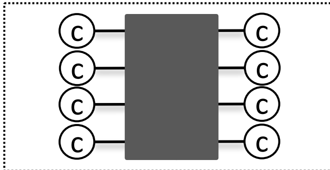
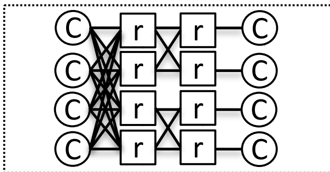
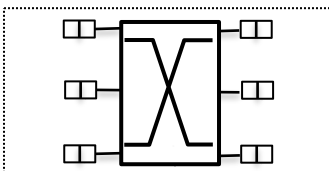
Modeling



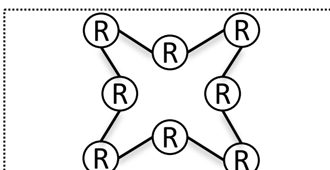
Functional-Level



Cycle-Level

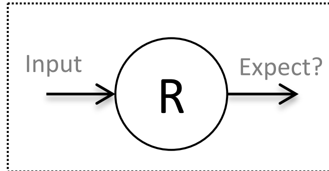


Register-Transfer-Level

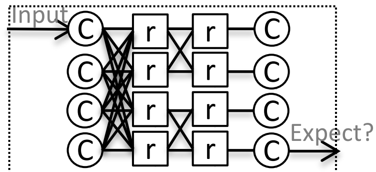


Physical-Level

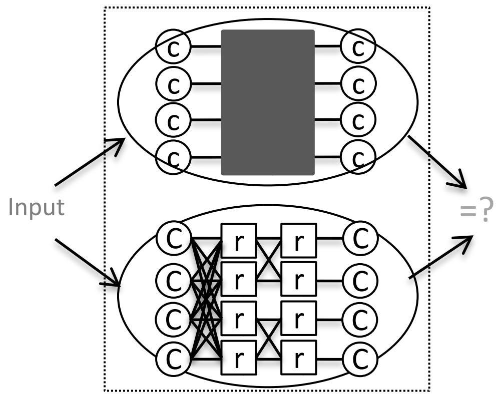
Testing



Unit

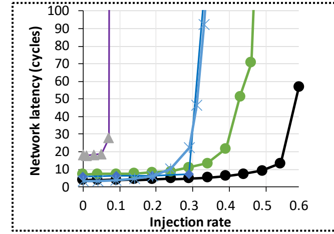


Integration



Property-Based Random

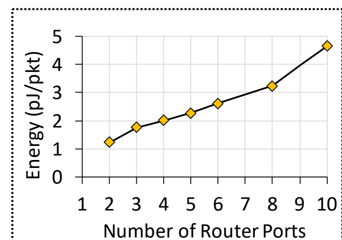
Evaluating



Simulation

```
always_ff @(posedge clk) begin : up_reg
  if ( reset ) begin
    head <= 1'd0;
    tail <= 1'd0;
    count <= 2'd0;
  end
  else begin
    head <= deq_xfer ? head_next : head;
    tail <= enq_xfer ? tail_next : tail;
    count <= ( enq_xfer & ( ~deq_xfer ) ) ?
      count + 2'd1 : ( deq_xfer & ( ~enq_xfer ) ) ?
      count - 2'd1 : count;
  end
end
```

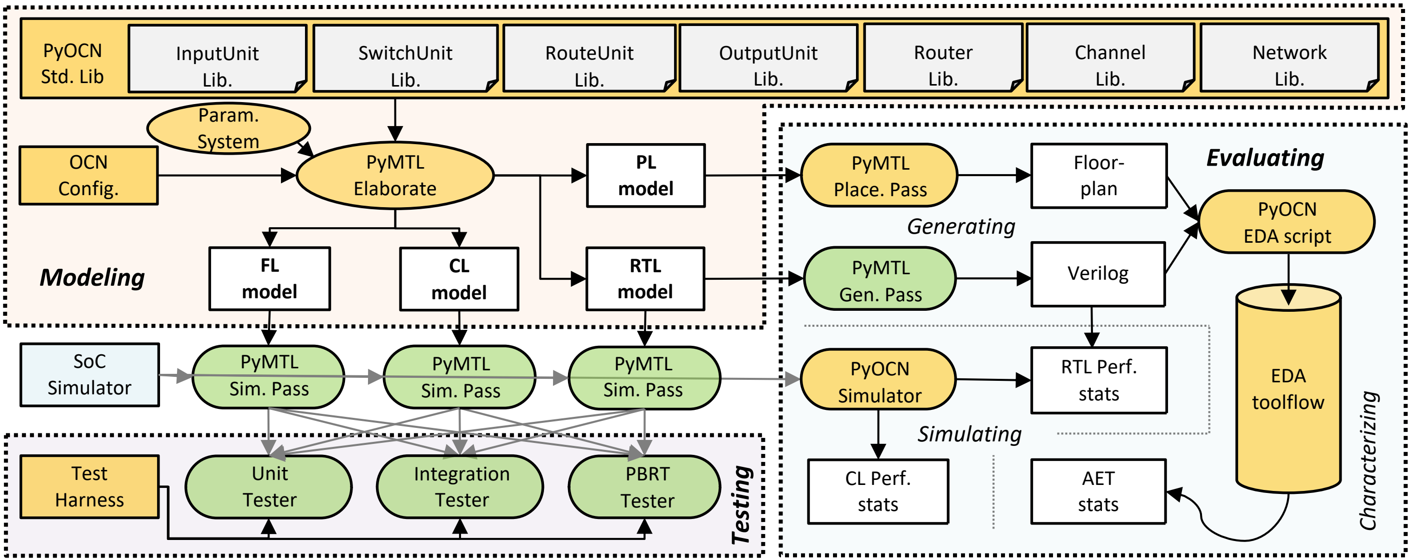
Generation



Characterization

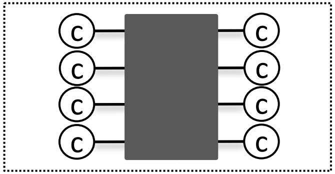
		Modeling					Testing			Evaluating		
	Lang.	Config.	Function-Level	Cycle-Level	RTL	Physical-Level	Unit	Int.	Property-based	Sim.	RTL Gen.	ASIC Char.
BookSim2	*C++*	●		●						●		
Garnet	*C++*	◐		●				●		●		
Noxim	*SysC*	●		●				●		●		●
Connect	*BSV*	◐			●					◐	●	
Netmaker	*Verilog*	◐			◐			●		◐	●	
OpenSMART	*Chisel*	◐			●		●	●		◐	●	
OpenSoC	*Chisel*	◐		●	●		●	●		◐	●	
DSENT	*C++*	◐										◐
Orion2	*C++*	◐										◐
COSI	*C++*	◐		●	●	●				●	●	◐
PyOCN	*PyMTL*	●	●	●	●	●	●	●	●	●	●	●

Overview of PyOCN Framework

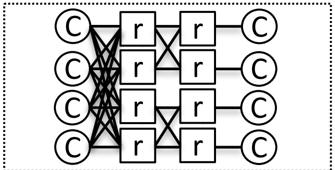


- Enables multi-level modeling to facilitate rapid design-space exploration
- Provides test harnesses for testing OCN designs modeled at different abstraction levels
- Can simulate OCNs at various abstraction levels, generate synthesizable Verilog, and drive a commercial standard-cell-based toolflow for characterizing OCN area, energy, and timing

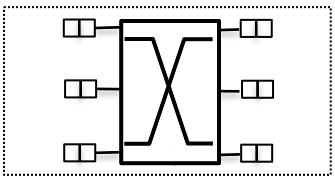
Modeling



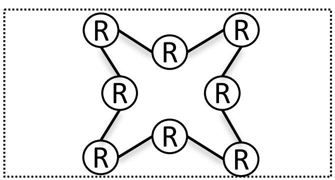
Function-Level



Cycle-Level

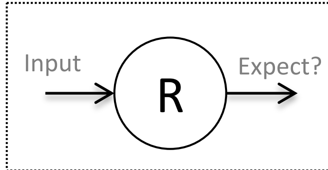


Register-Transfer-Level

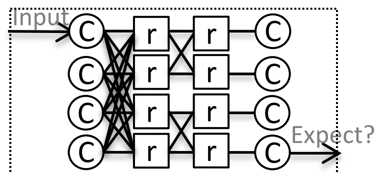


Physical-Level

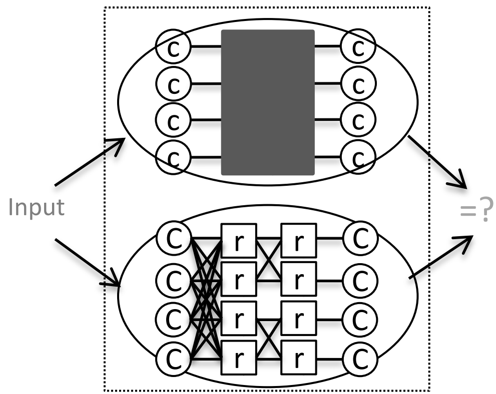
Testing



Unit

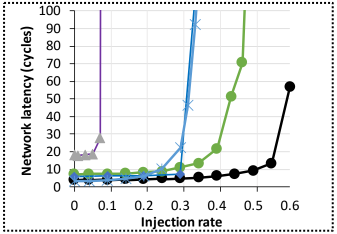


Integration



Property-Based Random

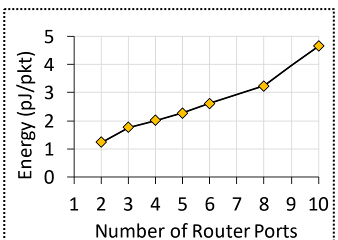
Evaluating



Simulation

```
always_ff @(posedge clk) begin : up_reg
  if ( reset ) begin
    head <= 1'd0;
    tail <= 1'd0;
    count <= 2'd0;
  end
  else begin
    head <= deq_xfer ? head_next : head;
    tail <= enq_xfer ? tail_next : tail;
    count <= ( enq_xfer & ( ~deq_xfer ) ) ?
      count + 2'd1 : ( deq_xfer & ( ~enq_xfer ) ) ?
      count - 2'd1 : count;
  end
end
```

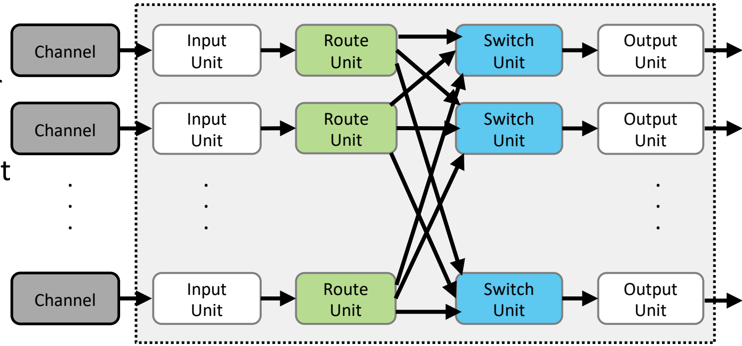
Generation



Characterization

PyOCN for Modeling OCNs

- New Modular Router Microarchitecture
  - Single unified router microarchitecture for all networks
  - Easily configure different units for different topologies, routing algorithms, and arbitration algorithms
  - Users can also provide their own units



- Multi-level modeling
  - Functional-Level
  - Cycle-Level
  - Register-Transfer-Level
  - Physical-Level

written in Python

Function-Level Modeling

- New Modular Router Microarchitecture
  - Single unified router microarchitecture for all networks
  - Easily configure different units for different topologies, routing algorithms, and arbitration algorithms
  - Users can also provide their own units
- Multi-level modeling
  - Functional-Level
  - Cycle-Level
  - Register-Transfer-Level
  - Physical-Level

```python
def ringnet_fl( src_pkts ):
  nterminals = len( src_pkts )
  dst_pkts = [ [] for _ in range( nterminals ) ]

  for packets in src_pkts:
    for pkt in packets:
      dst_pkts[ pkt.dst ].append( pkt )
  return dst_pkts
```

FL Implementation of Ring Network

Python function

Cycle-Level Modeling

- New Modular Router Microarchitecture
  - Single unified router microarchitecture for all networks
  - Easily configure different units for different topologies, routing algorithms, and arbitration algorithms
  - Users can also provide their own units
- Multi-level modeling
  - Functional-Level
  - Cycle-Level
  - Register-Transfer-Level
  - Physical-Level

```
class SwitchUnitCL( Component ):
  def construct( s, pkt_t, num_inports ):

    # Local parameters
    s.num_inports = num_inports

    # Interface
    s.get = [ \
        CallerIfc(pkt_t) for _ in range(num_inports) ]
    s.give = \
        CalleeIfc(pkt_t, method=s.give_, rdy=s.give_rdy)

    # Components
    s.priority = list( range(num_inports) )

    for i in range( num_inports ):
      s.add_constraints( M( s.get[i] ) == M( s.give ) )

  def give_rdy( s ):
    for i in range( s.num_inports ):
      if s.get[i].rdy():
        return True
    return False

  def give_( s ):
    for i in s.priority:
      if s.get[i].rdy():
        s.priority.append( s.priority.pop(i) )
        return s.get[i]()
```

CL Implementation of Switch Unit

Register-Transfer-Level Modeling

- New Modular Router Microarchitecture
  - Single unified router microarchitecture for all networks
  - Easily configure different units for different topologies, routing algorithms, and arbitration algorithms
  - Users can also provide their own units

- Multi-level modeling
  - Functional-Level
  - Cycle-Level
  - Register-Transfer-Level
  - Physical-Level

```python
class SwitchUnitRTL( Component ):
  def construct( s, pkt_t, num_inports ):
    # Local Parameters
    sel_width = clog2( num_inports )
    sel_t     = mk_bits( sel_width   )
    grant_t   = mk_bits( num_inports )

    # Interface
    s.get = [GetIfc(pkt_t) for _ in range(num_inports)]
    s.send = SendIfc(pkt_t)

    # Components
    s.arbiter = RoundRobinArbiterEn( num_inports )
    s.mux = Mux( pkt_t, num_inports )(
      out = s.send.msg,
    )
    s.encoder = Encoder( num_inports, sel_width )(
      in_ = s.arbiter.grants,
      out = s.mux.sel,
    )

    # Connections
    for i in range( num_inports ):
      s.connect( s.get[i].rdy, s.arbiter.reqs[i] )
      s.connect( s.get[i].msg, s.mux.in_[i]      )

    @s.update
    def up_arb_send_en():
      s.arbiter.en = \
        ( s.arbiter.grants > grant_t(0) ) & s.send.rdy
      s.send.en = \
        ( s.arbiter.grants > grant_t(0) ) & s.send.rdy

    @s.update
    def up_get_en():
      for i in range( num_inports ):
        s.get[i].en = s.get[i].rdy & s.send.rdy & \
                      ( s.mux.sel == sel_t(i) )
```

RTL Implementation of Switch Unit

Physical-Level Modeling

- New Modular Router Microarchitecture
  - Single unified router microarchitecture for all networks
  - Easily configure different units for different topologies, routing algorithms, and arbitration algorithms
  - Users can also provide their own units

- Multi-level modeling
  - Functional-Level
  - Cycle-Level
  - Register-Transfer-Level
  - Physical-Level

```
class RingNetworkRTL(Component):
  def construct(s, pkt_t, pos_t, nrouters, chnl_lat=0):
  ...
  def elaborate_physical(s):
    N = s.nrouters
    chnl_len = s.channels[0].dim.w
    for i, r in enumerate(s.routers):
      if i < (N / 2):
        r.dim.x = i * (r.dim.w + chnl_len)
        r.dim.y = 0
      else:
        r.dim.x = (N - i - 1) * (r.dim.w + chnl_len)
        r.dim.y = r.dim.h + chnl_len
    s.dim.w = N/2 * r.dim.w + (N/2 - 1) * chnl_len
    s.dim.h = 2 * r.dim.h + chnl_len
```

PL Implementation of Ring Network

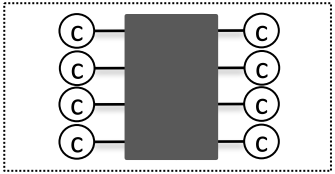
PyOCN for Modeling OCNs

- New Modular Router Microarchitecture
  - Single unified router microarchitecture for all networks
  - Easily configure different units for different topologies, routing algorithms, and arbitration algorithms
  - Users can also provide their own units
- Multi-level modeling
  - Functional-Level
  - Cycle-Level
  - Register-Transfer-Level
  - Physical-Level

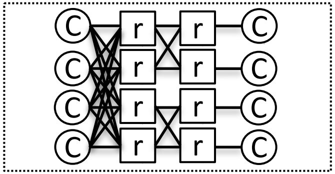
Injection Rate	**Speedup**	**Accuracy**
0.01	17.9X	86%
0.1	15.5X	87%
0.2	14.2X	87%
0.3	13.3X	97%
0.4	13.0X	74%

Multi-level simulation speedup and accuracy

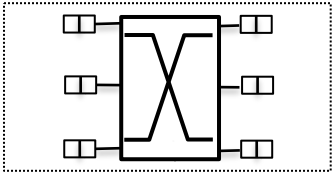
Modeling



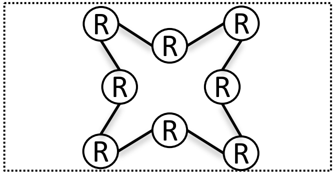
Function-Level



Cycle-Level

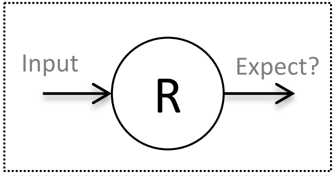


Register-Transfer-Level

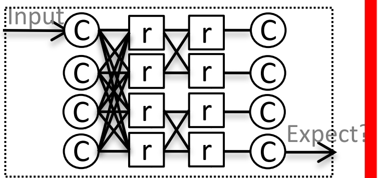


Physical-Level

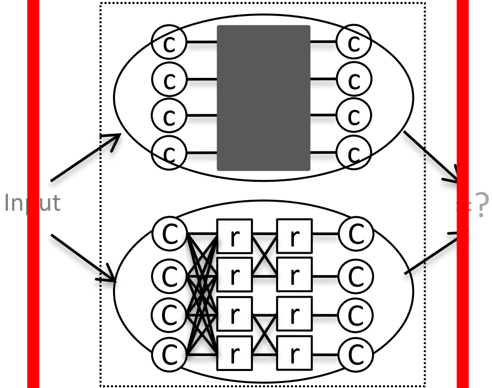
Testing



Unit

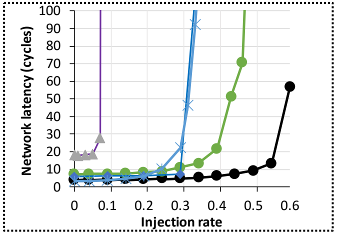


Integration



Property-Based Random

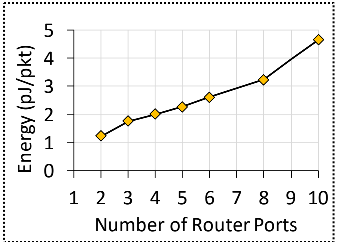
Evaluating



Simulation

```
always_ff @(posedge clk) begin : up_reg
  if ( reset ) begin
    head <= 1'd0;
    tail <= 1'd0;
    count <= 2'd0;
  end
  else begin
    head <= deq_xfer ? head_next : head;
    tail <= enq_xfer ? tail_next : tail;
    count <= ( enq_xfer & ( ~deq_xfer ) ) ?
      count + 2'd1 : ( deq_xfer & ( ~enq_xfer ) ) ?
      count - 2'd1 : count;
  end
end
```

Generation

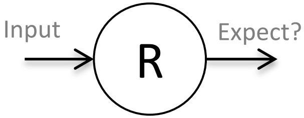


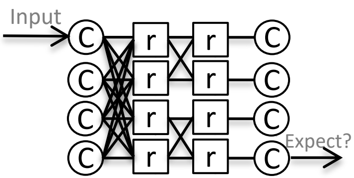
Characterization

Unit and Integration Test

PyOCN provides extensive test suites to unit test the basic network components.

PyOCN also enables integration test on complete network instances.

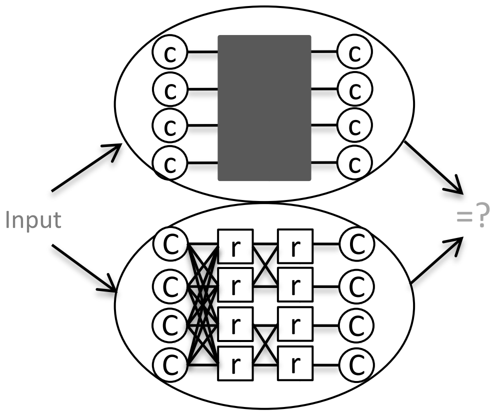





```
@pytest.mark.parametrize(
  'pos_x, pos_y',
  product( [ 0, 1, 2, 3 ], [ 0, 1, 2, 3 ] )
)
def test_simple_4x4( pos_x, pos_y ):
  ncols = 4; nrows = 4
  pkt_t = mk_mesh_pkt( ncols, nrows, nvcs=2 )

  src_pkts = [
    #      src_x  y  dst_x  y  opaque  vc  payload
    pkt_t(    0, 0,     1, 1,      0,  0, 0xfaceb00c ),
    pkt_t(    0, 2,     3, 3,      0,  0, 0xdeadface ),
  ]

  th = TestHarness( pkt_t, src_pkts )
  # Use the elegant parameter system
  th.set_param( "top.construct",
    ncols=ncols, nrows=nrows,
    pos_x=pos_x, pos_y=pos_y,
  )
  run_sim( th )
```

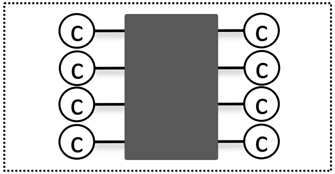
Property-Based Random Test

PyOCN uses a type-based random data generator for all inputs and checking if the DUT violates the given specification.

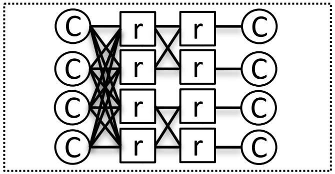



```
@hypothesis.given(
  ncols = st.integers(2, 8),
  nrows = st.integers(2, 8),
  pkts  = st.data(),
)
def test_hypothesis( ncols, nrows, pkts ):
  Pkt = mk_mesh_pkt( ncols, nrows, nvcs=2 )

  pkts_lst = pkts.draw(
    st.lists( mesh_pkt_strat( ncols, nrows ) ),
    label= "pkts"
  )

  src_pkts = mk_src_pkts( ncols, nrows, pkts_lst )
  dst_pkts = meshnet_fl( ncols, nrows, src_pkts )
  th = TestHarness( Pkt, ncols, nrows,
                    src_pkts, dst_pkts )
  run_sim( th )
```

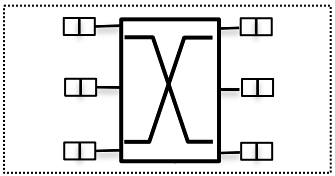
Modeling



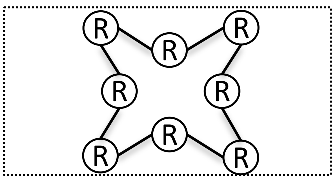
Function-Level



Cycle-Level

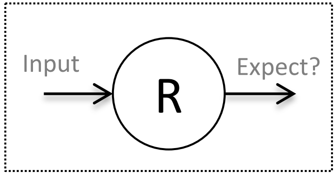


Register-Transfer-Level

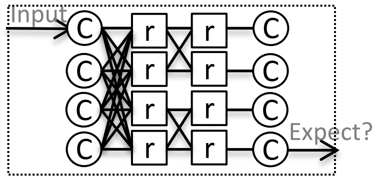


Physical-Level

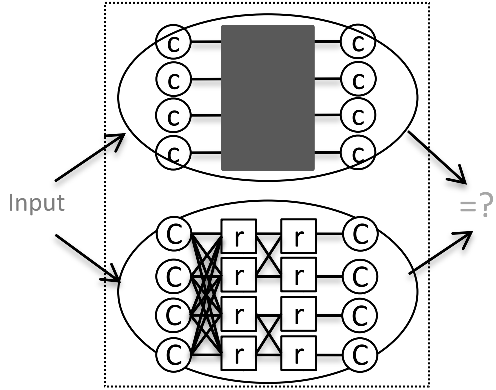
Testing



Unit

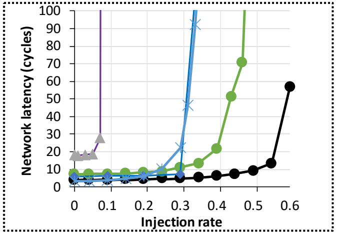


Integration



Property-Based Random

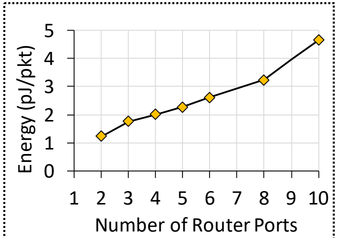
Evaluating



Simulation

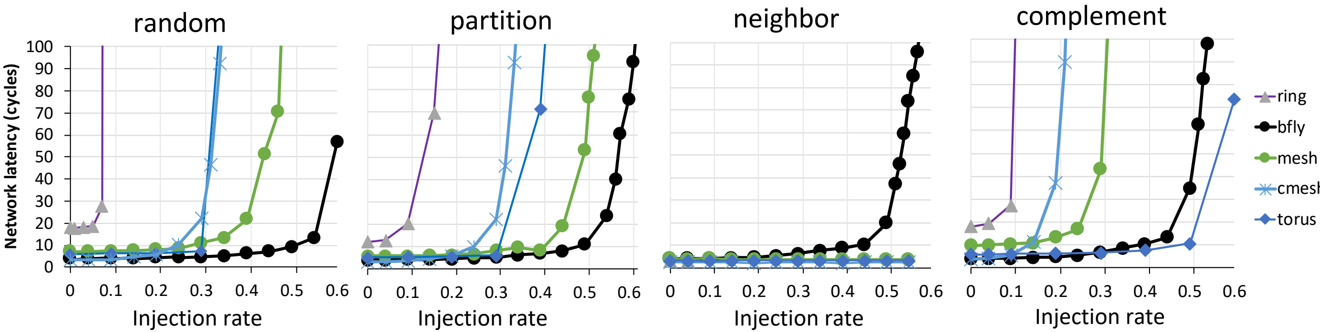
```
always_ff @(posedge clk) begin : up_reg
  if ( reset ) begin
    head <= 1'd0;
    tail <= 1'd0;
    count <= 2'd0;
  end
  else begin
    head <= deq_xfer ? head_next : head;
    tail <= enq_xfer ? tail_next : tail;
    count <= ( enq_xfer & ( ~deq_xfer ) ) ?
      count + 2'd1 : ( deq_xfer & ( ~enq_xfer ) ) ?
      count - 2'd1 : count;
  end
end
```

Generation



Characterization

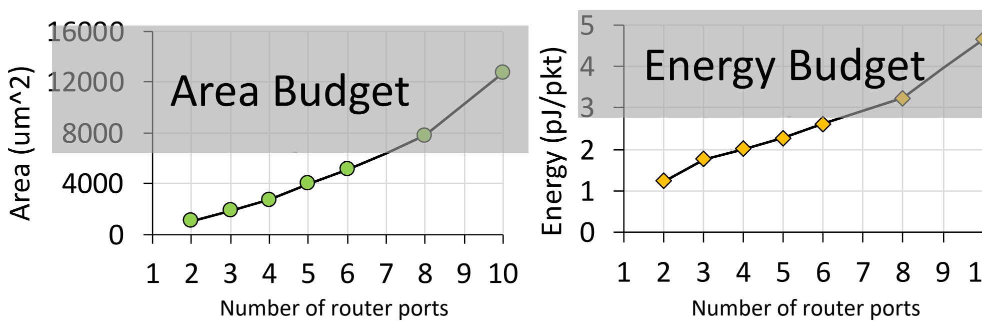
PyOCN for Evaluating OCNs



Simulation of different topologies at different injection rates

```
module Encoder__in_width_5__out_width_3
(
   input logic [0:0] clk,
   input logic [4:0] in_,
   output logic [2:0] out,
   input logic [0:0] reset
);

   localparam [31:0] __const$in_width = 32'd5;

   always_comb begin : encode
      out = 3'd0;
      for ( int i = 0; i < __const$in_width; i += 1 )
         out = in_[i] ? 3'( i ) : out;
   end
endmodule
```

Generated RTL



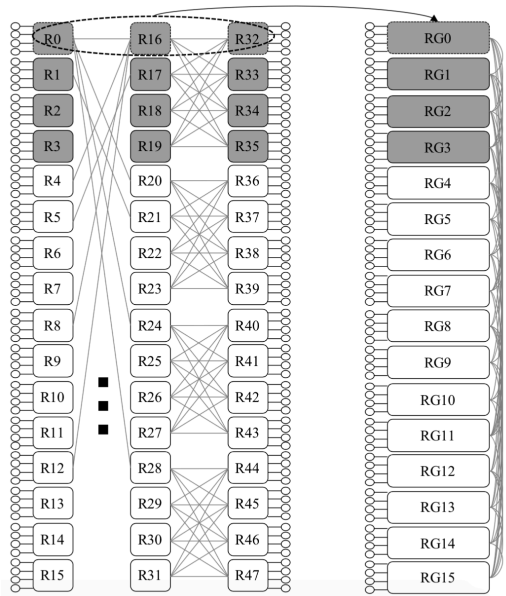
Area/Energy analysis of router

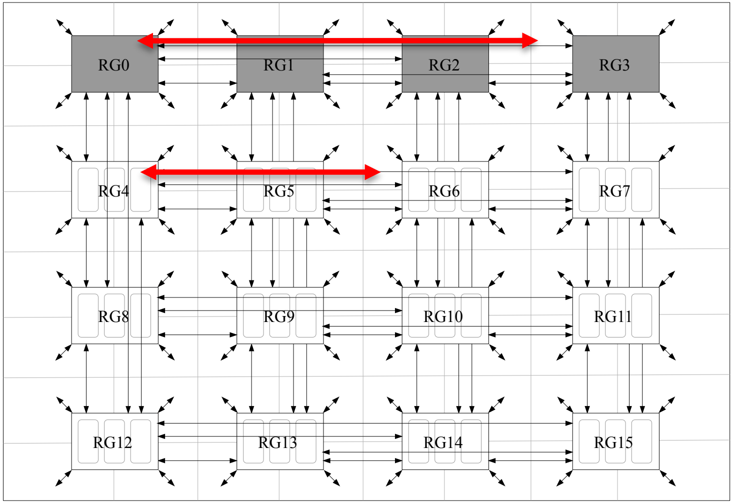
PyOCN for Evaluating OCNs

- Placement of butterfly topology
  - 64-terminal 4-ary 3-fly butterfly topology
  - 3 routers in the same row can be recognized as a router group

Case study to show the features of PyOCN

Critical paths





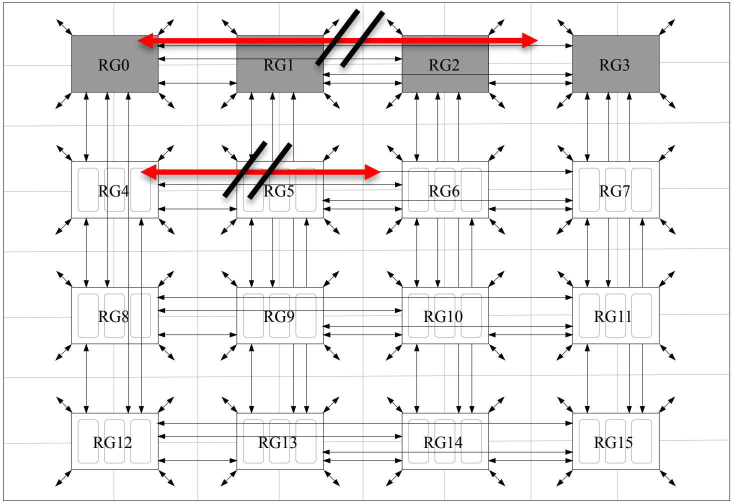
PyOCN for Evaluating OCNs

Case study to show the features of PyOCN

- Placement of butterfly topology
  - 64-terminal 4-ary 3-fly butterfly topology
  - 3 routers in the same row can be recognized as a router group
- Parameterization system
  - use *set_param()* to break down.

```
net = BFlyNetworkRTL( pkt_t, k_ary=4, n_fly=3 )
critical_paths = [
  "channels[82]",
  "channels[114]",
  ...
]
for c in critical_paths:
  net.set_param( f'top.{c}.construct', hops=2 )
net.elaborate()
```

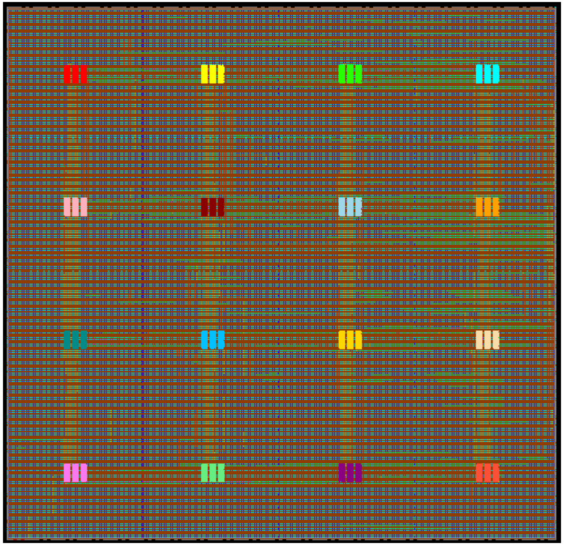
Critical paths

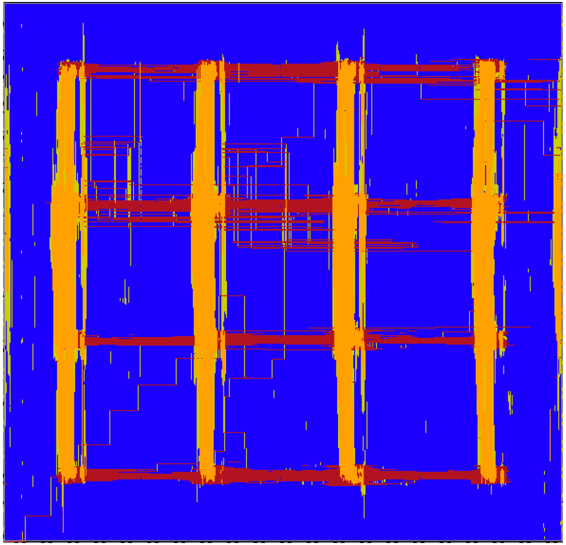


PyOCN for Evaluating OCNs

Case study to show the features of PyOCN

- Placement of butterfly topology
  - 64-terminal 4-ary 3-fly butterfly topology
  - 3 routers in the same row can be recognized as a router group
- Parameterization system
  - use *set_param()* to break down.





Open-Source PyOCN

- Open-source
  - https://github.com/cornell-brg/pymtl3-net
- Demo

To create a virtual environment and install pymtl3-net:

% python3 -m venv ${HOME}/venv

% source ${HOME}/venv/bin/activate

% pip3 install pymtl3-net

To test a 4-terminal ring using single-pkt with dumped vcd:

% pymtl3-net test ring --nterminals 4 --dump-vcd

To simulate a 2x2 mesh with specific injection rate:

% pymtl3-net sim mesh --ncols 2 --nrows 2 --injection-rate 10 -v

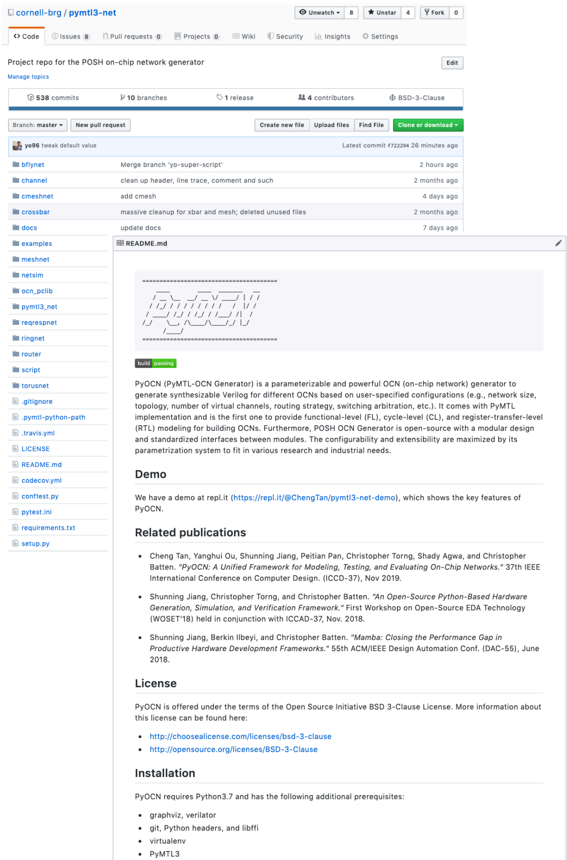
To simulate a 2x2 mesh across different injection rates:

% pymtl3-net sim mesh --ncols 2 --nrows 2 --sweep -v

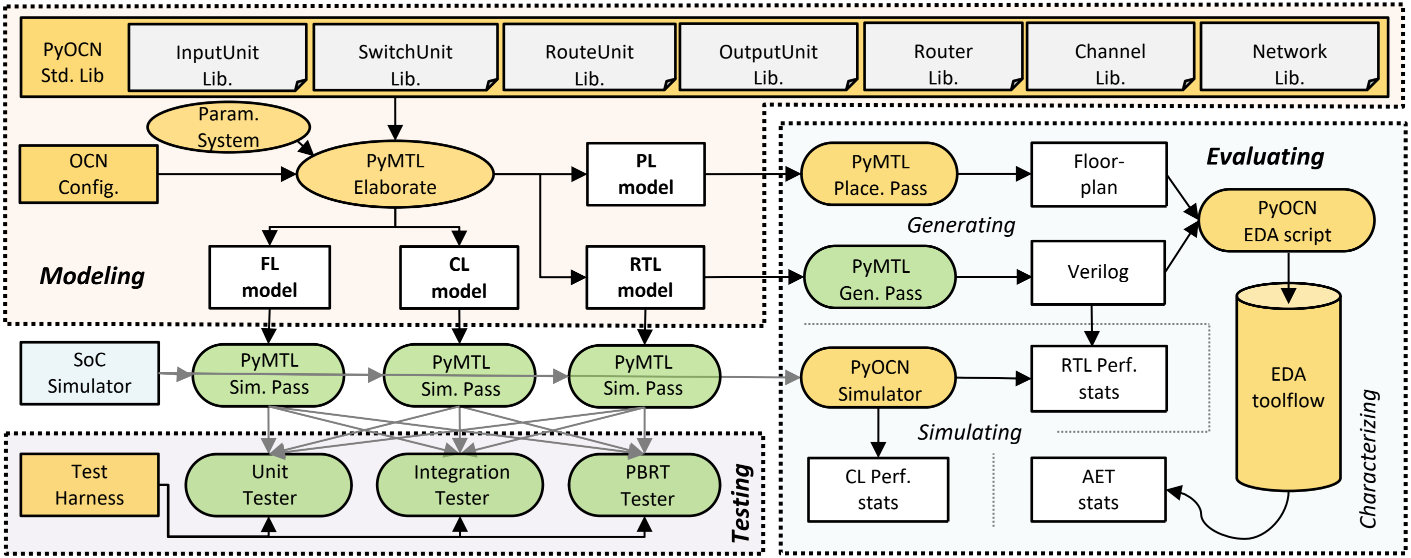
To generate a 4x4 mesh:

% pymtl3-net gen mesh --ncols 4 --nrows 4

Repl.it: https://repl.it/@ChengTan/pyocn-demo



PyOCN: A Unified Framework for Modeling, Testing, and Evaluating On-Chip Networks



- Enables multi-level modeling to facilitate rapid design-space exploration
- Provides test harnesses for testing OCN designs modeled at different abstraction levels
- Can simulate OCNs at various abstraction levels, generate synthesizable Verilog, and drive a commercial standard-cell-based toolflow for characterizing OCN area, energy, and timing

This work was supported in part by NSF CRI Award #1512937, DARPA POSH Award #FA8650-18-2-7852, and equipment, tool, and/or physical IP donations from Intel, Synopsys, and Cadence.