# Lessons Learned from PyMTL:

## Insights from Academia and Industry

Derek Lockhart

# Outline

Part 1: The View from Academia

- Motivations for PyMTL
- Design of PyMTL
- Features of PyMTL
- Insights from PyMTL

Part 2: The View from Industry

- Design Challenges
- Insights from Industry

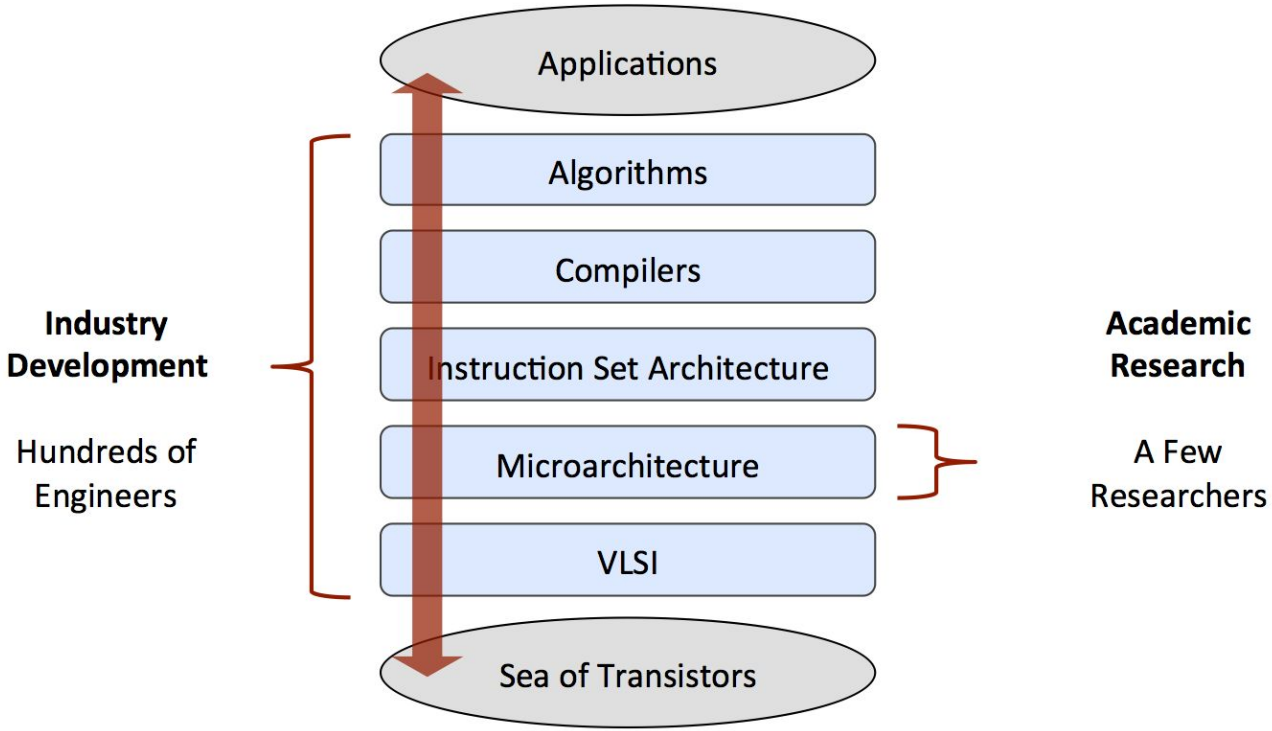
# Part 1: The View from Academia

## Motivations for PyMTL

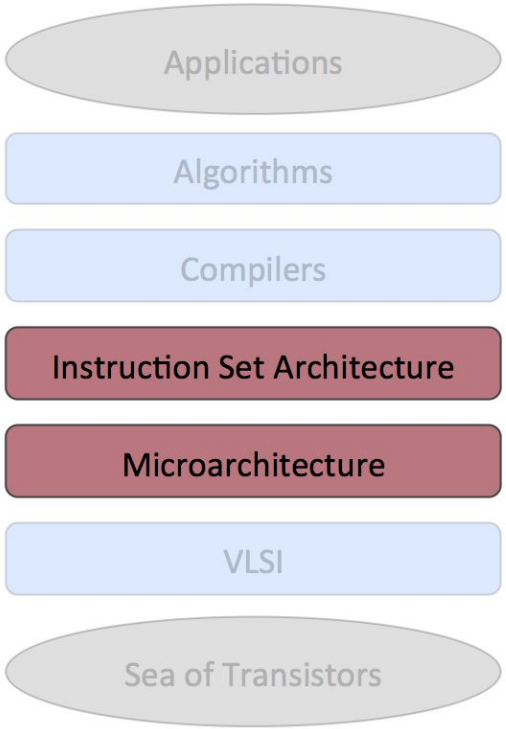
# Strong (?) Claims

- There is value in researchers implementing RTL.
  (*Many academic computer architects don't agree with this!*)
- There is value in higher-level (functional and cycle-approximate) modeling.
  *(A few academic computer architects don't agree with this either.)*
- Doing novel accelerator research requires both.

# Computer Architecture Research Abstractions



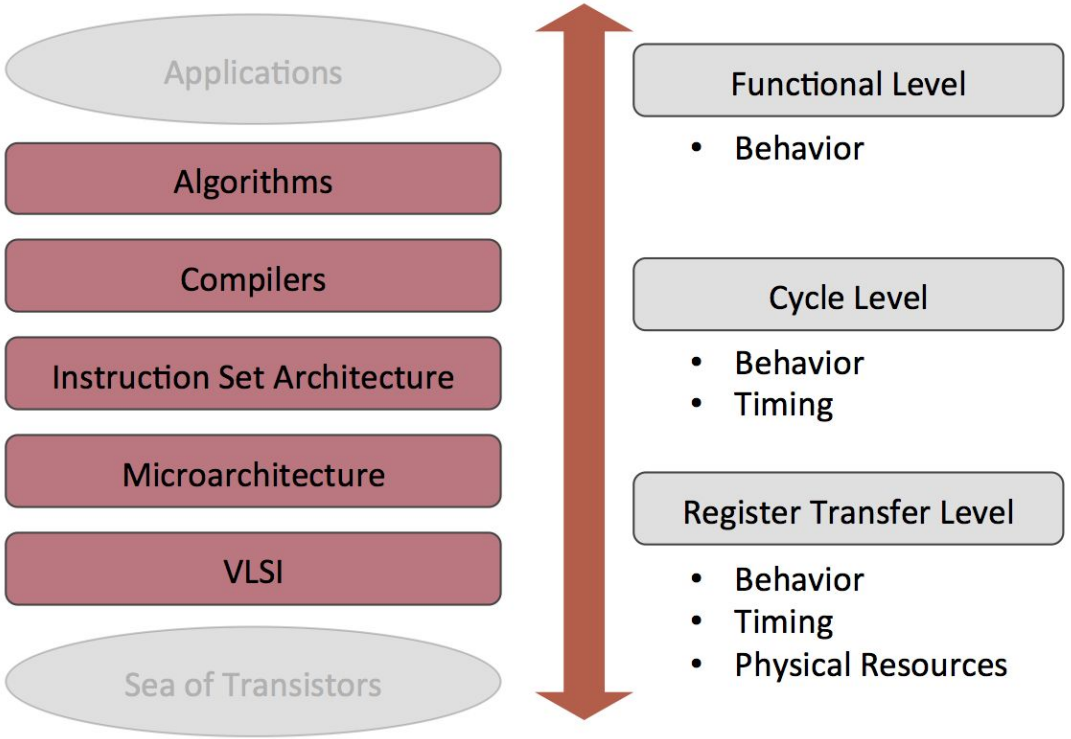
# Computer Architecture Research Methodologies



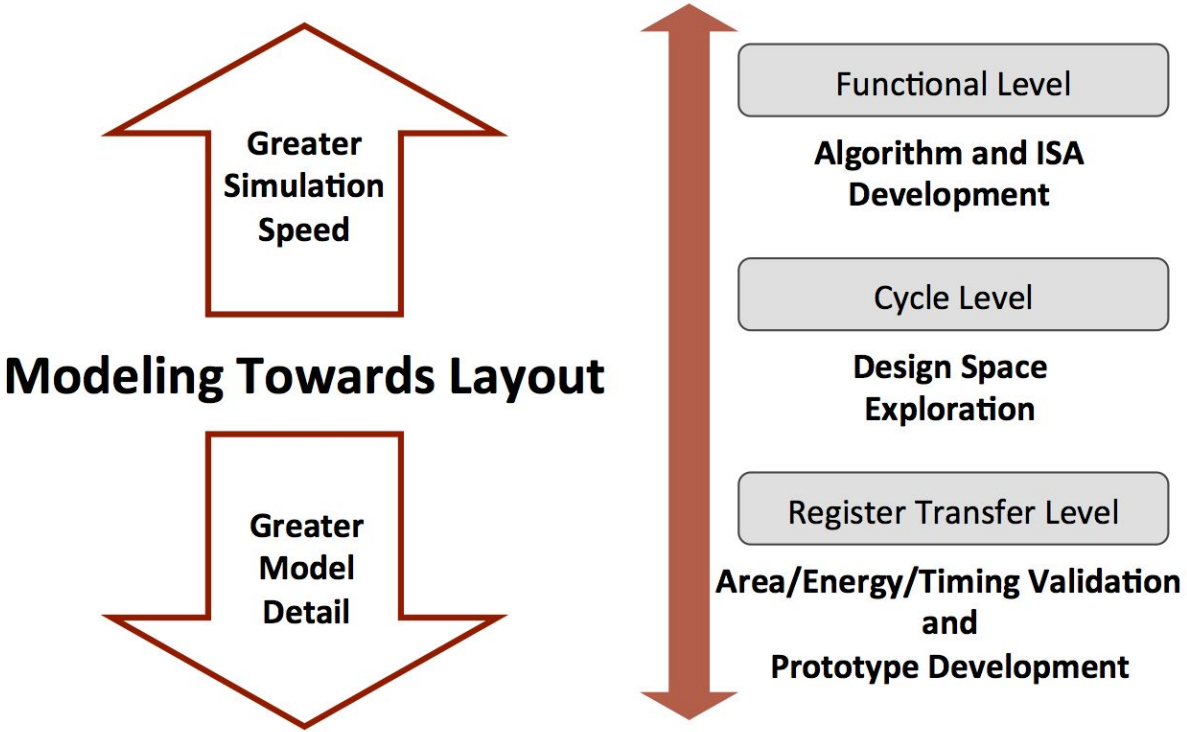

Cycle Level

- Behavior
- Timing

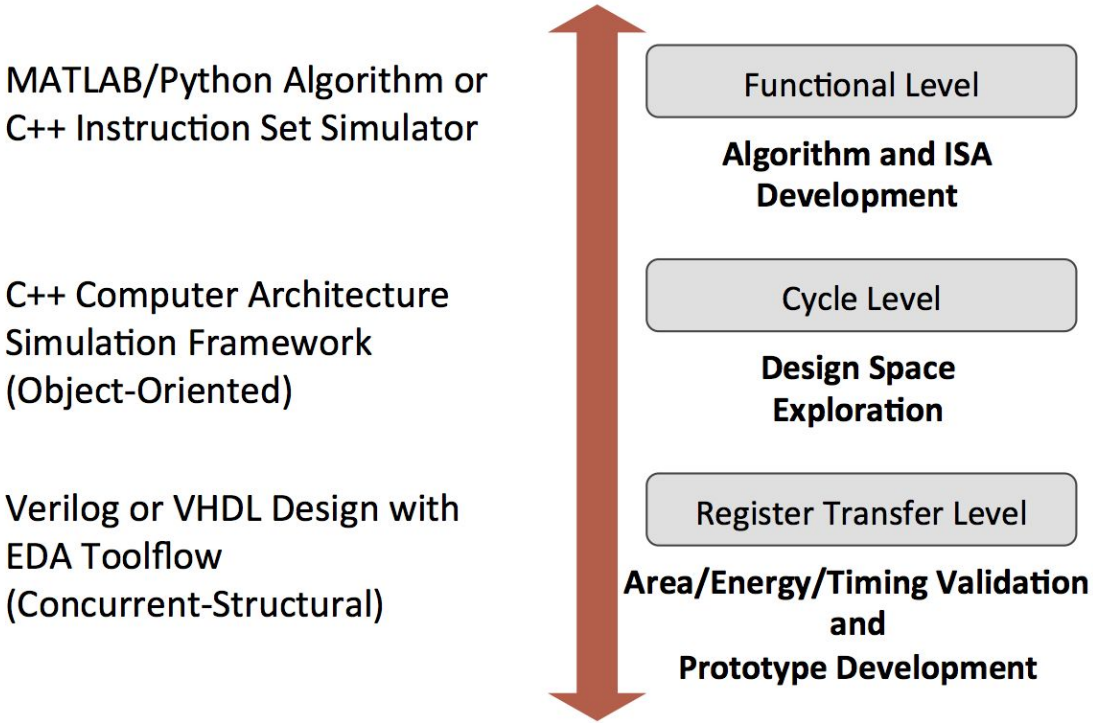
# Computer Architecture Research Methodologies



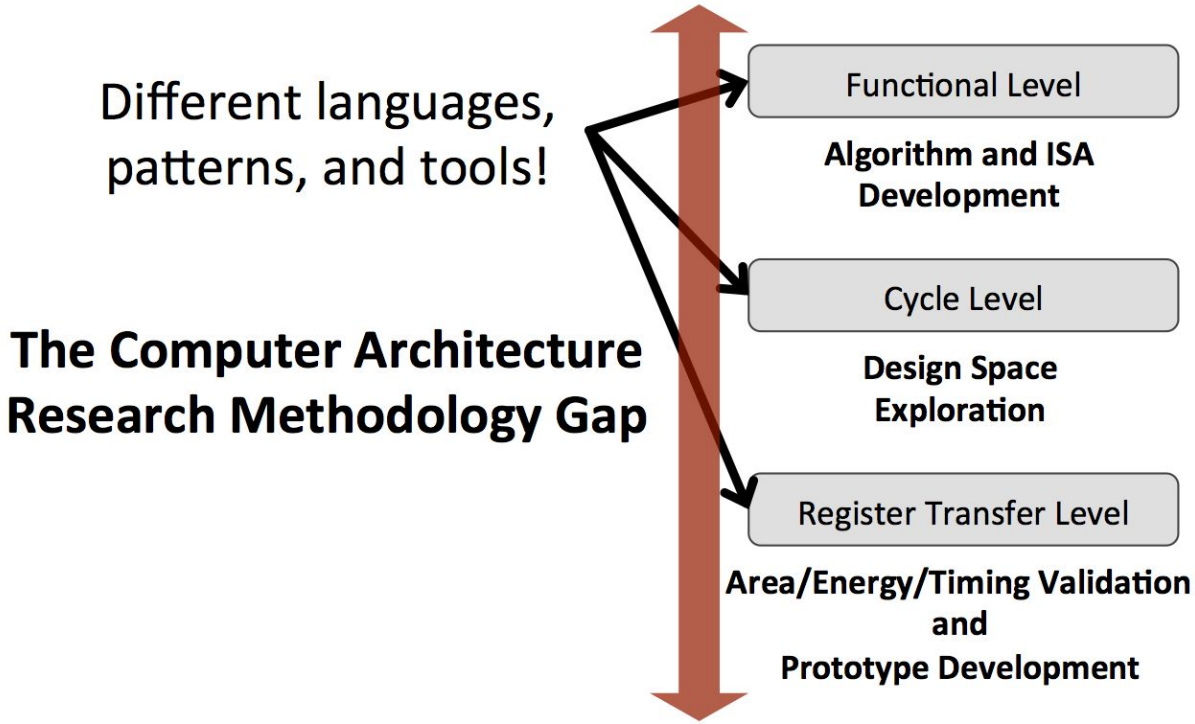
# Computer Architecture Research Toolflows



# Computer Architecture Research Toolflows



# Computer Architecture Research Toolflows



# Computer Architecture Research Pain Points

Higher-Level Modeling:

- Algorithmic Exploration in C++
- Accurate Architectural Modeling
- Unit-Testing C++ and Matlab
- Debugging C++

RTL Implementation:

- Parameterizing RTL
- Unit-Testing RTL
- Simulating and Debugging RTL
- Driving RTL With Software Tests
- Co-simulating C++ and RTL

# PyMTL Influences

| | |
|---|---|
| • **Concurrent-Structural Modeling** (Liberty, Cascade, SystemC) | Consistent interfaces across abstractions |
| • **Unified Modeling Languages** (SystemC) | Unified design environment for FL, CL, RTL |
| • **Hardware Generation Languages** (Chisel, Genesis2, BlueSpec, MyHDL) | Productive RTL design space exploration |
| • **HDL-Integrated Simulation Frameworks** (Cascade) | Productive RTL validation and cosimulation |
| • **Latency-Insensitive Interfaces** (Liberty, BlueSpec) | Component and test bench reuse |

# PyMTL Influences

Claim: Current hardware design methodologies are not sufficient for productively researching next generation hardware.

- Need for more productive modeling frameworks and design languages.
- Also need fast simulation strategies for rapid design space exploration and verification.

# PyMTL Influences

Claim: Current hardware design methodologies are not sufficient for productively researching next generation hardware.

- Need for more productive modeling frameworks and design languages.
- Also need fast simulation strategies for rapid design space exploration and verification

Approach: Selective Embedded Just-In-Time Specialization (SEJITS)

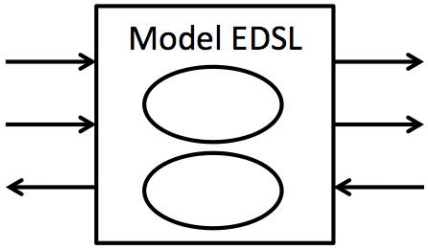
- Provide domain-specific abstractions using **embedded-DSLs.**
- Create **just-in-time specializers** to generate optimized simulator code at runtime.

# Part 1: The View from Academia

## Design of PyMTL

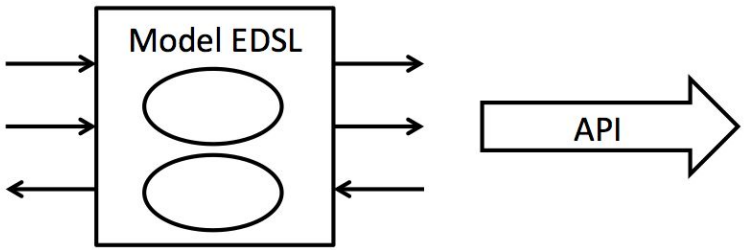
# PyMTL Components

- A Python EDSL for concurrent-structural hardware modeling



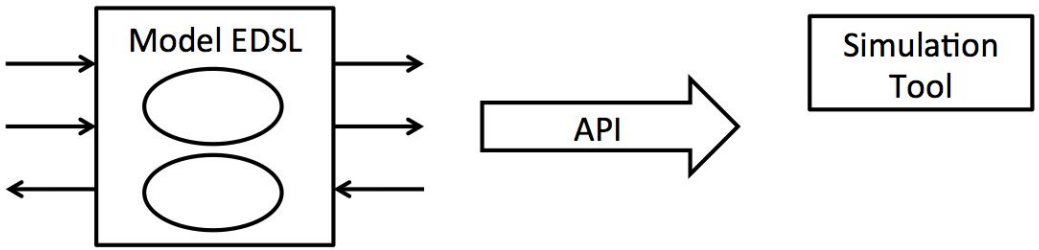
# PyMTL Components

- A Python EDSL for concurrent-structural hardware modeling
- A Python API for analyzing models described in the PyMTL EDSL



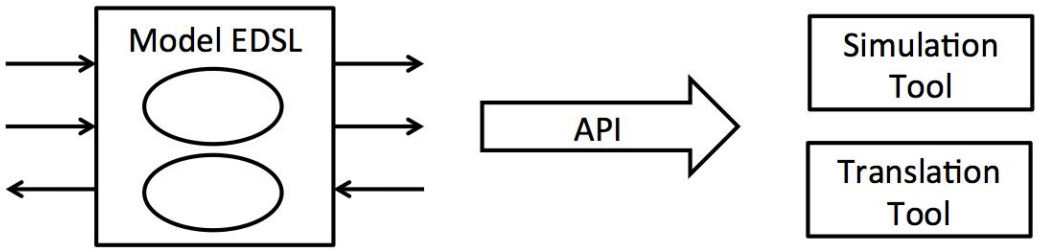
# PyMTL Components

- A Python EDSL for concurrent-structural hardware modeling
- A Python API for analyzing models described in the PyMTL EDSL
- A Python tool for simulating PyMTL FL, CL, and RTL models



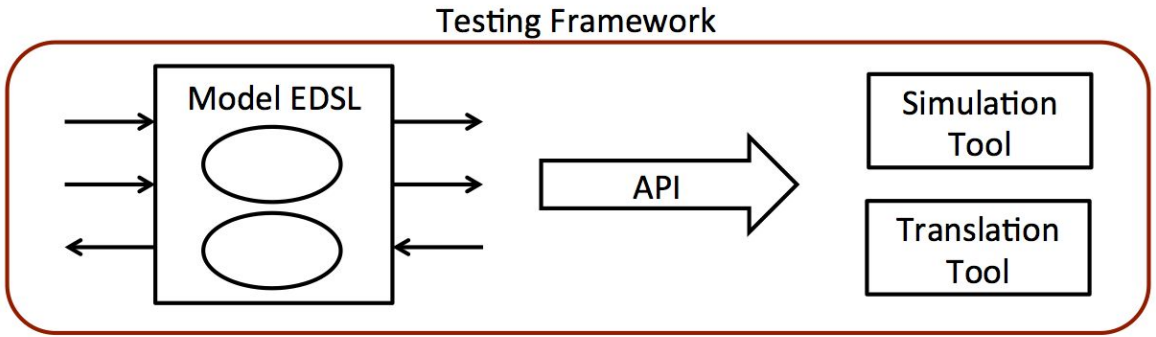
# PyMTL Components

- A Python EDSL for concurrent-structural hardware modeling
- A Python API for analyzing models described in the PyMTL EDSL
- A Python tool for simulating PyMTL FL, CL, and RTL models
- A Python tool for translating PyMTL RTL models into Verilog

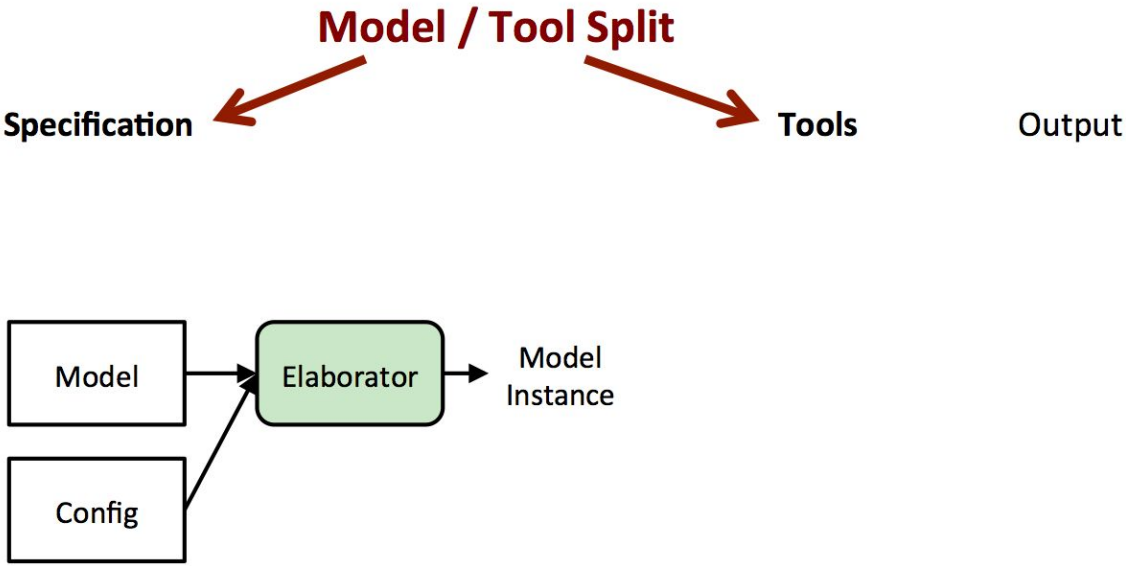


# PyMTL Components

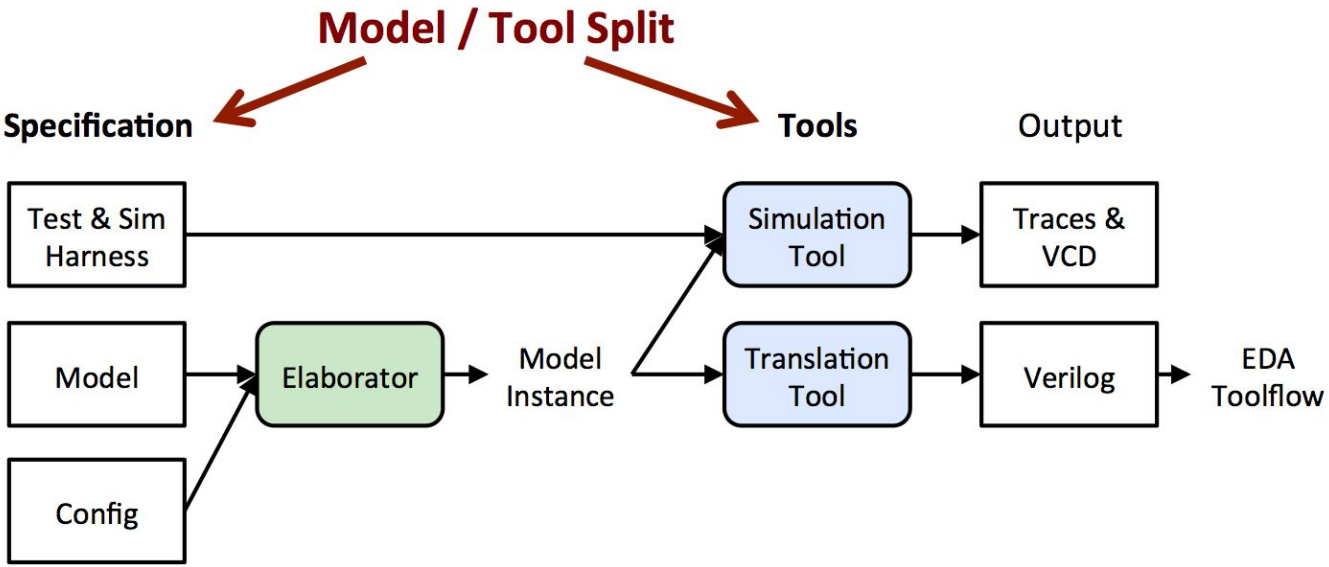
- A Python EDSL for concurrent-structural hardware modeling
- A Python API for analyzing models described in the PyMTL EDSL
- A Python tool for simulating PyMTL FL, CL, and RTL models
- A Python tool for translating PyMTL RTL models into Verilog
- A Python testing framework for model validation



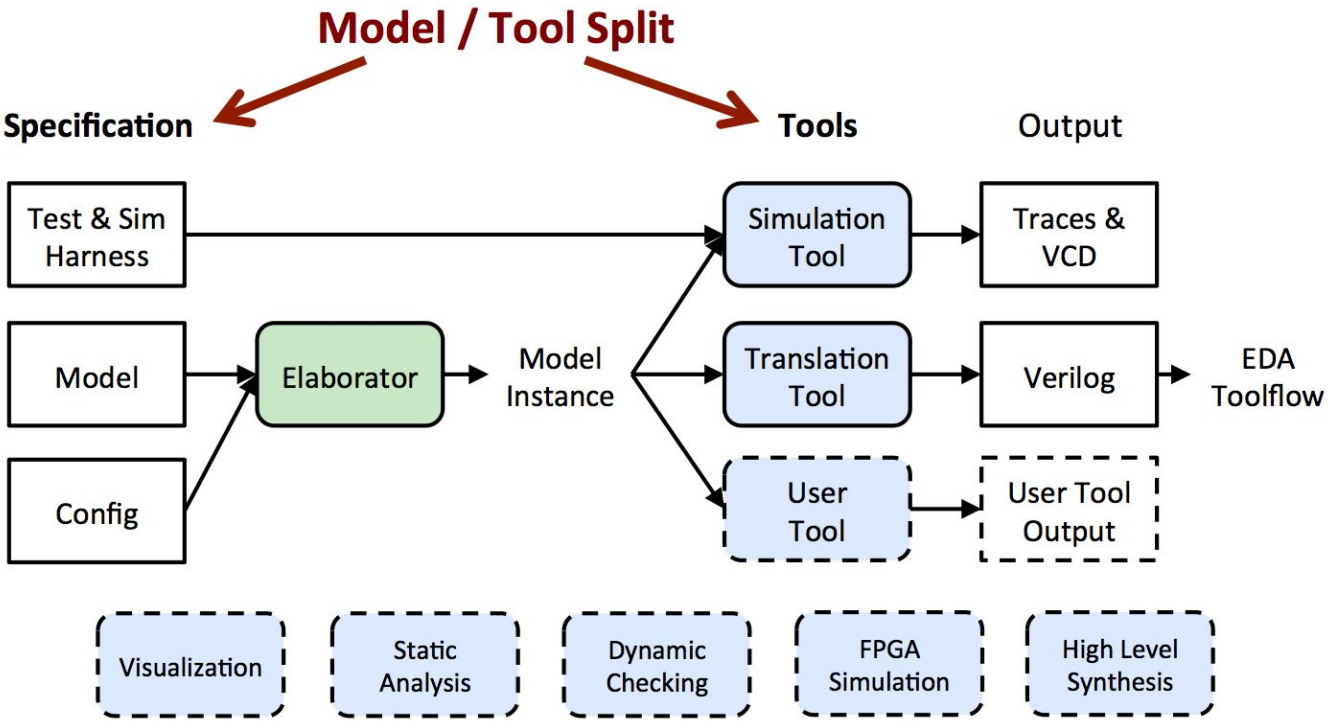
# PyMTL Architecture



# PyMTL Architecture

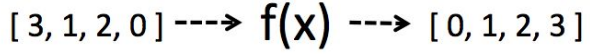


# PyMTL Architecture



# PyMTL Embedded-DSL: FL Modeling

```
def sorter_network( input_list ):
  return sorted( input_list )
```

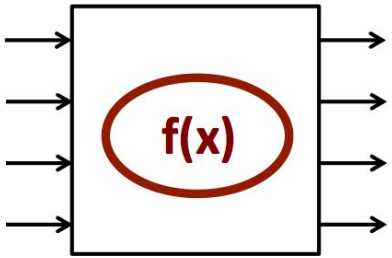


# PyMTL Embedded-DSL: FL Modeling

```
def sorter_network( input_list ):
  return sorted( input_list )
```

[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

```
class SorterNetworkFL( Model ):
  def __init__( s, nbits, nports ):

    s.in_  = InPort [nports]( nbits )
    s.out  = OutPort[nports]( nbits )


    @s.tick_fl
    def logic():
      for i, v in enumerate( sorted( s.in_ ) ):
        s.out[i].next = v
```

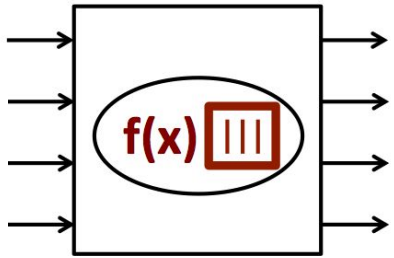


# PyMTL Embedded-DSL: CL Modeling

```
def sorter_network( input_list ):
  return sorted( input_list )
```

[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

```
class SorterNetworkCL( Model ):
  def __init__( s, nbits, nports ):

    s.in_  = InPort [nports]( nbits )
    s.out  = OutPort[nports]( nbits )


    @s.tick_cl
    def logic():
      # behavioral logic + timing delays
```

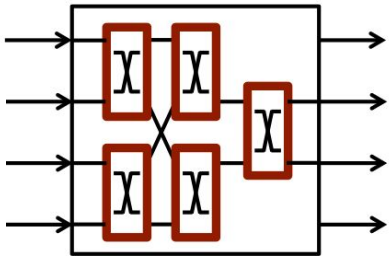


# PyMTL Embedded-DSL: RTL Implementation

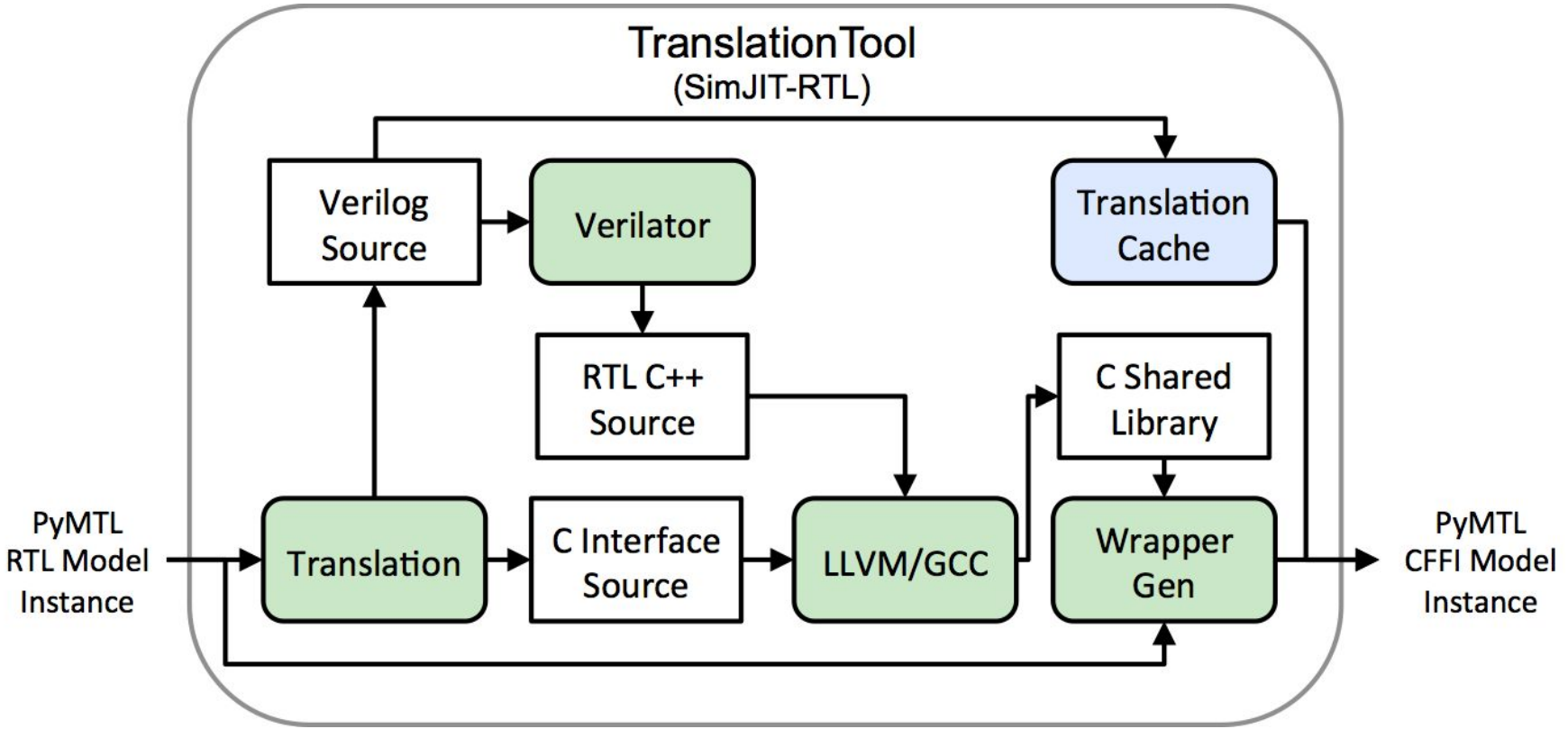
```
def sorter_network( input_list ):
  return sorted( input_list )
```

[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

```
class SorterNetworkRTL( Model ):
  def __init__( s, nbits, nports ):

    s.in_  = InPort [nports]( nbits )
    s.out  = OutPort[nports]( nbits )


    @s.tick_rtl
    def seq_logic():
      # sequential logic

    @s.combinational
    def comb_logic():
      # combinational logic
```

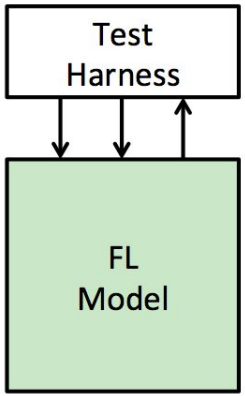


# PyMTL Tool: Verilog Generator



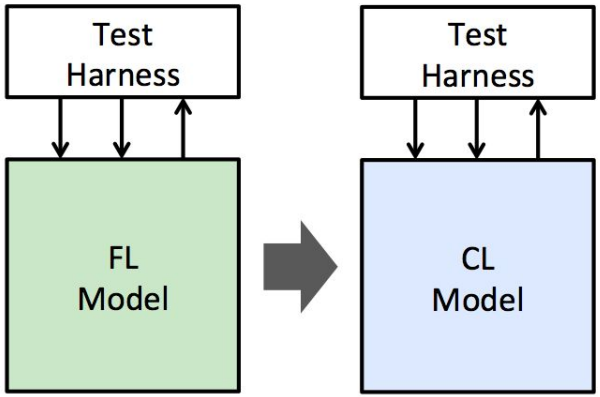
# PyMTL Capabilities

- Incremental refinement from algorithm to accelerator implementation



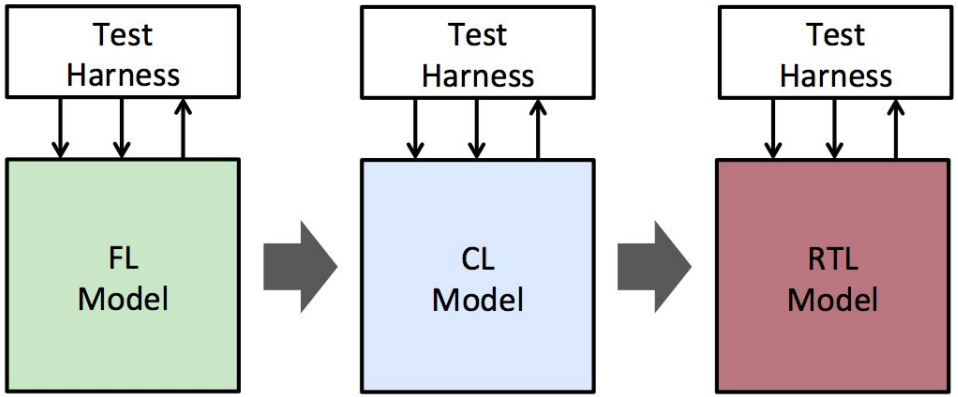
# PyMTL Capabilities

- Incremental refinement from algorithm to accelerator implementation



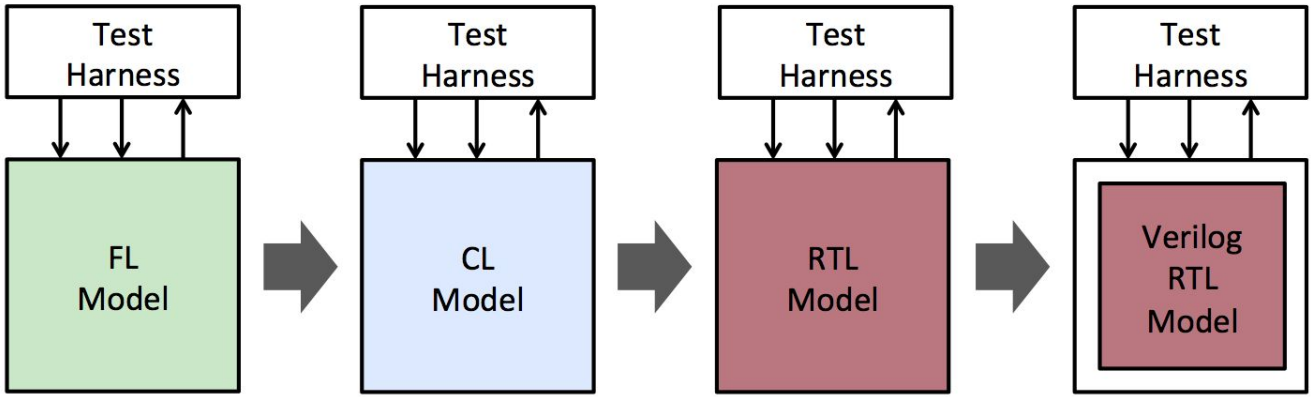
# PyMTL Capabilities

- Incremental refinement from algorithm to accelerator implementation



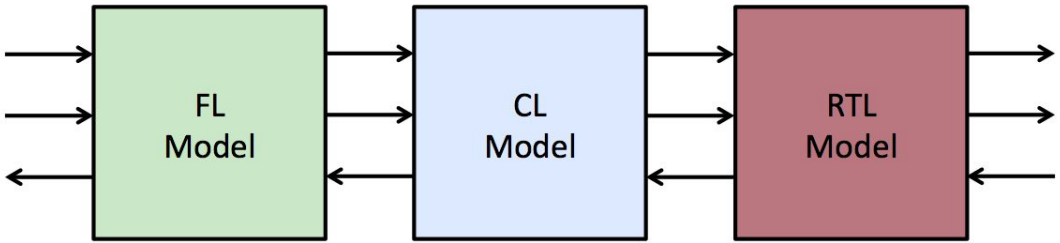
# PyMTL Capabilities

- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog



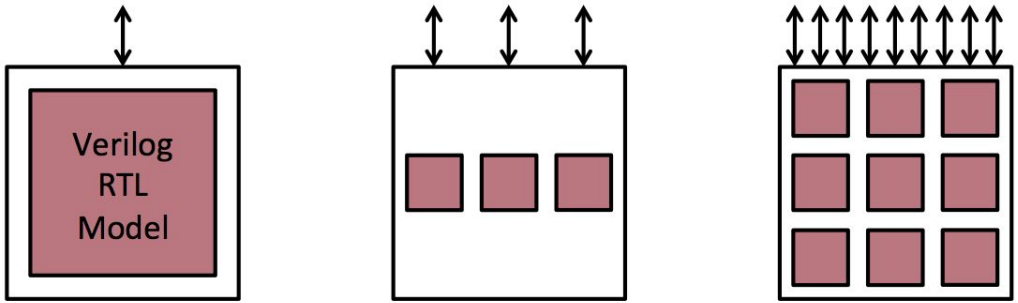
# PyMTL Capabilities

- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
- Multi-level co-simulation of FL, CL, and RTL models



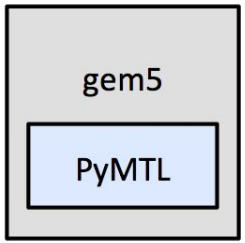
# PyMTL Capabilities

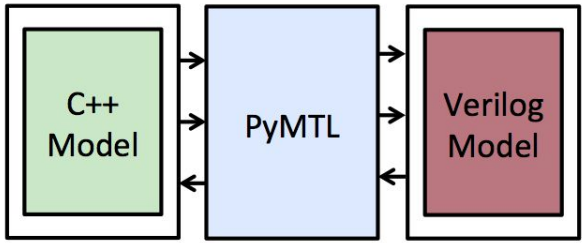
- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
- Multi-level co-simulation of FL, CL, and RTL models
- Construction of highly-parameterized RTL chip generators



# PyMTL Capabilities

- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
- Multi-level co-simulation of FL, CL, and RTL models
- Construction of highly-parameterized RTL chip generators
- Embedding within C++ frameworks & integration of C++/Verilog models



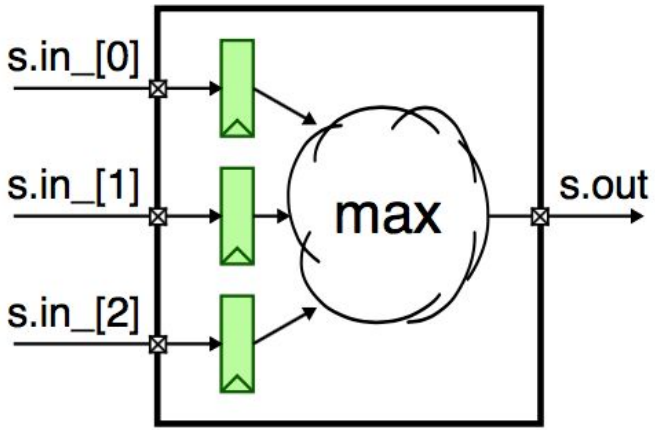



# Part 1: The View from Academia

## Features of PyMTL

# PyMTL Features: Interpretive Execution

```
% python

>>> from pymtl import *
>>> from MaxUnitFL import MaxUnitFL
>>> model = MaxUnitFL( nbits=8, nports=3 )
>>> model.elaborate()
>>> sim = SimulationTool(model)
>>> sim.reset()
>>> model.in_[0].value = 2
>>> model.in_[1].value = 5
>>> model.in_[2].value = 3
>>> sim.cycle()
>>> model.out
Bits( 8, 0x05 )
```

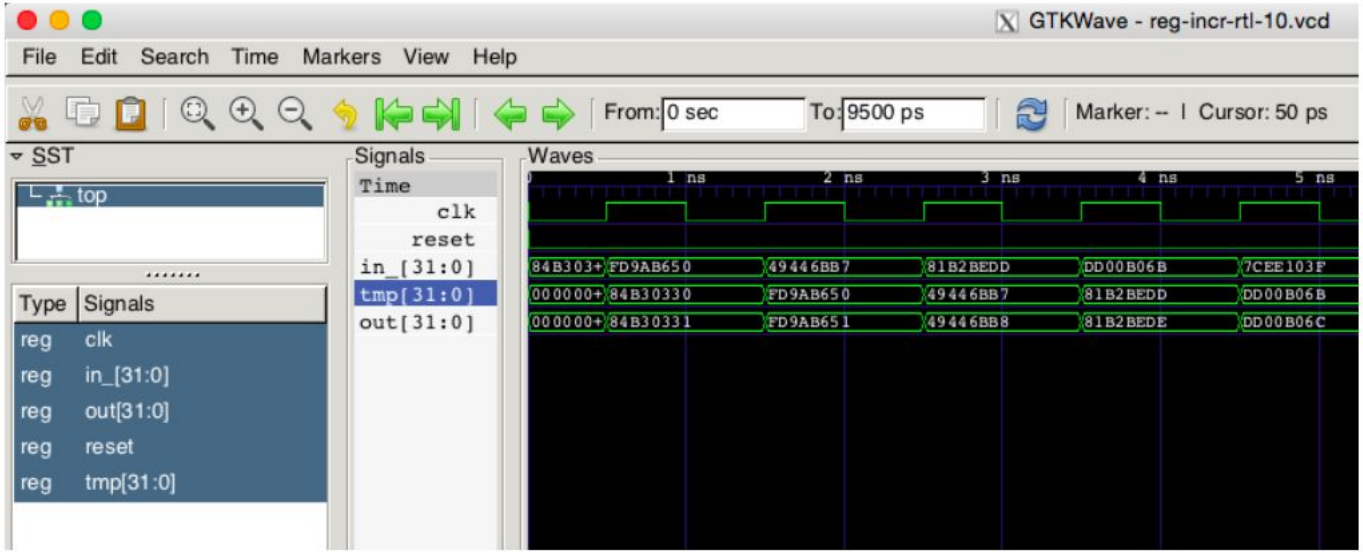


# PyMTL Features: Line Tracing

```
% python ../regincr/reg-incr-sim.py 20 --trace

  0: 04e5f14d (00000000) 00000000
  1: 7839d4fc (04e5f14d) 04e5f14e
  2: 996ab63d (7839d4fc) 7839d4fd
  3: 6d146dfc (996ab63d) 996ab63e
  4: 9cb87fec (6d146dfc) 6d146dfd
  5: ba43a338 (9cb87fec) 9cb87fed
  6: a0c394ff (ba43a338) ba43a339
  7: f72041ee (a0c394ff) a0c39500
  ...
```

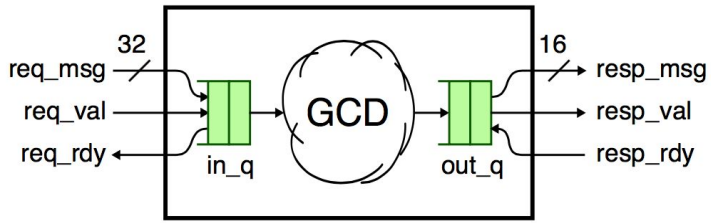
# PyMTL Features: VCD Dumping

```
% python ../regincr/reg-incr-sim.py 10 --dump-vcd
% gtkwave ./reg-incr-rtl-10.vcd
```



# PyMTL Features: Port Bundles

```
s.req   = InValRdyBundle ( dtype )
s.resp  = OutValRdyBundle( dtype )

s.child = ChildModel( dtype )

# connecting bundled request ports individually

s.connect( s.req.msg, s.child.req.msg )
s.connect( s.req.val, s.child.req.val )
s.connect( s.req.rdy, s.child.req.rdy )

# connecting bundled response ports in bulk

s.connect( s.resp,    s.child.resp )
```



# PyMTL Features: Bitstructs

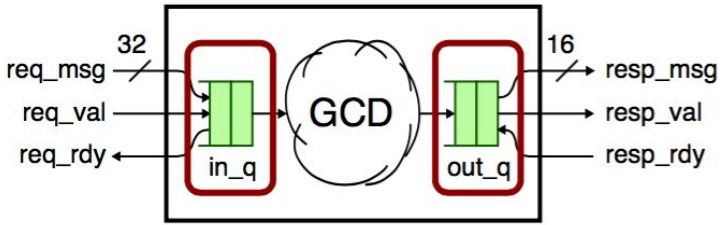
```
# MemReqMsg(addr_nbits, data_nbits) is a BitStruct datatype:
# +------+----------+------+----------+
# | type | addr     | len  | data     |
# +------+----------+------+----------+
dtype = MemReqMsg( 32, 32 )
s.in_ = InPort( dtype )

@s.tick
def logic():

  # BitStructs are subclasses of Bits, we can slice them
  addr, data = s.in_[34:66], s.in_[0:32]

  # ... but it's usually more convenient to use fields!
  addr, data = s.in_.addr, s.in_.data
```

# PyMTL Features: Interface Proxies

Wrap PortBundles with queue-like interfaces, greatly simplifying interaction with latency insensitive interfaces.

```
# Interface

s.req    = InValRdyBundle  ( GcdUnitReqMsg() )
s.resp   = OutValRdyBundle ( Bits(16)        )

# Adapters

s.req_q  = InValRdyQueueAdapter  ( s.req  )
s.resp_q = OutValRdyQueueAdapter ( s.resp )
```

```
@s.tick_fl
def block():

  # Use adapter to pop value from request queue
  req_msg = s.req_q.popleft()

  # Use gcd function from Python's standard library
  result = gcd( req_msg.a, req_msg.b )

  # Use adapter to append result to response queue
  s.resp_q.append( result )
```



# PyMTL Features: Pausable Continuations

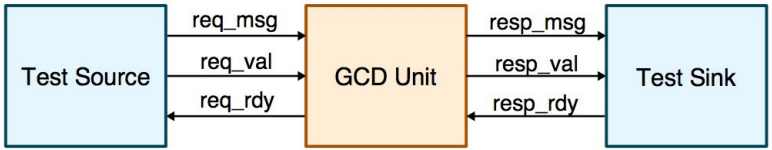
Greenlets (python coroutines) allows function calls to suspect execution of a particular module. This module will “pause” until another module (e.g. memory) responds to its request.

```
def __init__( s, mem_ifc_types, cpu_ifc_types ):
  s.cpu_ifc = ChildReqRespBundle ( cpu_ifc_types )
  s.mem_ifc = ParentReqRespBundle( mem_ifc_types )

  s.cpu  = ChildReqRespQueueAdapter( s.cpu_ifc )
  s.src0 = ListMemPortAdapter      ( s.mem_ifc )
  s.src1 = ListMemPortAdapter      ( s.mem_ifc )
```

```
@s.tick_fl
def logic():
  s.cpu.xtick()
  if not s.cpu.req_q.empty() and not s.cpu.resp_q.full():
    req = s.cpu.get_req()


    elif req.ctrl_msg == 0:
      result = numpy.dot( s.src0, s.src1 )
      s.cpu.push_resp( result )
```

# PyMTL Features: Latency Insensitive Testers

```
22  class TestHarness (Model):
23
24    def __init__( s, src_msgs, sink_msgs ):
25
26      s.src  = TestSource (GcdUnitReqMsg(), src_msgs)
27      s.gcd  = GcdUnitFL  ()
28      s.sink = TestSink   (Bits(16), sink_msgs)
29
30      s.connect( s.src.out,  s.gcd.req  )
31      s.connect( s.gcd.resp, s.sink.in_ )
```




```
% py.test ../gcd/GcdUnitFL_simple_test.py -vs
```

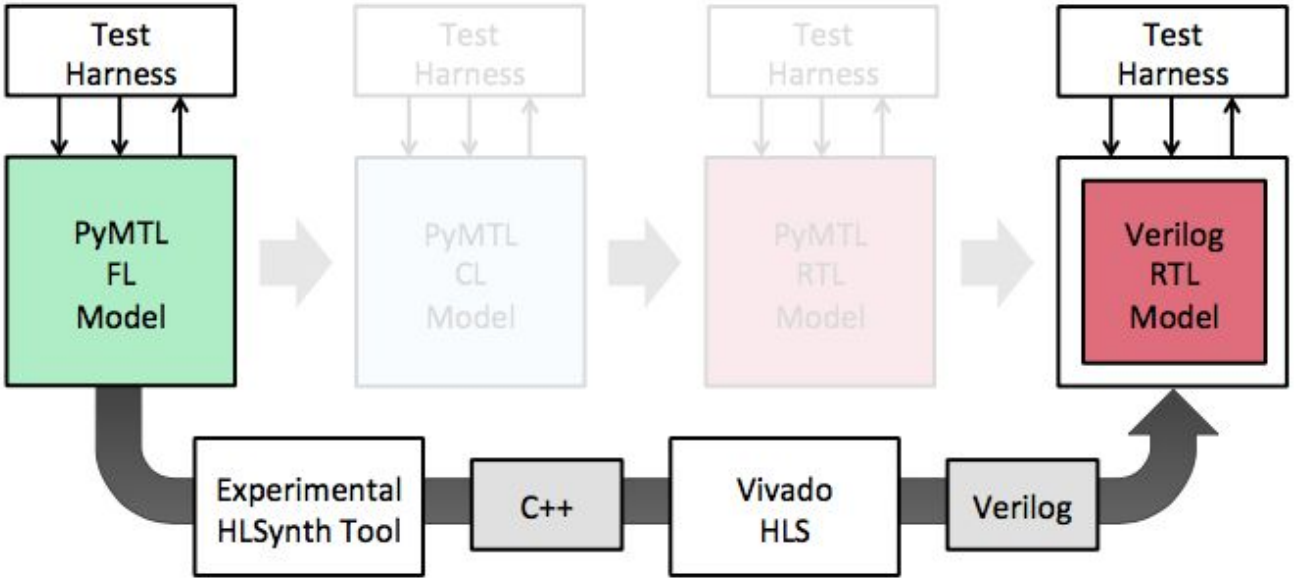
# PyMTL Experiments: Physical Placement

Approach: annotate PyMTL models with methods containing procedural logic for physical layout.

```
class RingNetwork( Model ):
  ...
  def create_layout( s ):
    max = len( s.routers )
    for i, r in enumerate( s.routers ):
      if i < (max / 2):
        r.dim.x = i * ( r.dim.w + s.link_len )
        r.dim.y = 0
      else:
        r.dim.x = (max - i - 1) * ( r.dim.w + s.link_len )
        r.dim.y = r.dim.h + s.link_len

    s.dim.w = max/2 * r.dim.w + (max/2 - 1) * s.link_len
    s.dim.h = 2 * r.dim.h + s.link_len
```

# PyMTL Experiments: High-Level Synthesis

Approach: generate Vivado HLS compatible C++ from PyMTL FL. Vivado generated RTL is wrapped back into PyMTL for testing and cosimulation.



# PyMTL Experiments: High-Level Synthesis

```
class GcdXcelFL( Model ):
  def __init__( s ):
    s.xcelreq  = InValRdyBundle ( XcelReqMsg()  )
    s.xcelresp = OutValRdyBundle( XcelRespMsg() )

    s.req_queue  = InQueuePortProxy ( s.xcelreq  )
    s.resp_queue = OutQueuePortProxy( s.xcelresp )

    @s.tick_fl
    def PyGcdXcelHLS():
      # ... receive input values

      # receive message containing go signal, send ack
      req  = s.req_queue.popleft()
      resp = XcelRespMsg() \
             .mk_msg( req.opaque, req.type_, 0, req.id )
      s.resp_queue.append( resp )

      result   = gcd( a, b )

      # ... respond with result
```

```
# 116 Cycles!
def gcd( a, b ):
  while b != 0:
    if a < b:
      # a, b = b, a
      t = b; b = a; a = t
    a = a - b
  return a

# 668 Cycles!
def gcd( a, b ):
  while b != 0:
    # a, b = b, a % b
    t = b; b = a % b; a = t
  return a

# 63 Cycles! Winner!
def gcd( a, b ):
  while a != b:
    if ( a > b ): a = a - b
    else:         b = b - a
  return a
```

# Part 1: The View from Academia

## Insights from PyMTL

# Insights

- Python was the right language choice. (For us).
  - Huge productivity boosts for research and building infrastructure!
  - In academia, velocity is more important than safety.
  - Was able to rapidly implement with weird infrastructure ideas (proxies, continuations, HLS).

# Insights

- Python was the right language choice. (For us).
  - Huge productivity boosts for research and building infrastructure!
  - In academia, velocity is more important than safety.
  - Was able to rapidly implement with weird infrastructure ideas (proxies, continuations, HLS).
- Debugging with line tracing is almost always faster than VCD.
  - Run/test/debug loop is much more instantaneous.
  - Can navigate and grep for transactions much more easily.

# Insights

- Python was the right language choice. (For us).
  - Huge productivity boosts for research and building infrastructure!
  - In academia, velocity is more important than safety.
  - Was able to rapidly implement with weird infrastructure ideas (proxies, continuations, HLS).
- Debugging with line tracing is almost always faster than VCD.
  - Run/test/debug loop is much more instantaneous.
  - Can navigate and grep for transactions much more easily.
- Using the same language for design, test, and tools boosted productivity.
  - Far less mental context switching.
  - Interfacing various tools and models became much simpler.

# Insights

- Python was the right language choice. (For us).
  - Huge productivity boosts for research and building infrastructure!
  - In academia, velocity is more important than safety.
  - Was able to rapidly implement with weird infrastructure ideas (proxies, continuations, HLS).
- Debugging with line tracing is almost always faster than VCD.
  - Run/test/debug loop is much more instantaneous.
  - Can navigate and grep for transactions much more easily.
- Using the same language for design, test, and tools boosted productivity.
  - Far less mental context switching.
  - Interfacing various tools and models became much simpler.
- Build abstractions up from the right primitives.
  - You can’t escape interfacing with Verilog, so there must be primitives to match.
  - BlueSpec builds down rather from up, so interfacing with Verilog is painful.

# Insights

- Host language simulation was a killer feature.
  - Python debug environment is amazing thanks to third party tools (py.test, etc.)
  - Could verify functionality before worrying about Verilog generation bugs.

# Insights

- Host language simulation was a killer feature.
  - Python debug environment is amazing thanks to third party tools (py.test, etc.)
  - Could verify functionality before worrying about Verilog generation bugs.
- HDL integration and cosimulation was a killer feature.
  - Allowed test artifacts built for software to be reused for verification, out of the box.
  - Seemless development experience switching from Python to Verilog simulation.

# Insights

- Host language simulation was a killer feature.
  - Python debug environment is amazing thanks to third party tools (py.test, etc.)
  - Could verify functionality before worrying about Verilog generation bugs.
- HDL integration and cosimulation was a killer feature.
  - Allowed test artifacts built for software to be reused for verification, out of the box.
  - Seemless development experience switching from Python to Verilog simulation.
- Testing infrastructure was a killer feature.
  - The use of py.test was a huge win, greatly simplified execution of large test suites.

# Insights

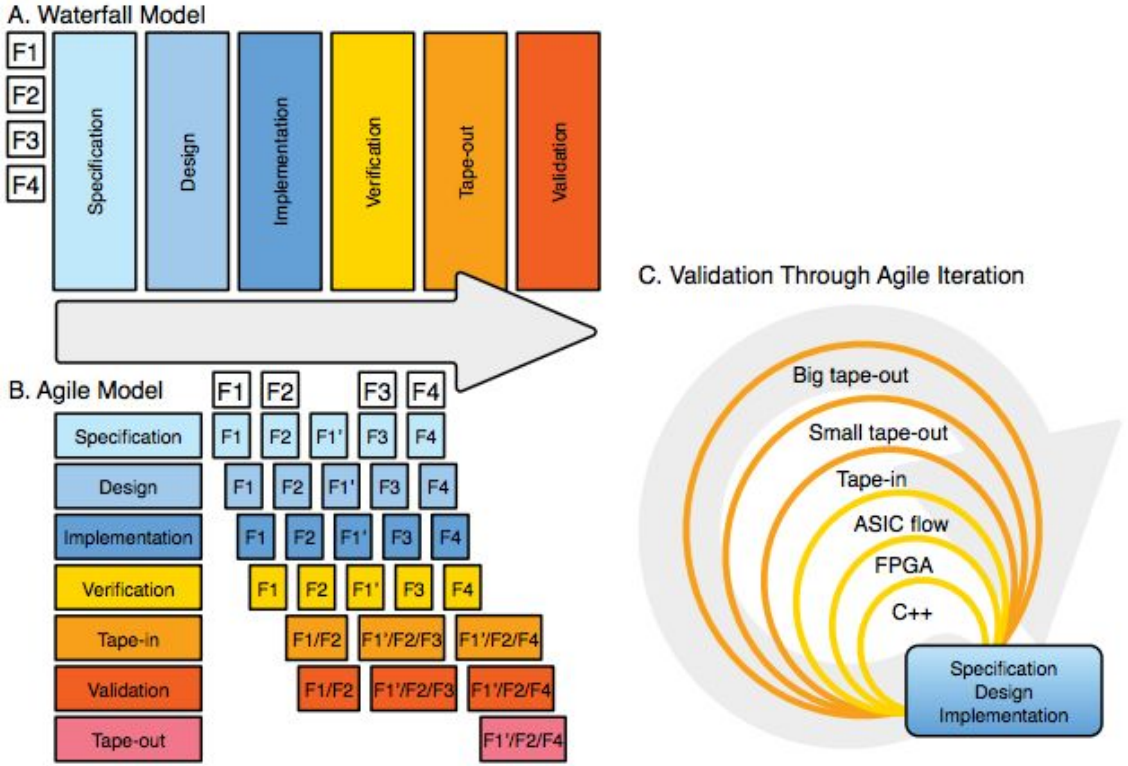
- Host language simulation was a killer feature.
  - Python debug environment is amazing thanks to third party tools (py.test, etc.)
  - Could verify functionality before worrying about Verilog generation bugs.
- HDL integration and cosimulation was a killer feature.
  - Allowed test artifacts built for software to be reused for verification, out of the box.
  - Seemless development experience switching from Python to Verilog simulation.
- Testing infrastructure was a killer feature.
  - The use of py.test was a huge win, greatly simplified execution of large test suites.
- Aggressively eliminate boilerplate.
  - The less you copy-paste-modify, the fewer chances for mismatches.

# Insights

- Host language simulation was a killer feature.
  - Python debug environment is amazing thanks to third party tools (py.test, etc.)
  - Could verify functionality before worrying about Verilog generation bugs.
- HDL integration and cosimulation was a killer feature.
  - Allowed test artifacts built for software to be reused for verification, out of the box.
  - Seemless development experience switching from Python to Verilog simulation.
- Testing infrastructure was a killer feature.
  - The use of py.test was a huge win, greatly simplified execution of large test suites.
- Aggressively eliminate boilerplate.
  - The less you copy-paste-modify, the fewer chances for mismatches.
- Doing development in a general-purpose language is a lot more enjoyable.

# Part 2: The View from Industry

# Industry Design Process



# Industry Design Process

- Software/Firmware/Compiler
- Golden Reference Model
- Performance Model
- Top-Level, IP Integration, IP Versioning
- RTL Implementation
- RTL Unit Testing
- Design Verification (UVM)
- Formal Property Verification
- Gate-Level Simulation
- Physical Design
- Emulation
- FPGA

# Insights from Industry

- Design entry is an overstated problem.
  - Plenty of resources and expertise, once architecture set can move rather quickly.
  - The long-pole is DV, and sometimes PD.

# Insights from Industry

- Design entry is an overstated problem.
  - Plenty of resources and expertise, once architecture set can move rather quickly.
  - The long-pole is DV, and sometimes PD.
- Once you parameterize your logic, you need to parameterize your tests too!
  - Parameterized RTL does you no good if you can’t verify all supported parameterizations.

# Insights from Industry

- Design entry is an overstated problem.
  - Plenty of resources and expertise, once architecture set can move rather quickly.
  - The long-pole is DV, and sometimes PD.
- Once you parameterize your logic, you need to parameterize your tests too!
  - Parameterized RTL does you no good if you can’t verify all supported parameterizations.
- Physical design needs stability.
  - Single-line changes in source resulting in huge changes in Verilog is an absolute nightmare.
  - HLS causes huge problems for area estimation and floorplanning.

# Insights from Industry

- Design entry is an overstated problem.
  - Plenty of resources and expertise, once architecture set can move rather quickly.
  - The long-pole is DV, and sometimes PD.
- Once you parameterize your logic, you need to parameterize your tests too!
  - Parameterized RTL does you no good if you can’t verify all supported parameterizations.
- Physical design needs stability.
  - Single-line changes in source resulting in huge changes in Verilog is an absolute nightmare.
  - HLS causes huge problems for area estimation and floorplanning.
- It is critical that generated RTL be readable.
  - Often not a problem for RTL debug unless there’s a bug in the tools.
  - Huge problem for Formal Property Verification.

# Insights from Industry

- Design entry is an overstated problem.
  - Plenty of resources and expertise, once architecture set can move rather quickly.
  - The long-pole is DV, and sometimes PD.
- Once you parameterize your logic, you need to parameterize your tests too!
  - Parameterized RTL does you no good if you can’t verify all supported parameterizations.
- Physical design needs stability.
  - Single-line changes in source resulting in huge changes in Verilog is an absolute nightmare.
  - HLS causes huge problems for area estimation and floorplanning.
- It is critical that generated RTL be readable.
  - Often not a problem for RTL debug unless there’s a bug in the tools.
  - Huge problem for Formal Property Verification.
- Generated code should always be X-free.
  - Debugging 2-state verse 4-state simulation mismatches is a huge pain.

# Insights from Industry

- Configuration of tools should be driven top-down from a single specification.
  - Maintaining multiple tops for DV, emulation, FPGA, gate-level is a nightmare.
  - Hardware parameters exposed to the ISA must match compiler parameters!

# Insights from Industry

- Configuration of tools should be driven top-down from a single specification.
  - Maintaining multiple tops for DV, emulation, FPGA, gate-level is a nightmare.
  - Hardware parameters exposed to the ISA must match compiler parameters!
- Safety wins over conciseness.
  - Inferred bit widths and automatic truncation behaviors are too surprising and bug prone.

# Insights from Industry

- Configuration of tools should be driven top-down from a single specification.
  - Maintaining multiple tops for DV, emulation, FPGA, gate-level is a nightmare.
  - Hardware parameters exposed to the ISA must match compiler parameters!
- Safety wins over conciseness.
  - Inferred bit widths and automatic truncation behaviors are too surprising and bug prone.
- Engineers would rather have boilerplate than magic.
  - If the automation implementation is too complex, it will be rejected.

# Insights from Industry

- Configuration of tools should be driven top-down from a single specification.
  - Maintaining multiple tops for DV, emulation, FPGA, gate-level is a nightmare.
  - Hardware parameters exposed to the ISA must match compiler parameters!
- Safety wins over conciseness.
  - Inferred bit widths and automatic truncation behaviors are too surprising and bug prone.
- Engineers would rather have boilerplate than magic.
  - If the automation implementation is too complex, it will be rejected.
- Builder APIs are strongly preferred over embedded-DSLs.
  - Maintainability and comprehensibility of the backend are critical.

# Insights from Industry

- Configuration of tools should be driven top-down from a single specification.
  - Maintaining multiple tops for DV, emulation, FPGA, gate-level is a nightmare.
  - Hardware parameters exposed to the ISA must match compiler parameters!
- Safety wins over conciseness.
  - Inferred bit widths and automatic truncation behaviors are too surprising and bug prone.
- Engineers would rather have boilerplate than magic.
  - If the automation implementation is too complex, it will be rejected.
- Builder APIs are strongly preferred over embedded-DSLs.
  - Maintainability and comprehensibility of the backend are critical.
- Expose a tool API, not a compiler IR.
  - The real wins come from democratizing tool design: let RTL engineers empower themselves.
  - This is only possible if graph analysis and transformation is simple and accessible.

# Conclusion

- PyMTL was built from the perspective of academia, and was explicitly designed to meet the needs of academics.
- Many of the insights learned from building PyMTL apply directly to industry design flows… but many do not!
- The constraints of industry EDA tools are very strong, you can’t ignore them.
- A successful design methodology must be comprehensible and accessible to both hardware **and** software engineers.

# Backup Slides

# References

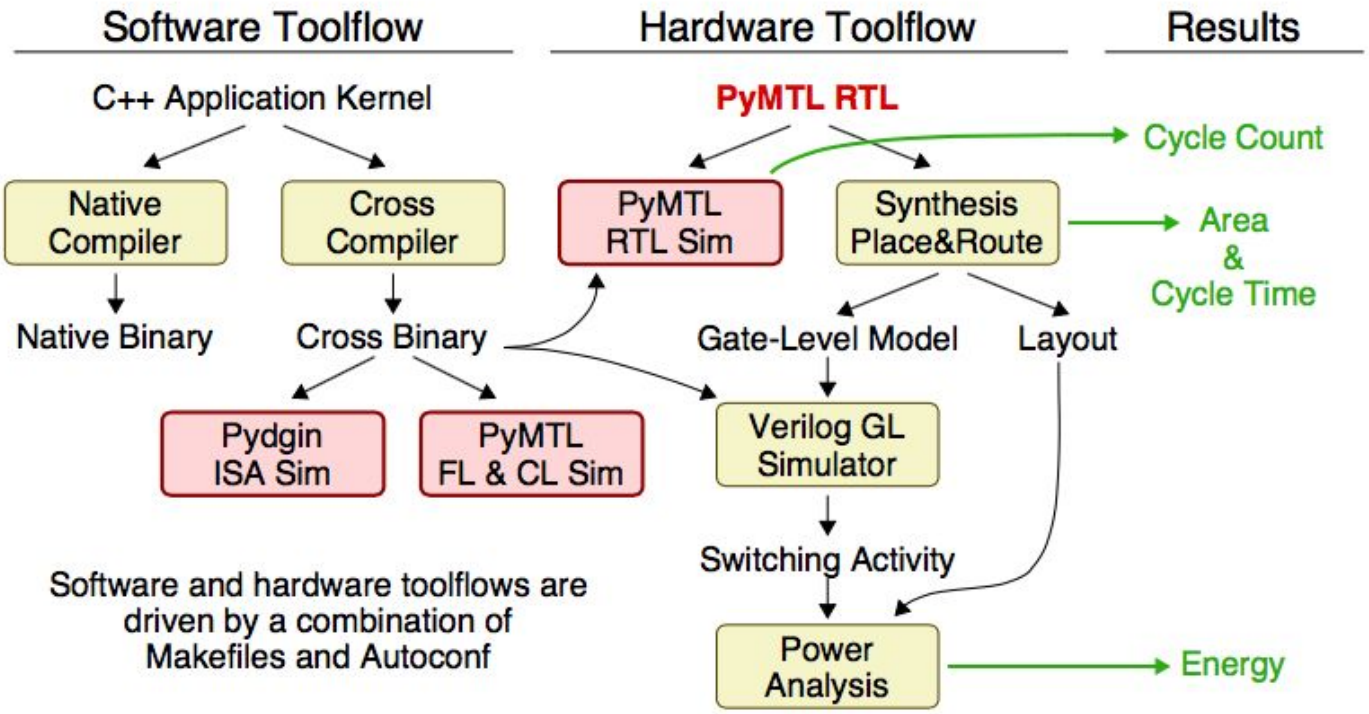

- B. Catanzaro et al. “SEJITS: Getting Productivity And Performance With Selective Embedded JIT Specialization” Proc. 2009 Workshop on Programming Models for Emerging Architectures (PMEA 2009).
- O. Shacham et al., "Avoiding game over: Bringing design to the next level," DAC Design Automation Conference 2012, San Francisco, CA, 2012, pp. 623-629.
- O. Shacham, O. Azizi, M. Wachs, S. Richardson and M. Horowitz, "Rethinking Digital Design: Why Design Must Change," in IEEE Micro, vol. 30, no. 6, pp. 9-24, Nov.-Dec. 2010.
- Y. Lee et al., "An Agile Approach to Building RISC-V Microprocessors," in IEEE Micro, vol. 36, no. 2, pp. 8-20, Mar.-Apr. 2016.

# Why Python

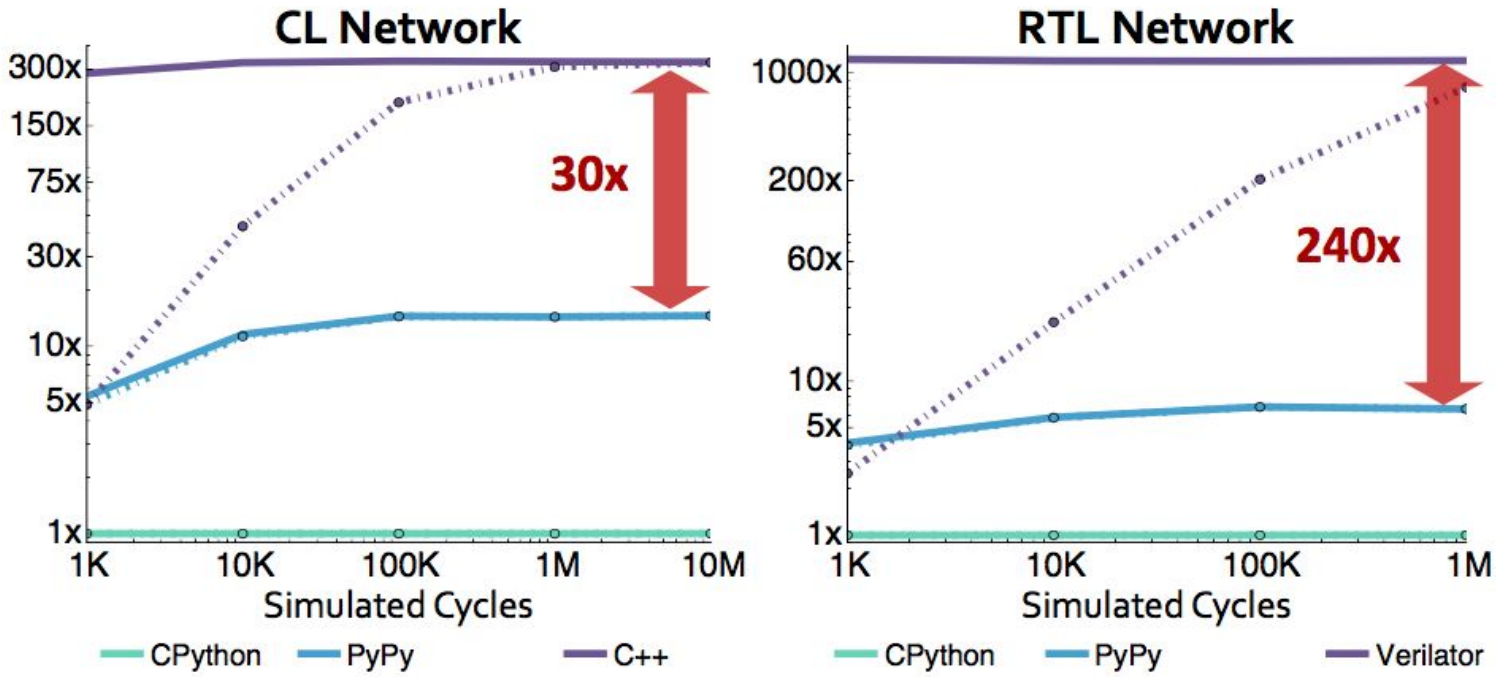
- Python is well regarded as a highly productive language with lightweight, pseudocode-like syntax
- Python supports modern language features to enable rapid, agile development (dynamic typing, reflection, metaprogramming)

- Python has a large and active developer and support community
- Python includes extensive standard and third-party libraries
- Python enables embedded domain-specific languages
- Python facilitates engaging application-level researchers
- Python includes built-in support for integrating with C/C++
- Python performance is improving with advanced JIT compilation

# Vertically Integrated Design Methodology



# SimJIT Performance



# SimJIT Performance