TUHH
Hamburg University of Technology

Institute of Embedded Systems

PyMTL Tutorial

Wolfgang Brandt

November 5, 2019

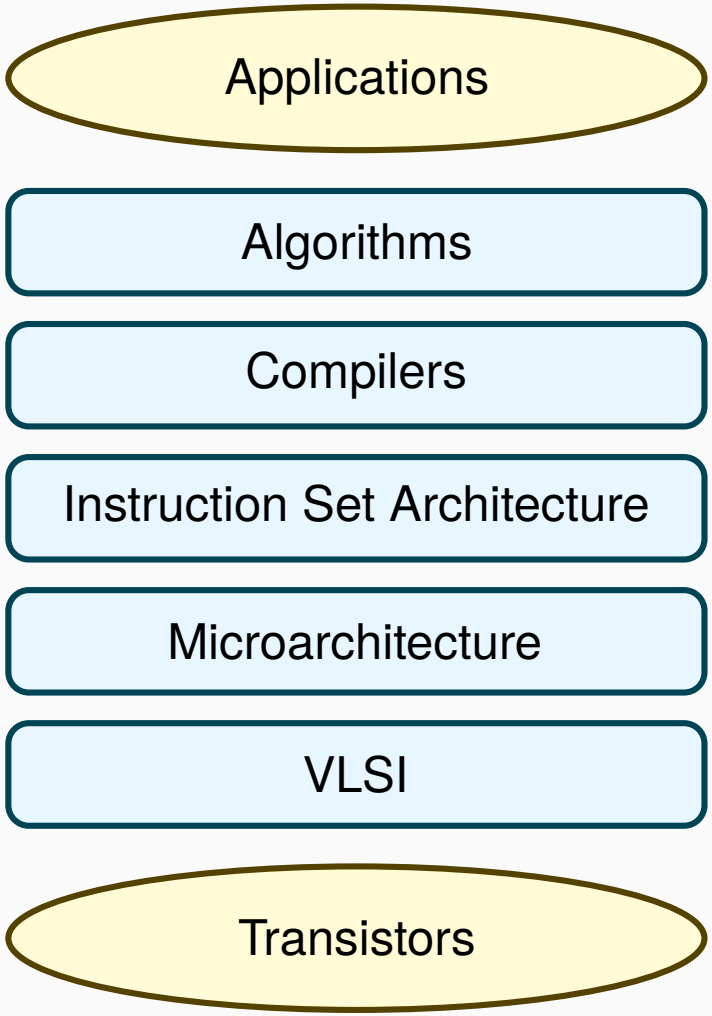
Institute of Embedded Systems (E-13)
Hamburg University of Technology

What is PyMTL?

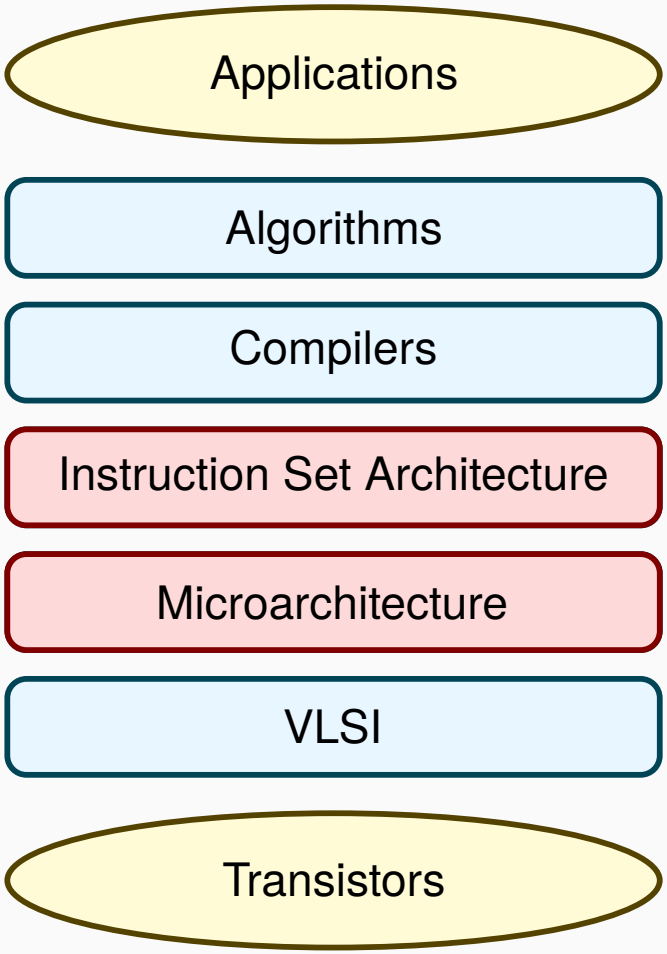
Python-based hardware **generation**, **simulation**, and **verification** **framework** which enables productive **multi-level modeling** and VLSI design

- 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 (System)Verilog
- A Python testing framework for model validation

Stack of Abstraction levels



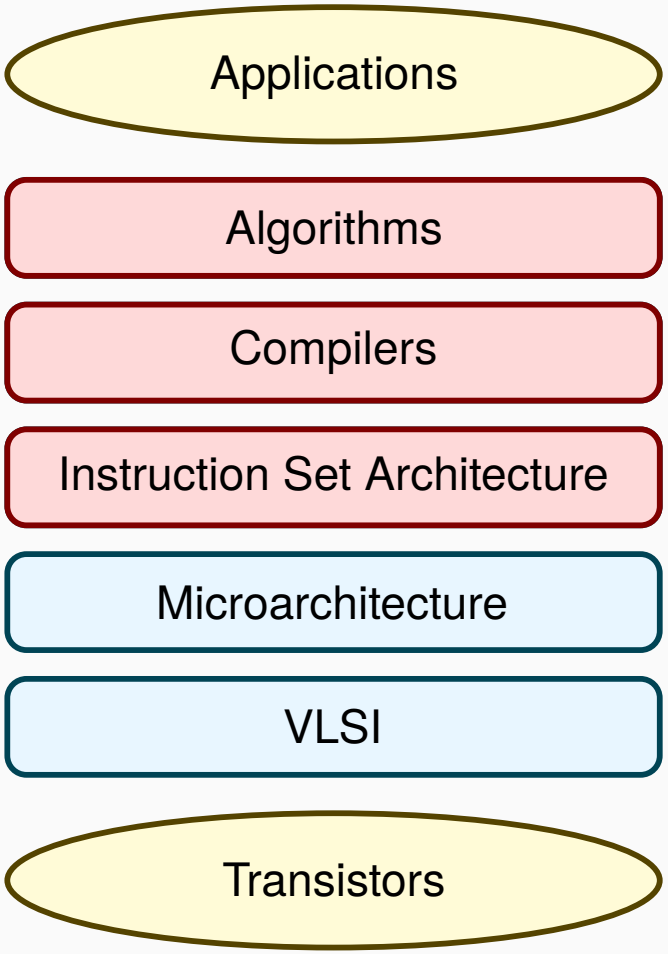
Modeling levels



Cycle-Level Modeling

- Behavior
- Cycle-Approximate
- Analytical Area, Energy, Timing

Modeling levels



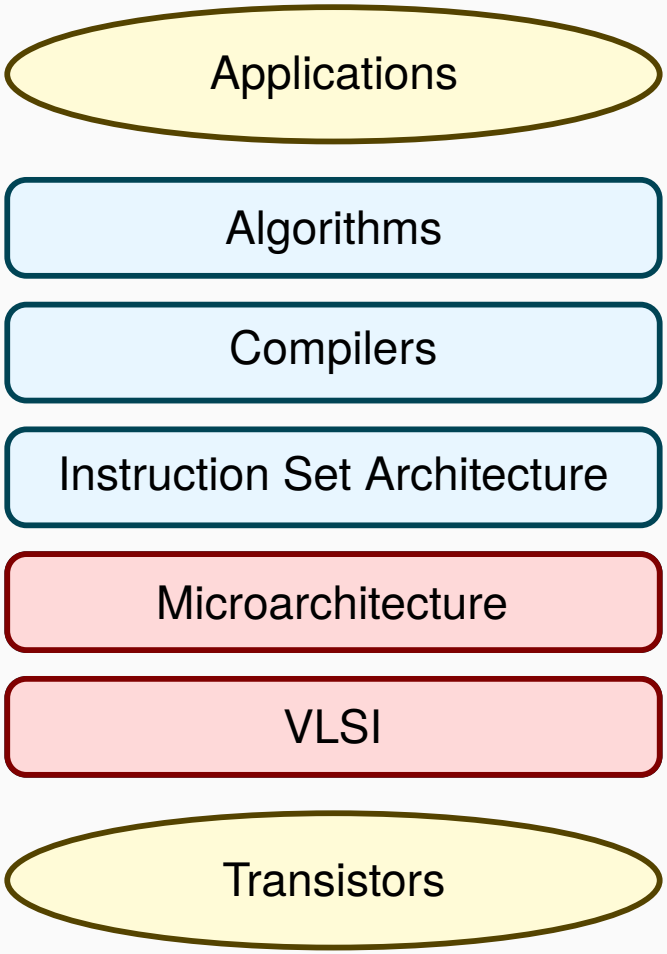
Functional-Level Modeling

– Behavior

Cycle-Level Modeling

– Behavior
– Cycle-Approximate
– Analytical Area, Energy, Timing

Modeling levels



Functional-Level Modeling

– Behavior

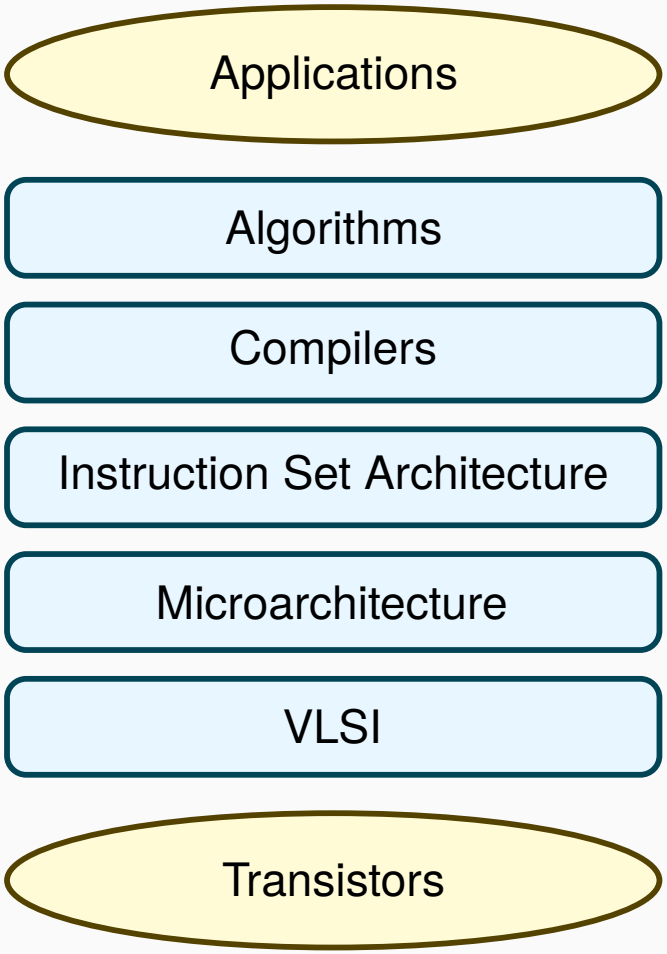
Cycle-Level Modeling

– Behavior
– Cycle-Approximate
– Analytical Area, Energy, Timing

Register-Transfer-Level Modeling

– Behavior
– Cycle-Accurate Timing
– Gate-Level Area, Energy, Timing

Modeling Tools



Functional-Level Modeling

- Algorithm/ISA Development
- MATLAB/Python, C++ ISA Sim

Cycle-Level Modeling

- Design-Space Exploration
- C++ Simulation Framework
- SW-Focused Object-Oriented
- gem5, SESC, McPAT

Register-Transfer-Level Modeling

- Prototyping & AET Validation
- Verilog, VHDL Languages
- HW-Focused Concurrent Structural
- EDA Toolflow

Modeling Gap

Computer Architecture Modeling Gap

FL, CL, RTL modeling use very different languages, patterns, tools, and methodologies

Functional-Level Modeling

- Algorithm/ISA Development
- MATLAB/Python, C++ ISA Sim

Cycle-Level Modeling

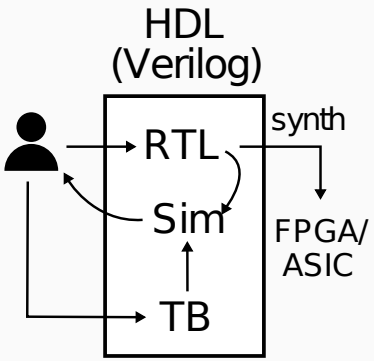
- Design-Space Exploration
- C++ Simulation Framework
- SW-Focused Object-Oriented
- gem5, SESC, McPAT

Register-Transfer-Level Modeling

- Prototyping & AET Validation
- Verilog, VHDL Languages
- HW-Focused Concurrent Structural
- EDA Toolflow

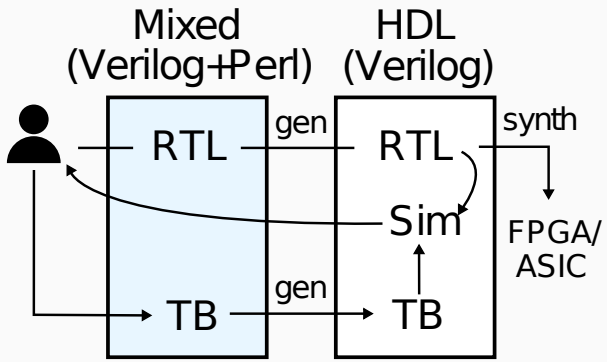
Traditional VLSI Design Methodologies

HDL
Hardware Description
Language



- ✓ Fast edit-sim-debug loop
- ✓ Single language for structural, behavioral + TB
- ✗ Difficult to create highly parameterized generators

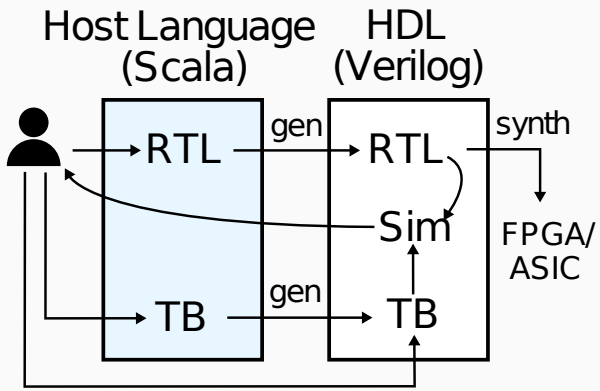
HPF
Hardware Preprocessing
Framework



Example: Genesis2

- ✗ Slower edit-sim-debug loop
- ✗ Multiple languages create "semantic gap"
- ✓ Easier to create highly parameterized generators

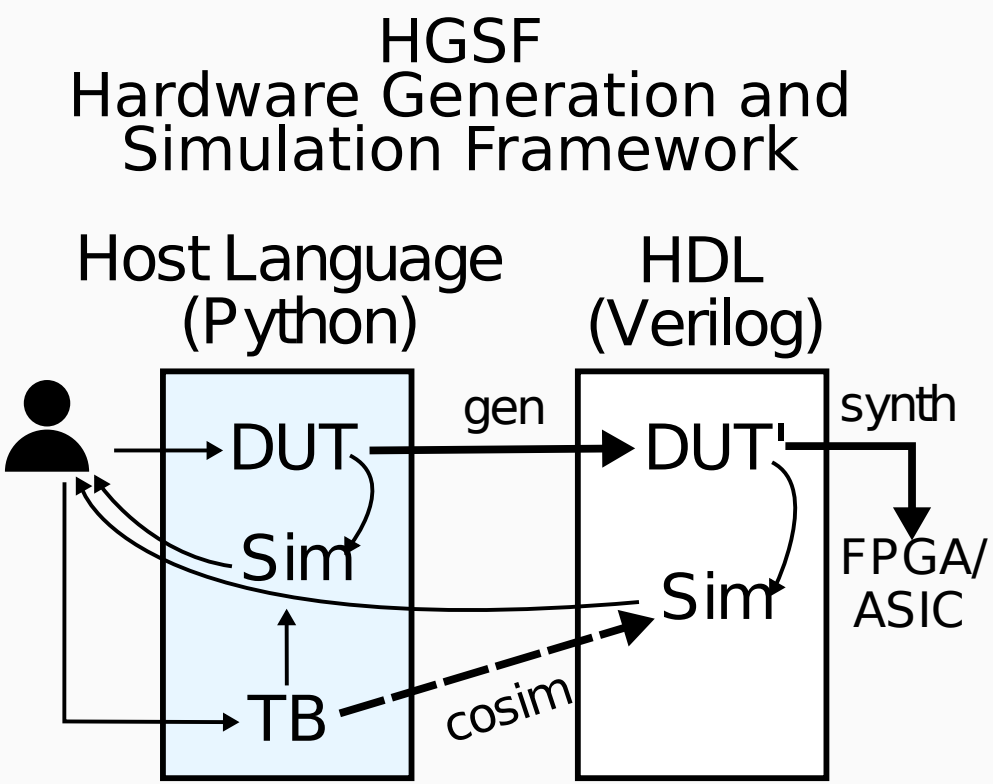
HGF
Hardware Generation
Framework



Example: Chisel

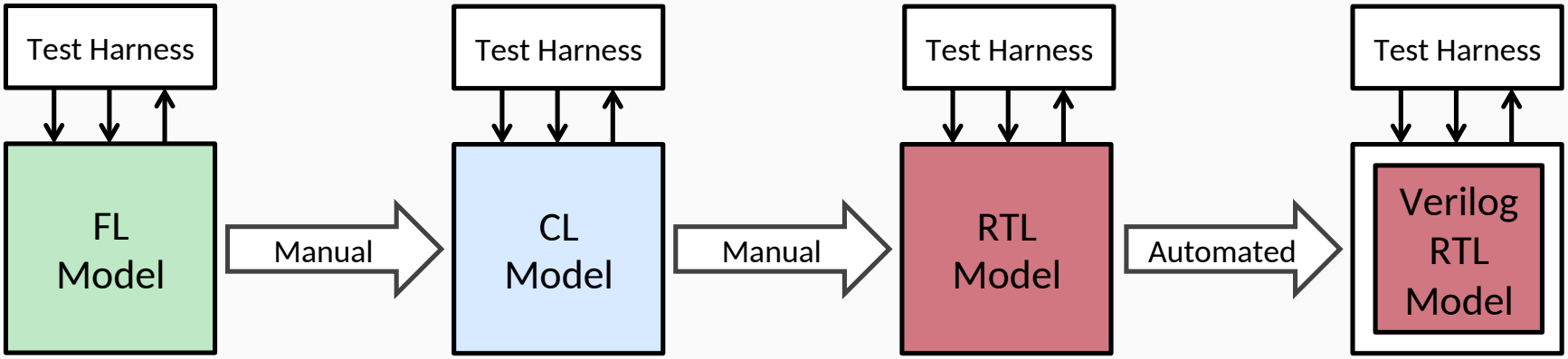
- ✗ Slower edit-sim-debug loop
- ✓ Single language for structural + behavioral
- ✓ Easier to create highly parametrized generators
- ✗ Cannot use power of host language for verification

Productive Multi-Level Modeling and VLSI Design



- ✓ Single framework for ML modeling & VLSI Design
- ✓ Fast edit-sim-debug loop
- ✓ Single language for structural, behavioral, + TB
- ✓ Easy to create highly parametrized generators
- ✓ Use power of host language for verification

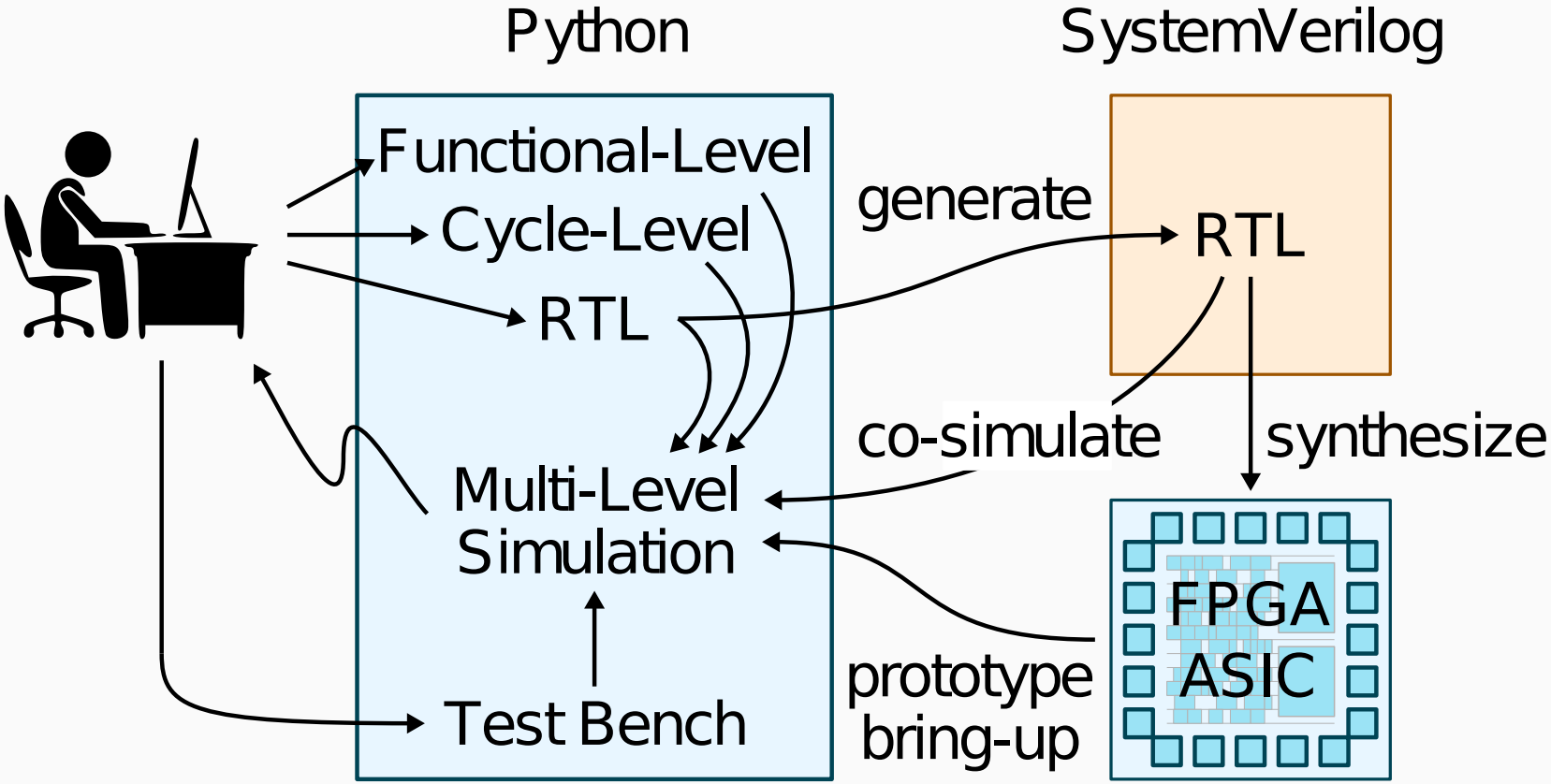
Multi-Level Modeling in PyMTL



- FL modeling allows for the rapid creation of a working model. Designers can quickly experiment with interfaces and protocols.
- This design is manually refined into a PyMTL CL model that includes timing, which is useful for rapid design space exploration.
- Promising architectures can again be manually refined into a PyMTL RTL implementation to accurately model resources.
- Verilog generated from PyMTL RTL can be passed to an EDA toolflow for accurate area, energy, and timing estimates.
- Throughout this process, the same PyMTL test harnesses can used to verify each model!

What is PyMTL?

Python-based hardware **generation**, **simulation**, and **verification framework** which enables productive **multi-level modeling** and VLSI design



Why Python?

- Is well regarded as a highly productive language with pseudocode-like syntax
- Supports modern language features to enable rapid, agile development (dynamic typing, reflection, metaprogramming)
- Has a large and active developer and support community
- Includes extensive standard and third-party libraries
- Enables embedded domain-specific languages (EDSL)
- Facilitates engaging application-level researchers
- Includes built-in support for integrating with C/C++
- Performance is improving with advanced JIT compilation

What is PyMTL for and not (currently) for?

- PyMTL is for ...
  - Taking an accelerator design from concept to implementation
  - Construction of highly-parameterizable RTL chip generators
  - Rapid design, testing, and exploration of hardware mechanisms
  - Interfacing models with imported (System)-Verilog
- PyMTL is not (currently) for ...
  - Python high-level synthesis (HLS)
  - Many-core simulations with hundreds of cores
  - Full-system simulation with real OS support
  - Users needing a complex OOO processor model e.g. ARM/X86 “out of the box”

`Bits` Class

- The `Bits` class represents fixed-bitwidth values.
- Each bit can only take on one of two values (i.e., 0, 1)
- Creating a `Bits` Class object:

```
% python
>>> from pymtl3 import *
>>> a = Bits( 16, 37 )
>>> a
Bits16( 0x0025 )
>>> a.nbits
16
>>> a.uint()
37
>>> a.int()
37
>>> a.oct()
'0o00000045'
>>> a.hex()
'0x0025'
>>> a.bin()
'0b0000000000100101'
```

```
>>> a = Bits16( -37 )
>>> a.int()
-37
>>> a.uint()
65499
>>> a.bin()
'0b1111111111011011'
>>> d = Bits256( 0xff )
NameError: 'Bits256' is not defined
>>> Bits256 = mk_bits(256)
>>> d = Bits256( 0xff )
>>> d Bits256( 0x0000000000000000...00ff )
```

Numerical Literals in `Bits` Class

Specify numeric literals in binary, hexadecimal or octal form:

```
>>> Bits( 8, 0b10101100 )
Bits( 8, 0xac )
>>> Bits( 32, 0xabcd0123 )
Bits32( 0xabcd0123 )
>>> Bits( 8, 0o34)
Bits8( 0x1c )
```

Negative values stored in two's complement:

```
>>> Bits( 8, -1 )
Bits8( 0xff )
>>> Bits( 8, -2 )
Bits8( 0xfe )
>>> Bits( 8, -128 )
Bits8( 0x80 )
```

Slicing `Bits` Class objects

Specify bit slices for reading or writing fields within a Bits object:

```
>>> a = Bits( 32, 0xabcd0123 )
>>> a[4:24]
Bits20( 0xcd012 )
>>> a = Bits( 32, 0xabcd0123 )
>>> a[4:24] = 0x210cd
>>> a
Bits32( 0xfab210cd3 )
```

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
a	b	c	d	0	1	2	3

Referencing `Bits` Class objects

Assigning a to b creates two names that refer to the same Bits object:

```
>>> a = Bits( 32, 0xabcd0123 )
>>> b = a
>>> b
Bits32( 0xabcd0123 )
>>> a[24:32] = 0x67
>>> a
Bits32( 0x67cd0123 )
>>> b
Bits32( 0x67cd0123 )
```

Copying `Bits` Class objects

To copy the object, we must create a new Bits object:

```
>>> a = Bits( 32, 0xabcd0123 )
>>> b = Bits( 32, a )
>>> a
Bits32( 0xabcd0123 )
>>> b
Bits32( 0xabcd0123 )
>>> a[24:32] = 0x67
>>> a
Bits32( 0x67cd0123 )
>>> b
Bits32( 0xabcd0123 )
```

Concatenating and Extending `Bits` objects

```
>>> a = Bits( 4, 0xd )
>>> b = Bits( 12, 0xead )
>>> c = Bits( 12, 0xbee )
>>> d = Bits( 4, 0xf )
>>> concat( a, b, c, d )
Bits32( 0xdeadbeef )
```

```
>>> a = Bits( 4, 0xa )
>>> sext( a, 8 )
Bits8( 0xfa )
>>> zext( a, 8 )
Bits8( 0x0a )
```

Reducing and Shifting `Bits` objects

```
>>> a = Bits( 8, 0b10101100 )
>>> reduce_and(a)
Bits( 1, 0x0 )
>>> reduce_or(a)
Bits( 1, 0x1 )
>>> reduce_xor(a)
Bits( 1, 0x0 )
>>> a >> 2
Bits8( 0x2b )
```

`Bits` Class Operators

Logical Operators

&	bitwise AND
|	bitwise OR
^	bitwise XOR
^~	bitwise XNOR
~	bitwise NOT

Arith. Operators

+	addition
-	subtraction
*	multiplication
/	division
%	modulo

Shift Operators

>>	shift right
<<	shift left

Slice Operators

`[x]`	get/set bit x
`[x:y]`	get/set bits x upto y

Reduction Operators

`reduce_and`	reduce via AND
`reduce_or`	reduce via OR
`reduce_xor`	reduce via XOR

Relational Operators

==	equal
!=	not equal
>	greater than
>=	greater than or equals
<	less than
<=	less than or equals

Other Functions

`concat`	concatenate
`sext`	sign-extension
`zext`	zero-extension

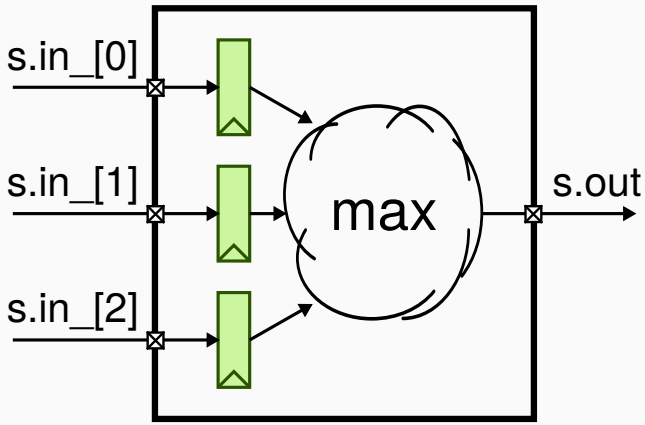
First PyMTL Model (Functional Level)

Pure Python Model:

```
def max_unit( input_list ):
  return max( input_list )

>>> max_unit ([1,2,3,4])
4
```

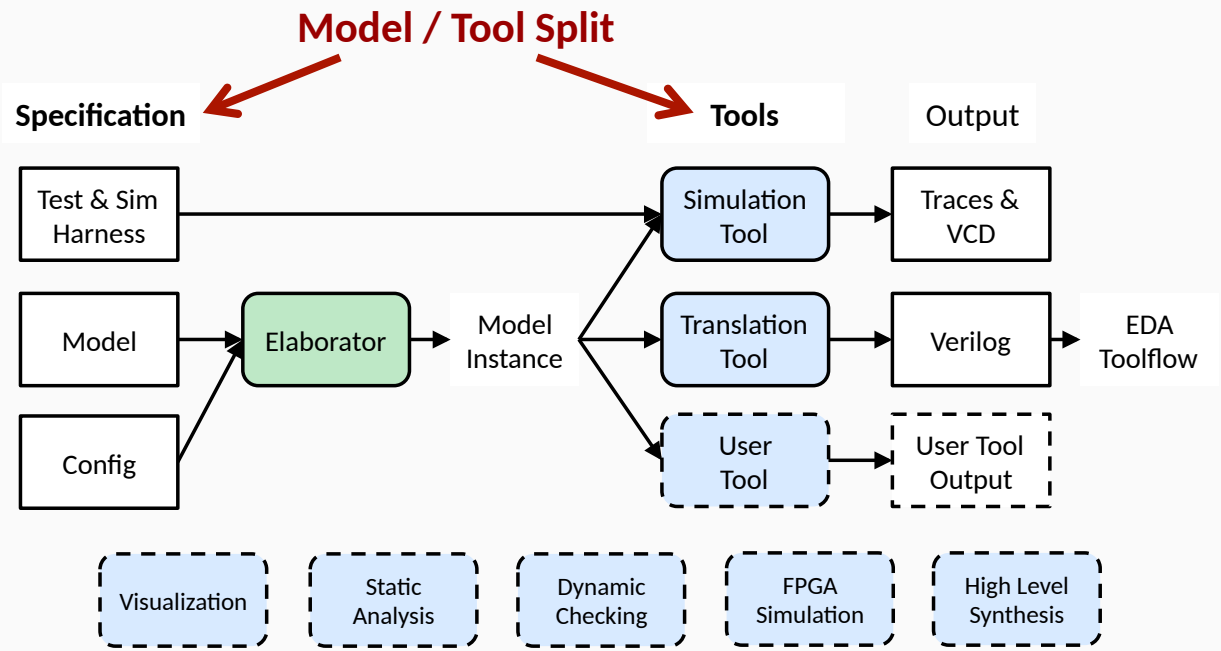
PyMTL Embedded DSL Model:

```
from pymtl3 import *

class MaxUnitFL( Component ):
  def construct( s, Type, ninputs ):
    s.in_ = [InPort( Type ) \
             for _ in range(ninputs)]
    s.out = OutPort( Type )

    @s.update
    def logic():
      s.out = max( s.in_ )
```



Static elaboration

- Everything inside `construct()` that is not in a decorated function (@update, @update_on_edge)
- Constructs a connectivity graph of components
- Can use the full expressiveness of Python
- Is always Verilog translatable as long as leaf modules are translatable
- Enables the creation of powerful and highly-parameterizable hardware generators

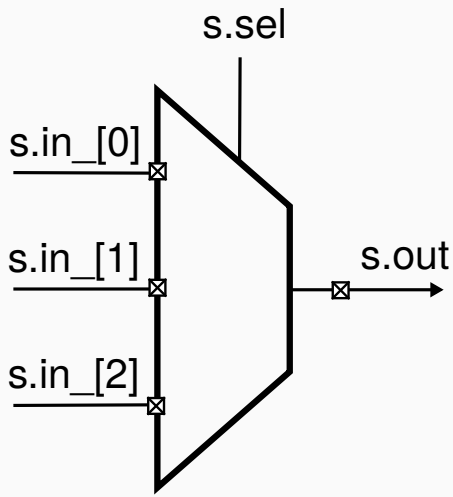


Simulating a Model

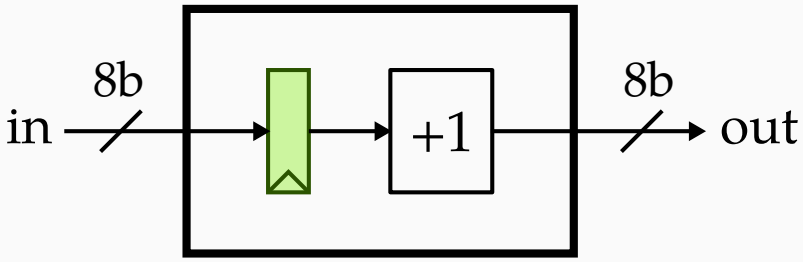
```
from pymtl3 import *
from MaxUnitFL import MaxUnitFL

model = MaxUnitFL( Bits8, ninputs=3 )
model.elaborate()
sim = model.apply(SimulationPass)
model.reset()
model.in_[0] = 2
model.in_[1] = 5
model.in_[2] = 3
model.tick()
print (model.out)
```

Multiplexer Model

```
class Mux( Component ):

  def construct( s, Type, ninputs ):
    s.in_ = [ InPort( Type ) for _ in range(ninputs) ]
    s.sel = InPort( int if Type is int else mk_bits( clog2(ninputs) ) )
    s.out = OutPort( Type )

    @s.update
    def up_mux():
      s.out = s.in_[ s.sel ]
```



Modeling a Registered Incrementer




```
from pymtl3 import *

class RegIncrRTL( Component ):

  def construct( s, dtype ):  # Constructor
    s.in_ = InPort ( dtype )
    s.out = OutPort( dtype )

    s.tmp = Wire( dtype )     # Wires modeling the register output

    @s.update_on_edge         # Sequential block modeling the register
    def seq_logic():
      s.tmp = s.in_

    @s.update                 # Concurrent block modeling incrementer
    def comb_logic():
      s.out = s.tmp + dtype(1)
```

Simulating a Registered Incrementer

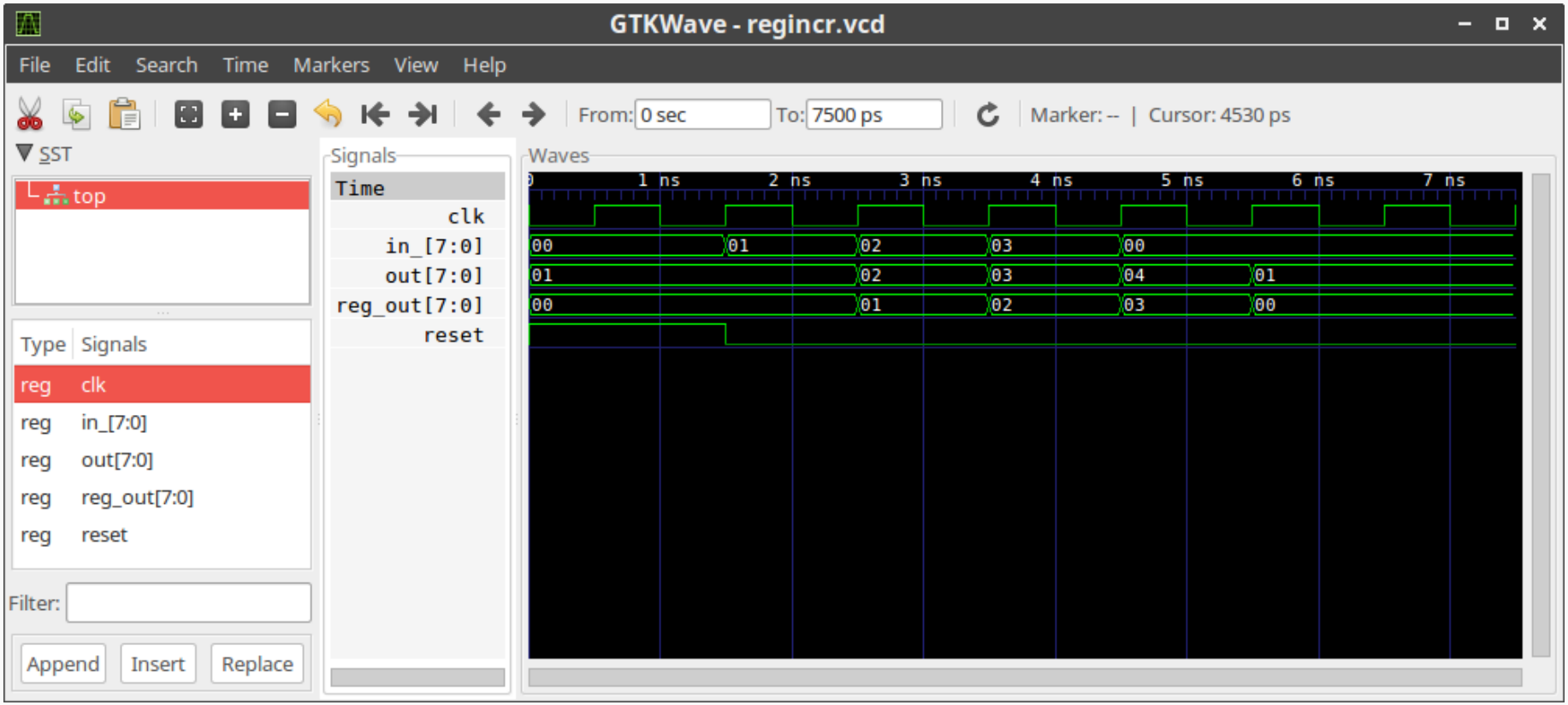
```
from RegIncrRTL import RegIncrRTL

# Get list of input values from command line
input_values = [ int(x,0) for x in argv[1:] ]

# Add three zero values to end of list of input values
input_values.extend( [0]*3 )

# Instantiate and Elaborate the model
model = RegIncrRTL(b8)
model.elaborate()
model.dump_vcd = True
model.vcd_file_name = "regincr"

# Create and reset simulator
model.apply(SimulationPass)
model.sim_reset()

for in_val in input_values:
  model.in_ = b8(in_val)
  print (" in = {}, out = {}".format( model.in_, model.out ))
  # Tick simulator one cycle
  model.tick()
```

Simulating a Registered Incrementer

```
$ python RegIncr_sim.py 1 2 3
 in = 01, out = 00
 in = 02, out = 02
 in = 03, out = 03
 in = 00, out = 04
 in = 00, out = 01
 in = 00, out = 01
```

Simulating a Registered Incrementer with Gtkwave

```
$ gtkwave ./regincr.vcd
```



Unit Testing with pytest

- pytest is a state-of-the-art Python-based testing framework
- Significantly simplifies process of writing unit tests
- Scales to large-scale sophisticated functional testing
- Powerful facilities for writing highly parameterizable unit tests

```
def add(a,b):
  return a + b

def test_add():
  assert add(2,2) == 5
```

```
$ pytest pytest_test.py
...
collected 1 item

pytest_test.py F

== FAILURES ==
__ test_add __

    def test_add():
>     assert add(2,2) == 5
E     assert 4 == 5
E      +  where 4 = add(2, 2)

pytest_test.py:5: AssertionError
== 1 failed in 0.17 seconds ==
```

Verification of a Model with Unit Tests using Pytest

- If you do not specify a file name in which the test functions are defined, Pytest searches the current directory and all subdirectories for files that begin with "test_" and end with "_test.py". If these are present, pytest automatically executes all functions starting with "test_".
- If you only want to run functions that belongs to a specific name pattern, you can use the -k switch.
  E.g. `pytest RegIncr_test.py -k simple` runs all functions that start with `test_simple`.
- By decorating your test functions with @pytest.mark.parametrize () you can keep your test functions very generic and reuse them frequently.
- For more information read the document about pytest located at studIP.

Verification of a Model with Unit Tests using Pytest

```
# test vector: list of tuples of input values and expected out values
test_vector = [(4,5), (6,7), (2,3), (8,9), (0,1)]

@pytest.mark.parametrize( "dtype", [b8])
def test_simple( dtype):

  model = RegIncr( dtype )     # instantiate the model

  model.elaborate()            # build the circuit

  model.apply(SimulationPass ) # create the simulator

  print()

  for val_in, expected_out in test_vector:
    model.in_.value = val_in
    print(model.line_trace())
    model.tick()



$ pytest RegIncrRTL_test.py -v
```



```
...
```

Verification with random numbers

```
def gen_test_vector( nbits, size=10 ):

  random.seed( 0x5750 )

  test_vector = []
  for i in range( size ):
    in_val = Bits( nbits, random.randrange( 2**nbits ) )
    test_vector.append( (in_val, in_val + 1) )

  return test_vector
```

Verification with random numbers

```
def do_random_test( ModelType, dtype ):

  model = ModelType( dtype )     # instantiate the model
  model.elaborate()              # elaborate model
  model.apply( SimulationPass )  # create the simulator

  print()
  for val_in, expected_out in gen_test_vector( dtype.nbits, 100 ):
    model.in_.value = val_in
    print(model.line_trace())
    model.tick()
    assert model.out == expected_out
  print(model.line_trace())

@pytest.mark.parametrize( "dtype", [b32] )
def test_random( dtype ):
  do_random_test( RegIncr, dtype )
```

```
$ pytest RegIncrRTL_test.py --verbose
...
RegIncrRTL_test.py::test_simple[Bits8]    PASSED [ 50%]
RegIncrRTL_test.py::test_random[Bits32]   PASSED [100%]
```

Unit Tests vs. Simulators

Unit Tests: `pytest ModelName_test.py`

- Tests that verify the simulation behavior of a model isolation
- Test functions are executed by the `pytest` testing framework
- Unit tests should always be written before simulator scripts!

Simulators: `./model-name-sim.py`

- Simulators are meant for model evaluation and stats collection
- Simulation scripts take commandline arguments for configuration
- Used for experimentation and (design space) exploration!

Line Tracing vs. VCD Dumping

Line Tracing

- Shows a single cycle per line and uses text characters to indicate state and how data moves through a system
- Provides a way to visualize the high-level behavior of a system (e.g., pipeline diagrams, transaction diagrams)
- Enables quickly debugging high-level functionality and performance bugs at the commandline
- Can be used for FL, CL, and RTL models

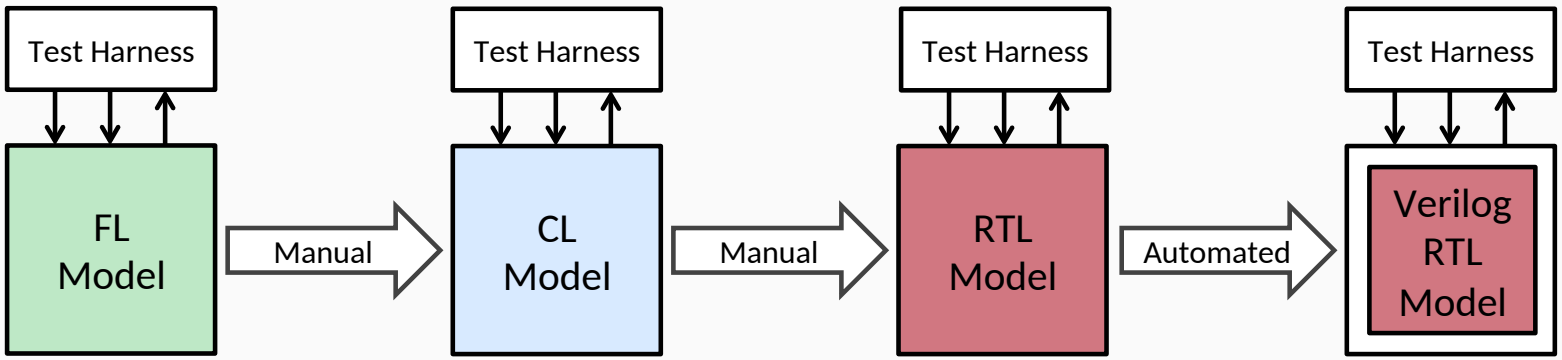
VCD Dumping

- Captures the bit-level activity of every signal on every cycle
- Requires a separate waveform viewer to visualize the signals
- Provides a much more detailed view of a design
- Mostly used for RTL models

Translating PyMTL RTL into SystemVerilog

- PyMTL models written at the register-transfer level of abstraction can be translated into SystemVerilog source using the `TranslationPass`
- Generated SystemVerilog can be used with commercial EDA toolflows to characterize area, energy, and timing

```
from pymtl3                 import *
from RegIncrRTL             import RegIncrRTL
from pymtl3.passes.yosys    import TranslationPass

model = RegIncrRTL(b8)
model.yosys_translate = True
model.elaborate()
model.apply( TranslationPass())
```



Translating PyMTL RTL into Verilog

The TranslationTool has limitations on what it can translate:

- Static elaboration can use arbitrary Python (connections $\Rightarrow$ connectivity graph $\Rightarrow$ structural Verilog)
- Concurrent logic blocks must abide by language restrictions
  - Communication between blocks only using signals (InPorts, OutPorts, and Wires)
  - *Signals* may only contain types (`Bits/BitStructs`)
  - Only pre-defined, translatable operators/functions may be used (no user-defined operators or functions)
  - Any variables that don't refer to *signals* must be integer constants

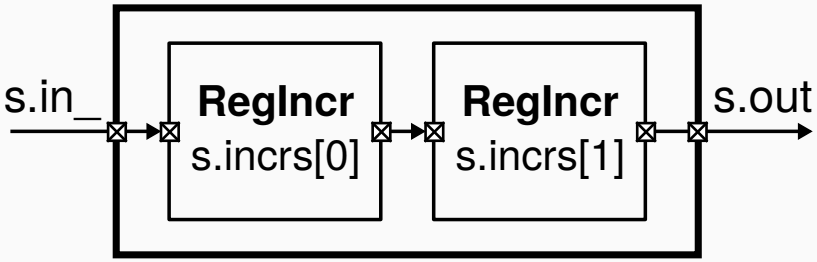
Structural Composition in PyMTL

- In PyMTL, more complex designs can be created by hierarchically composing models using structural composition
- Models are structurally composed by connecting their ports using `connect()` or `connect_pairs()` statements
  - `connect(signal1, signal2)` connects the two signals with each other. The direction is irrelevant.
  - `signal1 //= signal2` is a shortform notation for `connect()`
  - `connect_pairs(sig_1, sig_2, sig_3, sig_4,...,sig_n)` connects `sig_1` with `sig_2`, `sig_3` with `sig_4` and so on.
- Function compositions like `f(g())` not allowed, instead you have to instantiate the two components and connect all the needed signals.

```
def class example(Component):
  def construct(s, dtype):
    in_ = InPort(dtype)
    out = OutPort(dtype)
```

```
s.f = f(); s.g = g()
s.f.in_ //= s.in_
s.g.in_ //= s.f.out
s.out   //= s.g.out
```

Pipelining two Registered Incrementer




```
class RegIncrPipeline( Component ):

  def construct ( s, dtype, nstages ):
    s.in_ = InPort ( dtype )
    s.out = OutPort( dtype )

    s.incrs = [RegIncr( dtype ) for _ in range( nstages )]

    assert len( s.incrs ) > 0

    connect( s.in_, s.incrs[0].in_ )
    for i in range( nstages - 1 ):
      connect( s.incrs[i].out, s.incrs[i+1].in_ )
    connect( s.out, s.incrs[-1].out )
```

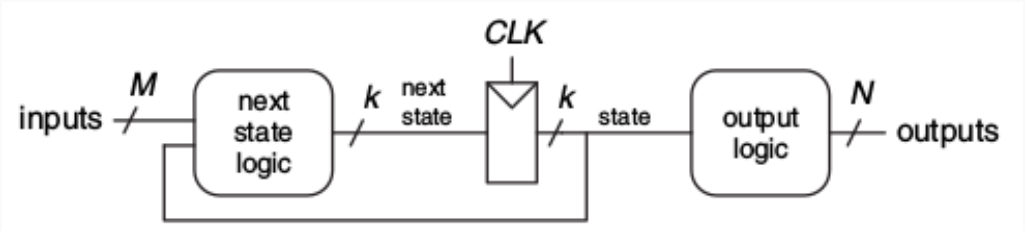
Use "_" when "`range()`" index is unused

Testing the pipelined Registered Incrementer

```
def gen_test_vectors( nstages ):

  test_vectors = [
    ( 4,  4 + nstages ),
    ( 6,  6 + nstages ),
    ( 2,  2 + nstages ),
    (15, 15 + nstages ),
    ( 8,  8 + nstages ),
    ( 0,  0 + nstages ),
    (10, 10 + nstages ),
  ]
```

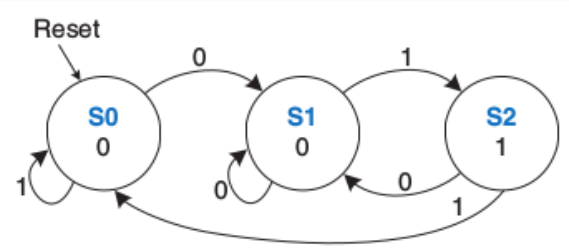
```
$ py.test RegIncrPipeline_test.py -sv

0: 04 (00 00) 00
1: 06 (05 02) 02
2: 02 (07 06) 06
3: 0f (03 08) 08
4: 08 (10 04) 04
5: 00 (09 11) 11
6: 0a (01 0a) 0a
7: 0a (0b 02) 02
8: 0a (0b 0c) 0c
PASSED
```

Finite State Machine (FSM)




```
class FSMRTL0(Component ):

  def construct( s, dtype ):
    s.a         = InPort ( dtype )
    s.y         = OutPort( dtype )

    s.state     = Wire(b2)
    s.S0        = b2(0)
    s.S1        = b2(1)
    s.S2        = b2(2)
    s.nextState = Wire(b2)

    @s.update_on_edge
    def state_memory():
      if s.reset: s.state = s.S0
      else      : s.state = s.nextState
```

Finite State Machine (FSM)




```
@s.update
def transition_logic():
  if s.state == s.S0:
    if s.a == dtype(1): s.nextState = s.S0
    else              : s.nextState = s.S1
  elif s.state == s.S1:
    if s.a == dtype(1): s.nextState = s.S2
    else              : s.nextState = s.S1
  elif s.state == s.S2:
    if s.a == dtype(1): s.nextState = s.S0
    else              : s.nextState = s.S1

@s.update
def output_logic():
  if s.state == s.S2  : s.y = dtype(1)
  else                : s.y = dtype(0)
```

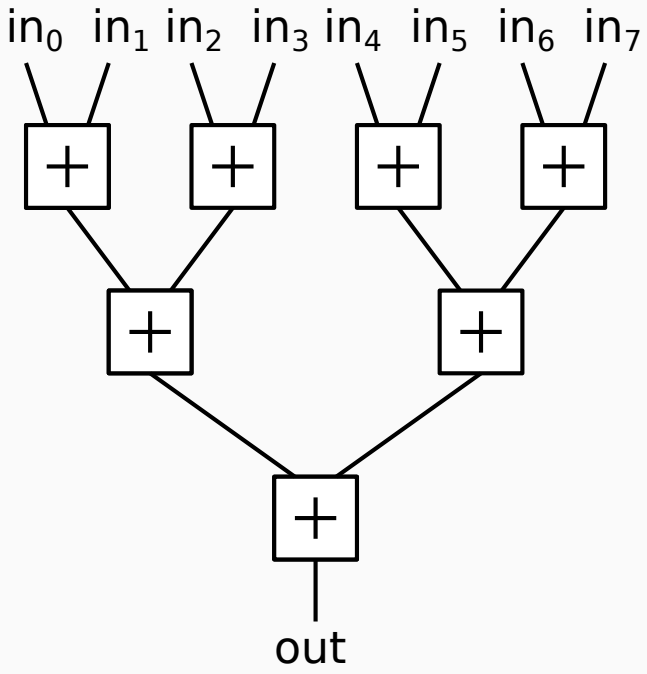
Finite State Machine (FSM)

```
@s.update_on_edge
def state_memory():
  s.state = s.S0          if s.reset else \
            s.nextState

@s.update
def transition_logic():
 s.nextState = s.S0 if (s.state == s.S0) & (s.a == dtype(1))        \
                     | (s.state == s.S2) & (s.a == dtype(1))  else \
               s.S2 if (s.state == s.S1) & (s.a == dtype(1))  else \
               s.S1

@s.update
def output_logic():
  s.y = dtype(1) if s.state == s.S2 else \
        dtype(0)
```

RAM

```
class Ram(Component):
  def construct(s, dtype, atype, ramType = 'B'):
    s.addr = InPort (atype)
    s.in_  = InPort (dtype)
    s.we   = InPort (b1)
    s.out  = OutPort(dtype)

    s.ram = [Wire(dtype) for _ in range(2**atype.nbits)]

    @s.update_on_edge
    def wr():
     if s.we:
       s.ram[s.addr] = s.in_

    if ramType == 'B': # BlockRAM, dedicated blocks of memory
      @s.update_on_edge
      def rdBram():
        s.out = s.ram[s.addr]

    if ramType == 'D': # Distributed RAM, created from slices
      @s.update
      def rdDram():
        s.out = s.ram[s.addr]
```

Recursive Defined Tree Adder (FL)

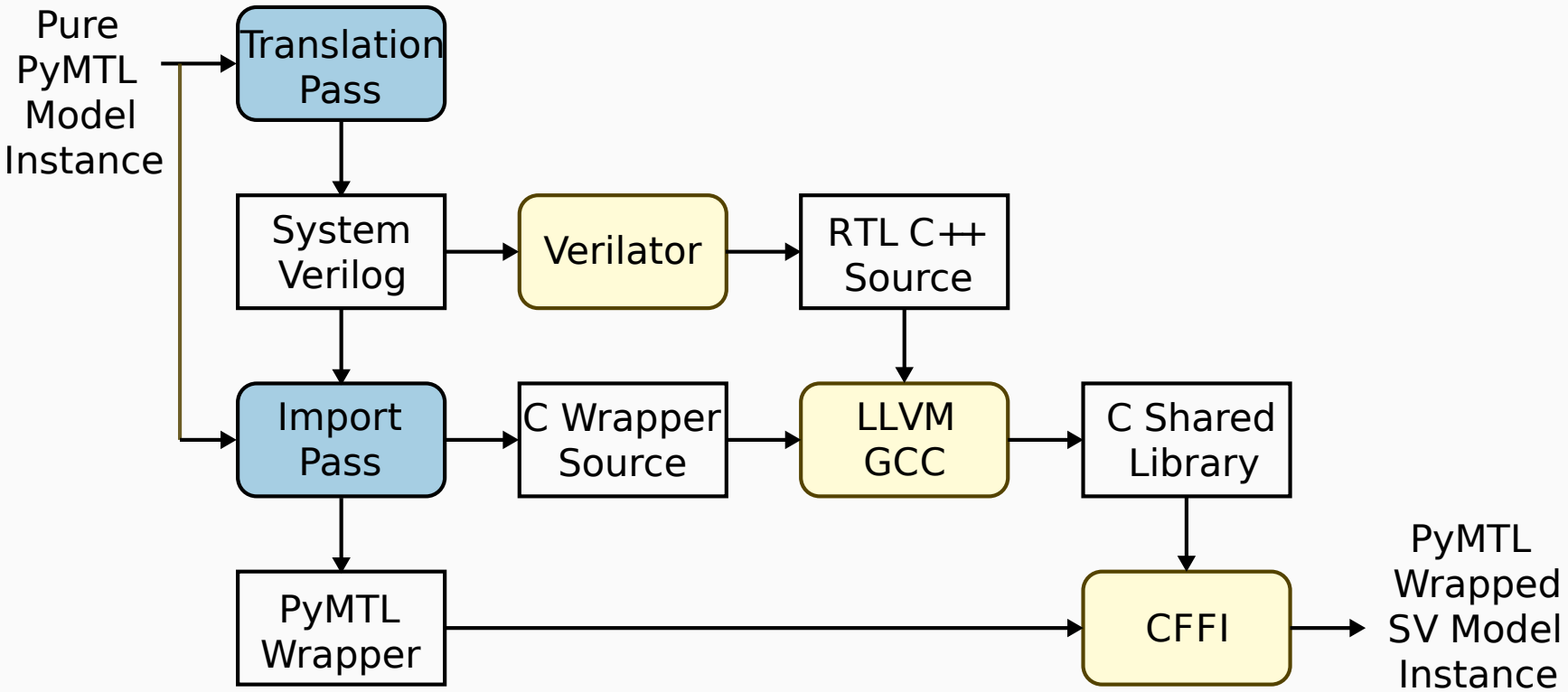



```
def reduceT(f,xs,n):
  if n == 1:
    return xs[0]
  else:
    n2 = int(n/2)
    res = f(reduceT(f,xs[:n2],n2),reduceT(f,xs[n2:],n2))
    return res
```

Recursive Defined Tree Adder (RTL)

```
class reduceT(Component):
  def construct(s,Type,n):
    s.in_ = [InPort (Type) for _ in range(n)]
    s.out = OutPort(Type)

    n2 = int(n/2)

    if n == 1:
      s.in_[0] //= s.out
      return
    else:
      s.l = reduceT(Type,n2)
      for i in range(n2):
        s.l.in_[i] //= s.in_[i]

      s.r = reduceT(Type,n2)
      for i in range(n2):
        s.r.in_[i] //= s.in_[i+n2]

      s.f = add(Type)
      s.f.inl //= s.l.out
      s.f.inr //= s.r.out
      s.f.out //= s.out

      return
```

Testing with Verilator



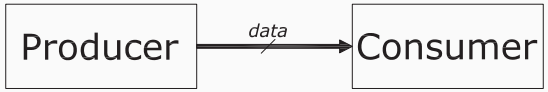
- Translation+import enables easily testing translated SystemVerilog
- Also acts like a JIT compiler for improved RTL simulation speed
- Can also import external SystemVerilog IP for co-simulation

Testing with Verilator

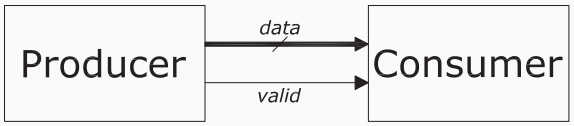
```
from pymtl3    import *
from Fibonacci import Fib
from pymtl3.passes.yosys import TranslationImportPass

# Instantiate and Elaborate the model
dut = Fib(b16)
dut.elaborate()

# Translate to SystemVerilog and import the SV model into PyMTL
dut.yosys_translate_import = True
dut = TranslationImportPass()( dut )

# Create a simulator
dut.dump_vcd        = True
dut.vcd_file_name   = "Fibonacci"
dut.elaborate()
dut.apply( SimulationPass )
dut.sim_reset()

for _ in range(10):
  print(dut.out.int(), ", ",end="")
  dut.tick()
```

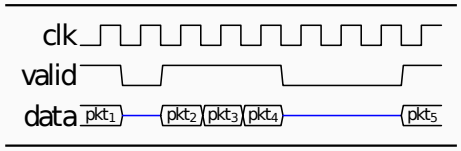
Handshake signaling for data transfer

- Producer data rate = Consumer data rate: No handshake signaling is needed

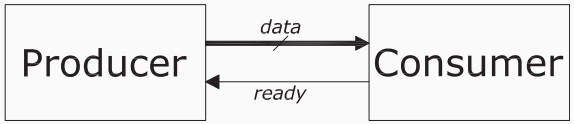


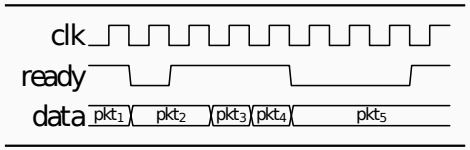
- Producer data rate < Consumer data rate: Valid signal is added to Producer



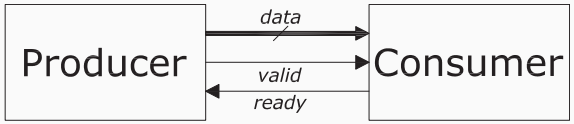


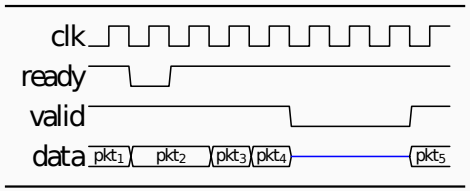
- Producer data rate > Consumer data rate: Ready signal is added to Consumer





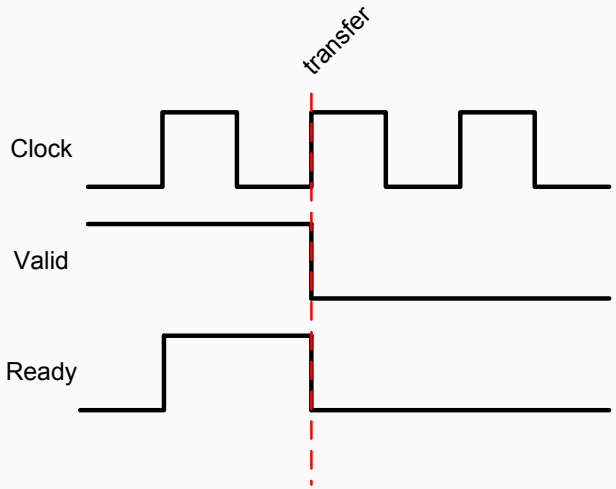
- Latency insensitive interface:

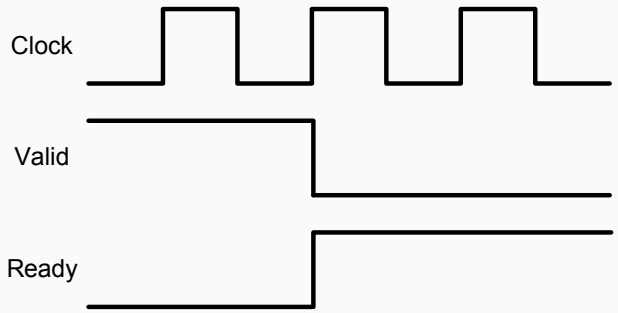




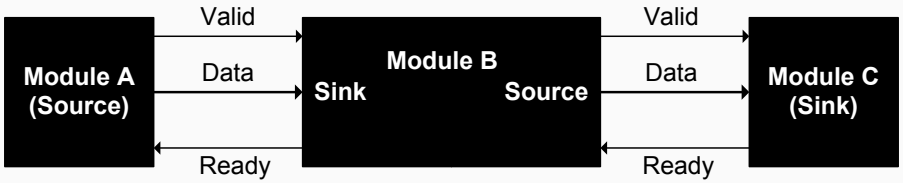
Handshake signaling for data transfer

- Valid indicates that the source has put valid data on the Data line this cycle
  - Valid is high only when data is valid
- Ready (output from the sink and input to the source) indicates that the sink is ready to receive new data
  - Ready can be asserted as soon as the sink is ready to receive new data
- A data transfer only takes place if both Valid and Ready are high before the start of the next clock cycle



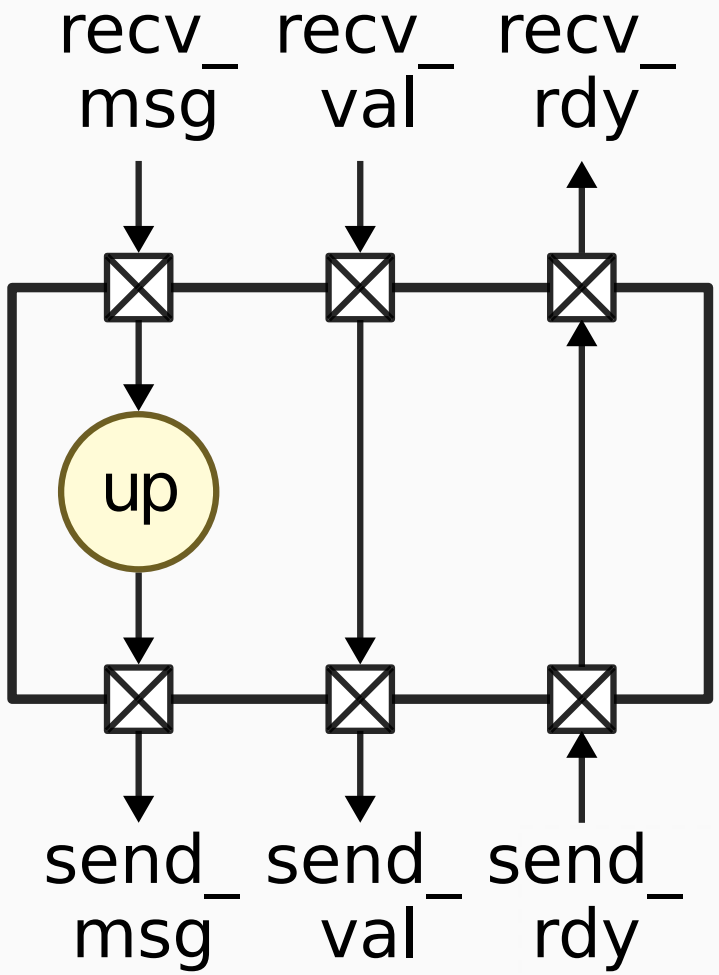


Out of phase: No transfer



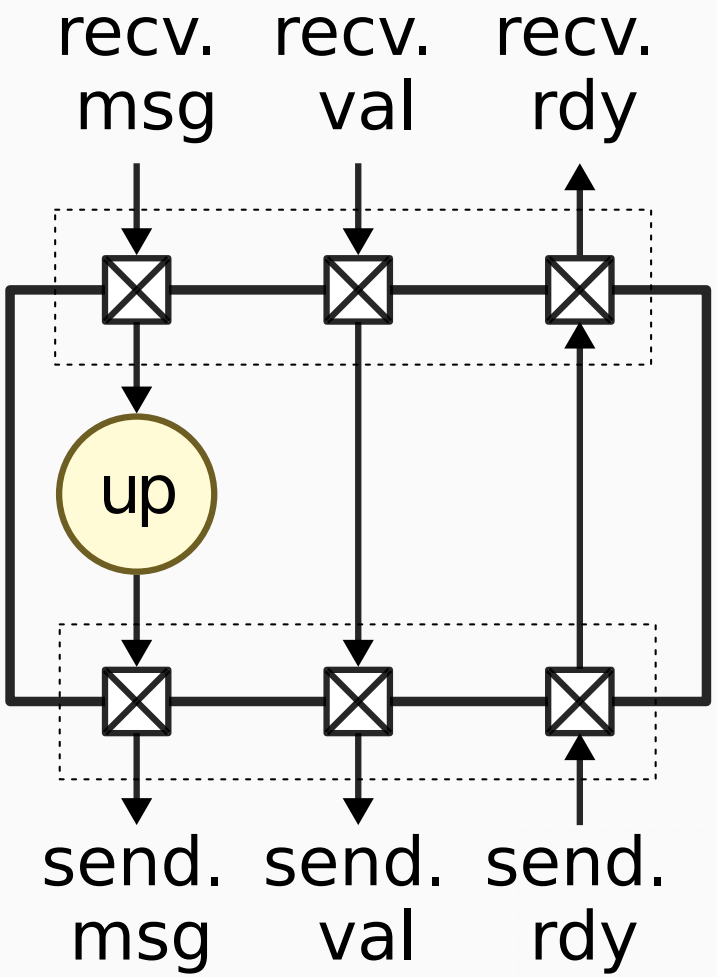
Handshake Interfaces

```
class IncrValueModular( Component ):
  def construct( s ):
    s.recv_msg = InPort ( Bits8 )
    s.recv_val = InPort ( Bits1 )
    s.recv_rdy = OutPort( Bits1 )
    s.send_msg = OutPort( Bits8 )
    s.send_val = OutPort( Bits1 )
    s.send_rdy = InPort ( Bits1 )

    s.send_val //= s.recv_val
    s.recv_rdy //= s.send_rdy

    @s.update
    def up():
      s.send_msg = s.recv_msg + b8(1)
```



Handshake Interfaces

```
class RecvIfcRTL( Interface ):
  def construct( s, Type ):
    s.msg = InPort ( Type )
    s.val = InPort ( Bits1 )
    s.rdy = OutPort( Bits1 )

class SendIfcRTL( Interface ):
  def construct( s, Type ):
    s.msg = OutPort( Type )
    s.val = OutPort( Bits1 )
    s.rdy = InPort ( Bits1 )

class IncrValModular( Component ):
  def construct( s ):
    s.recv = RecvIfcRTL( Bits8 )
    s.send = SendIfcRTL( Bits8 )

    s.send.val //= s.recv.val
    s.recv.rdy //= s.send.rdy

    @s.update
    def up():
      s.send.msg = s.recv.msg + b8(1)
```



Fletcher's Checksum Generator

- Accumulates input words
- also accumulates the cummulative sum values and
- produces a different checksum if the input data is rearranged.
- Input is a list of 16-bit words.

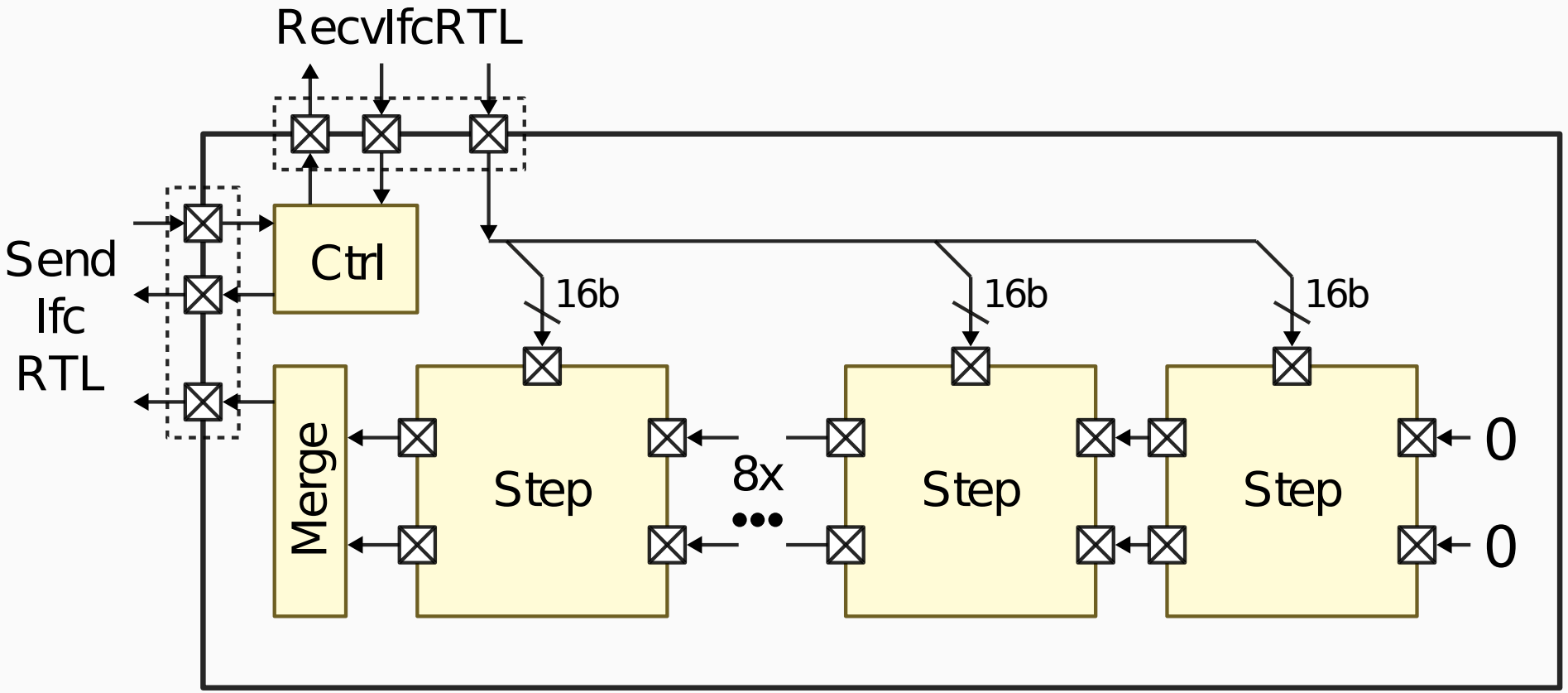
```
from pymtl3 import *

def checksum( words ):

  sum1 = b32(0)
  sum2 = b32(0)
  for word in words:
    sum1 = ( sum1 + word ) & 0xffff
    sum2 = ( sum2 + sum1 ) & 0xffff

  return ( sum2 << 16 ) | sum1
```

Testing the Checksum Generator

```
from Checksum import checksum

def test_simple( ):
    words = [ b16(x) for x in [ 1, 2, 3, 4, 5, 6, 7, 8 ] ]
    assert checksum( words ) == b32( 0x00780024 )

def test_overflow( ):
    words = [ b16(x) for x in [ 0xf000, 0xff00, 0x1000, 0x2000,
                                0x5000, 0x6000, 0x7000, 0x8000 ] ]
    assert checksum( words ) == b32( 0x3900bf00 )

def test_order( ):
    words0 = [ b16(x) for x in [ 1, 2, 3, 4, 5, 6, 7, 8 ] ]
    words1 = [ b16(x) for x in [ 1, 2, 3, 4, 8, 7, 6, 5 ] ]
    assert checksum( words0 ) != checksum( words1 )
    assert checksum( words0 ) == b32( 0x00780024 )
    assert checksum( words1 ) == b32( 0x00820024 )
```

Checksum Generator RTL

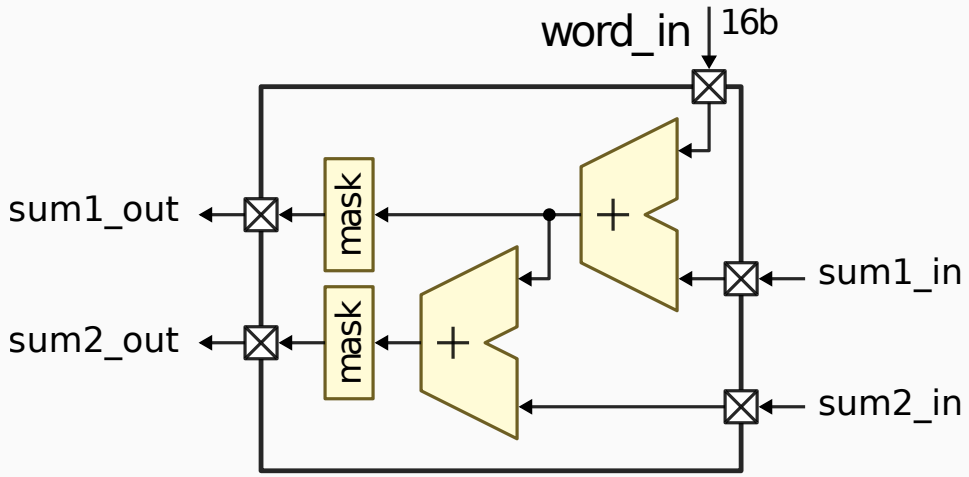
- single-cycle checksum generator on RTL
- 8 words (16 bit) as input
- latency-insensitive interfaces



Step Unit

Each step unit basically does one iteration of the algorithm (i.e., calculates both `sum1` and `sum2`).

```
class StepUnit( Component ):
  def construct( s ):

    s.word_in  = InPort ( Bits16 )
    s.sum1_in  = InPort ( Bits32 )
    s.sum2_in  = InPort ( Bits32 )
    s.sum1_out = OutPort( Bits32 )
    s.sum2_out = OutPort( Bits32 )
```




```
    @s.update
    def up_step():
      temp1      = b32(s.word_in) + s.sum1_in
      s.sum1_out = temp1 & b32(0xffff)
      temp2      = s.sum1_out + s.sum2_in
      s.sum2_out = temp2 & b32(0xffff)
```

Checksum Generator (RTL)

```
from pymtl3.stdlib.ifcs import RecvIfcRTL, SendIfcRTL
from pymtl3.stdlib.rtl.queues import PipeQueueRTL
from StepUnit import *

class ChecksumRTL( Component ):

  def construct( s ):

    # Interface
    s.recv  = RecvIfcRTL( Bits128 )
    s.send  = SendIfcRTL( Bits32  )

    # Component
    s.words = [ Wire( Bits16 ) for _ in range( 8 ) ]
    s.sum1  = Wire( Bits32 )
    s.sum2  = Wire( Bits32 )

    # Instantiate a FIFO / queue with one 128-bit entry
    s.in_q  = PipeQueueRTL( Bits128, num_entries = 1 )

    # Instantiate 8 step units
    s.steps = [ StepUnit() for _ in range( 8 ) ]
```

Checksum Generator (RTL)

```
# Register input
s.in_q.enq //= s.recv

# Decompose input message into 8 words
for i in range( 8 ):
  s.words[i] //= s.in_q.deq.msg[i*16:(i+1)*16]

# Connect step units

for i in range( 8 ):
  s.steps[i].word_in //= s.words[i]
  if i == 0:
    s.steps[i].sum1_in //= b32(0)
    s.steps[i].sum2_in //= b32(0)
  else:
    s.steps[i].sum1_in //= s.steps[i-1].sum1_out
    s.steps[i].sum2_in //= s.steps[i-1].sum2_out

s.sum1 //= s.steps[-1].sum1_out
s.sum2 //= s.steps[-1].sum2_out
```

Checksum Generator (RTL)

```
    # Control Unit
    @s.update
    def up_rtl_send():
      # data are valid (en = true) only if the queue is not empty
      s.send.en     = s.in_q.deq.rdy & s.send.rdy
      s.in_q.deq.en = s.in_q.deq.rdy & s.send.rdy

    # Merge
    @s.update
    def up_rtl_sum():
      s.send.msg = ( s.sum2 << 16 ) | s.sum1

  def line_trace( s ):
    return "{}(){}".format( s.recv, s.send )
```

Testing the Checksum Generator (RTL)

```
def checksum_rtl( words ):
  # Convert list of 8 16-bit words to one 128bit word
  bits_in = words_to_b128( words )
  # bits_in[5] = 0  # Inject a bug forcing bits_in[5] to be zero
  # Create a simulator
  dut = ChecksumRTL()
  dut.elaborate()
  dut.dump_vcd        = True
  dut.vcd_file_name   = "Checksum"
  dut.apply( SimulationPass )
  dut.sim_reset()

  dut.send.rdy = b1(1)
  while not dut.recv.rdy: # Wait until ready to receive input
    dut.recv.en = b1(0); dut.tick()

  dut.recv.en = b1(1)
  dut.recv.msg = bits_in; dut.tick()

  while not dut.send.en:  # Wait until send the message
    dut.recv.en = b1(0); dut.tick()

  return dut.send.msg
```

Testing the Checksum Generator (RTL)

```
def test_simple( ):
    words = [ b16(x) for x in [ 1, 2, 3, 4, 5, 6, 7, 8 ] ]
    assert checksum_rtl( words ) == b32( 0x00780024 )
    assert checksum_rtl( words ) == checksum( words )

def test_overflow( ):
    words = [ b16(x) for x in [ 0xf000, 0xff00, 0x1000, 0x2000,
                                0x5000, 0x6000, 0x7000, 0x8000 ] ]
    assert checksum_rtl( words ) == b32( 0x3900bf00 )
    assert checksum_rtl( words ) == checksum( words )

def test_order( ):
    words0 = [ b16(x) for x in [ 1, 2, 3, 4, 5, 6, 7, 8 ] ]
    words1 = [ b16(x) for x in [ 1, 2, 3, 4, 8, 7, 6, 5 ] ]
    assert checksum_rtl( words0 ) != checksum_rtl( words1 )
    assert checksum_rtl( words0 ) == b32( 0x00780024 )
    assert checksum_rtl( words1 ) == b32( 0x00820024 )
    assert checksum_rtl( words0 ) != checksum( words1 )
    assert checksum_rtl( words0 ) == checksum( words0 )
    assert checksum_rtl( words1 ) == checksum( words1 )
```

Testing the Checksum Generator (RTL)

```
$ pytest ChecksumRTL_test.py -sv

ChecksumRTL_test.py::test_simple PASSED
ChecksumRTL_test.py::test_overflow PASSED
ChecksumRTL_test.py::test_order PASSED
```

Property-Based Random Testing

- First popularized with the Haskell QuickCheck library
- In Python using Hypothesis
- Dynamically generate test vectors that satisfy constraints (strategies)
- Auto-shrinking (generate minimal failing test cases)
- Failing test cases stored in database (start next test cycle with failing test cases)
- Excellent documentation: https://hypothesis.readthedocs.io

Property-Based Random Testing (PBRT)

```
# golden reference model
def incr_8bit( x ):
  return b8(x) + b8(1)

# design under test
class Incr8bitFL( Component ):
  def construct( s ):
    s.in_ = InPort ( Bits8 )
    s.out = OutPort( Bits8 )
    @s.update
    def up():
      tmp   = s.in_ + b8(1)
      s.out = tmp & b8(0xef)

def incr_8bit_fl( x ):
  incr = Incr8bitFL()
  incr.apply( SimpleSim )
  incr.in_ = x
  incr.tick()
  return incr.out
```

Property-Based Random Testing

```
from hypothesis                 import given
from hypothesis                 import strategies as st
import pymtl3.datatypes.strategies as pst
from Incr8bit                   import *
from pymtl3                     import *

def test_simple():
  assert incr_8bit_fl( b8(2) ) == b8(3)

@given( x=pst.bits(8) )
def test_hypothesis(x):
  print("x =",x)
  assert incr_8bit_fl(x) == incr_8bit(x)
```

- The @given decorator turns test function into a parametrized one over a wide range of matching data from that strategy.

Property-Based Random Testing (Stateless)

```python
from functools import reduce
from pymtl3 import *

#-------------------------------------------------------------------------
# Converts a list of Bits16 to Bits128
#-------------------------------------------------------------------------
def words_to_b128( words ):
  assert len( words ) == 8
  bits = reduce( lambda x, y: concat( y, x ), words )
  return bits

#-------------------------------------------------------------------------
# Converts Bits128 to a list of Bits16
#-------------------------------------------------------------------------
def b128_to_words( bits ):
  assert bits.nbits == 128
  words = [ bits[i*16:(i+1)*16] for i in range( 8 ) ]
  return words
```

Property-Based Random Testing (Stateless)

```
from hypothesis             import given
from hypothesis.strategies import text
from hypothesis             import strategies as st
import pymtl3.datatypes.strategies as pst
from pymtl3                 import *
from utils                  import *

@given( words=st.lists( pst.bits(16), min_size=8, max_size=8 ) )
def test_hypothesis( words ):
  print(words)
  assert b128_to_words( words_to_b128( words ) ) == words
```

```
$ pytest PBRT.py -sv
convert_test.py::test_hypothesis
[Bits16(0x0000), Bits16(0x0000), Bits16(0x0000), ... , Bits16(0x0000)]
[Bits16(0xff6d), Bits16(0xd8b2), Bits16(0xa839), ... , Bits16(0x167d)]
...
PASSED
```

Property-Based Random Testing (Stateless)

```
import hypothesis
from hypothesis import strategies as st
import pymtl3.datatypes.strategies as pst

@hypothesis.settings( deadline=None )
@hypothesis.given( words = st.lists( pst.bits(16), min_size=8, max_size=8
def test_hypothesis( words ):
  print( [ int(x) for x in words ] )
  assert checksum_rtl( words ) == checksum( words )
```

if you want to observe the shrinking process, uncomment line 11 in ChecksumRTL_test.py

```
def checksum_rtl( words ):
  # Convert list of 8 16-bit words to one 128bit word
  bits_in = words_to_b128( words )
  # bits_in[5] = 0  # Inject a bug forcing bits_in[5] to be zero
```

Bitstructs

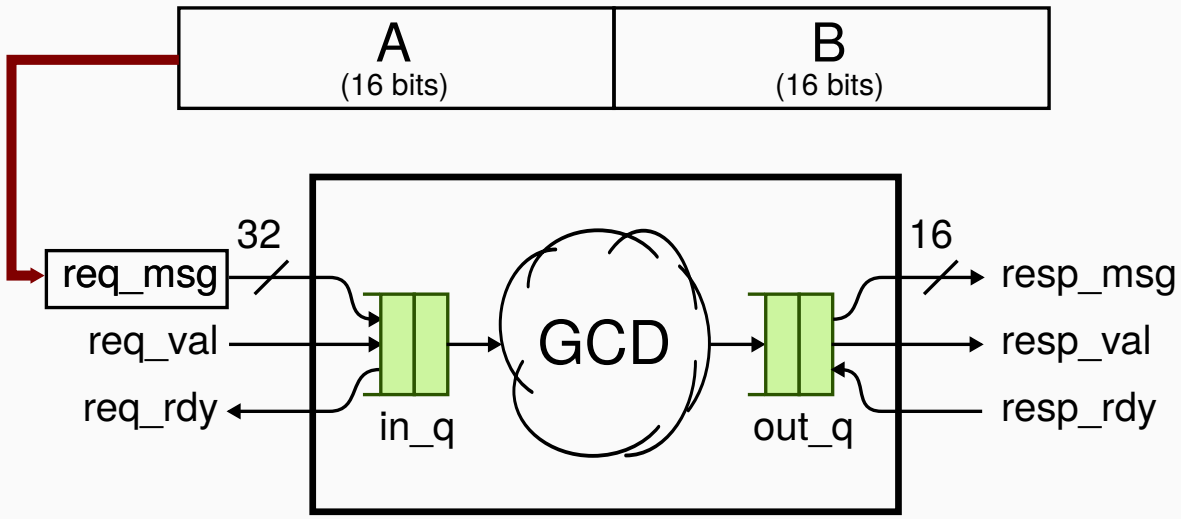
```
from pymtl3.datatypes.BitStruct import *

class Ops( BitStruct ):
  fields = [('x', Bits8), ('y', Bits8)]; nbits = 16

  def __init__(s, x = Bits8(), y = Bits8()):
    s.x = x; s.y = y

class Multiplier( Component ):
  def construct( s ):

    s.in_ = InPort ( Ops )
    bits  = Ops.nbits # Ops.field_nbits('x') + Ops.field_nbits('y')
    s.out = OutPort( mk_bits(bits) )

    s.x  = Wire( mk_bits(bits) )
    s.y  = Wire( mk_bits(bits) )

    @s.update
    def mul():
      s.x = zext(s.in_.x, bits)
      s.y = zext(s.in_.y, bits)
      s.out = s.x * s.y
```

Bitstructs

```
model = Multiplier()
model.yosys_translate=True
model.elaborate()
model.apply(TranslationPass())
model.apply(SimulationPass)
model.in_.x = b8(117)
model.in_.y = b8(159)
model.tick()
print(model.line_trace())
```

GCD Unit with latency insensitive I/O

- 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.



Modeling queues on functional level (FL)

Python Collections.deque

Create `collections.deque()` creates a list optimized for inserting and removing items

Insert To add an element to the right of the deque, you have to use `append()` method. `appendleft()` does same for the left

Remove to remove an element from left, you can use `popleft()`. `pop()` does the same for the right

Clearing If you want to remove all elements from a deque, you can use `clear()` function

Counting If you want to find the count of a specific element, use `count()` function

Modeling of a GCD Unit

```
  # Concurrent block

  @s.update
  def logic():

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

    result = gcd( req_msg.a, req_msg.b )

    # append result to response queue
    s.resp_q.append( result )

# Line tracing

def line_trace( s ):
  return "{}(){}".format( s.req, s.resp )
```

Testing Latency Insensitive Models

- 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.