PyMTL Tutorial Schedule

1:45 – 2:00pm	Virtual Machine Installation and Setup
2:00 – 2:15pm	*Presentation:* PyMTL Overview
2:15 – 2:55pm	*Part 1:* PyMTL Basics
2:55 – 3:30pm	*Part 2:* Multi-Level Modeling with PyMTL
3:30 – 4:00pm	Coffee Break
4:00 – 4:45pm	*Part 3:* Processor Modeling with PyMTL
4:45 – 5:30pm	*Part 4:* Multi-Level Composition in PyMTL

`Bits` Classes for Fixed-Bitwidth Values

PyMTL Bits 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

★ Task 1.1: Experiment with Bits ★

```
% mkdir -p $HOME/pymtl-tut/examples/build
% cd $HOME/pymtl-tut/examples/build
% python
```

```
>>> from pymtl3 import *
>>> a = Bits8( 5 )
>>> b = Bits8( 3 )
>>> a + b
Bits8( 0x08 )
>>> a - b
Bits8( 0x02 )
>>> a | b
Bits8( 0x07 )
>>> a & b
Bits8( 0x01 )
>>> a + b8(1)
Bits8( 0x06 )
```

```
>>> c = concat( a, b )
>>> c
Bits16( 0x0503 )
>>> c[0:8]
Bits8( 0x03 )
>>> c[8:16]
Bits8( 0x05 )
>> d = Bits256( 0xff )
NameError: 'Bits256' is not defined
>> Bits256 = mk_bits(256)
>> d = Bits256( 0xff )
>> d
Bits256( 0x00000000000000...00ff )
>> exit()
```

Unit Testing with `pytest`

- `pytest` is a state-of-the-art Python-based testing framework
  - Enables boilerplate-less small-scale testing
  - Scales to large-scale sophisticated functional testing
  - Provides extensive configuration and plugin support

- Example simple test

```
1 def add( a, b ):
2   return a + b
3
4 def test_add():
5   assert add(2,2) == 5
```

```
% pytest example_test.py
========== test session starts ==========
...
collected 1 item
example_test.py F                    [100%]
=============== FAILURES ===============
_____________ test_answer _____________
def test_add():
>       assert add(2,2) == 5
E       assert 4 == 5
E        +  where 4 = add(2, 2)

example_test.py:5: AssertionError
======= 1 failed in 0.12 seconds ========
```

★ Task 1.2: Write a Simple Function and Unit Test ★

```
% cd $HOME/pymtl-tut/examples/build
% geany ../ex01_basics/incr_test.py
```

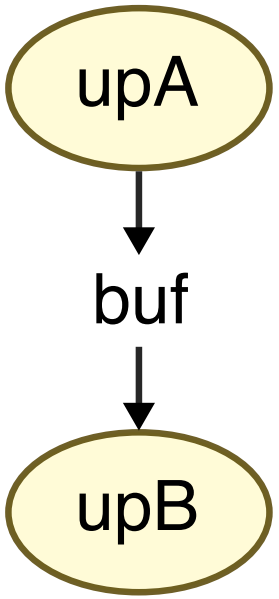
```
1 def incr_8bit( x ):
2   return b8(x) - b8(1)

3 def test_incr_8bit_simple():
4   assert incr_8bit( b8(2) ) == b8(3)

5 def test_incr_8bit_overflow():
6   assert incr_8bit( b8(0xff) ) == b8(0)
```

```
% pytest ../ex01_basics/incr_test.py
% pytest ../ex01_basics/incr_test.py -k simple
% pytest ../ex01_basics/incr_test.py -k simple --tb=long
```

- Use `-k pattern` to test cases which match the given `pattern`
- Use `--tb=long` to display a more detailed traceback
- Fix the bug and then rerun `pytest`

Update Blocks Communicating via Variables

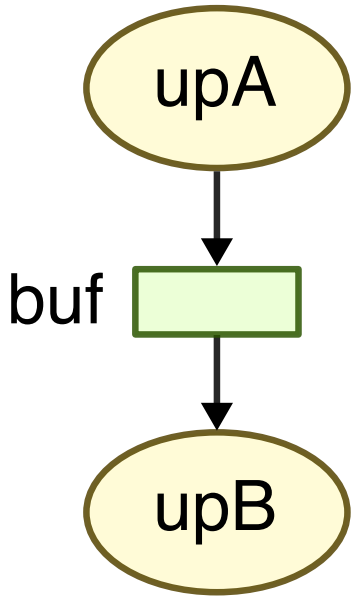
```
1  class FooPyVars( Component ):
2   def construct( s ):
3     s.count = b8(10)
4     s.buf   = b8(0)
5     @s.update
6     def upA():
7       s.buf = s.count
8       s.count += b8(10)
9     @s.update
10    def upB():
11      print(s.buf)
12    s.add_constraints( U(upA) < U(upB) )
```



- `upX < upY` means `upX` should "happen before" `upY` within a cycle
- In this example ...
  - `upA < upB` will model combinational communication
  - `upB < upA` will model an edge between update blocks

Update Blocks Communicating via Wires

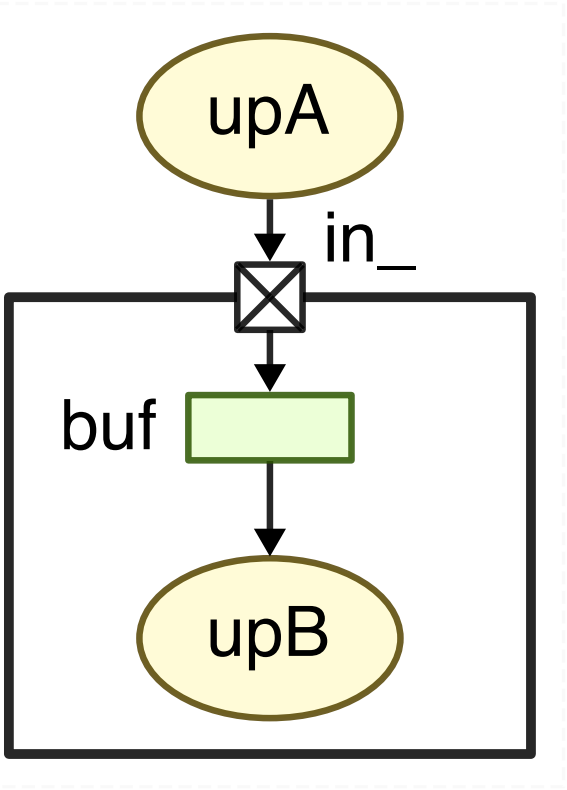
```
1  class FooWires( Component ):
2   def construct( s ):
3
4     s.count = Wire( Bits8 )
5     s.buf   = Wire( Bits8 )
6
7     @s.update
8     def upA():
9       s.buf = s.count
10      s.count += b8(10)
11
12    @s.update
13    def upB():
14      print(s.buf)
```



- Wires enable the framework to automatically infer the constraints
- `s.update` produces an implicit `upA < upB` constraint
- `s.update_on_edge` produces an implicit `upB < upA` constraint

Implementing Modularity with Value Ports

```
1  class FooValuePorts( Component ):
2   def construct( s ):
3
4     s.in_ = InPort ( Bits8 )
5     s.buf = Wire   ( Bits8 )
6
7     s.connect( s.in_, s.buf )
8
9     @s.update
10    def upB():
11      print(s.buf)
12
13 class FooTestBench( Component ):
14  def construct( s ):
15
16    s.count = Wire( Bits8 )
17    s.foo   = FooValuePorts()
18
19    @s.update
20    def upA():
21      s.foo.in_ = s.count
22      s.count += b8(10)
```



Simulating a PyMTL Component

```
1  class FooWires( Component ):
2   def construct( s ):
3
4     s.count = Wire( Bits8 )
5     s.buf   = Wire( Bits8 )
6
7     @s.update
8     def upA():
9       s.buf = s.count
10      s.count += b8(10)
11
12    @s.update
13    def upB():
14      print(s.buf)
15
16 def test_foo_wires():
17   incr = FooWires()
18   incr.elaborate()
19   incr.apply( SimpleSim )
20   for i in range(6):
21     incr.tick()
```

Initialize – Calls `__init__()` which is defined by the framework

Elaborate – Calls `construct()` and uses reflection to create metadata to be used in passes

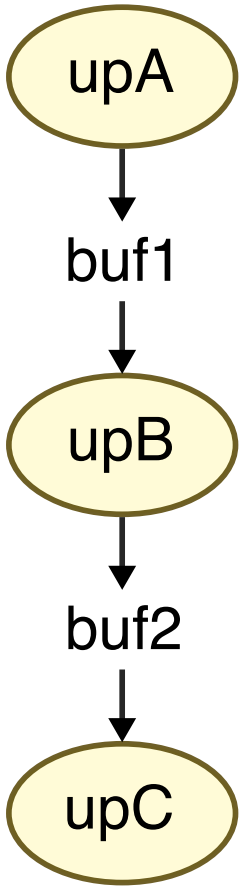
Apply Passes – `SimpleSim` pass queries metadata to analyze all update blocks, create a valid schedule, and monkey patch new methods (`tick()`)

Simulate – Simulate one cycle by executing update blocks according to the pre-determined schedule

★ Task 1.3: Communicating via Variables ★

```
% cd $HOME/pymtl-tut/examples/build
% geany ../ex01_basics/IncrPyVars_test.py
```

```
1  class IncrPyVars( Component ):
2   def construct( s ):
3     s.incr_in = b8(10); s.incr_out = b8(0)
4     s.buf1 = b8(0); s.buf2 = b8(0)

5     @s.update
6     def upA():
7       s.buf1 = s.incr_in
8       s.incr_in += b8(10)

9     @s.update
10    def upB():
11      s.buf2 = s.buf1 + b8(1)

12    @s.update
13    def upC():
14      s.incr_out = s.buf2

15    s.add_constraints( U(upA) < U(upB), U(upB) < U(upC) )
```

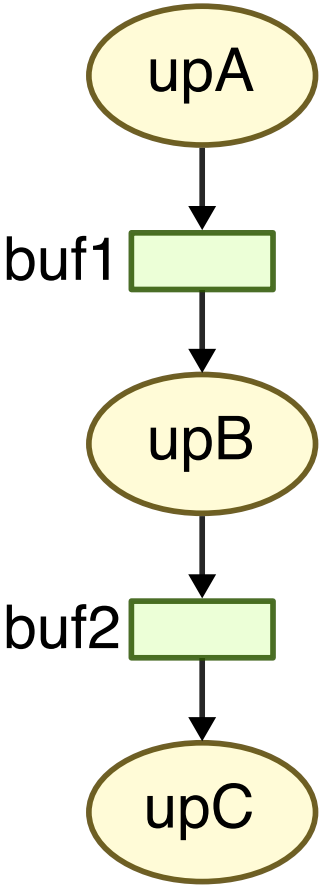



```
% pytest ../ex01_basics/IncrPyVars_test.py -s
```

★ Task 1.4: Communicating via Wires ★

```
% cd $HOME/pymtl-tut/examples/build
% geany ../ex01_basics/IncrWires_test.py
```

```
1  class IncrWires( Component ):
2   def construct( s ):
3     s.incr_in = b8(10); s.incr_out = b8(0)
4     s.buf1 = Wire( Bits8 )
5     s.buf2 = Wire( Bits8 )

6     @s.update
7     def upA():
8       s.buf1 = s.incr_in
9       s.incr_in += b8(10)

10    @s.update
11    def upB():
12      s.buf2 = s.buf1 + b8(1)

13    @s.update
14    def upC():
15      s.incr_out = s.buf2
```

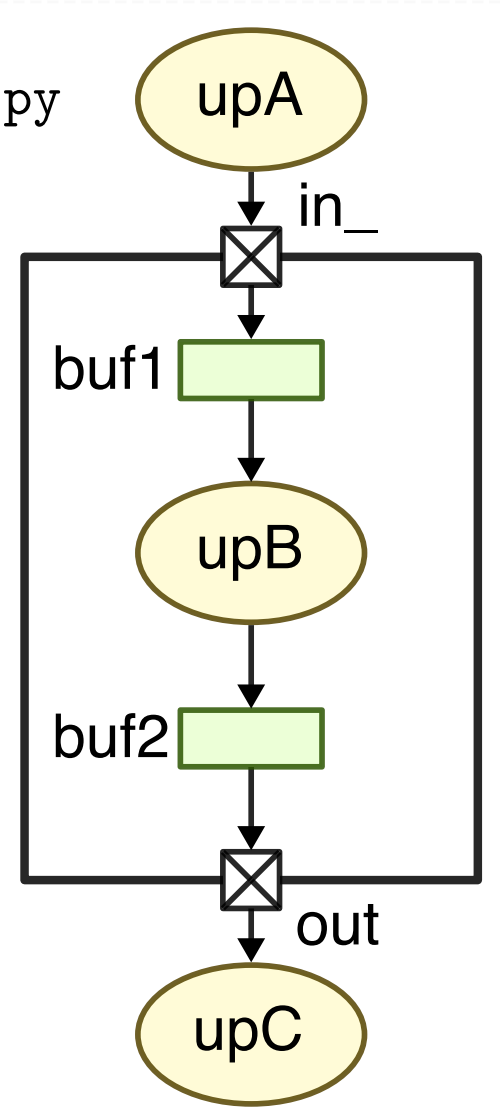



```
% pytest ../ex01_basics/IncrWires_test.py -s
```

★ Task 1.5: Modularity with Value Ports ★

```
% cd $HOME/pymtl-tut/examples/build
% geany ../ex01_basics/IncrValueModular_test.py
```

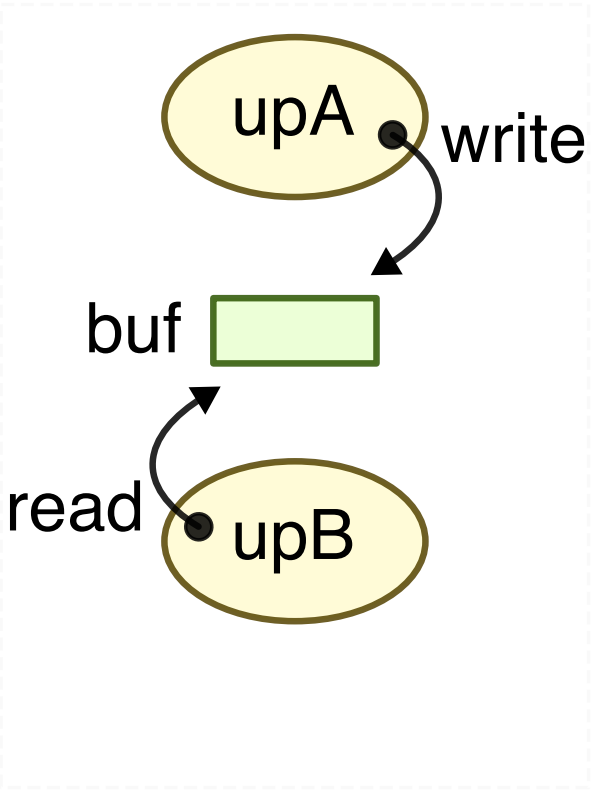
```
1  class IncrValueModular( Component ):
2   def construct( s ):
3     s.in_  = InPort ( Bits8 )
4     s.out  = OutPort( Bits8 )
5     s.buf1 = Wire( Bits8 )
6     s.buf2 = Wire( Bits8 )
7     s.connect( s.in_, s.buf1 )
8     s.connect( s.out, s.buf2 )
9     @s.update
10    def upB():
11      s.buf2 = s.buf1 + b8(1)
12 class IncrTestBench( Component ):
13  def construct( s ):
14    ...
15    s.incr = IncrValueModular()
```




```
% pytest ../ex01_basics/IncrValueModular_test.py -s
```

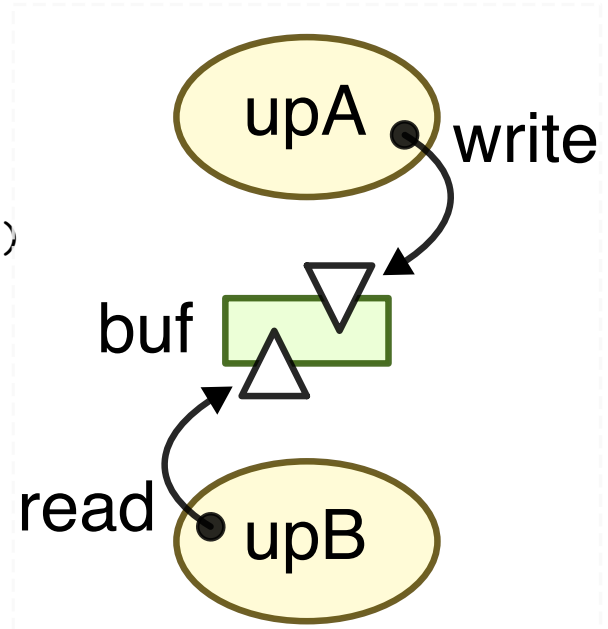
Update Blocks Communicating via Objects

```
1  class Buffer( object ):
2   def __init__( s ):
3     s.data = b8(0)
4   def write( s, value ):
5     s.data = value
6   def read( s ):
7     return s.data
8  class FooPyObjs( Component ):
9   def construct( s ):
10    s.count = b8(10)
11    s.buf   = Buffer()
12    @s.update
13    def upA():
14      s.buf.write( s.count )
15      s.count += b8(10)
16    @s.update
17    def upB():
18      print(s.buf.read())
19    s.add_constraints( U(upA) < U(upB) )
```



Update Blocks Communicating via Method Ports

```
1  class Buffer( Component ):
2   def construct( s ):
3     s.write = CalleePort( method=s.write_ )
4     s.read  = CalleePort( method=s.read_  )
5     s.add_constraints( M(s.write) < M(s.read) )
6     s.data = b8(0)
7   def write_( s, value ):
8     s.data = value
9   def read_( s ):
10    return s.data
11 class FooMethodPorts( Component ):
12  def construct( s ):
13    s.count = b8(10)
14    s.buf   = Buffer()
15    @s.update
16    def upA():
17      s.buf.write( s.count )
18      s.count += b8(10)
19    @s.update
20    def upB():
21      print(s.buf.read())
```

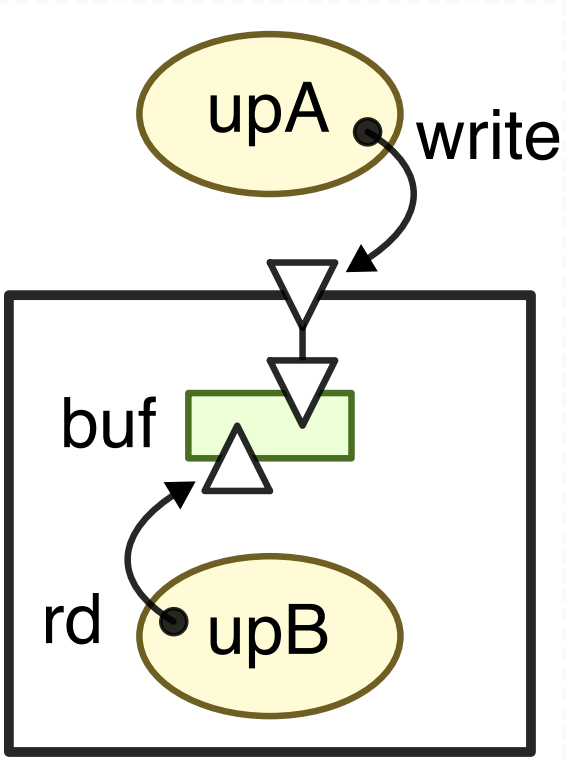


Method-Port Syntatic Sugar

```
1  class Buffer( Component ):
2   def construct( s ):
3     s.add_constraints(
4       M(s.write) < M(s.read) )
5     s.data = b8(0)
6   @method_port
7   def write( s, value ):
8     s.data = value
```

Implementing Modularity with Method Ports

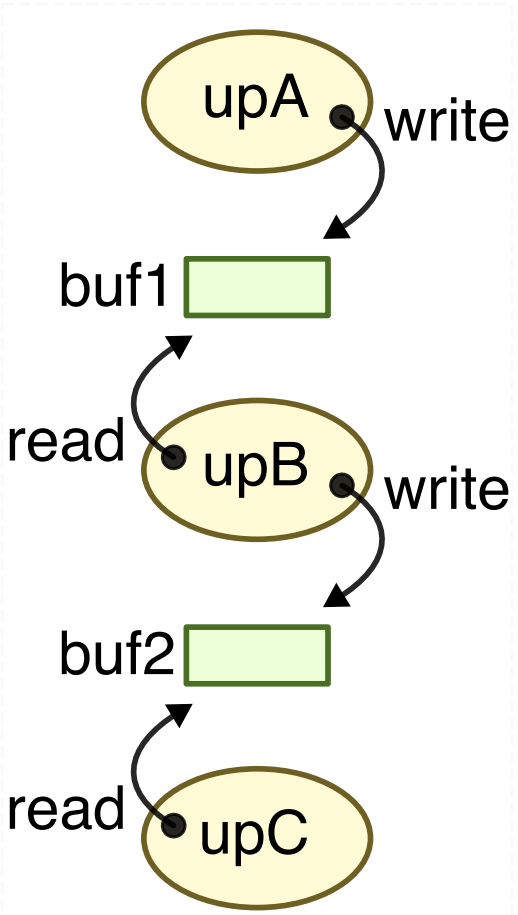
```
1  class FooMethodModular( Component ):
2   def construct( s ):
3     s.write = CalleePort()
4     s.buf   = Buffer()
5     s.connect( s.write, s.buf.write )
6     @s.update
7     def upB():
8       print(s.buf.read())
9  class FooTestBench( Component ):
10  def construct( s ):
11    s.count = b8(10)
12    s.foo   = FooMethodModular()
13    @s.update
14    def upA():
15      s.foo.write( s.count )
16      s.count += 10
```



★ Task 1.6: Communicating via Objects ★

```
% cd $HOME/pymtl-tut/examples/build
% geany ../ex01_basics/IncrPyObjs_test.py
```

```
1  class IncrPyObjs( Component ):
2   def construct( s ):
3     s.incr_in = b8(10); s.incr_out = b8(0)
4     s.buf1 = Buffer(); s.buf2 = Buffer()

5     @s.update
6     def upA():
7       s.buf1.write( s.incr_in )
8       s.incr_in += b8(10)

9     @s.update
10    def upB():
11      tmp = s.buf1.read()
12      s.buf2.write( tmp + b8(1) )

13    @s.update
14    def upC():
15      s.incr_out = s.buf2.read()

16    s.add_constraints( U(upA) < U(upB), U(upB) < U(upC) )
```

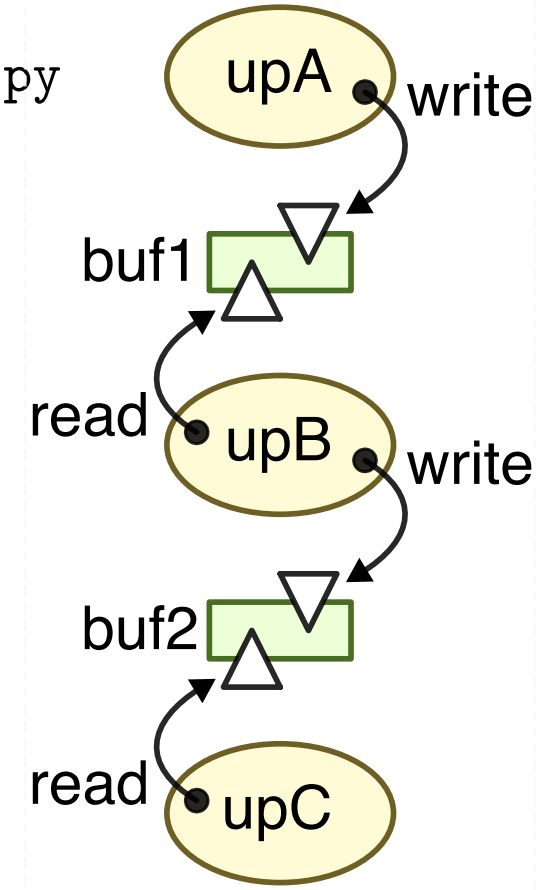



```
% pytest ../ex01_basics/IncrPyObjs_test.py -s
```

★ Task 1.7: Communicating via Method Ports ★

```
% cd $HOME/pymtl-tut/examples/build
% geany ../ex01_basics/IncrMethodPorts_test.py
```

```
1  class Buffer( Component ):
2   def construct( s ):
3     s.data = b8(0)
4
5     s.add_constraints(
6       M(s.write) < M(s.read) )
7
8   @method_port
9   def write( s, value ):
10    s.data = value
11
12  @method_port
13  def read( s ):
14    return s.data
```

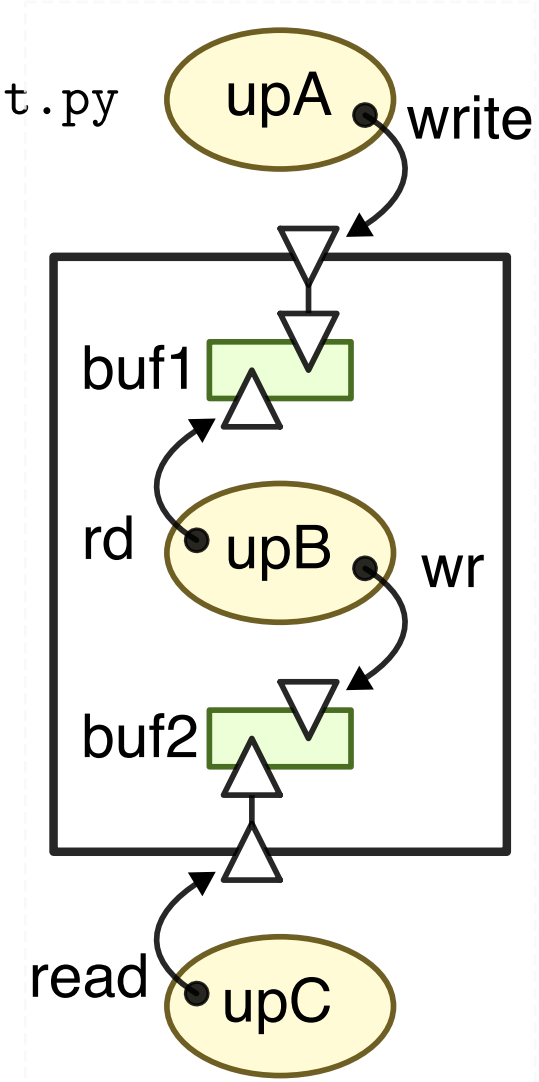



```
% pytest ../ex01_basics/IncrMethodPorts_test.py -s
```

★ Task 1.8: Modularity with Method Ports ★

```
% cd $HOME/pymtl-tut/examples/build
% geany ../ex01_basics/IncrMethodModular_test.py
```

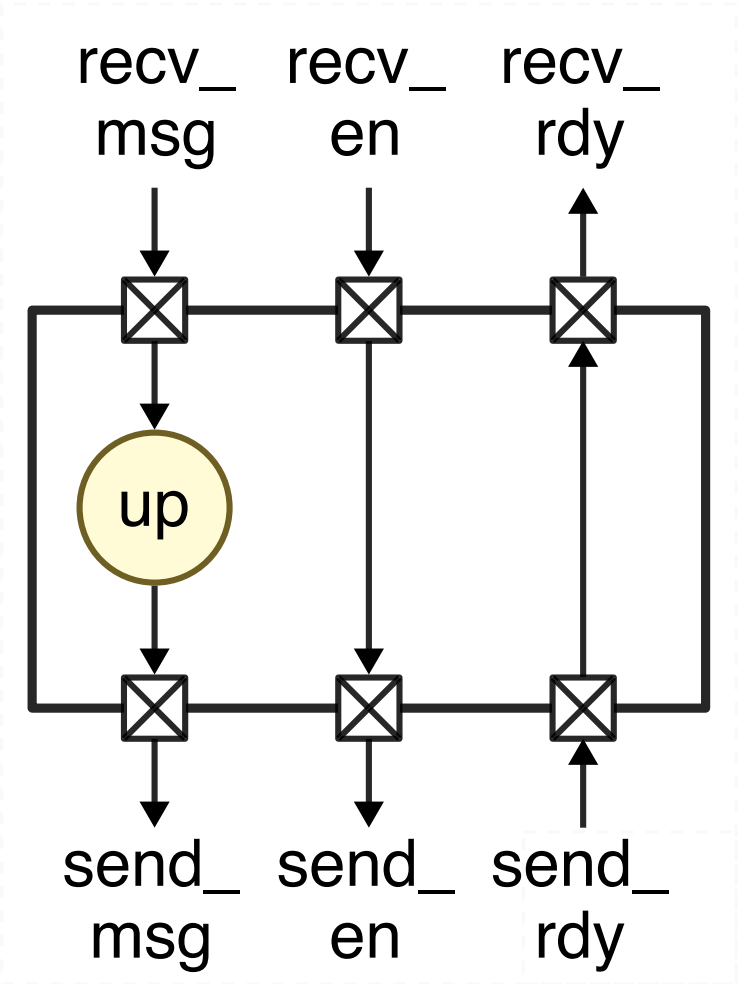
```
1  class IncrMethodModular( Component ):
2   def construct( s ):
3     s.write = CalleePort()
4     s.read  = CalleePort()
5     s.buf1  = Buffer()
6     s.buf2  = Buffer()
7     s.connect( s.write, s.buf1.write )
8     s.connect( s.read,  s.buf2.read  )
9     @s.update
10    def upB():
11      s.buf2.write( s.buf1.read() + b8(1) )
12 class IncrTestBench( Component ):
13  def construct( s ):
14    ...
15    s.incr = IncrMethodModular()
```




```
% pytest ../ex01_basics/IncrMethodModular_test.py -s
```

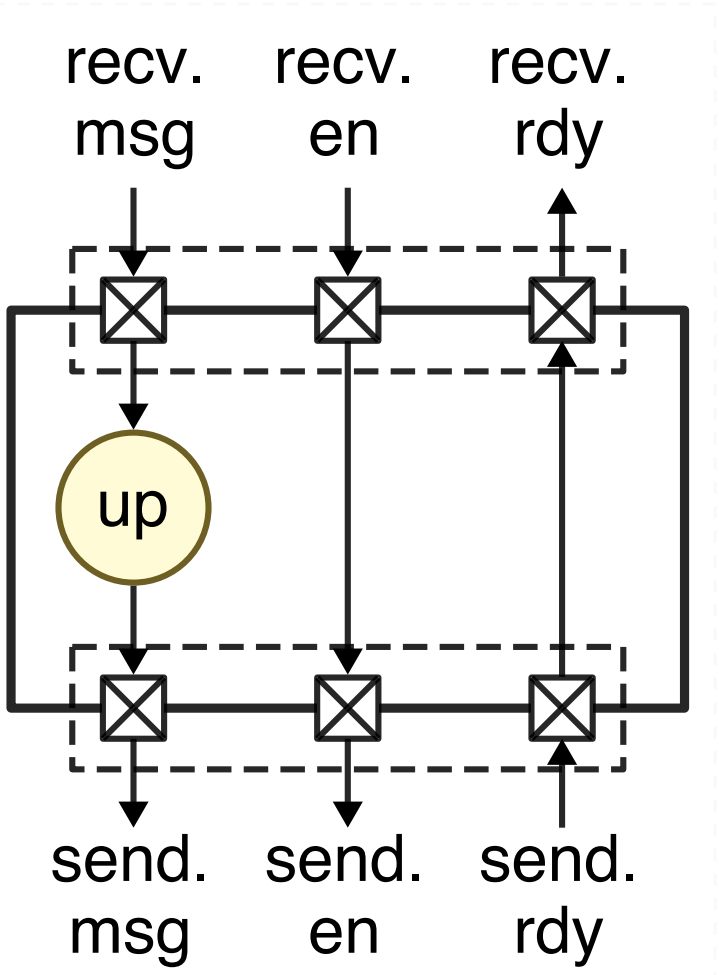
Value Ports for En/Rdy Micro-Protocol

```
1  class IncrValueModular( Component ):
2   def construct( s ):
3
4     s.recv_msg = InPort ( Bits8 )
5     s.recv_en  = InPort ( Bits1 )
6     s.recv_rdy = OutPort( Bits1 )
7
8     s.send_msg = OutPort( Bits8 )
9     s.send_en  = OutPort( Bits1 )
10    s.send_rdy = InPort ( Bits1 )
11
12    s.connect( s.recv_en,  s.send_en  )
13    s.connect( s.recv_rdy, s.send_rdy )
14
15    @s.update
16    def upB():
17      s.send_msg = s.recv_msg + b8(1)
```



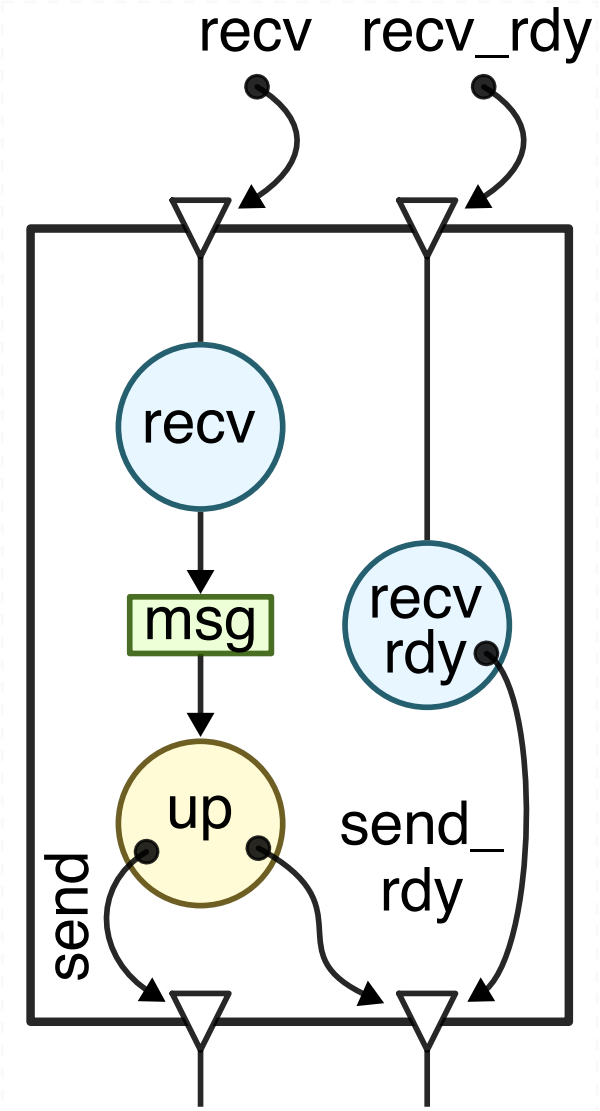
Send/Recv Value-Based Interfaces

```
1  class RecvIfcRTL( Interface ):
2   def construct( s, Type ):
3     s.msg = InPort ( Type  )
4     s.en  = InPort ( Bits1 )
5     s.rdy = OutPort( Bits1 )
6
7  class SendIfcRTL( Interface ):
8   def construct( s, Type ):
9     s.msg = OutPort( Type  )
10    s.en  = OutPort( Bits1 )
11    s.rdy = InPort ( Bits1 )
12
13 class IncrValueModular( Component ):
14  def construct( s ):
15    s.recv = RecvIfcRTL( Bits8 )
16    s.send = SendIfcRTL( Bits8 )
17
18    s.connect( s.recv.en,  s.send.en  )
19    s.connect( s.recv.rdy, s.send.rdy )
20
21    @s.update
22    def up():
23      s.send.msg = s.recv.msg + b8(1)
```



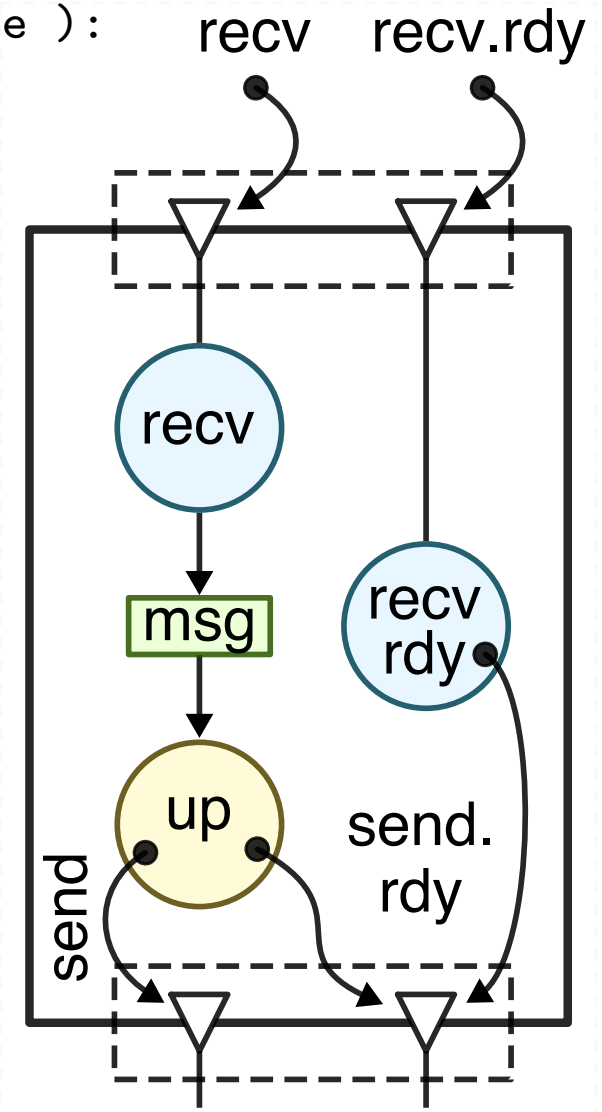
Method Ports for Non-Blocking Micro-Protocol

```
1  class IncrMethodModular( Component ):
2   def construct( s ):
3     s.send     = CallerPort()
4     s.send_rdy = CallerPort()
5     s.msg      = None
6     @s.update
7     def up():
8       if s.msg is not None and s.send_rdy():
9         s.send( s.msg + b8(1) )
10        s.msg = None
11    s.add_constraint( M(s.recv) < U(up) )
12  @method_port
13  def recv( s, msg ):
14    s.msg = msg
15  @method_port
16  def recv_rdy( s ):
17    return s.send_rdy()
```



Non-Blocking Method-Based Interfaces

```
1  class NonBlockingCallerIfc( NonBlockingInterface ):
2   def construct( s, Type ):
3     s.method = CallerPort( Type )
4     s.rdy    = CallerPort()
5
6  class IncrMethodModular( Component ):
7   def construct( s ):
8     s.send = NonBlockingCallerIfc( Bits8 )
9     s.msg  = None
10
11    @s.update
12    def up():
13      if s.msg_en is not None \
14        and s.send.rdy():
15          s.send( s.msg + b8(1) )
16
17  @non_blocking( lambda s: s.send.rdy() )
18  def recv( s, msg ):
19    s.msg = msg
```



Equivalence Between En/Rdy Value-Based Interfaces and Non-Blocking Method-Based Interfaces

- En/rdy value-based interfaces and non-blocking method-based interfaces are semantically equivalent with different implementations
  - Setting the `en` signal high is equivalent to "calling" the method
  - The `msg` signal is equivalent to the method's arguments
  - The `rdy` signal is equivalent to the return value of the `rdy()` method
  - This equivalence enables elegant interface adapters