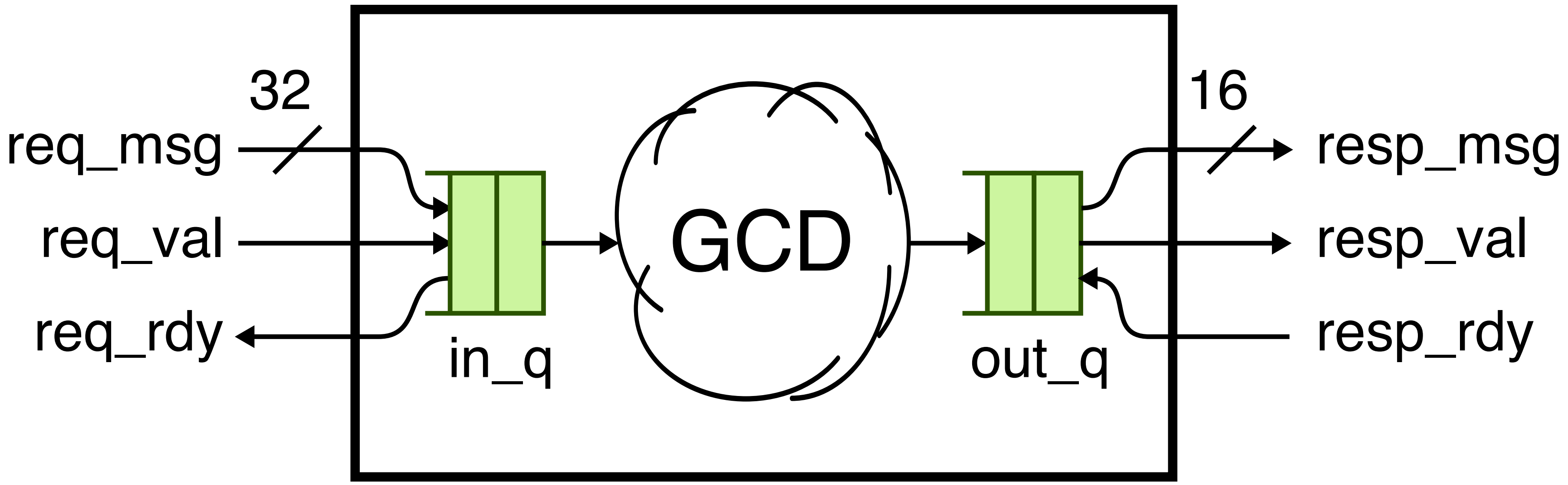
PyMTL/Pydgin Tutorial Schedule

8:30am	– 8:50am	Virtual Machine Installation and Setup
8:50am	– 9:00am	*Presentation:* PyMTL/Pydgin Tutorial Overview
9:00am	– 9:10am	*Presentation:* Introduction to Pydgin
9:10am	– 10:00am	*Hands-On:* Adding a GCD Instruction using Pydgin
10:00am	– 10:10am	*Presentation:* Introduction to PyMTL
10:10am	– 11:00am	*Hands-On:* PyMTL Basics with Max/RegIncr
11:00am	– 11:30am	Coffee Break
11:30am	– 11:40am	*Presentation:* Multi-Level Modeling with PyMTL
11:40am	– 12:30pm	*Hands-On:* FL, CL, RTL Modeling of a GCD Unit

PyMTL 102: The GCD Unit

- Computes the greatest-common divisor of two numbers.
- Uses a latency insensitive input protocol to accept messages only when sender has data available and GCD unit is ready.
- Uses a latency insensitive output protocol to send results only when result is done and receiver is ready.



PyMTL 102: Bundled Interfaces

- **PortBundles** are used to simplify the handling of multi-signal interfaces, such as ValRdy:

```
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 )
```

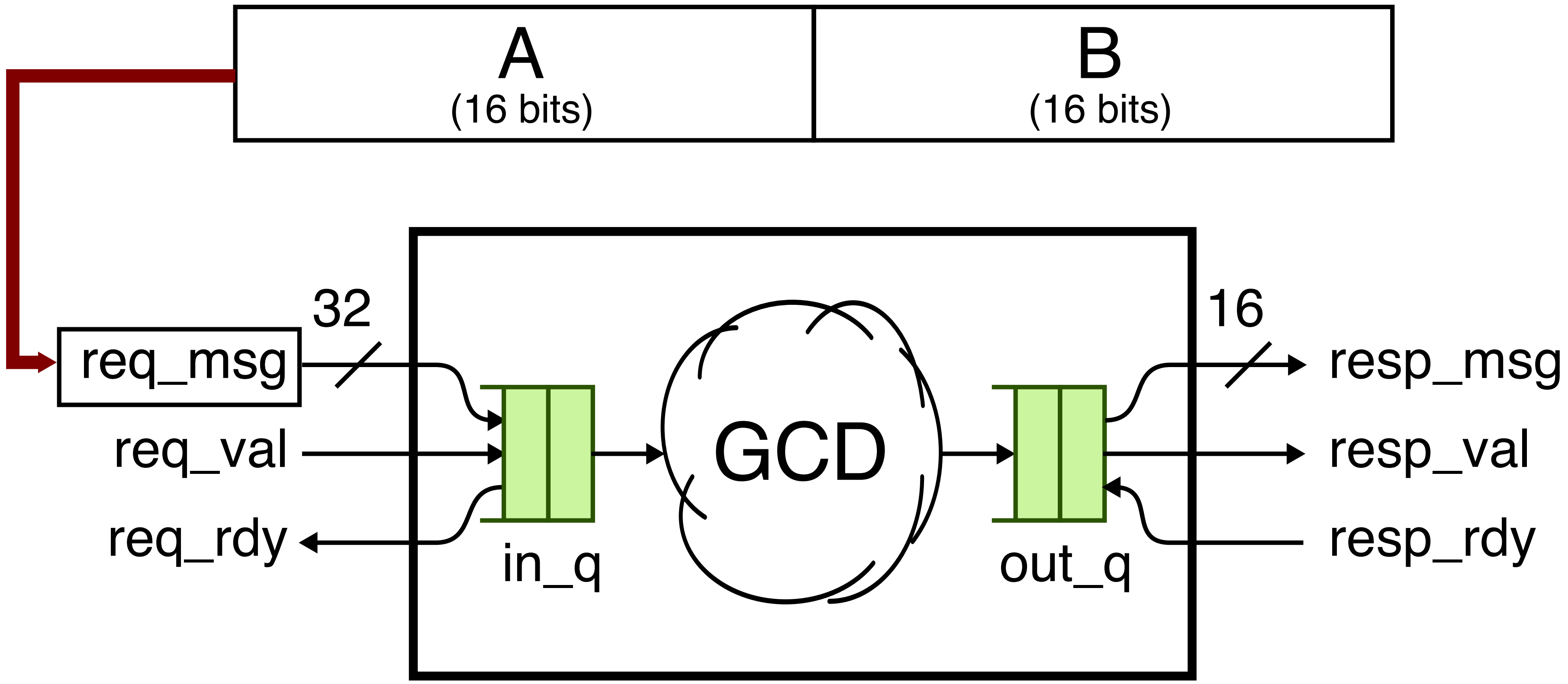
PyMTL 102: Complex Datatypes

- **BitStructs** are used to simplify communicating and interacting with complex packages of data:

```
# 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 102: Complex Datatypes

The GCD request message can be implemented as a BitStruct that has two fields, one for each operand:



Hands-On: FL, CL, RTL Modeling of a GCD Unit

- Task 3.1: Create a BitStruct for the GCD request
- Task 3.2: Build an FL model for the GCD unit
- Task 3.3: Create a latency insensitive test
- Task 3.4: Add timing to the GCD CL model
- Task 3.5: Fix the bug in the GCD RTL model
- Task 3.6: Verify generated Verilog GCD RTL
- Task 3.7: Experiment with the GCD simulator

Hands-On: FL, CL, RTL Modeling of a GCD Unit

- **Task 3.1: Create a BitStruct for the GCD request**
- Task 3.2: Build an FL model for the GCD unit
- Task 3.3: Create a latency insensitive test
- Task 3.4: Add timing to the GCD CL model
- Task 3.5: Fix the bug in the GCD RTL model
- Task 3.6: Verify generated Verilog GCD RTL
- Task 3.7: Experiment with the GCD simulator

★ Task 3.1: Create a BitStruct for the GCD request ★

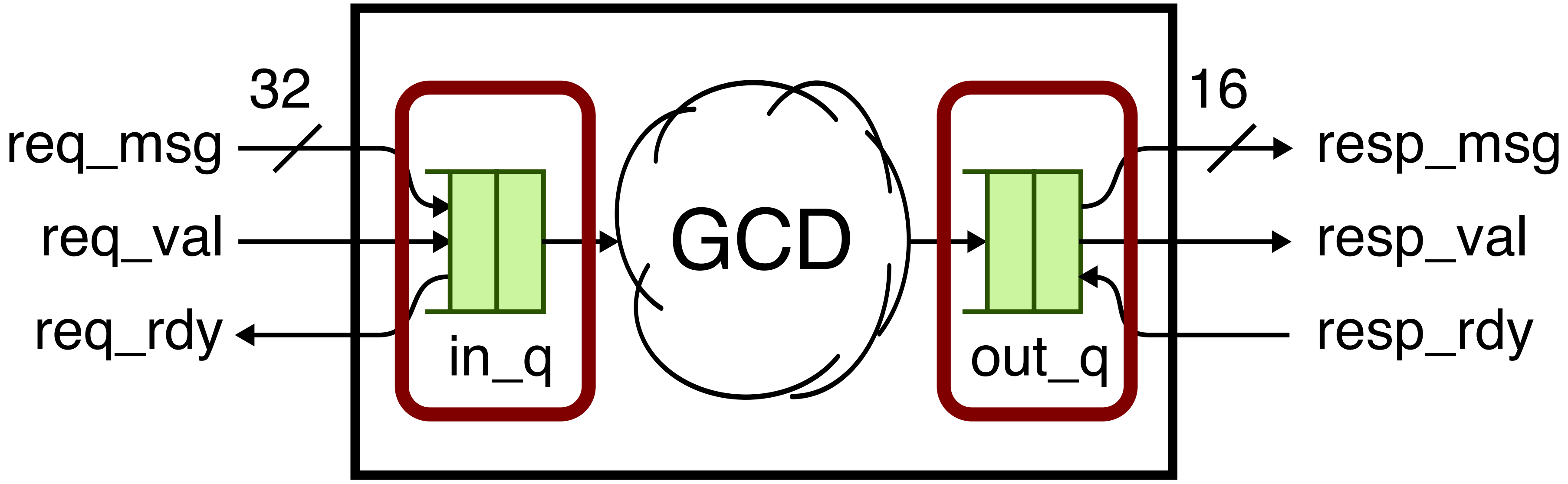
```
% cd ~/pymtl-tut/build
% gedit ../gcd/GcdUnitMsg.py
```

```
#-------------------------------------------------------------
# TASK 3.1: Comment out the Exception below.
#           Implement GcdUnitMsg code shown on the slides.
#-------------------------------------------------------------
class GcdUnitReqMsg( BitStructDefinition ):

  def __init__( s ):
    s.a = BitField( 16 )
    s.b = BitField( 16 )

  def __str__( s ):
    return "{}:{}".format( s.a, s.b )
```

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

PyMTL 102: Latency Insensitive FL Models

- Implementing latency insensitive communication protocols can be complex to implement and a challenge to debug.
- PyMTL provides **Interface Adapters** which abstract away the complexities of ValRdy, and expose simplified method interfaces.



PyMTL 102: Latency Insensitive FL Models

- Implementing latency insensitive communication protocols can be complex to implement and a challenge to debug.
- PyMTL provides **Interface Adapters** which abstract away the complexities of ValRdy, and expose simplified method interfaces.

```
    # Interface

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

    # Adapters

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

Hands-On: FL, CL, RTL Modeling of a GCD Unit

- Task 3.1: Create a BitStruct for the GCD request
- **Task 3.2: Build an FL model for the GCD unit**
- **Task 3.3: Create a latency insensitive test**
- Task 3.4: Add timing to the GCD CL model
- Task 3.5: Fix the bug in the GCD RTL model
- Task 3.6: Verify generated Verilog GCD RTL
- Task 3.7: Experiment with the GCD simulator

★ Task 3.2: Build an FL model for the GCD unit ★

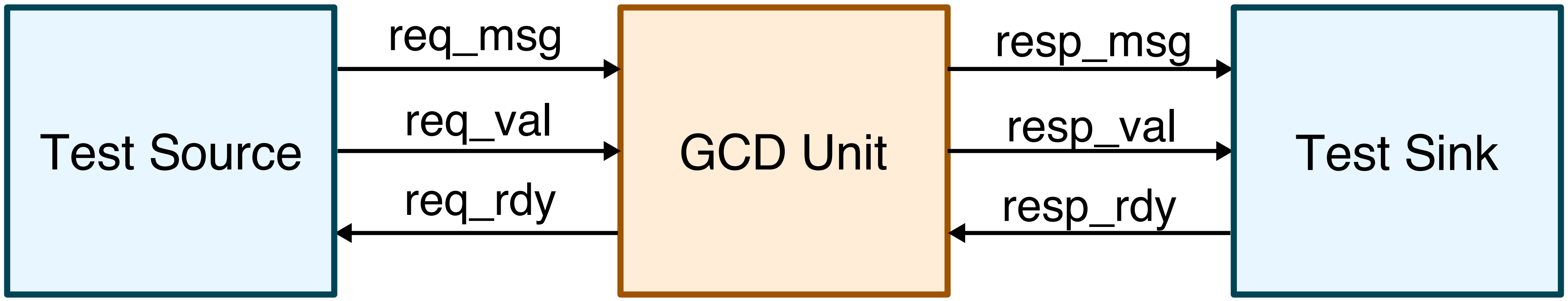
```
% cd ~/pymtl-tut/build
% gedit ../gcd/GcdUnitFL.py
```

```
    @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 )
```

```
% py.test ../gcd/GcdUnitFL_test.py -v
```

PyMTL 102: Testing Latency Insensitive Models

- To simplify testing of latency insensitive designs, PyMTL provides TestSources and TestSinks with ValRdy interfaces.
- TestSources/TestSinks only transmit/accept data when the “design under test” is ready/valid.
- Can be configured to insert random delays into valid/ready signals to verify latency insensitivity under various conditions.



★ Task 3.3: Create a latency insensitive test ★

```
% cd ~/pymtl-tut/build
% gedit ../gcd/GcdUnitFL_simple_test.py
```

```
class TestHarness (Model):

  def __init__( s, src_msgs, sink_msgs ):

    s.src  = TestSource (GcdUnitReqMsg(), src_msgs)
    s.gcd  = GcdUnitFL  ()
    s.sink = TestSink   (Bits(16), sink_msgs)

    s.connect( s.src.out,  s.gcd.req  )
    s.connect( s.gcd.resp, s.sink.in_ )
```

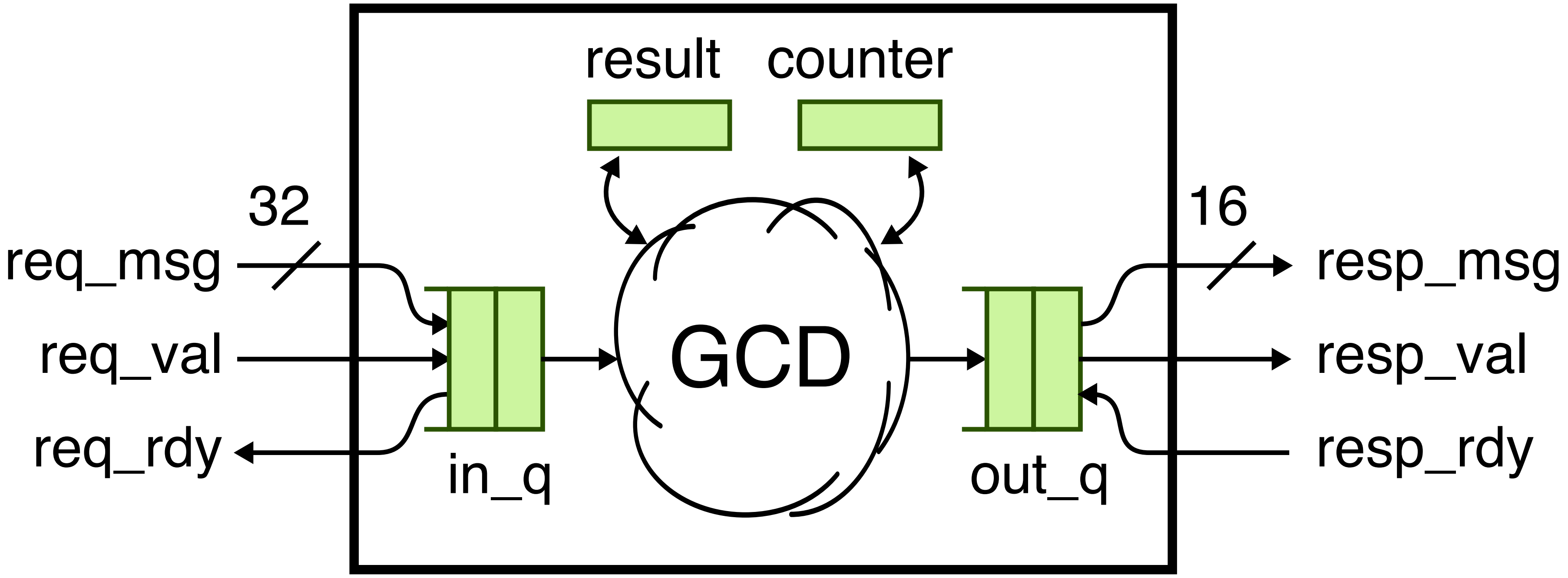
```
% py.test ../gcd/GcdUnitFL_simple_test.py -vs
```

PyMTL 102: Latency Insensitive FL Models

```
../gcd/GcdUnitFL_simple_test.py::test

  2:           >          ().    > .
  3: 000f:0005 > 000f:0005()     >
  4: #         > #         ()0005 > 0005
  5: #         > #         ()     >
  6: 0003:0009 > 0003:0009()     >
  7: #         > #         ()0003 > 0003
  8: #         > #         ()     >
  9: 001b:000f > 001b:000f()     >
 10: #         > #         ()0003 > 0003
 11: #         > #         ()     >
 12: 0015:0031 > 0015:0031()     >
 13: .         > .         ()0007 > 0007

PASSED
```

PyMTL 102: Latency Insensitive CL Models

- Cycle-level models add timing information to a functional model and can provide a cycle-approximate estimation of performance.
- Useful for rapid, initial exploration of an architectural design space.
- We'll use a simple GCD algorithm to provide timing info.



Hands-On: FL, CL, RTL Modeling of a GCD Unit

- Task 3.1: Create a BitStruct for the GCD request
- Task 3.2: Build an FL model for the GCD unit
- Task 3.3: Create a latency insensitive test
- **Task 3.4: Add timing to the GCD CL model**
- **Task 3.5: Fix the bug in the GCD RTL model**
- Task 3.6: Verify generated Verilog GCD RTL
- Task 3.7: Experiment with the GCD simulator

★ Task 3.4: Add timing to the GCD CL model ★

```
% cd ~/pymtl-tut/build
% py.test ../gcd/GcdUnitCL_test.py
% py.test ../gcd/GcdUnitCL_test.py -k basic_0x0 -sv
% gedit ../gcd/GcdUnitCL.py
```

```
# Handle delay to model the gcd unit latency

if s.counter > 0:
  s.counter -= 1
  if s.counter == 0:
    s.resp_q.enq( s.result )

# If we have a new message and the output queue is not full

elif not s.req_q.empty() and not s.resp_q.full():
  req_msg = s.req_q.deq()
  s.result,s.counter = gcd( req_msg.a, req_msg.b )
```

```
def gcd( a, b ):

  ncycles = 1

  while b:
    ncycles += 1
    a, b = b, a%b

  return (a, ncycles)
```

```
% py.test ../gcd/GcdUnitCL_test.py -k basic_0x0 -sv
```

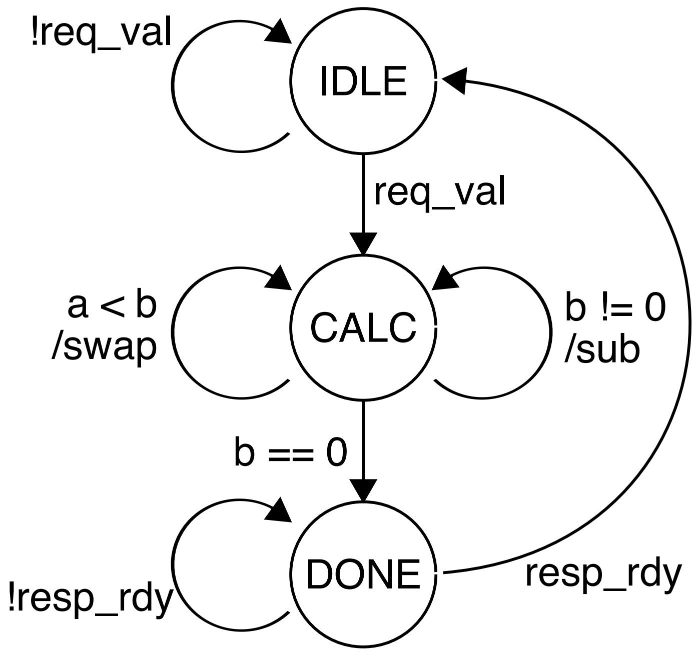
PyMTL 102: Latency Insensitive CL Models

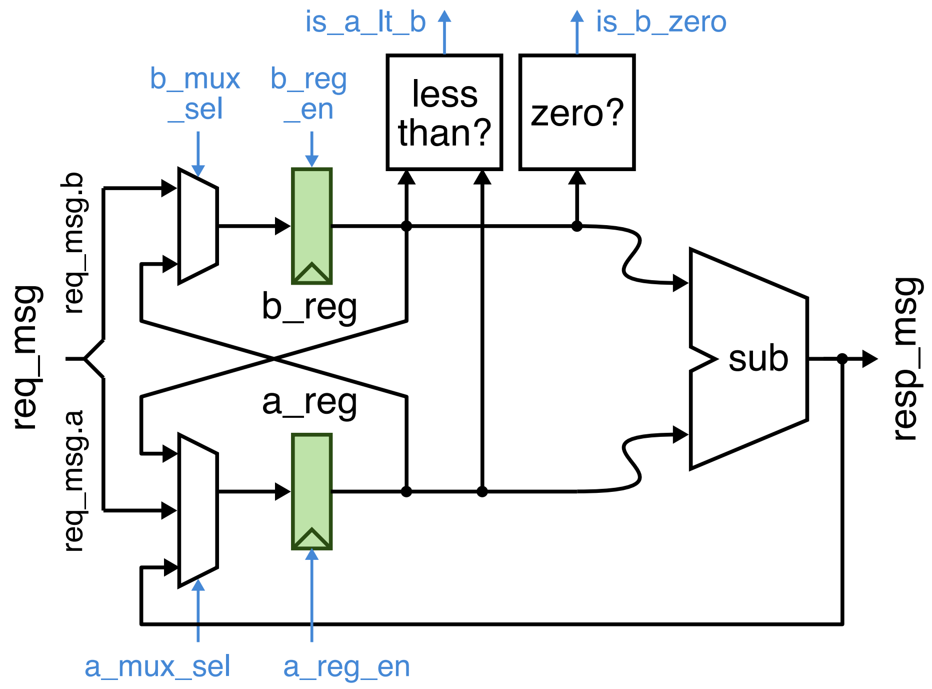
```
../gcd_soln/GcdUnitCL_test.py::test[basic_0x0] ()

  2:           >           ().    > .
  3: 000f:0005 > 000f:0005()      >
  4: 0003:0009 > 0003:0009()      >
  5: #         > #         ()0005 > 0005
  6: 0000:0000 > 0000:0000()      >
  7: #         > #         ()0003 > 0003
  8: 001b:000f > 001b:000f()      >
  9: #         > #         ()0000 > 0000
 10: 0015:0031 > 0015:0031()      >
 11: #         > #         ()0003 > 0003
 12: 0019:001e > 0019:001e()      >
 13: #         > #         ()0007 > 0007
 14: 0013:001b > 0013:001b()      >
 15: #         > #         ()0005 > 0005
 16: 0028:0028 > 0028:0028()      >
 17: #         > #         ()0001 > 0001
 18: 00fa:00be > 00fa:00be()      >
 19: #         > #         ()0028 > 0028
 20: 0005:00fa > 0005:00fa()      >
 21: #         > #         ()000a > 000a
 22: ffff:00ff > ffff:00ff()      >
 23: .         > .         ()0005 > 0005
 24:           >           ()     >
 25:           >           ()00ff > 00ff

PASSED
```

```
../gcd_soln/GcdUnitCL_test.py::test[basic_0x0]

  2:           >           ().    > .
  3: 000f:0005 > 000f:0005()      >
  4: 0003:0009 > 0003:0009()      >
  5: #         > #         ()     >
  6: #         > #         ()0005 > 0005
  7: 0000:0000 > 0000:0000()      >
  8: #         > #         ()     >
  9: #         > #         ()     >
 10: #         > #         ()0003 > 0003
 11: 001b:000f > 001b:000f()      >
 12: #         > #         ()0000 > 0000
 13: 0015:0031 > 0015:0031()      >
 14: #         > #         ()     >
 15: #         > #         ()     >
 16: #         > #         ()     >
 17: #         > #         ()0003 > 0003
 18: 0019:001e > 0019:001e()      >
 19: #         > #         ()     >
 20: #         > #         ()     >
 21: #         > #         ()     >
 22: #         > #         ()0007 > 0007
 23: 0013:001b > 0013:001b()      >
 24: #         > #         ()     >
 25: #         > #         ()     >
 26: #         > #         ()     >
 27: #         > #         ()0005 > 0005
 28: 0028:0028 > 0028:0028()      >
 29: #         > #         ()     >
```

PyMTL 102: Latency Insensitive RTL Models

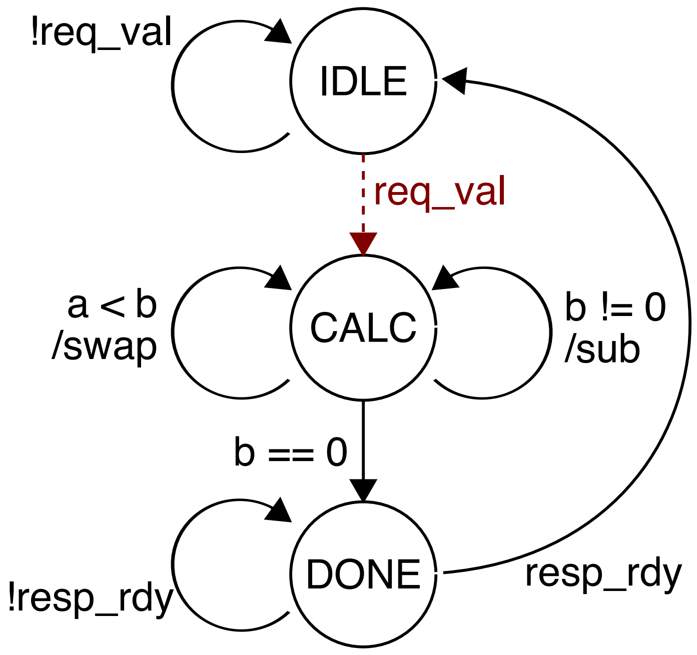
- RTL models allow us to accurately estimate executed cycles, cycle-time, area and energy when used with an EDA toolflow.
- Constructing is time consuming! PyMTL tries to make it it more productive by providing a better design and testing environment.

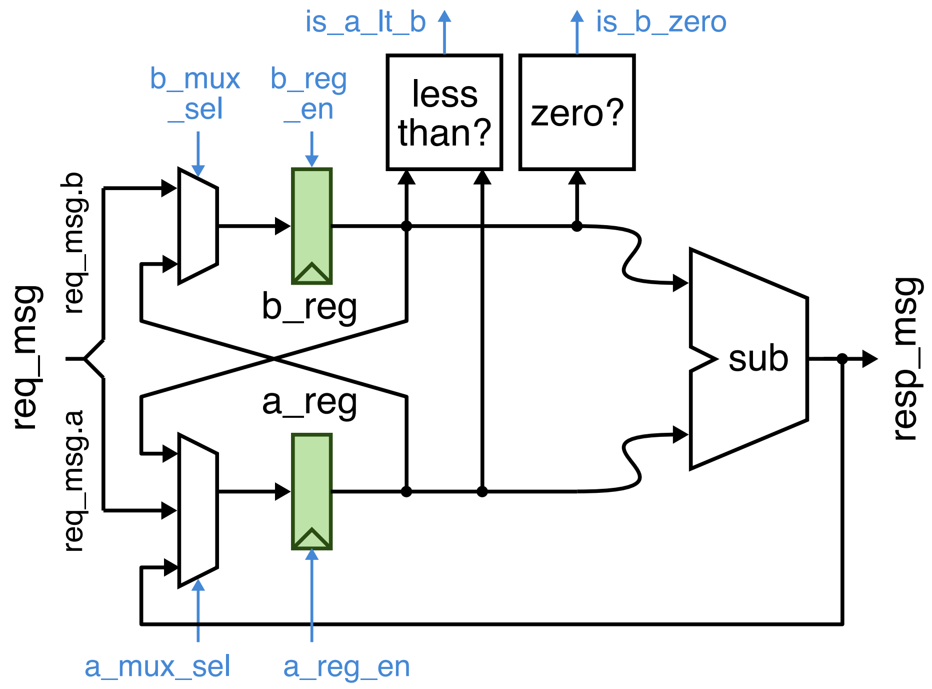




PyMTL 102: Latency Insensitive RTL Models

- Latency insensitive hardware generally separates logic into control and datapath (shown below).
- Today, we won't be writing RTL for GCD, but well be fixing a bug in the RTL implementation of the state machine.

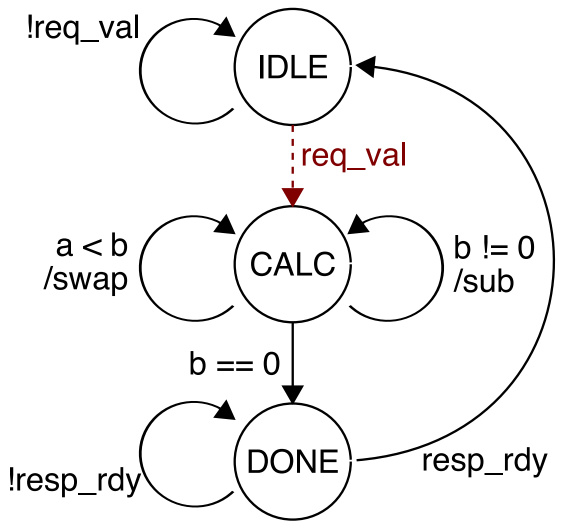




★ Task 3.5: Fix the bug in the GCD RTL model ★

```
% cd ~/pymtl-tut/build
% py.test ../gcd/GcdUnitRTL_test.py -k basic_0x0 -v
% gedit ../gcd/GcdUnitRTL.py
```

```
# Transitions out of IDLE state

if ( curr_state == s.STATE_IDLE ):
  pass

# Transitions out of CALC state

if ( curr_state == s.STATE_CALC ):
  if ( not s.is_a_lt_b and s.is_b_zero ):
    next_state = s.STATE_DONE

# Transitions out of DONE state
```



```
% py.test ../gcd/GcdUnitRTL_test.py -k basic_0x0 -v
```

PyMTL 102: Latency Insensitive RTL Models

```
../gcd_soln/GcdUnitRTL_test.py::test[basic_0x0]

  2:           >           (000f 0005 I ).    > .
  3: 000f:0005 > 000f:0005(000f 0005 I )      >
  4: #         > #         (000f 0005 C-)     >
  5: #         > #         (000a 0005 C-)     >
  6: #         > #         (0005 0005 C-)     >
  7: #         > #         (0000 0005 Cs)     >
  8: #         > #         (0005 0000 C )     >
  9: #         > #         (0005 0000 D )0005 > 0005
 10: 0003:0009 > 0003:0009(0005 0000 I )      >
 11: #         > #         (0003 0009 Cs)     >
 12: #         > #         (0009 0003 C-)     >
 13: #         > #         (0006 0003 C-)     >
 14: #         > #         (0003 0003 C-)     >
 15: #         > #         (0000 0003 Cs)     >
```

Hands-On: FL, CL, RTL Modeling of a GCD Unit

- Task 3.1: Create a BitStruct for the GCD request
- Task 3.2: Build an FL model for the GCD unit
- Task 3.3: Create a latency insensitive test
- Task 3.4: Add timing to the GCD CL model
- Task 3.5: Fix the bug in the GCD RTL model
- **Task 3.6: Verify generated Verilog GCD RTL**
- **Task 3.7: Experiment with the GCD simulator**

★ Task 3.6: Verify generated Verilog GCD RTL ★

```
% cd ~/pymtl-tut/build
% py.test ../gcd/GcdUnitRTL_test.py --test-verilog -sv
% gedit GcdUnitRTL_*.v
```

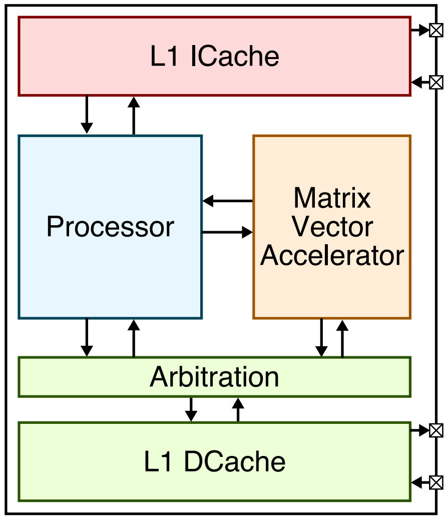
```
module GcdUnitRTL_0x791afe0d4d8c
(
  input  wire [   0:0] clk,
  input  wire [  31:0] req_msg,
  output wire [   0:0] req_rdy,
  input  wire [   0:0] req_val,
  input  wire [   0:0] reset,
  output wire [  15:0] resp_msg,
  input  wire [   0:0] resp_rdy,
  output wire [   0:0] resp_val
);

  // ctrl temporaries
  wire   [   0:0] ctrl$is_b_zero;
  wire   [   0:0] ctrl$resp_rdy;
  wire   [   0:0] ctrl$clk;
  wire   [   0:0] ctrl$is_a_lt_b;
  wire   [   0:0] ctrl$req_val;
  wire   [   0:0] ctrl$reset;
  wire   [   1:0] ctrl$a_mux_sel;
  wire   [   0:0] ctrl$resp_val;
  wire   [   0:0] ctrl$b_mux_sel;
  wire   [   0:0] ctrl$b_reg_en;
  wire   [   0:0] ctrl$a_reg_en;
  wire   [   0:0] ctrl$req_rdy;

  GcdUnitCtrlRTL_0x791afe0d4d8c ctrl
  (
    .is_b_zero ( ctrl$is_b_zero ),
    .resp_rdy  ( ctrl$resp_rdy ),
    .clk       ( ctrl$clk ),
    .is_a_lt_b ( ctrl$is_a_lt_b ),
    .req_val   ( ctrl$req_val ),
    .reset     ( ctrl$reset ),
    .a_mux_sel ( ctrl$a_mux_sel ),
    .resp_val  ( ctrl$resp_val ),
    .b_mux_sel ( ctrl$b_mux_sel ),
    .b_reg_en  ( ctrl$b_reg_en ),
    .a_reg_en  ( ctrl$a_reg_en ),
    .req_rdy   ( ctrl$req_rdy )
  );

  // dpath temporaries
  wire   [   1:0] dpath$a_mux_sel;
```

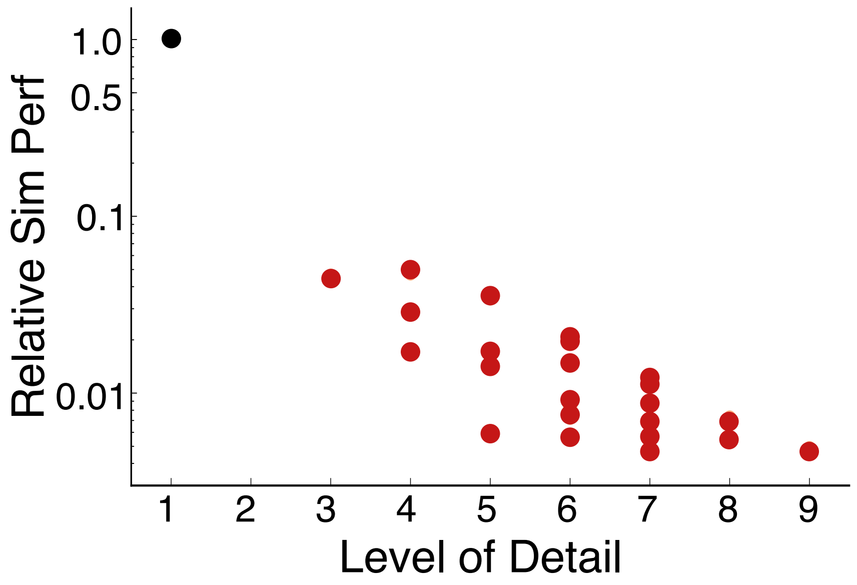
★ Task 3.7: Experiment with the GCD simulator ★

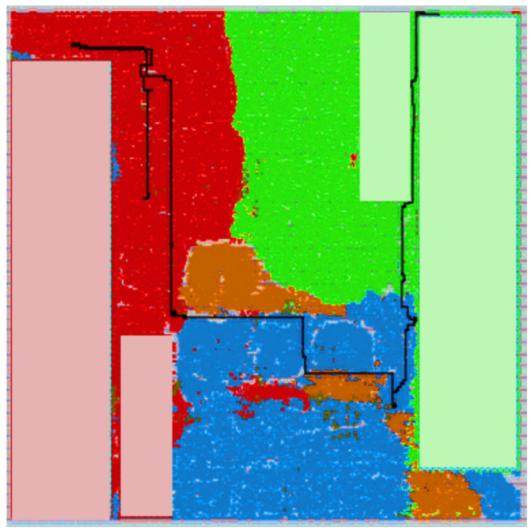
```
# Simulating both the CL and RTL models

% cd ~/pymtl-tut/build
% ../gcd/gcd-sim --stats --impl fl  --input random
% ../gcd/gcd-sim --stats --impl cl  --input random
% ../gcd/gcd-sim --stats --impl rtl --input random

# Experimenting with various datasets

% ../gcd/gcd-sim --impl rtl --input random --trace
% ../gcd/gcd-sim --impl rtl --input small  --trace
% ../gcd/gcd-sim --impl rtl --input zeros  --trace
```

PyMTL In Practice: Matrix Vector Accelerator

- In the PyMTL paper [MICRO'14], we discuss how multi-level modeling in PyMTL can facilitate the design of coprocessors.
- Selecting FL/CL/RTL models for the cache/processor/accelerator allows designers to tradeoff simulation speed and accuracy.
- PyMTL-generated Verilog passed into Synopsys toolflow for area/energy/timing estimates.



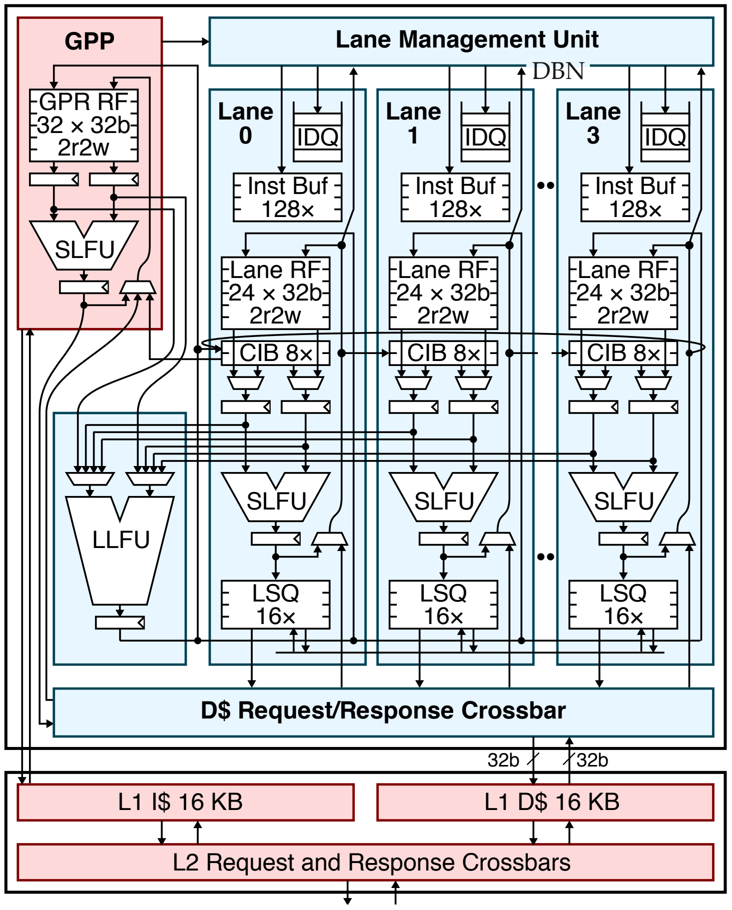




PyMTL In Practice: XLOOPS Loops Specialization

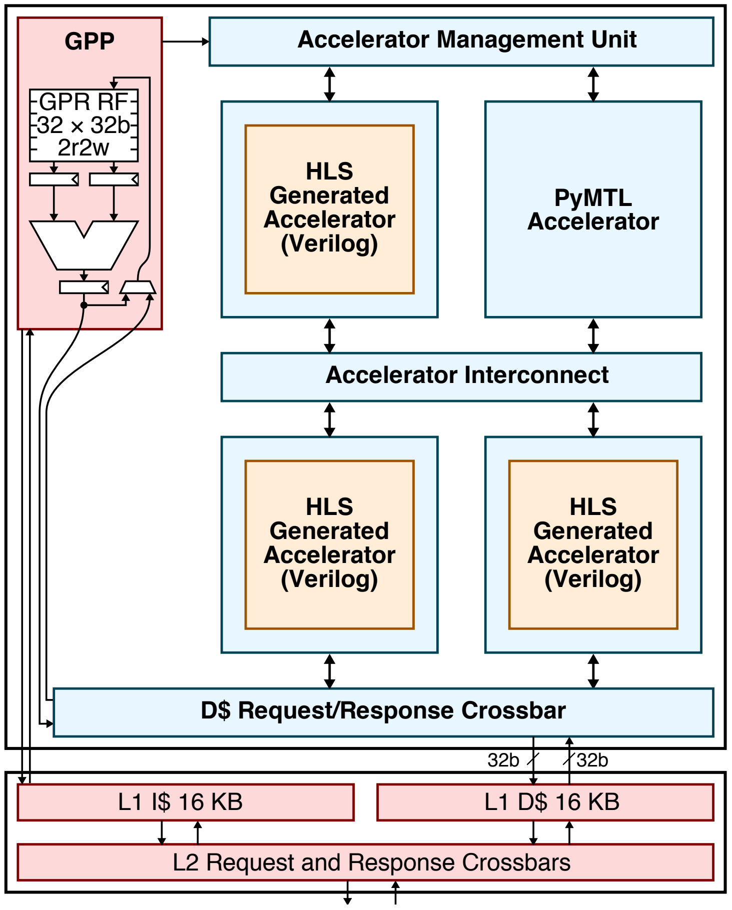
- In the XLOOPS paper (published in MICRO'14), PyMTL was combined with gem5 to evaluate an architecture for loop acceleration.
- gem5 provided access to complex out-of-order processor and memory system models (red).
- PyMTL was used to quickly build and iterate on a CL model for the loop acceleration unit (blue).

S. Srinath, B. Ilbeyi, et al., "Architectural Specialization for Inter-Iteration Loop Dependence Patterns." 47th ACM/IEEE Int'l Symp. on Microarchitecture, Dec. 2014.



PyMTL In Practice: HLS Accelerators

- We are currently experimenting with accelerators generated using high-level synthesis
- We can import the HLS-generated Verilog into PyMTL, and then use PyMTL to verify these accelerators and compose accelerators using various interconnects
- We can also include our own accelerators written in PyMTL using FL, CL, and RTL modeling


GPP
GPR RF 32 × 32b 2r2w
Accelerator Management Unit
HLS Generated Accelerator (Verilog)
PyMTL Accelerator
Accelerator Interconnect
HLS Generated Accelerator (Verilog)
HLS Generated Accelerator (Verilog)
D$ Request/Response Crossbar
32b
32b
L1 I$ 16 KB
L1 D$ 16 KB
L2 Request and Response Crossbars

- We then use PyMTL+gem5 integration to experiment with tightly integrated general-purpose processors with accelerators

PyMTL Next Steps and More Resources

Next Steps:

- See the detailed tutorial on the Cornell ECE5745 website:
  http://www.csl.cornell.edu/courses/ece5745/handouts/ece5745-tut-pymtl.pdf

Check out the /**docs** directory in the PyMTL repo for guides on:

- Writing Pythonic PyMTL Models and Tests
- Writing Verilog Translatable PyMTL RTL
- Importing Verilog Components into PyMTL
- **Coming Soon**: Embedding PyMTL Models into gem5

Become a contributor! We'd love your PyMTL hacks and models!

- https://github.com/cornell-brg/pymtl
- https://github.com/cornell-brg/pydgin

Thank you for coming!

PyMTL

Pydgin

PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research

[MICRO 2014]

https://github.com/cornell-brg/pymtl

Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers

[ISPASS 2015]

https://github.com/cornell-brg/pydgin