**Modeling**

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

**Testing**

- **Unit**
- **Integration**
- **Simulation**

```verilog
testbench.v:
always_ff @(posedge clk) begin : up_reg
    if (reset) begin
        head <= 1'd0;
        tail <= 1'd0;
        count <= 2'd0;
    end
    else begin
        head <= head + 1'd0;
        tail <= tail + 1'd0;
        count <= count + 1'd0;
    end
end
```

**Evaluating**

- **Generation**
- **Characterization**

**PyOCN: A Unified Framework for Modeling, Testing, and Evaluating On-Chip Networks**
<table>
<thead>
<tr>
<th></th>
<th>Modeling</th>
<th>Testing</th>
<th>Evaluating</th>
</tr>
</thead>
<tbody>
<tr>
<td>BookSim2</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Garnet</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Noxim</td>
<td>SysC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connect</td>
<td>BSV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Netmaker</td>
<td>Verilog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenSMART</td>
<td>Chisel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenSoC</td>
<td>Chisel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSENT</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orion2</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COSI</td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PyOCN</td>
<td>PyMTL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
PyOCN: A Unified Framework for Modeling, Testing, and Evaluating On-Chip Networks

**Modeling**
- Function-Level
- Cycle-Level
- Register-Transfer-Level
- Physical-Level

**Testing**
- Input
- Expect?
- Integration

**Evaluating**
- Network latency (cycles)
- Injection rate
- Simulation
- Generation

**Property-Based Random**
- Input
- =?
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

```python
class SwitchUnitCL( Component ):
    def construct( s, pkt_t, num_imports ):
        # Local parameters
        s.num_imports = num_imports

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

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

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

    def give_rdy( s ):
        for i in range( s.num_imports ):
            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( self, pkt_t, num_imports ):
        # Local Parameters
        sel_width = clog2( num_imports )
        sel_t = mk_bits( sel_width )
        grant_t = mk_bits( num_imports )

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

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

        # Connections
        for i in range( num_imports ):
            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_imports ):
            s.get[i].en = s.get[i].rdy & s.send.rdy & \  
                ( s.mux.sel == sel_t(0) )
```

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

```python
class RingNetworkRTL(Component):
    def construct(s, pkt_t, pos_t, nroutes, chnl_lat=0):
        ...  
        def elaborate_physical(s):
            N = s.nroutes
            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

<table>
<thead>
<tr>
<th>Injection Rate</th>
<th>Speedup</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>17.9X</td>
<td>86%</td>
</tr>
<tr>
<td>0.1</td>
<td>15.5X</td>
<td>87%</td>
</tr>
<tr>
<td>0.2</td>
<td>14.2X</td>
<td>87%</td>
</tr>
<tr>
<td>0.3</td>
<td>13.3X</td>
<td>97%</td>
</tr>
<tr>
<td>0.4</td>
<td>13.0X</td>
<td>74%</td>
</tr>
</tbody>
</table>

Multi-level simulation speedup and accuracy
PyOCN: A Unified Framework for Modeling, Testing, and Evaluating On-Chip Networks

**Modeling**

- Function-Level
- Cycle-Level
- Register-Transfer-Level
- Physical-Level

**Testing**

- Input
- Expect?

**Evaluating**

- Network latency (cycles)
- Injection rate

**Simulation**

```verilog
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 <= (eng_xfer ? tail_next : tail);
count <= (eng_xfer & (~deg_xfer)) ?
count + '2'd1 : (deg_xfer & (~eng_xfer)) ?
count - '2'd1 : count;
end
end
```

**Generation**

- Energy (pJ/pkt)

**Characterization**

- Number of Router Ports
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.

```python
@unittest.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)

    srcPkts = [
        # 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, srcPkts)
    # 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.

```python
@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_f1( ncols, nrows, src_pkts )
    th = TestHarness( Pkt, ncols, nrows,
    src_pkts, dst_pkts )
    run_sim( th )
```
PyOCN: A Unified Framework for Modeling, Testing, and Evaluating On-Chip Networks

Modeling

- Function-Level
- Cycle-Level
- Register-Transfer-Level
- Physical-Level

Testing

- Unit
- Integration
- Property-Based Random

Evaluating

- Simulation
- Generation
- Characterization

- Network latency (cycles)
- Injection rate
- Energy (pJ/pkt)
- Number of Router Ports
PyOCN: A Unified Framework for Modeling, Testing, and Evaluating On-Chip Networks

PyOCN for Evaluating OCNs

Simulation of different topologies at different injection rates

Generated RTL

Area Budget

Energy Budget

Area/Energy analysis of router

Network latency (cycles)

Injection rate

Area (um^2)

Energy (pJ/pkt)

Simulation of different topologies at different injection rates

Area Budget

Energy Budget

Area/Energy analysis of router

Network latency (cycles)

Injection rate

Area (um^2)

Energy (pJ/pkt)

Simulation of different topologies at different injection rates

Area Budget

Energy Budget

Area/Energy analysis of router

Network latency (cycles)

Injection rate

Area (um^2)

Energy (pJ/pkt)

Simulation of different topologies at different injection rates

Area Budget

Energy Budget

Area/Energy analysis of router

Network latency (cycles)

Injection rate

Area (um^2)

Energy (pJ/pkt)

Simulation of different topologies at different injection rates

Area Budget

Energy Budget

Area/Energy analysis of router

Network latency (cycles)

Injection rate

Area (um^2)

Energy (pJ/pkt)

Simulation of different topologies at different injection rates

Area Budget

Energy Budget

Area/Energy analysis of router

Network latency (cycles)

Injection rate

Area (um^2)

Energy (pJ/pkt)

Simulation of different topologies at different injection rates

Area Budget

Energy Budget

Area/Energy analysis of router

Network latency (cycles)

Injection rate

Area (um^2)

Energy (pJ/pkt)

Simulation of different topologies at different injection rates

Area Budget

Energy Budget

Area/Energy analysis of router

Network latency (cycles)

Injection rate

Area (um^2)

Energy (pJ/pkt)
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.

```python
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()
```

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
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 --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.