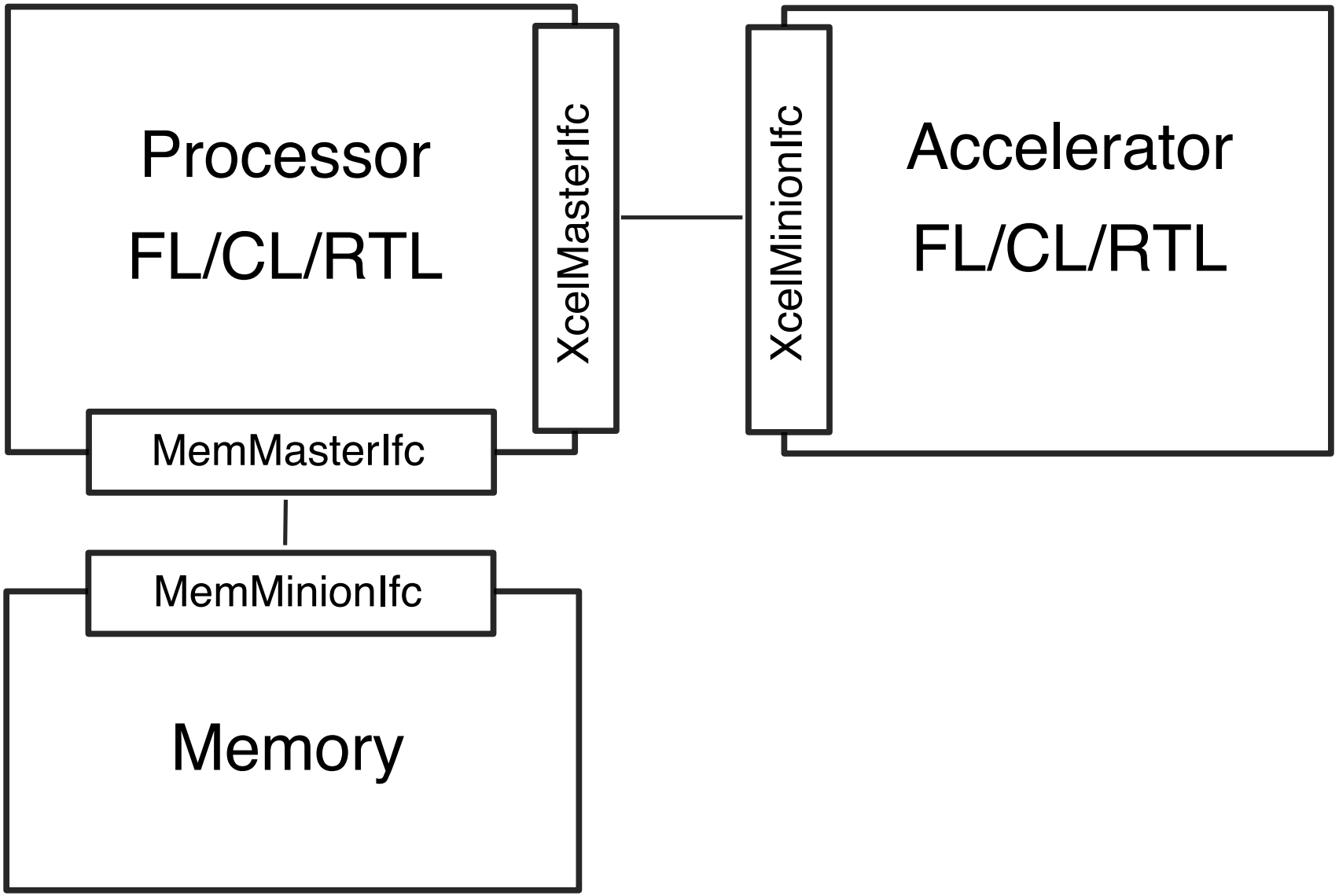
PyMTL Tutorial Schedule

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

Overview: Proc + Mem + Xcel Composition



Multi-Level Xcel Interface

- Functional-Level Xcel Interface

```
class XcelMinionIfcFL( Interface ):
 def construct( s, read=None, write=None ):
   s.read  = CalleePort( method=read )
   s.write = CalleePort( method=write )
```

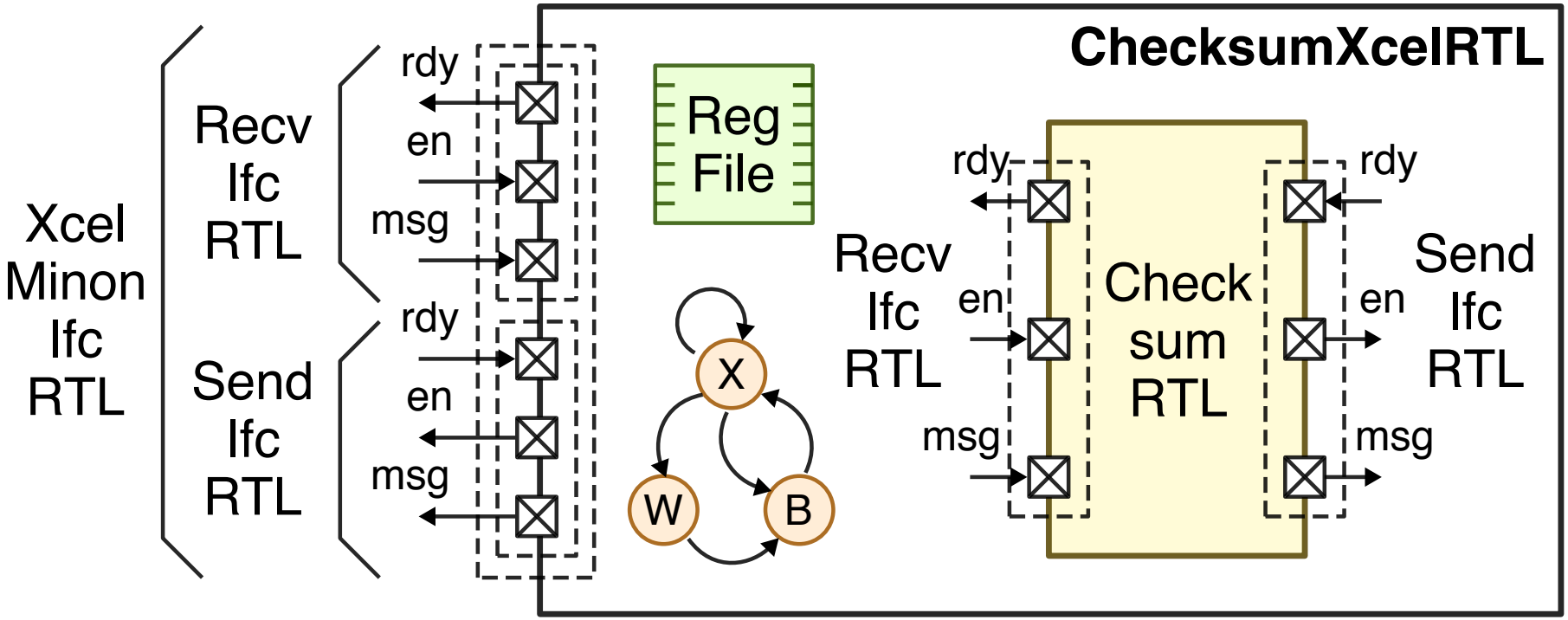
- Cycle-Level Xcel Interface

```
class XcelMinionIfcCL( Interface ):
 def construct( s, ReqType, RespType, req=None, req_rdy=None ):
   s.req  = NonBlockingCalleeIfc( ReqType, req, req_rdy )
   s.resp = NonBlockingCallerIfc( RespType )
```

- Register-Transfer-Level Xcel Interface

```
class XcelMinionIfcRTL( Interface ):
 def construct( s, ReqType, RespType ):
   s.req  = RecvIfcRTL( ReqType  )
   s.resp = SendIfcRTL( RespType )
```

Wrapping `ChecksumRTL` to Create a `ChecksumXcelRTL`

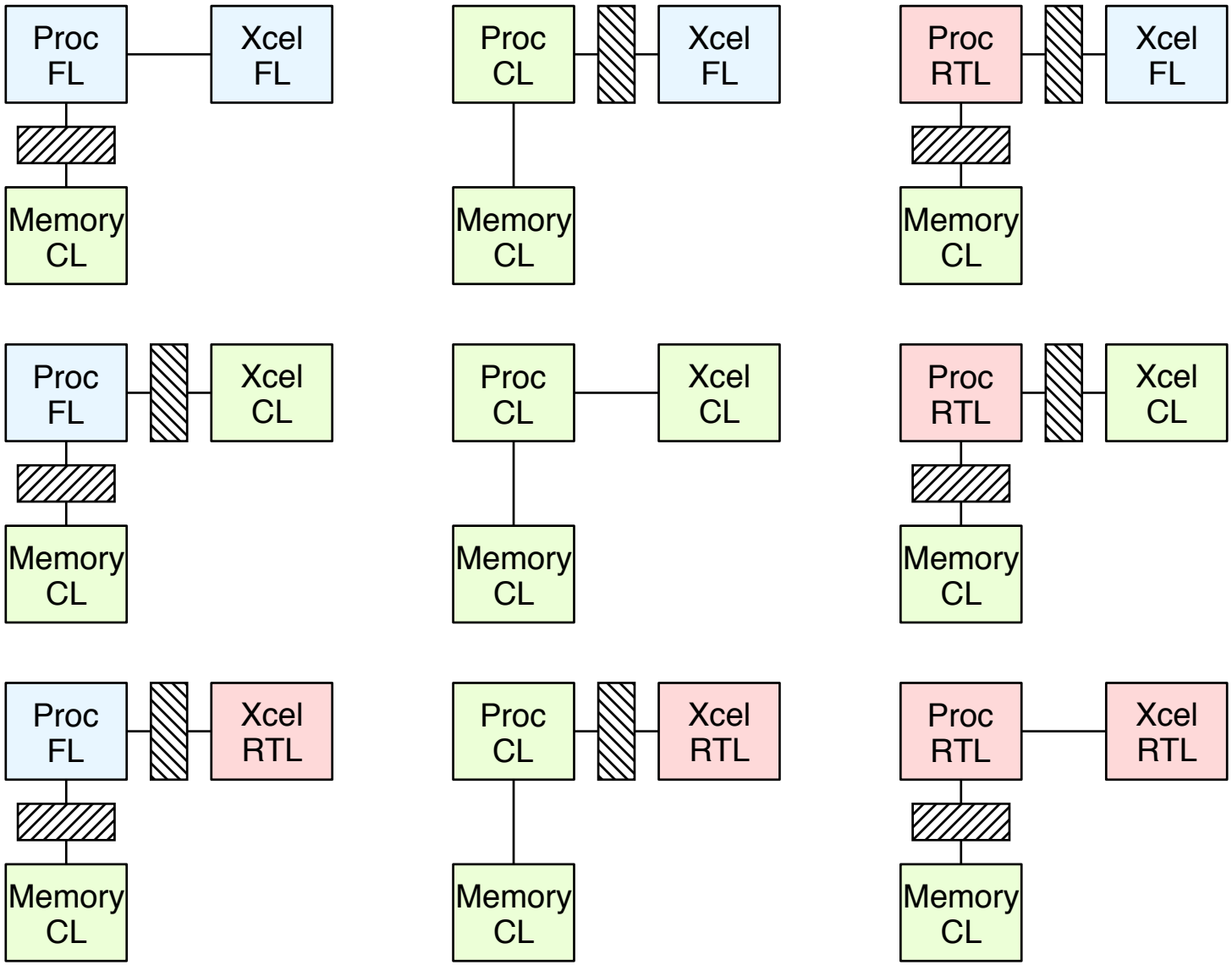


Composition Code in TestHarness

```
class ProcXcel( Component ):
  def construct( s, ProcClass, XcelClass ):
    ...
    s.proc = ProcClass()( ... )
    s.xcel = XcelClass()( xcel_minion = s.proc.xcel_master )
```

```
class TestHarness( Component ):
  def construct( s, ProcClass, XcelClass ):
    s.src  = TestSrcCL ( Bits32, [] )
    s.sink = TestSinkCL( Bits32, [] )
    s.mem  = MemoryCL  ( 2 )

    s.dut  = ProcXcel( ProcClass, XcelClass )(
      mngr2proc = s.src.send,
      proc2mngr = s.sink.recv,
      imem = s.mem.ifc[0],
      dmem = s.mem.ifc[1],
    )
    ...
```

Multi-Level Composition w/ Automatic Adapters

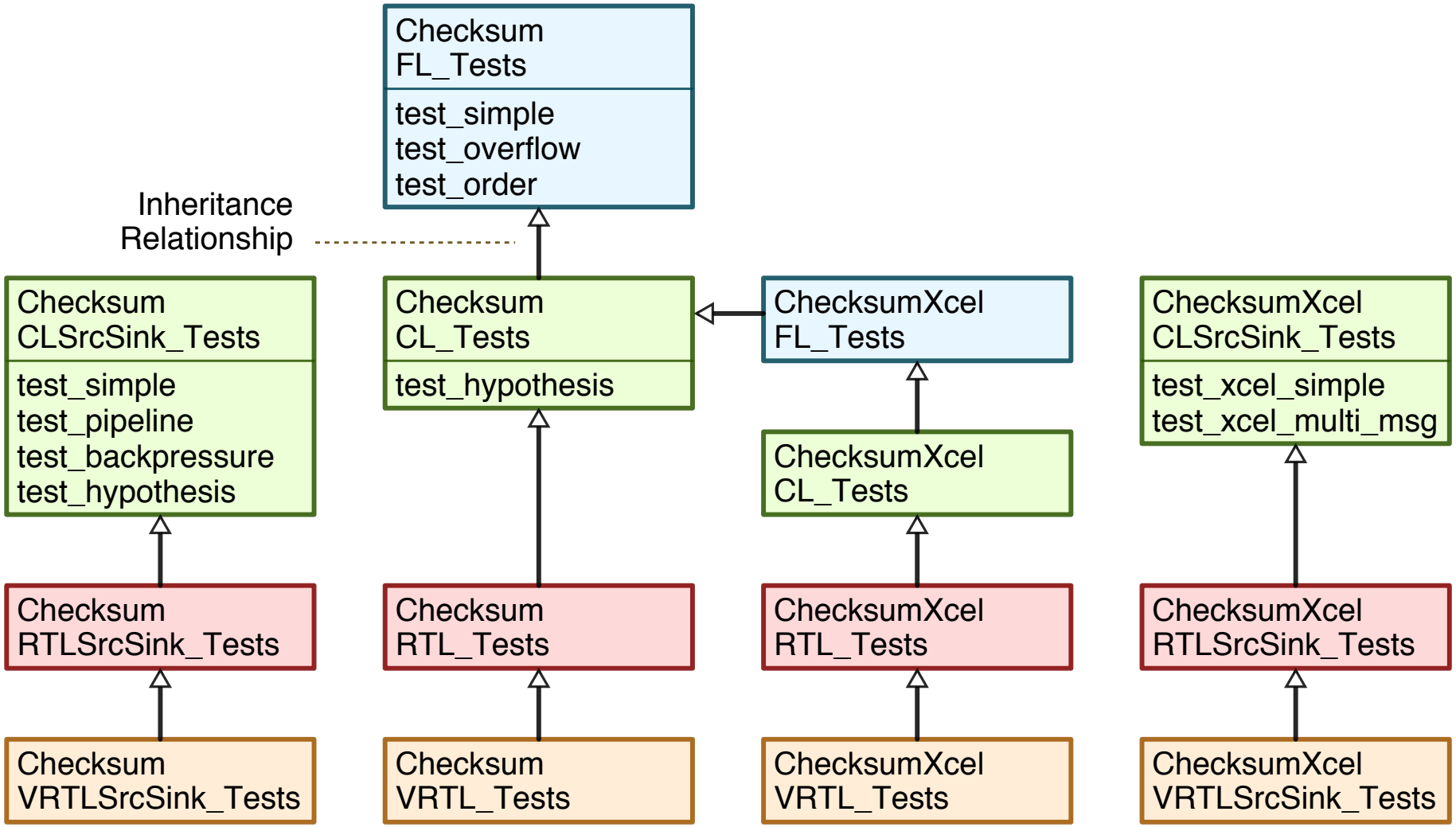


Why Multi-Level Composition is Important?

- All nine combinations are useful in certain phases of the hardware development lifecycle.

Proc	Xcel	Scenario
FL	FL	Developing end-to-end software that runs correctly on Proc+Xcel
FL	CL	Preliminary kernel-based performance evaluation
FL	RTL	Accurate kernel-based Xcel evaluation, end-to-end Xcel verification
CL	FL	Developing Proc/Xcel communication protocol
CL	CL	Preliminary end-to-end performance evaluation
CL	RTL	"Almost" cycle-accurate performance evaluation
RTL	FL	Developing Proc/Xcel communication protocol
RTL	CL	"Almost" cycle-accurate end-to-end performance evaluation
RTL	RTL	100% cycle-accurate end-to-end performance evaluation

Multi-Level Testing Class Hierachy



★ Task 4.1: Reusing Tests for Checksum Unit ★

```
% geany ../ex04_xcel/test/ChecksumXcelFL_test.py
```

```
56 from examples.ex02_cksum.test.ChecksumCL_test \
57   import ChecksumCL_Tests as BaseTests
58
59 class ChecksumXcelFL_Tests( BaseTests ):
60   def cksum_func( s, words ):
61     return checksum_xcel_fl( words )
```

```
% pytest ../ex04_xcel/test/ChecksumXcelFL_test.py -v
```

```
% geany ../ex04_xcel/test/ChecksumXcelCL_test.py
```

```
125 from examples.ex02_cksum.test.ChecksumCL_test \
126   import ChecksumCL_Tests as BaseTests
127
128 class ChecksumXcelCL_Tests( BaseTests ):
129   def cksum_func( s, words ):
130     return checksum_xcel_cl( words )
```

```
% pytest ../ex04_xcel/test/ChecksumXcelCL_test.py -v
```

★ Task 4.1: Reusing Tests for Checksum Unit (cont) ★

```
% geany ../ex04_xcel/test/ChecksumXcelRTL_test.py
```

```
72 from examples.ex02_cksum.test.ChecksumCL_test \
73   import ChecksumCL_Tests as BaseTests
74
75 class ChecksumXcelRTL_Tests( BaseTests ):
76   def cksum_func( s, words ):
77     return checksum_xcel_rtl( words )
```

```
% pytest ../ex04_xcel/test/ChecksumXcelRTL_test.py -v
```

★ Task 4.2: Experimenting with Nine Compositions ★

```
% cd $HOME/pymtl-tut/examples/build

% ../ex04_xcel/proc-xcel-sim --bmark cksum-xcel --proc-impl fl  --xcel-impl fl
% ../ex04_xcel/proc-xcel-sim --bmark cksum-xcel --proc-impl fl  --xcel-impl cl
% ../ex04_xcel/proc-xcel-sim --bmark cksum-xcel --proc-impl fl  --xcel-impl rtl

% ../ex04_xcel/proc-xcel-sim --bmark cksum-xcel --proc-impl cl  --xcel-impl fl
% ../ex04_xcel/proc-xcel-sim --bmark cksum-xcel --proc-impl cl  --xcel-impl cl
% ../ex04_xcel/proc-xcel-sim --bmark cksum-xcel --proc-impl cl  --xcel-impl rtl

% ../ex04_xcel/proc-xcel-sim --bmark cksum-xcel --proc-impl rtl --xcel-impl fl
% ../ex04_xcel/proc-xcel-sim --bmark cksum-xcel --proc-impl rtl --xcel-impl cl
% ../ex04_xcel/proc-xcel-sim --bmark cksum-xcel --proc-impl rtl --xcel-impl rtl
```

- Experiment with `--trace` to see a line trace

Processor + Xcel Design-Space Exploration

- We will use a `NullXcel` as a place-holder for our baseline design
- Analyze the performance using the cycle-accurate RTL composition
- Translate to SystemVerilog so we can use other tools
- Use open-source EDA toolflow to generate area and timing estimates
- Fill in the following table

	Performance (cycles)	**Area** (μm^2)	**Timing** (ns)
Processor + NullXcel			
Processor + ChecksumXcel			
Ratio			

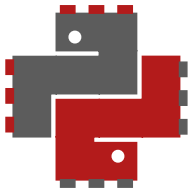
★ Task 4.3: Translate and Synthesize Proc+NullXcel ★

```
% cd $HOME/pymtl-tut/examples/build
% ../ex04_xcel/proc-xcel-sim --bmark cksum \
    --proc-impl rtl --xcel-impl null --translate

% cp ProcXcel__ProcClass_ProcRTL__XcelClass_NullXcelRTL.sv \
    $HOME/alloy-asic/designs/ProcNullRTL/rtl
% mkdir $HOME/alloy-asic/build_proc_null
% cd $HOME/alloy-asic/build_proc_null
% ../configure.py --design ProcNullRTL
% make open-yosys-synthesis

% grep "Chip area" 3-open-yosys-synthesis/run-step.log
% grep "Current delay" 3-open-yosys-synthesis/run-step.log
```

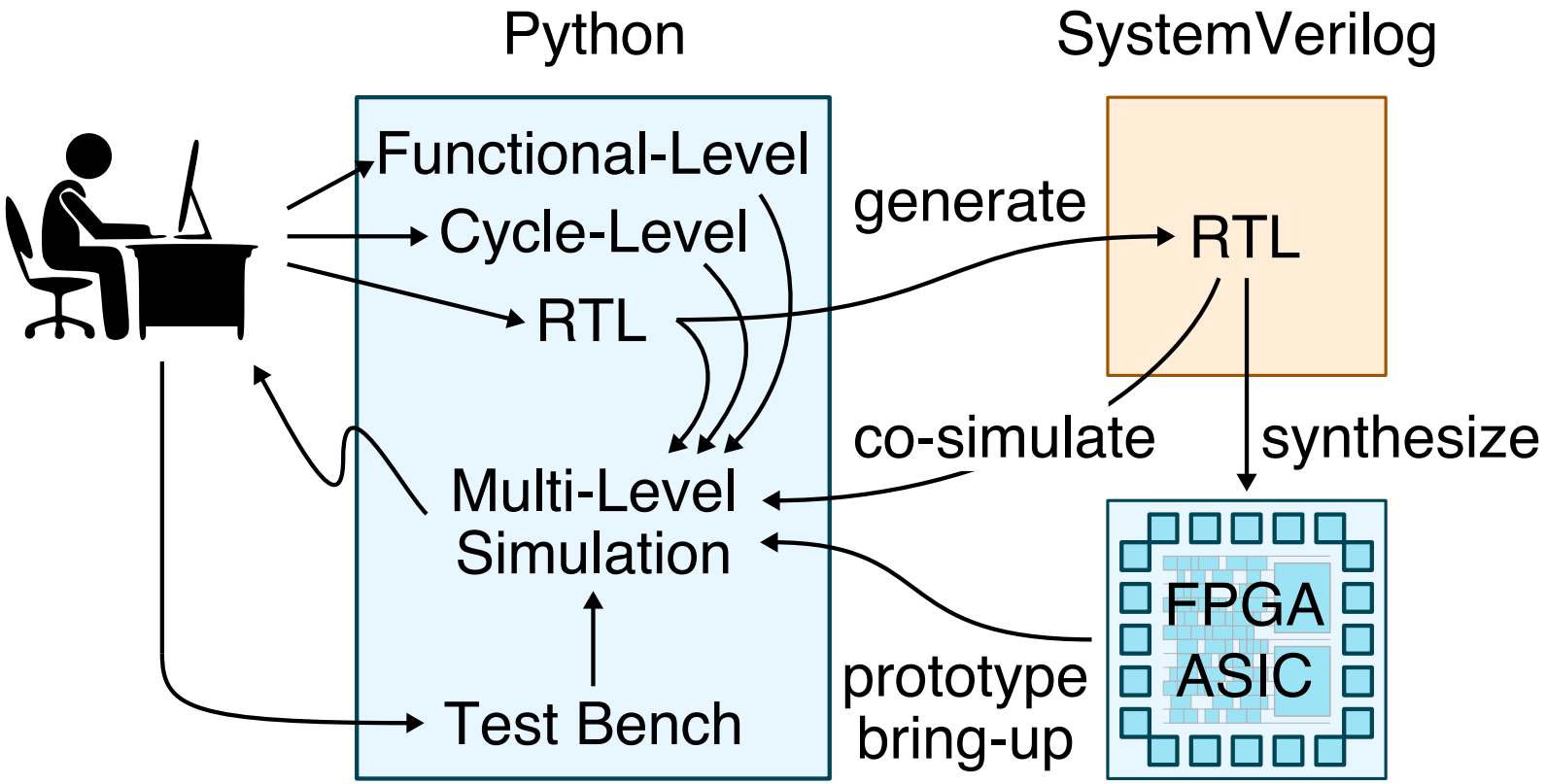
★ Task 4.4: Translate and Synthesize Proc+ChecksumXcel ★

```
% cd $HOME/pymtl-tut/examples/build
% ../ex04_xcel/proc-xcel-sim --bmark cksum-xcel \
    --proc-impl rtl --xcel-impl rtl --translate

% cp ProcXcel__ProcClass_ProcRTL__XcelClass_ChecksumXcelRTL.sv \
    $HOME/alloy-asic/designs/ProcCksumRTL/rtl
% mkdir $HOME/alloy-asic/build_proc_cksum
% cd $HOME/alloy-asic/build_proc_cksum
% ../configure.py --design ProcCksumRTL
% make open-yosys-synthesis

% grep "Chip area" 3-open-yosys-synthesis/run-step.log
% grep "Current delay" 3-open-yosys-synthesis/run-step.log
```

PyMTL



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



This work was supported in part by NSF XPS Award #1337240, NSF CRI Award #1512937, NSF SHF Award #1527065, AFOSR YIP Award #FA9550-15-1-0194, DARPA Young Faculty Award #N66001-12-1-4239, and the the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, and equipment, tool, and/or physical IP donations from Intel, NVIDIA, Xilinx, Synopsys, and ARM.

Thanks to Derek Lockhart, Ji Kim, Shreesha Srinath, Berkin Ilbeyi, Yixiao Zhang, Jacob Glueck, Aaron Wisner, Gary Zibrat, and Carl Friedrich Bolz for their help developing, testing, and using PyMTLv2 and PyMTLv3