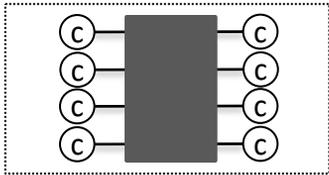
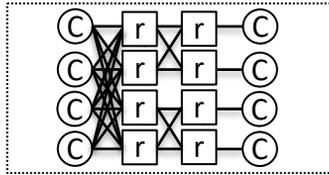


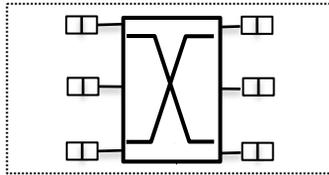
Modeling



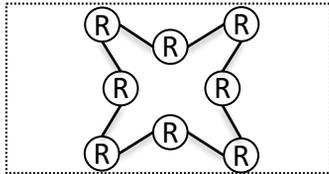
Functional-Level



Cycle-Level

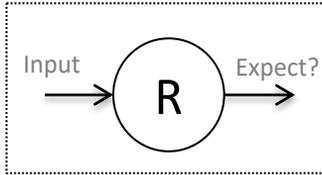


Register-Transfer-Level

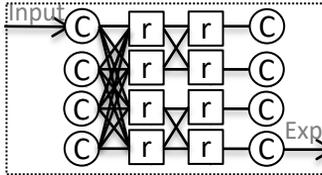


Physical-Level

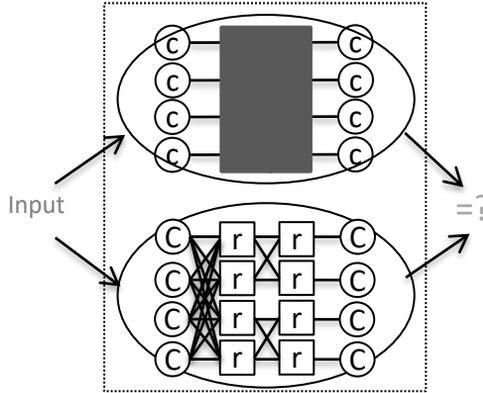
Testing



Unit

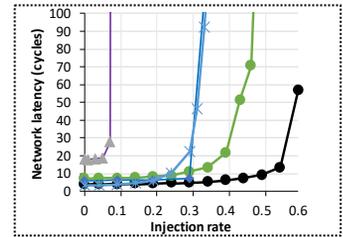


Integration



Property-Based Random

Evaluating

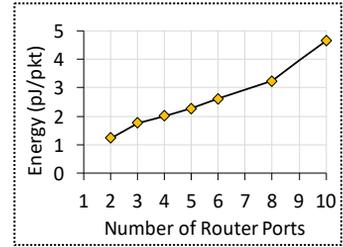


Simulation

```

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

Generation



Characterization

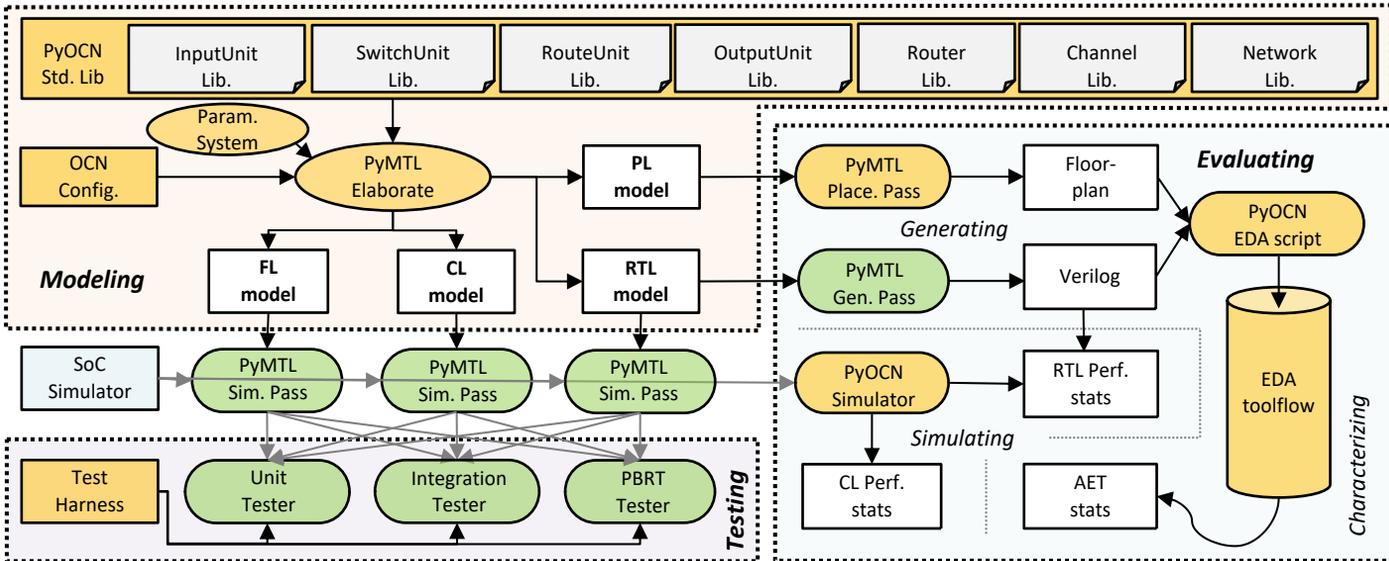
Modeling

Testing

Evaluating

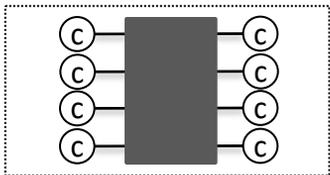
	Lang.	Config.	Function- Level	Cycle- Level	RTL	Physical- Level	Unit	Int.	Property- based	Sim.	RTL Gen.	ASIC Char.
BookSim2	C++	●		●						●		
Garnet	C++	◐		●				●		●		
Noxim	SysC	●		●				●		●		●
Connect	BSV	◐				●				◐	●	
Netmaker	Verilog	◐				◐		●		◐	●	
OpenSMART	Chisel	◐				●	●	●		◐	●	
OpenSoC	Chisel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

Overview of PyOCN Framework

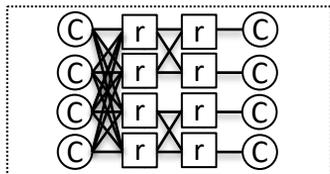


- Enables multi-level modeling to facilitate rapid design-space exploration
- Provides test harnesses for testing OCN designs modeled at different abstraction levels
- Can simulate OCNs at various abstraction levels, generate synthesizable Verilog, and drive a commercial standard-cell-based toolflow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

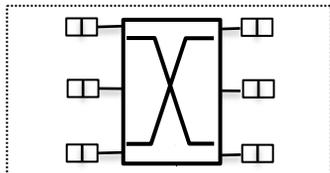
Modeling



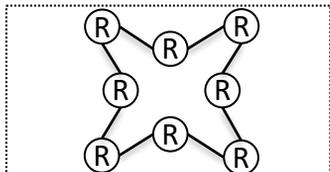
Function-Level



Cycle-Level

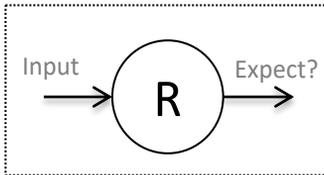


Register-Transfer-Level

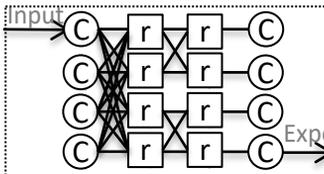


Physical-Level

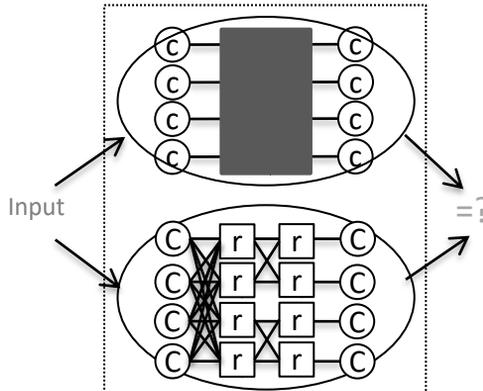
Testing



Unit

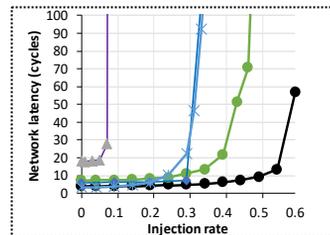


Integration



Property-Based Random

Evaluating

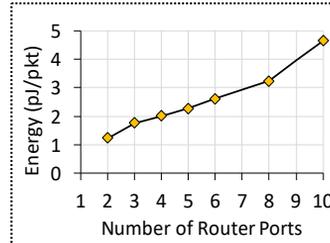


Simulation

```

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

Generation

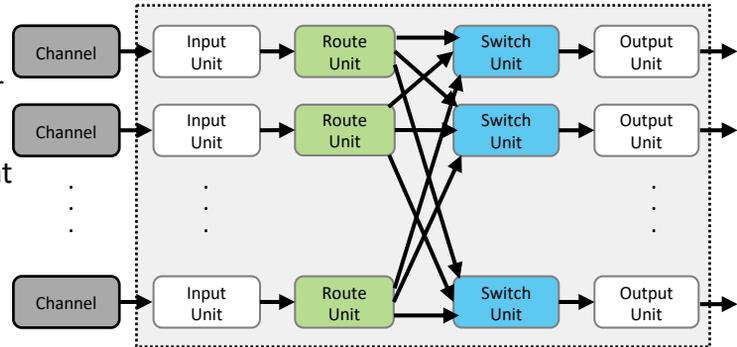


Characterization

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

```

1 def ringnet_fl( src_pkts ):
2     nterminals = len( src_pkts )
3     dst_pkts = [ [] for _ in range( nterminals ) ]
4
5     for packets in src_pkts:
6         for pkt in packets:
7             dst_pkts[ pkt.dst ].append( pkt )
8     return dst_pkts

```

FL Implementation of Ring Network

Python
function

- Multi-level modeling

- Functional-Level
- Cycle-Level
- Register-Transfer-Level
- Physical-Level

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

```

1 class SwitchUnitCL( Component ):
2     def construct( s, pkt_t, num_inports ):
3
4         # Local parameters
5         s.num_inports = num_inports
6
7         # Interface
8         s.get = [ \
9             CallerIfc(pkt_t) for _ in range(num_inports) ]
10        s.give = \
11            CalleeIfc(pkt_t, method=s.give_, rdy=s.give_rdy)
12
13        # Components
14        s.priority = list( range(num_inports) )
15
16        for i in range( num_inports ):
17            s.add_constraints( M( s.get[i] ) == M( s.give ) )
18
19        def give_rdy( s ):
20            for i in range( s.num_inports ):
21                if s.get[i].rdy():
22                    return True
23            return False
24
25        def give_( s ):
26            for i in s.priority:
27                if s.get[i].rdy():
28                    s.priority.append( s.priority.pop(i) )
29            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

```

1 class SwitchUnitRTL( Component ):
2     def construct( s, pkt_t, num_inports ):
3         # Local Parameters
4         sel_width = clog2( num_inports )
5         sel_t     = mk_bits( sel_width )
6         grant_t   = mk_bits( num_inports )
7
8         # Interface
9         s.get = [GetIfc(pkt_t) for _ in range(num_inports)]
10        s.send = SendIfc(pkt_t)
11
12        # Components
13        s.arbiter = RoundRobinArbiterEn( num_inports )
14        s.mux = Mux( pkt_t, num_inports )(
15            out = s.send.msg,
16        )
17        s.encoder = Encoder( num_inports, sel_width )(
18            in_ = s.arbiter.grants,
19            out = s.mux.sel,
20        )
21
22        # Connections
23        for i in range( num_inports ):
24            s.connect( s.get[i].rdy, s.arbiter.reqs[i] )
25            s.connect( s.get[i].msg, s.mux.in_[i] )
26
27        @s.update
28        def up_arb_send_en():
29            s.arbiter.en = \
30                ( s.arbiter.grants > grant_t(0) ) & s.send.rdy
31            s.send.en = \
32                ( s.arbiter.grants > grant_t(0) ) & s.send.rdy
33
34        @s.update
35        def up_get_en():
36            for i in range( num_inports ):
37                s.get[i].en = s.get[i].rdy & s.send.rdy & \
38                    ( 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

```

1 class RingNetworkRTL(Component):
2     def construct(s, pkt_t, pos_t, n_routers, chnl_lat=0):
3         ...
4     def elaborate_physical(s):
5         N = s.n_routers
6         chnl_len = s.channels[0].dim.w
7         for i, r in enumerate(s.routers):
8             if i < (N / 2):
9                 r.dim.x = i * (r.dim.w + chnl_len)
10                r.dim.y = 0
11            else:
12                r.dim.x = (N - i - 1) * (r.dim.w + chnl_len)
13                r.dim.y = r.dim.h + chnl_len
14                s.dim.w = N/2 * r.dim.w + (N/2 - 1) * chnl_len
15                s.dim.h = 2 * r.dim.h + chnl_len

```

PL Implementation of Ring Network

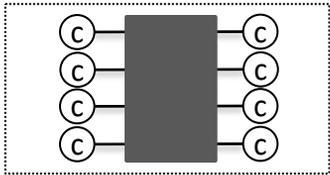
PyOCN for Modeling OCNs

- **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

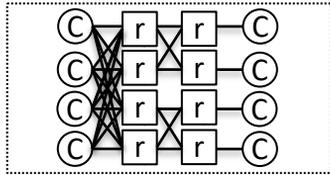
Injection Rate	Speedup	Accuracy
0.01	17.9X	86%
0.1	15.5X	87%
0.2	14.2X	87%
0.3	13.3X	97%
0.4	13.0X	74%

Multi-level simulation speedup and accuracy

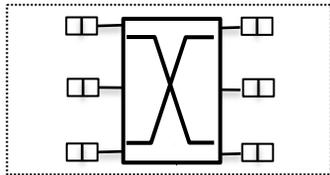
Modeling



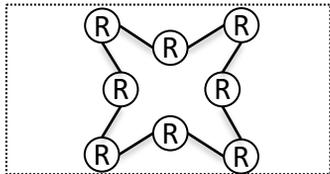
Function-Level



Cycle-Level

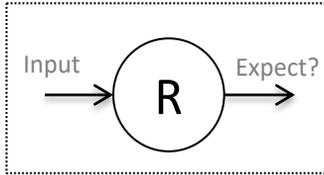


Register-Transfer-Level

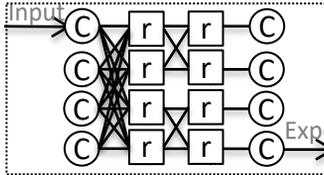


Physical-Level

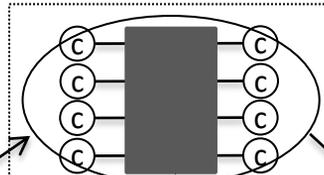
Testing



Unit

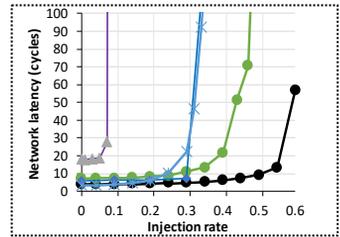


Integration



Property-Based Random

Evaluating

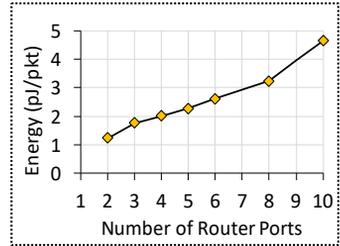


Simulation

```

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

Generation



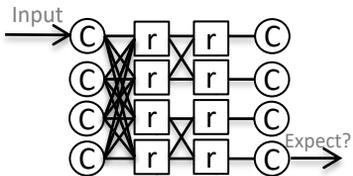
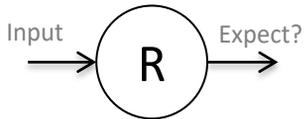
Characterization

Input ?

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.



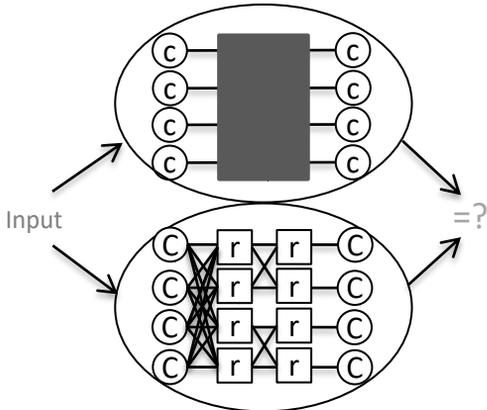
```

1 @pytest.mark.parametrize(
2     'pos_x, pos_y',
3     product( [ 0, 1, 2, 3 ], [ 0, 1, 2, 3 ] )
4 )
5 def test_simple_4x4( pos_x, pos_y ):
6     ncols = 4; nrows = 4
7     pkt_t = mk_mesh_pkt( ncols, nrows, nvcs=2 )
8
9     src_pkts = [
10        # src_x y dst_x y opaque vc payload
11        pkt_t( 0, 0, 1, 1, 0, 0, 0xfaceb00c ),
12        pkt_t( 0, 2, 3, 3, 0, 0, 0xdeadface )
13    ]
14
15    th = TestHarness( pkt_t, src_pkts )
16    # Use the elegant parameter system
17    th.set_param( "top.construct",
18                ncols=ncols, nrows=nrows,
19                pos_x=pos_x, pos_y=pos_y,
20            )
21    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.

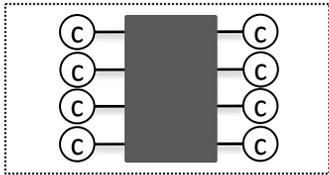


```

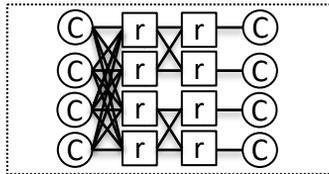
1 @hypothesis.given(
2     ncols = st.integers(2, 8),
3     nrows = st.integers(2, 8),
4     pkts = st.data(),
5 )
6 def test_hypothesis( ncols, nrows, pkts ):
7     Pkt = mk_mesh_pkt( ncols, nrows, nvcs=2 )
8
9     pkts_lst = pkts.draw(
10         st.lists( mesh_pkt_strat( ncols, nrows ) ),
11         label= "pkts"
12     )
13
14     src_pkts = mk_src_pkts( ncols, nrows, pkts_lst )
15     dst_pkts = meshnet_fl( ncols, nrows, src_pkts )
16     th = TestHarness( Pkt, ncols, nrows,
17                     src_pkts, dst_pkts )
18     run_sim( th )

```

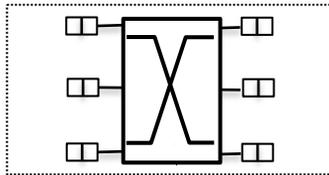
Modeling



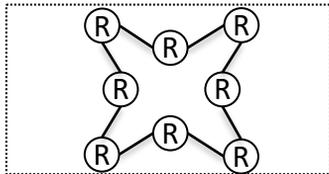
Function-Level



Cycle-Level

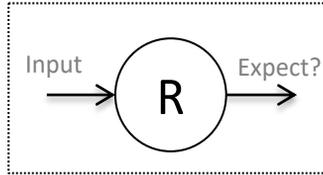


Register-Transfer-Level

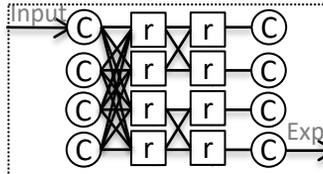


Physical-Level

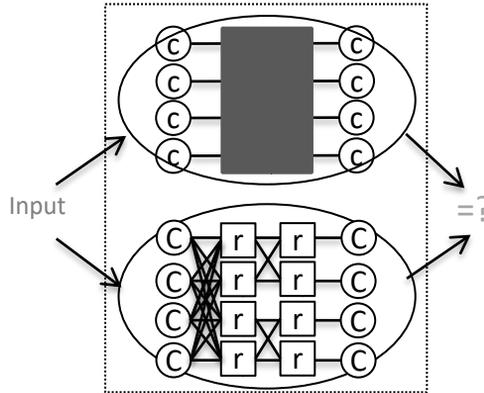
Testing



Unit

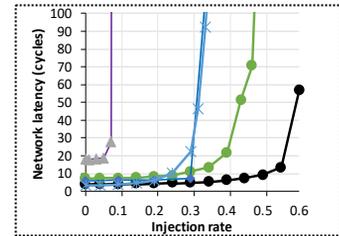


Integration



Property-Based Random

Evaluating

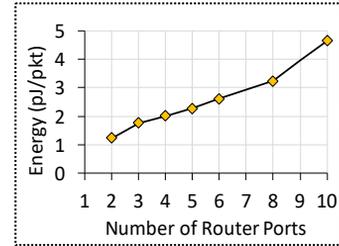


Simulation

```

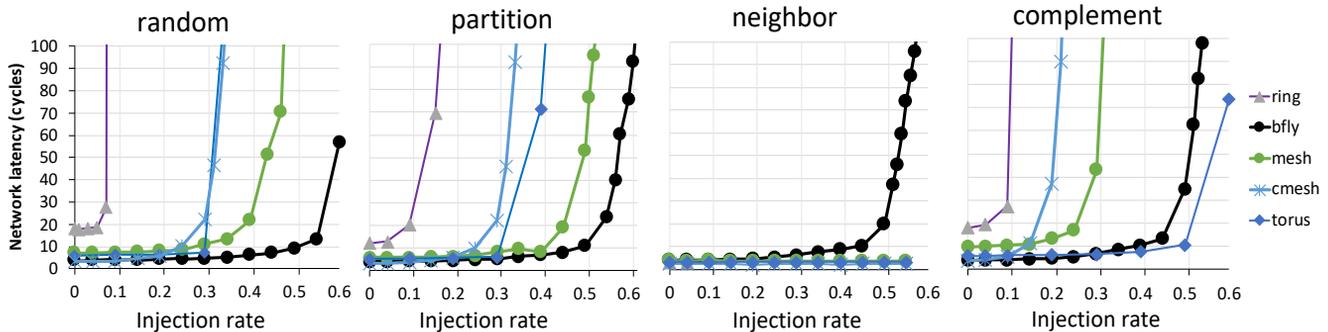
always_ff @(posedge clk) begin : up_reg
  if (reset) begin
    head <= 1'd0;
    tail <= 1'd0;
    count <= 2'd0;
  end
  else begin
    head <= deg_xfer ? head_next : head;
    tail <= enq_xfer ? tail_next : tail;
    count <= (enq_xfer & (~deg_xfer)) ?
      count + 2'd1 : (deg_xfer & (~enq_xfer))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

Generation



Characterization

PyOCN for Evaluating OCNs



Simulation of different topologies at different injection rates

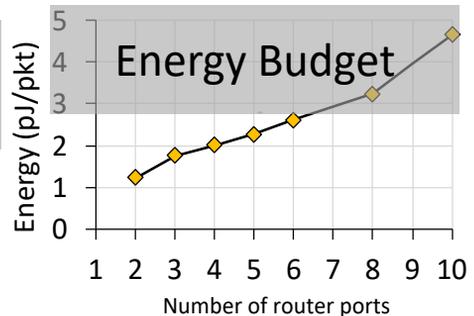
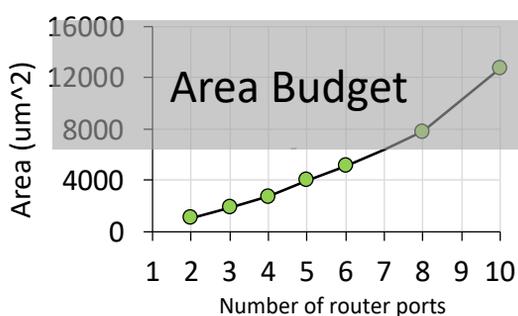
```

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



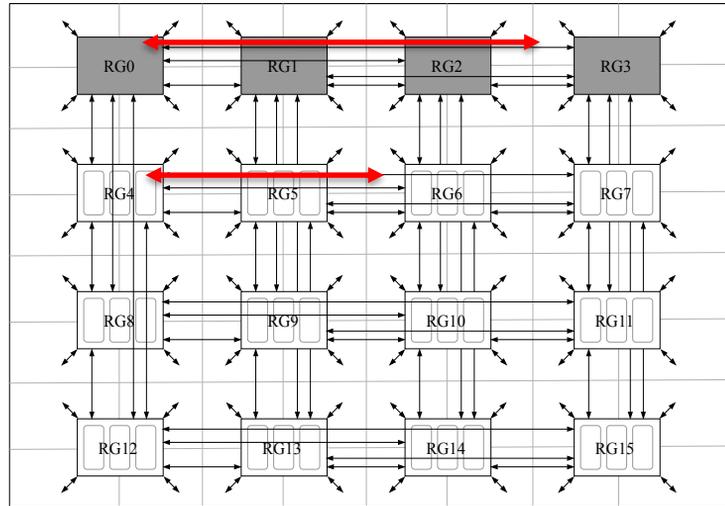
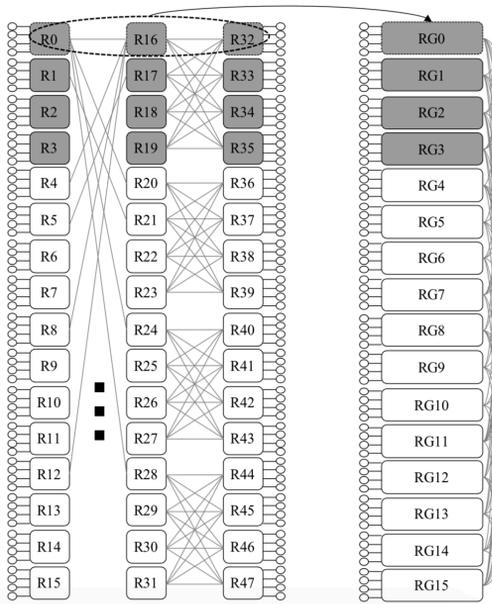
Area/Energy analysis of router

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

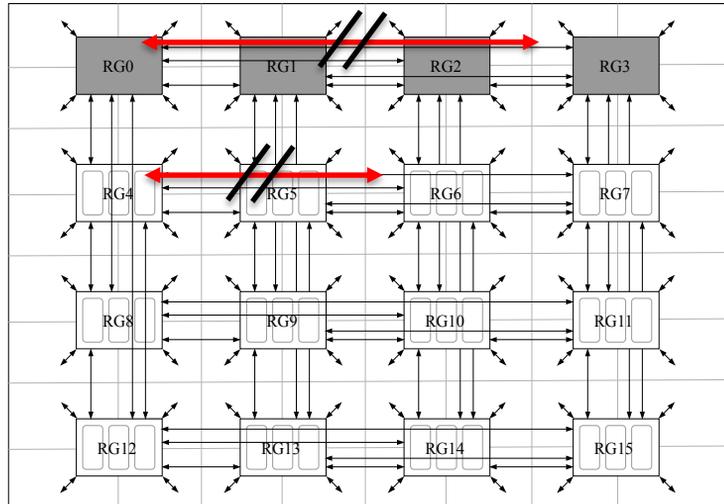
- Placement of butterfly topology
 - 64-terminal 4-ary 3-fly butterfly topology
 - 3 routers in the same row can be recognized as a router group
- Parameterization system
 - use `set_param()` to break down.

Case study to show the features of PyOCN

Critical paths

```

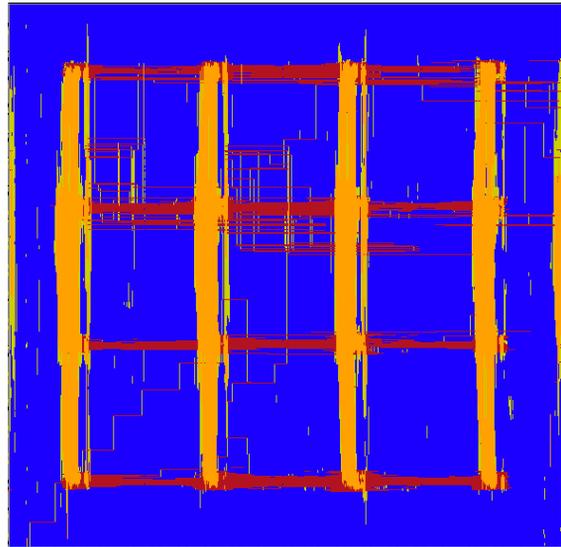
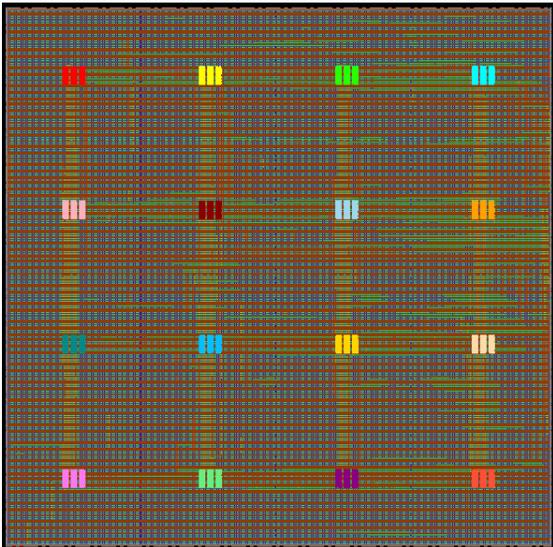
net = BFlyNetworkRTL( pkt_t, k_ary=4, n_fly=3 )
critical_paths = [
    "channels[82]",
    "channels[114]",
    ...
]
for c in critical_paths:
    net.set_param( f'top.{c}.construct', hops=2 )
net.elaborate()
  
```



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
- Parameterization system
 - use `set_param()` to break down.

Case study to show the features of PyOCN



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
```

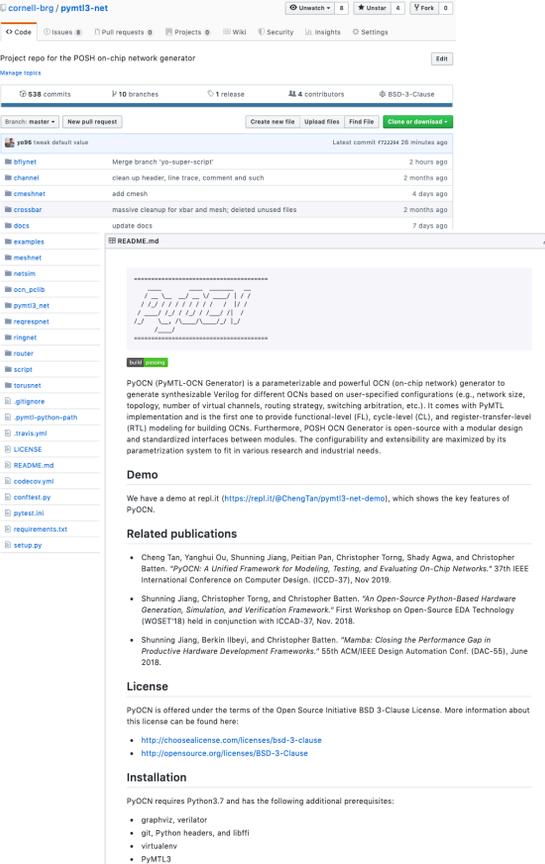
To simulate a 2x2 mesh across different injection rates:

```
% 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>



The screenshot shows the GitHub repository page for `cornell-brg/pymtl3-net`. The repository is described as the "Project repo for the POSH on-chip network generator" and is licensed under "BSD-3-Clause". It has 938 commits, 10 branches, 1 release, and 4 contributors.

The README content is as follows:

PyOCN (PyMTL-OCN Generator) is a parameterizable and powerful OCN (on-chip network) generator to generate synthesizable Verilog for different OCNs based on user-specified configurations (e.g., network size, topology, number of virtual channels, routing strategy, switching arbitration, etc.). It comes with PyMTL implementation and is the first one to provide functional-level (FL), cycle-level (CL), and register-transfer-level (RTL) modeling for building OCNs. Furthermore, PyOCN Generator is open-source with a modular design and standardized interfaces between modules. The configurability and extensibility are maximized by its parameterization system to fit in various research and industrial needs.

Demo

We have a demo at repl.it (<https://repl.it/@ChengTan/pymtl3-net-demo>), which shows the key features of PyOCN.

Related publications

- Cheng Tan, Yanghui Ou, Shunning Jiang, Peilian Pan, Christopher Torng, Shady Agwa, and Christopher Batten. "PyOCN: A Unified Framework for Modeling, Testing, and Evaluating On-Chip Networks." 37th IEEE International Conference on Computer Design. (ICCD-37), Nov 2019.
- Shunning Jiang, Christopher Torng, and Christopher Batten. "An Open-Source Python-Based Hardware Generation, Simulation, and Verification Framework." First Workshop on Open-Source EDA Technology (POSET'18) held in conjunction with ICCAD-37, Nov. 2018.
- Shunning Jiang, Berkin Ilbazi, and Christopher Batten. "Mamba: Closing the Performance Gap in Productive Hardware Development Frameworks." 55th ACM/IEEE Design Automation Conf. (DAC-55), June 2018.

License

PyOCN is offered under the terms of the Open Source Initiative BSD-3-Clause License. More information about this license can be found here:

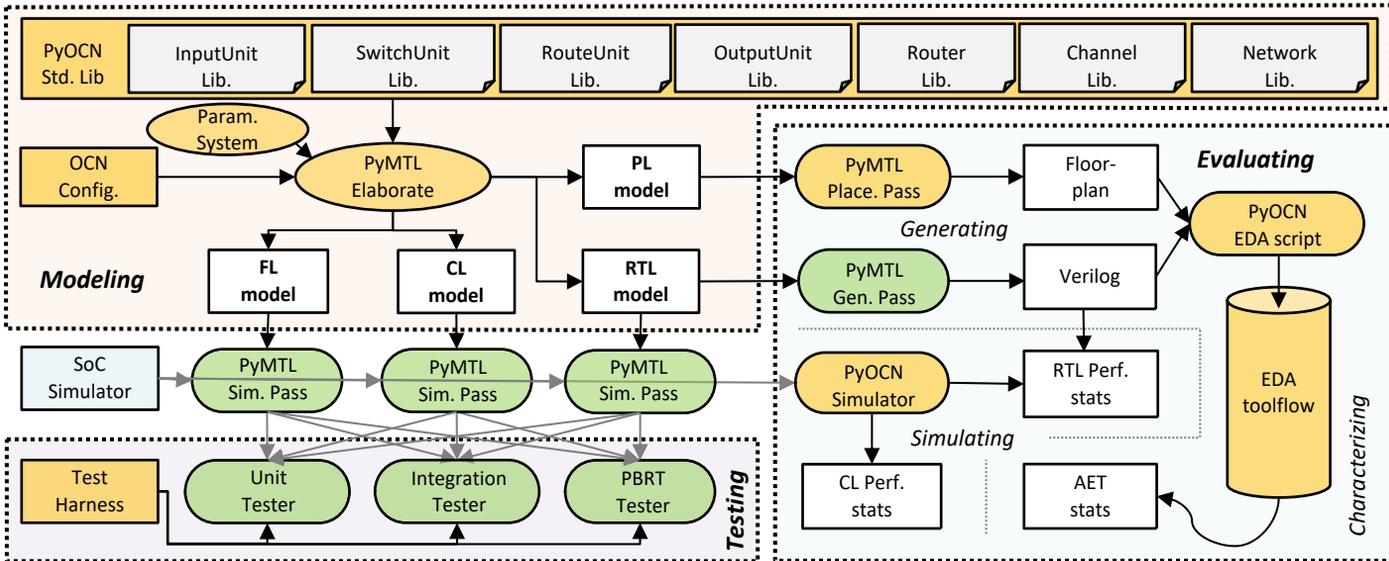
- <http://choosealicense.com/licenses/bsd-3-clause>
- <http://opensource.org/licenses/bsd-3-clause>

Installation

PyOCN requires Python3.7 and has the following additional prerequisites:

- graphviz, verilator
- git, Python headers, and libffi
- virtualenv
- PyMTL3

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.