00080007...00020001	> 00080007...00020001	().	> .
2: 00010002...00070008	> 00010002...00070008	()00780024	> 00780024
3: 00080007...00020001	> 00080007...00020001	()00cc0024	> 00cc0024
4: 00010002...00070008	> 00010002...00070008	()00780024	> 00780024
5:	>	()00cc0024	> 00cc0024

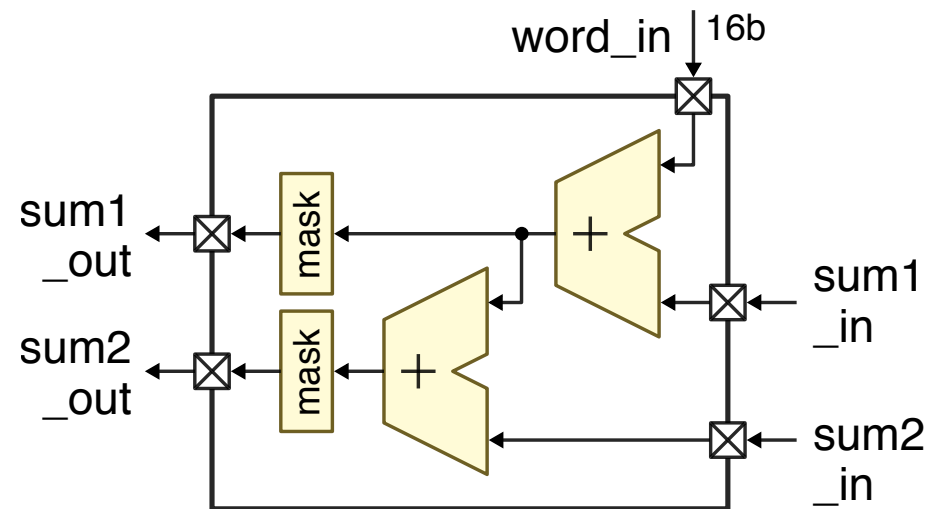
source	recv	send	sink
0: .	> .	([])	>
1: 00080007...00020001	> 00080007...00020001	([])	>
2: 00010002...00070008	> 00010002...00070008	([*])	>
3: 00080007...00020001	> 00080007...00020001	([**])	>
4: 00010002...00070008	> 00010002...00070008	([***])	>
5:	>	([****])	>
11:	>	([***])00780024	> 00780024
12:	>	([**])00cc0024	> 00cc0024
13:	>	([*])00780024	> 00780024
14:	>	([])00cc0024	> 00cc0024

Checksum Step RTL Implementation

```

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
10         @s.update
11         def up_step():
12             temp1 = b32(s.word_in) + s.sum1_in
13             s.sum1_out = temp1 & b32(0xffff)
14
15             temp2 = s.sum1_out + s.sum2_in
16             s.sum2_out = temp2 & b32(0xffff)

```

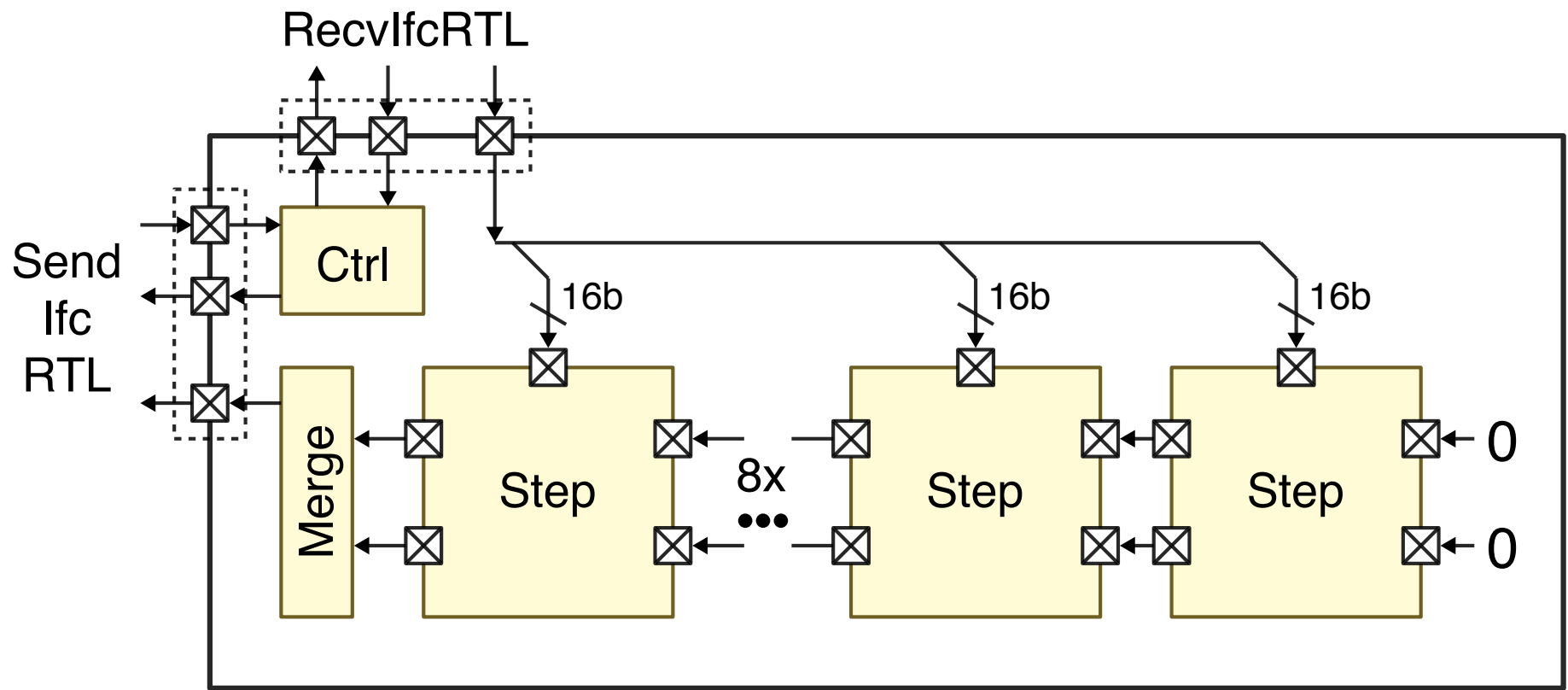


Checksum RTL Interface and Block Diagram

```

1 class ChecksumRTL( Component ):
2     def construct( s ):
3         s.recv = RecvIfcRTL( Bits128 )
4         s.send = SendIfcRTL( Bits32 )

```



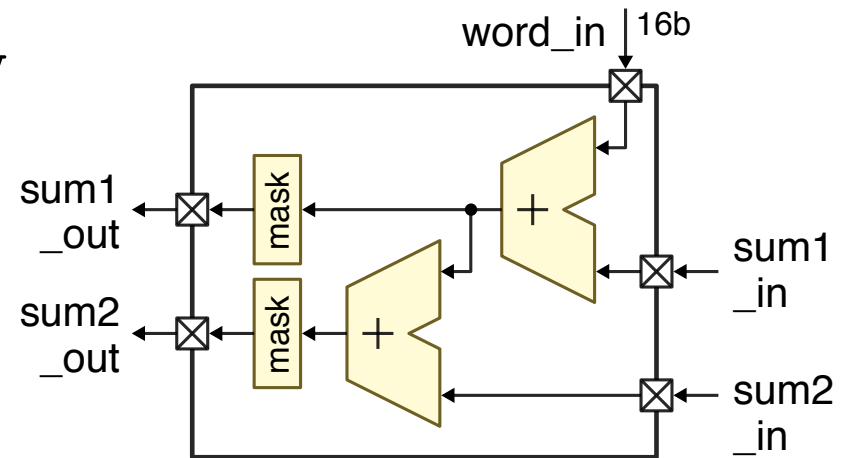
Checksum RTL Implementation

```
1 class ChecksumRTL( Component ):
2     def construct( s ):
3         s.recv = RecvIfcRTL( Bits128 )
4         s.send = SendIfcRTL( Bits32 )
5
6         # Register input and decompose input message into 8 words
7
8         s.in_q = PipeQueueRTL( Bits128, num_entries=1 )
9         s.connect( s.recv, s.in_q.enq )
10
11        s.words = [ Wire( Bits16 ) for _ in range( 8 ) ]
12        for i in range( 8 ):
13            s.connect( s.words[i], s.in_q.deq.msg[i*16:(i+1)*16] )
14
15        # Connect step units
16
17        s.steps = [ StepUnit() for _ in range( 8 ) ]
18        for i in range( 8 ):
19            s.connect( s.steps[i].word_in, s.words[i] )
20            if i == 0:
21                s.connect( s.steps[i].sum1_in, b32(0) )
22                s.connect( s.steps[i].sum2_in, b32(0) )
23            else:
24                s.connect( s.steps[i].sum1_in, s.steps[i-1].sum1_out )
25                s.connect( s.steps[i].sum2_in, s.steps[i-1].sum2_out )
26        ...
```

★ Task 2.3: Implement Checksum Step RTL Model ★

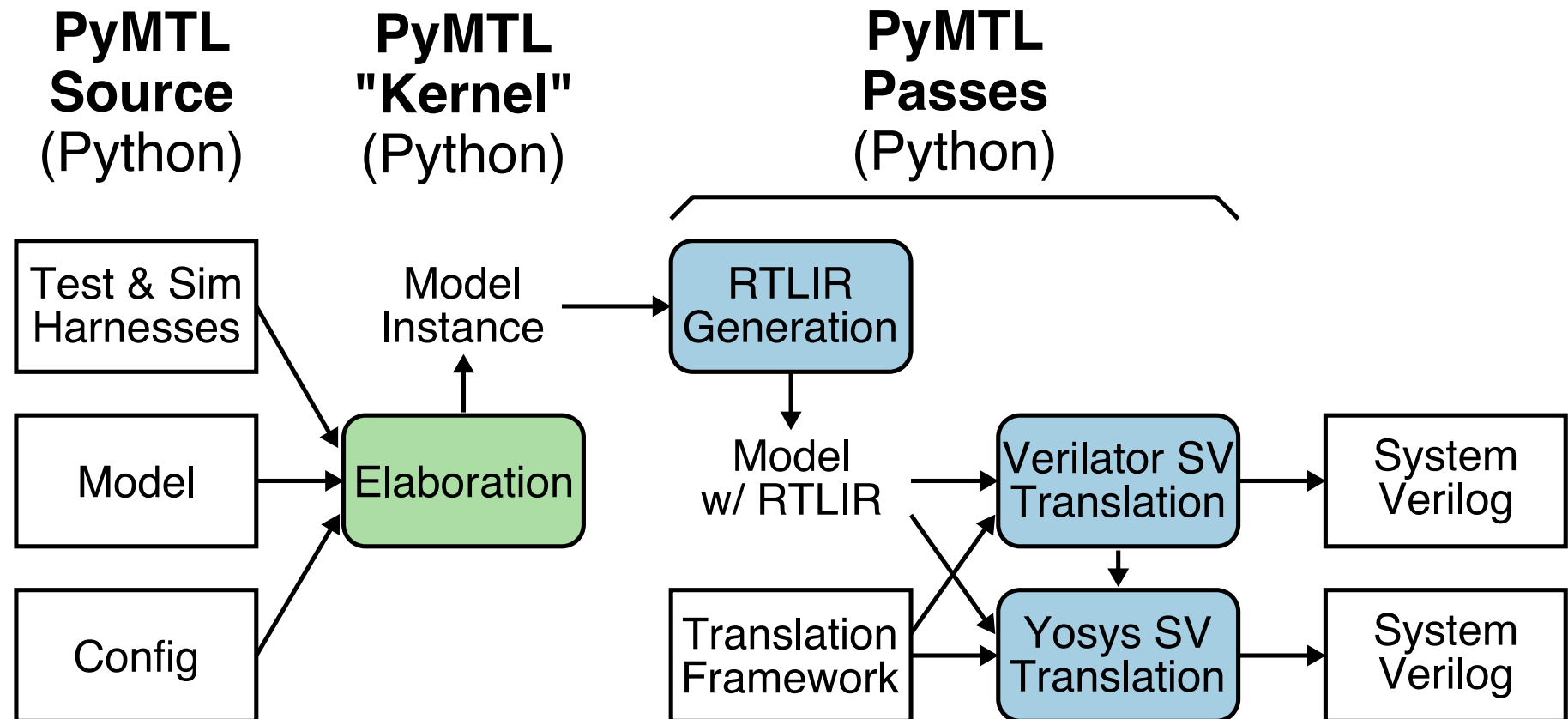
```
% cd $HOME/pymtl-tut/examples/build
% geany ../ex02_cksum/ChecksumRTL.py
```

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



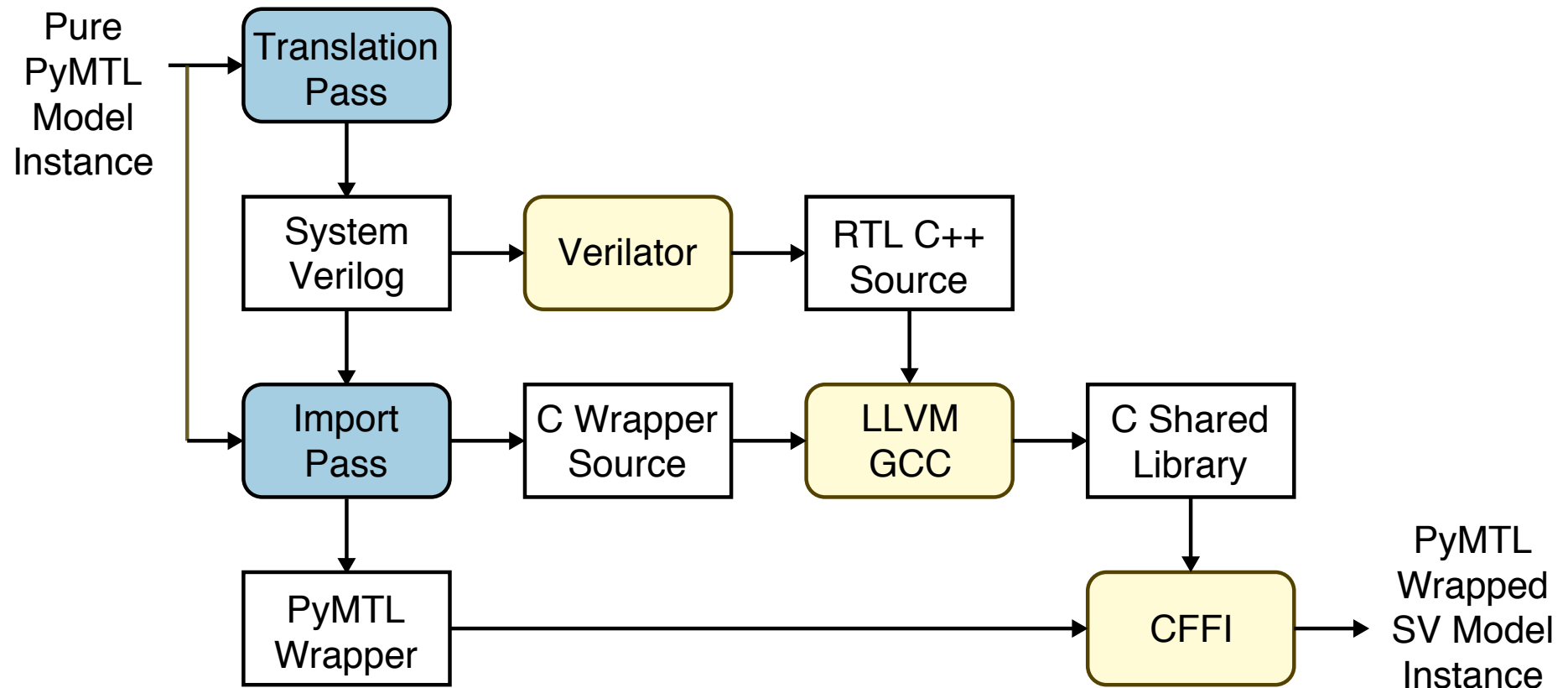
```
% pytest ../ex02_cksum/test/ChecksumRTL_test.py -k step_unit
% pytest ../ex02_cksum/test/ChecksumRTL_test.py -sv \
    -k test_srcsink_pipeline --dump-vcd
% gtkwave ChecksumRTL.test_srcsink_pipeline.vcd &
```

SystemVerilog RTLIR/Translation Framework



- ▶ RTLIR simplifies RTL analysis passes and translation
- ▶ Translation framework simplifies implementing new translation passes

SystemVerilog Translation and Import



- ▶ 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

```
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

```
module StepUnit
(
    input  logic [0:0]    clk,
    input  logic [0:0]    reset,
    input  logic [31:0]   sum1_in,
    output logic [31:0]   sum1_out,
    input  logic [31:0]   sum2_in,
    output logic [31:0]   sum2_out,
    input  logic [15:0]   word_in
);
    // Temporary wire definitions
    logic [31:0]    __up_step$temp1;
    logic [31:0]    __up_step$temp2;

    // PYMTL SOURCE:
    // ...

    always_comb begin : up_step
        __up_step$temp1 = {{16{1'b0}},word_in} + sum1_in;
        sum1_out = __up_step$temp1 & 32'd65535;
        __up_step$temp2 = sum1_out + sum2_in;
        sum2_out = __up_step$temp2 & 32'd65535;
    end

endmodule
```

★ 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
```

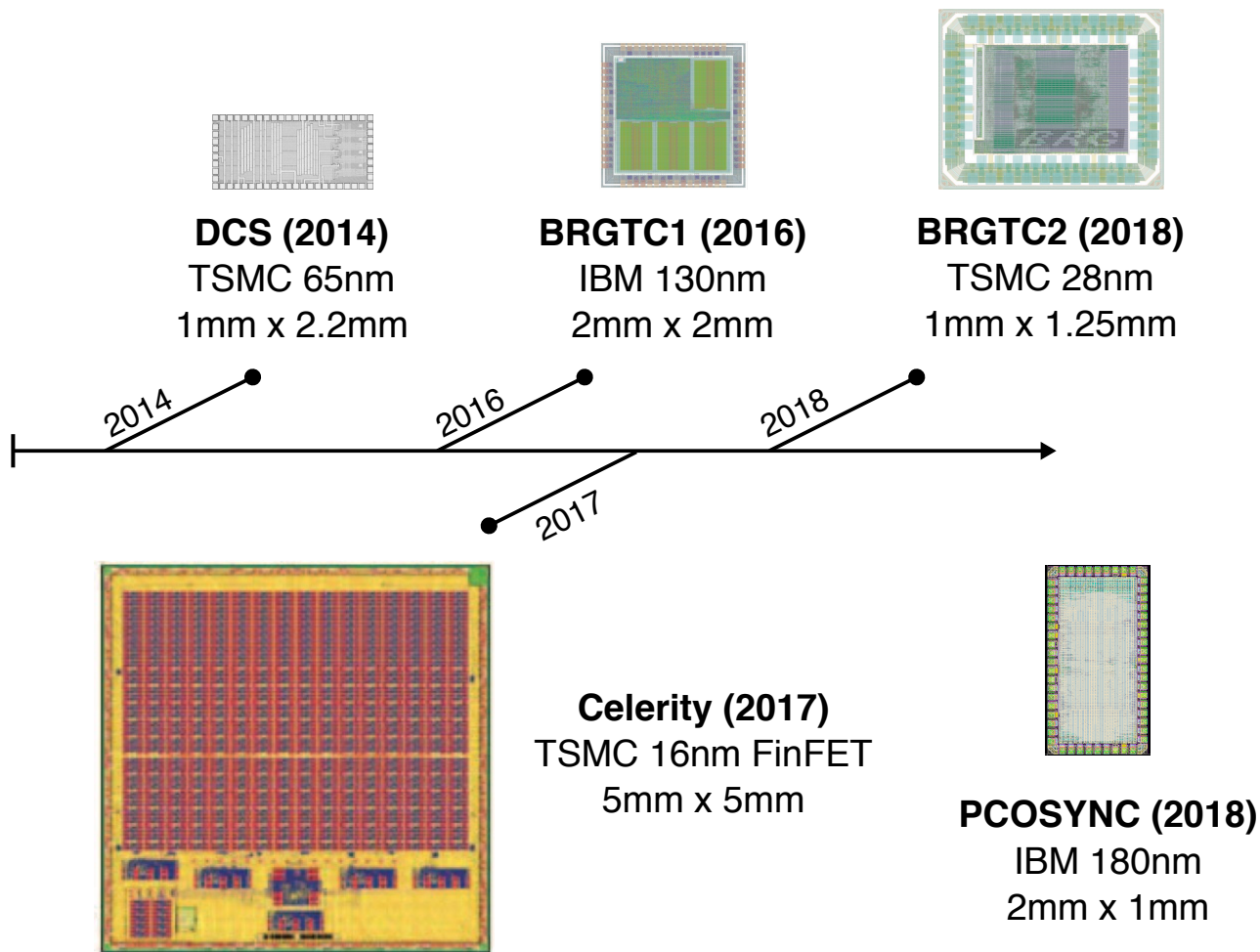
- ▶ Add code to apply translation, import, simulation pass for testing

```
% 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

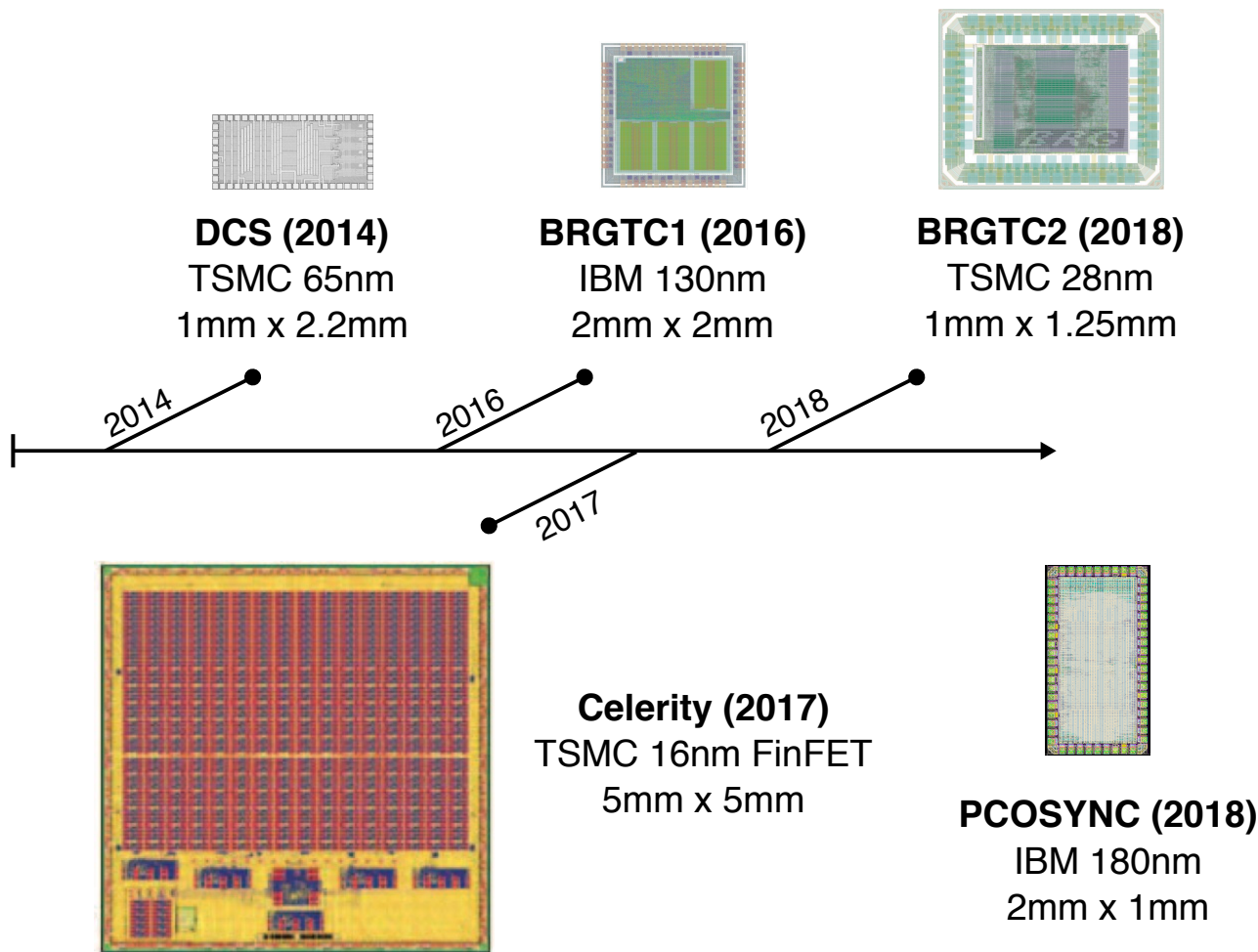


Why Prototype?

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

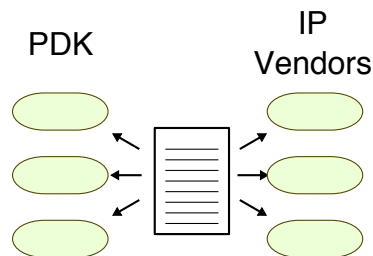
- ▶ Graphcore
- ▶ Nervana
- ▶ Cerebras
- ▶ Wave Computing
- ▶ Horizon Robotics
- ▶ Cambricon
- ▶ DeePhi
- ▶ Esperanto
- ▶ SambaNova
- ▶ Eyeriss
- ▶ Tenstorrent
- ▶ Mythic
- ▶ ThinkForce
- ▶ Groq
- ▶ Lightmatter

<https://github.com/cornell-brg/alloy-asic>

___ The ASIC Design Kit (ADK) ___

Problem: Bad portability across techs/libs

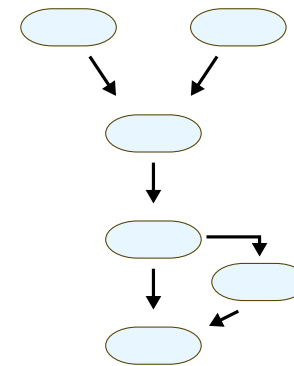
Solution: Standard interface to ASIC design kit



___ Modularizing the ASIC Flow ___

Problem: Rigid, static ASIC flows

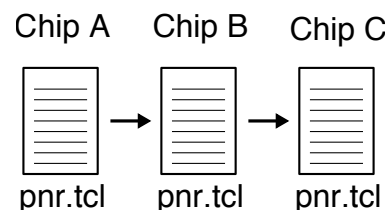
Solution: Modularize steps, assemble flows



___ Design-Space Exploration ___

Problem: Rewriting scripts for each design

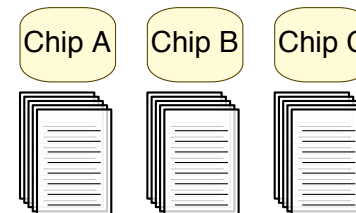
Solution: Fast parameterization



___ Helping Everyone Build Chips ___

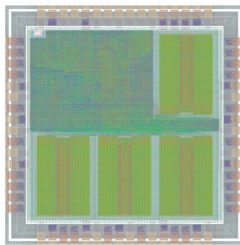
Problem: High barrier of entry for building chips

Solution: Open-source our flow as a reference



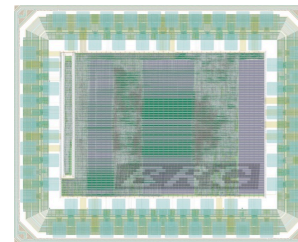
The ASIC Design Kit (ADK)

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

IO cell library

- ▶ iocells.db
- ▶ iocells.lef
- ▶ iocells.lib
- ▶ iocells.mwlib
- ▶ iocells.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

ADK-specific variables

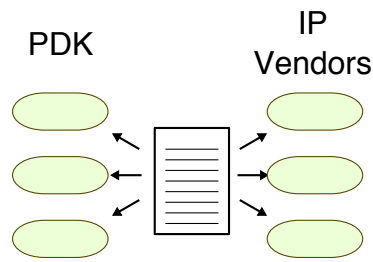
- ▶ adk.tcl

<https://github.com/cornell-brg/alloy-asic>

___ The ASIC Design Kit (ADK) ___

Problem: Bad portability across techs/libs

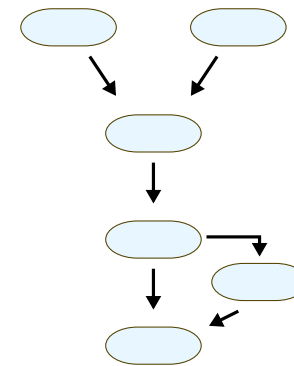
Solution: Standard interface to ASIC design kit



___ Modularizing the ASIC Flow ___

Problem: Rigid, static ASIC flows

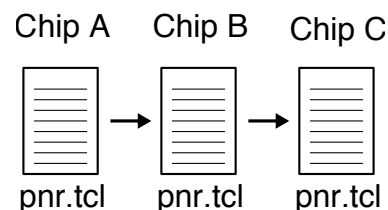
Solution: Modularize steps, assemble flows



___ Design-Space Exploration ___

Problem: Rewriting scripts for each design

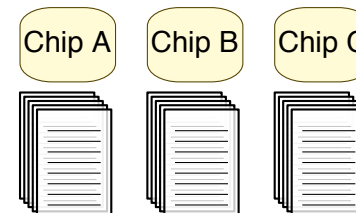
Solution: Fast parameterization



___ Helping Everyone Build Chips ___

Problem: High barrier of entry for building chips

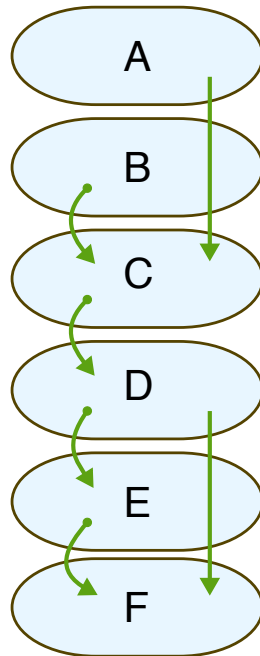
Solution: Open-source our flow as a reference



Modularizing the ASIC Flow

Problem: Rigid, static ASIC flows

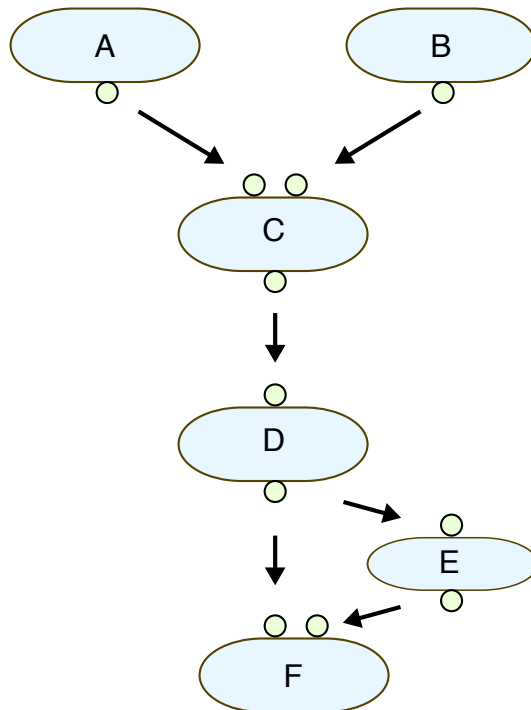
Typical ASIC Flows



- ▶ 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

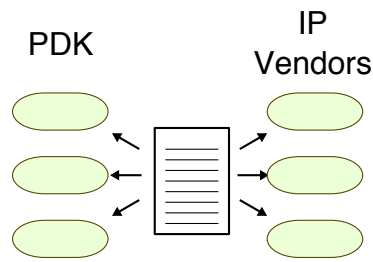
- ▶ Use the build system to mix, match, and assemble steps together
 - ▷ Create modular steps that know how to run/clean themselves
 - ▷ The build system can also check prerequisites and outputs before and after execution to make sure each step can run
- ▶ Assemble the ASIC flow as a graph
 - ▷ Can target *architecture* papers by assembling a minimal graph
 - ▷ Can target *VLSI* papers by assembling a medium graph w/ 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

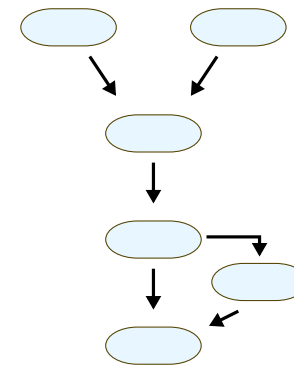
Solution: Standard interface to ASIC design kit



___ Modularizing the ASIC Flow ___

Problem: Rigid, static ASIC flows

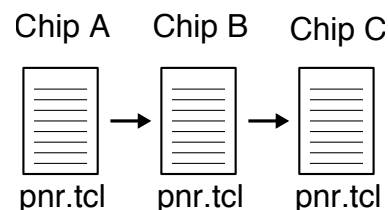
Solution: Modularize steps, assemble flows



___ Design-Space Exploration ___

Problem: Rewriting scripts for each design

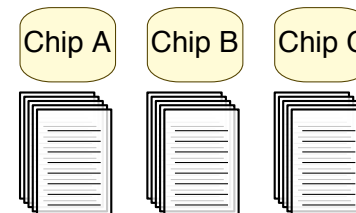
Solution: Fast parameterization



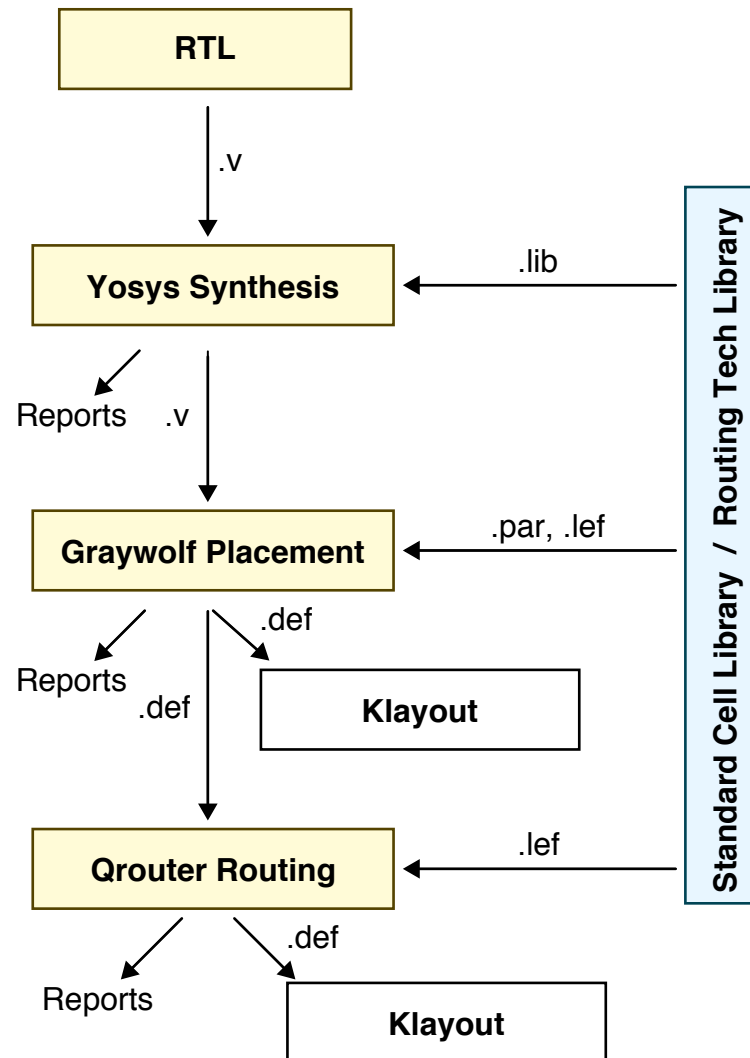
___ Helping Everyone Build Chips ___

Problem: High barrier of entry for building chips

Solution: Open-source our flow as a reference



The Open-Source Flow in this Tutorial



★ 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

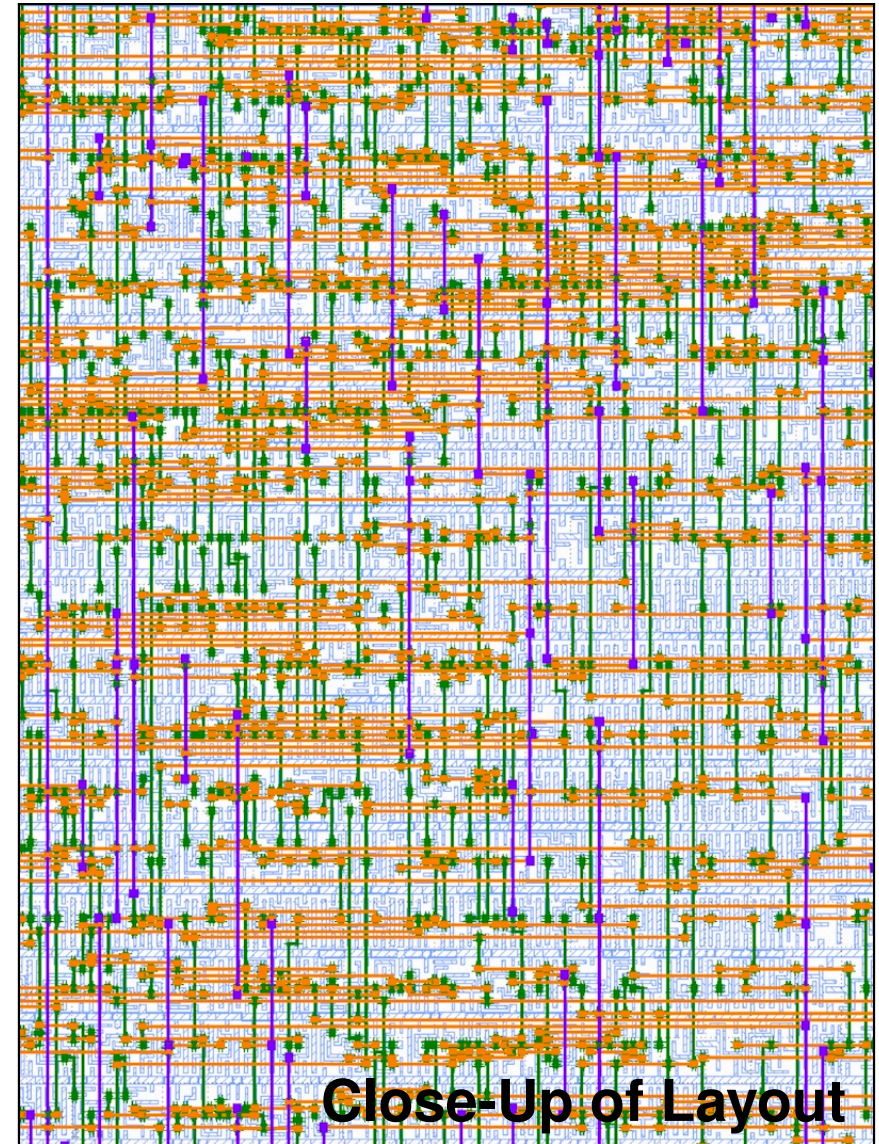
Using Klayout to View Place-and-Routed Design

- ▶ 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



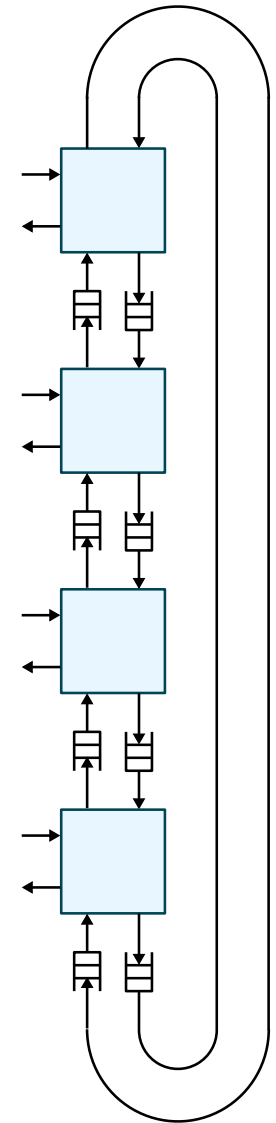
- ▶ Pure-PyMTL RTL design
- ▶ FreePDK45 Process
- ▶ Nangate standard cell library
- ▶ Yosys, graywolf, Qrouter, klayout
- ▶ 1.1 V, 625 MHz, 3400 μm^2



The Need for Productive Multi-Level Verification

```
1 class RingNet( Component ):
2     def construct( s, sz=4, ... many more parameters ... ):
3         s.in_ = [ RecvIfc( MsgType ) for _ in range(sz) ]
4         s.out = [ SendIfc( MsgType ) for _ in range(sz) ]
5
6         s.routers = [ RingRouter() for _ in range(sz) ]
7         s.channels_E = [ Queue(MsgType,2) for _ in range(sz) ]
8         s.channels_W = [ Queue(MsgType,2) for _ in range(sz) ]
9
10        for i in range( nrouters ):
11            i_n = (i+1) % nrouters
12
13            s.connect_pairs(
14                s.routers[i].router_id, i,
15                s.routers[i].out[DIR_C], s.out[i],
16                s.routers[i].in_[DIR_C], s.in_[i]
17            )
18
19            s.connect_pairs(
20                s.routers[i].out[DIR_E], s.channels_E[i].enq,
21                s.channels_E[i].deq, s.routers[i_n].in_[DIR_W],
22                s.routers[i].in_[DIR_E], s.channels_W[i].deq,
23                s.channels_W[i].enq, s.routers[i_n].out[DIR_W]
24            )
```

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



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

```
1 from hypothesis import given
2 from hypothesis.strategies \
3     import text
4
5 @given( text() )
6 def test_decode_inverts_encode(s):
7     assert decode(encode(s)) == s
```

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

<https://hypothesis.readthedocs.io>

```
1 from hypothesis.stateful \
2     import rule, precondition, \
3         RuleStateMachine
4
5 class QueueTH(RuleStateMachine):
6     def __init__( s ):
7         s.ref = []
8         s.dut = Queue()
9
10    @rule( value=integers() )
11    def enq( s, value ):
12        s.ref.appendleft( value )
13        s.dut.enq( value )
14
15    @rule()
16    @precondition( lambda s: s.ref )
17    def deq( s ):
18        correct = s.ref.popleft()
19        result = s.dut.deq()
20        assert correct == result
```

Using Stateless Hypothesis in PyMTL

```

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

4  # CL design under test
5  class Incr8bitCL( Component ):
6      def construct( s ):
7          s.in_ = InPort ( Bits8 )
8          s.out = OutPort( Bits8 )

9          @s.update
10         def up():
11             tmp = s.in_ + b8(1)
12             s.out = tmp & b8(0xef)

13  def incr_8bit_cl( x ):
14      incr = Incr8bitCL()
15      incr.apply( SimpleSim )
16      incr.in_ = x
17      incr.tick()
18      return incr.out

```

```

1  def test_simple():
2      assert incr_8bit_cl( b8(2) ) \
3          == b8(3)

4  @given( x=pst.bits(8) )
5  def test_hypothesis(x):
6      print("x =",x)
7      assert incr_8bit_cl(x) \
8          == incr_8bit(x)

% pytest example_test.py
===== test session starts =====
...
incr_8bit_test.py::test_hypothesis
x = 00
x = 88
... 19 other tries ...
x = 0e
Falsifying example:
  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/utlis_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/utlis_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(0x66c1), ... b16(0x1c11), b16(0xcee1)]
[b16(0x7a51), b16(0x3c92), b16(0x5cac), ... 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 ★

```
% 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, 0, 61711, 46, 0]
words = [3840, 15, 0, 0, 0, 61711, 0, 0]
words = [0, 0, 0, 0, 0, 61696, 0, 0]
words = [0, 0, 0, 0, 0, 61440, 0, 0]
words = [0, 0, 0, 0, 0, 256, 0, 0]
words = [0, 0, 0, 0, 0, 0, 0, 256]
words = [0, 0, 0, 0, 0, 0, 256, 0]
words = [0, 0, 0, 0, 0, 1, 0, 0]
words = [0, 0, 0, 0, 0, 0, 0, 1]
words = [0, 0, 0, 0, 0, 0, 1, 0]
words = [0, 0, 0, 0, 0, 1, 0, 0]
----- Hypothesis -----
Falsifying example:
test_hypothesis(words=[
    Bits16(0x0000), Bits16(0x0000), Bits16(0x0000), Bits16(0x0000),
    Bits16(0x0000), Bits16(0x0001), Bits16(0x0000), Bits16(0x0000) ])
```

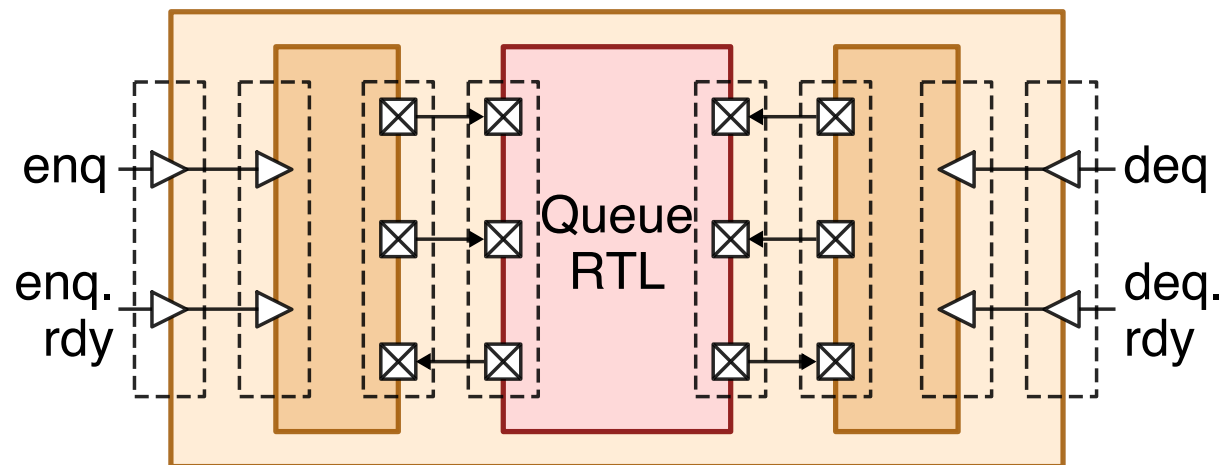
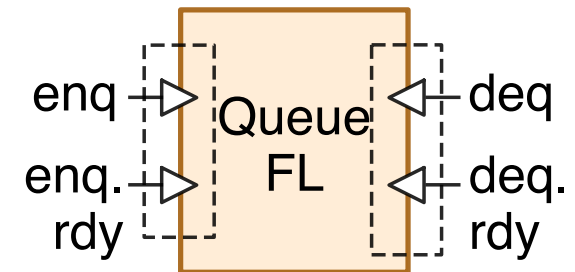
Using Stateful Hypothesis in PyMTL

```

1  class QueueFL( Component ):
2      def construct( s, maxsize ):
3          s.q = deque( maxlen=maxsize )
4
5      @non_blocking(
6          lambda s: len(s.q) < s.q.maxlen )
7      def enq( s, value ):
8          s.q.appendleft( value )
9
10     @non_blocking(
11         lambda s: len(s.q) > 0 )
12     def deq( s ):
13         return s.q.pop()
14
15 q = QueueFL( 4 )
16 q.elaborate()
17 assert q.enq.rdy()
18 q.enq(42)
19 assert q.deq.rdy()
20 assert q.deq() == 42

```

► 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!

```

1 def test_state_machine():
2     run_test_state_machine(
3         RTL2FLWrapper(QueueRTL(Bits16,4)),
4         QueueFL(Bits16,4) )

```

Example Random Test Sequence

enq(data=59488)	enq(data=37367)
deq() -> 59488	deq() -> 36011
enq(data=8637)	deq() -> 33990
deq() -> 8637	enq(data=227)
enq(data=18754)	enq(data=64506)
enq(data=37951)	deq() -> 43506
deq() -> 18754	enq(data=48999)
enq(data=23589)	deq() -> 50664
enq(data=17602)	deq() -> 227
deq() -> 37951	enq(data=45118)
enq(data=35695)	deq() -> 64506
enq(data=43506)	deq() -> 48999
deq() -> 23589	enq(data=62684)
deq() -> 17602	enq(data=29836)
deq() -> 35695	enq(data=33990)
enq(data=50664)	deq() -> 45118

Auto-Generating Minimal Counter-Examples

```

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

```

Injected Bug:
Do not update
head pointer



Trying example:

```

enq(data=9352)
enq(data=27368)
deq() -> 9352
enq(data=31419)
enq(data=46760)
enq(data=9228)
==== error ====
deq() -> 9228

```

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