

Functional and Single Event Effect (SEE) Triplication Verification of the ATLAS ITk Strip Tracker HCCStar and AMACStar Front-end ASICs

Paul T. Keener and Ben Rosser

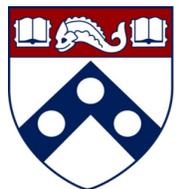
University of Pennsylvania and University of Chicago (formally Penn)

7 October 2025



Contents

1. What is CocomTB?
2. Why use CocomTB?
3. Triplication for SEE Mitigation
4. Verification of SEE Mitigation



CocoTB

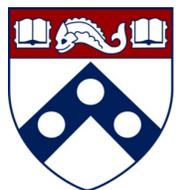
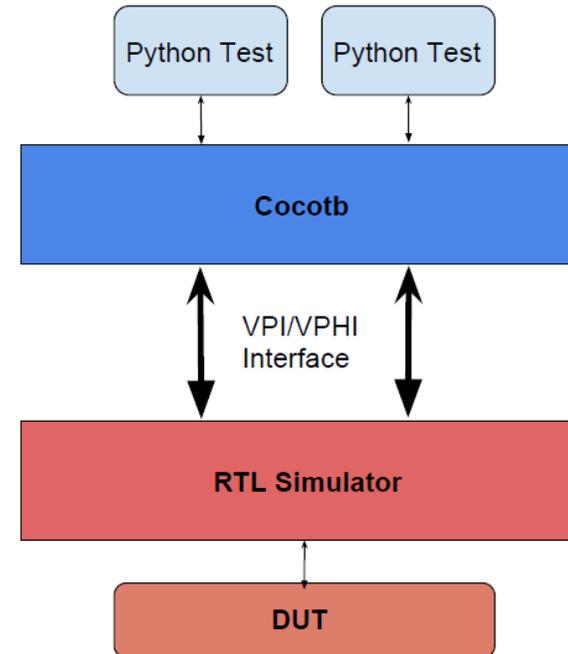
CocoTB is a library for digital logic verification in Python

Coroutine **c**osimulation **T**est**B**ench

Provides a Python interface to standard RTL simulators

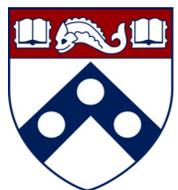
Offers an alternative to RTL or SystemVerilog testbenches

Especially UVM!



Simple Example

```
// example_mux.v  
// MUX taken from HCCStar design (ITk Strips)  
module example_mux(  
    output wire we_lp_muxed_o,  
    input wire readout_mode_i,  
    input wire L0_i,  
    input wire we_lp_i  
);  
  
    // Switch between inputs depending on  
    // value of readout mode.  
    assign we_lp_muxed_o =  
        readout_mode_i ?  
        L0_i : we_lp_i;  
endmodule
```



Simple Example

```
// example_mux.v
// MUX taken from HCCStar design (ITk Strips)
module example_mux(
    output wire we_lp_muxed_o,
    input wire readout_mode_i,
    input wire L0_i,
    input wire we_lp_i
);

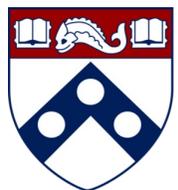
    // Switch between inputs depending on
    // value of readout mode.
    assign we_lp_muxed_o =
        readout_mode_i ?
        L0_i : we_lp_i;
endmodule
```

```
# mux_tester.py
import cocotb
from cocotb.triggers import Timer
from cocotb.result import TestFailure

@cocotb.test()
def mux_test(dut):
    dut.L0_i <= 0
    dut.we_lp_i <= 0

    dut.readout_mode_i <= 1
    dut.L0_i <= 1
    yield Timer(1, "ns")
    if dut.we_lp_muxed_o != 1:
        raise TestFailure("Failure!")

    dut.readout_mode_i <= 0
    yield Timer(1, "ns")
    if dut.we_lp_muxed_o != 0:
        raise TestFailure("Failure!")
```



Simple Example – Annotated

A few key points:

- `@cocotb.test()` declares a function as a test
- `dut` represents the design hierarchy
- `dut.L0_i <= 0` is a shorthand to assign to an RTL variable
- `yield Timer(1, "ns")` waits 1ns for the simulator to advance
- `raise TestFailure` fails the test if the MUX is not working

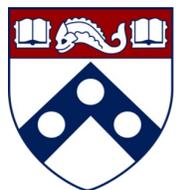
Caveat: This example is using an older version of the library

```
# mux_tester.py
import cocotb
from cocotb.triggers import Timer
from cocotb.result import TestFailure

@cocotb.test()
def mux_test(dut):
    dut.L0_i <= 0
    dut.we_lp_i <= 0

    dut.readout_mode_i <= 1
    dut.L0_i <= 1
    yield Timer(1, "ns")
    if dut.we_lp_muxed_o != 1:
        raise TestFailure("Failure!")

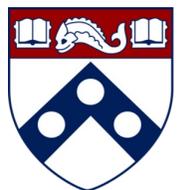
    dut.readout_mode_i <= 0
    yield Timer(1, "ns")
    if dut.we_lp_muxed_o != 0:
        raise TestFailure("Failure!")
```



Key Feature

Can create Python functions that run in parallel, for example, to drive complex protocols on inputs and analyze outputs

Greatly simplifies test writing



Real Test Example

```
@CocotbTest("HCC_301", skip=True)
def test_hcc_301_multiple_packets(dut):
    """ Tests sending multiple packets to the input
    channel. """
    tester = hccstar.HCCStarTester(dut)

    # Create a fake ABCStar on input channel 4.
    tester.add_fake_abc(4)

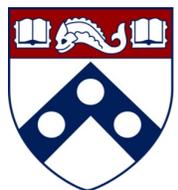
    tester.packetmon.set_automatic_packet_checking()
    yield tester.start()

    yield tester.configure_standalone()
```

```
# Turn on input channel 4.
tester.enqueue_reg_write_hcc(0b0010, 40, 0b10000)
yield tester.lcbdrv.wait()
yield triggers.ClockCycles(tester.clock, 100)

tester.add_abc_physics_data(4, num_clusters=8)
tester.enqueue_L0A(4, 0b0001, False)
yield tester.lcbdrv.wait()

# See if we got a packet with all 8+1 clusters.
packet = yield tester.get_parsed_packet('TYP_LP',
fail=True)
yield tester.stop()
```

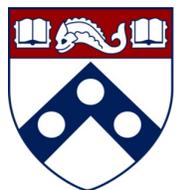
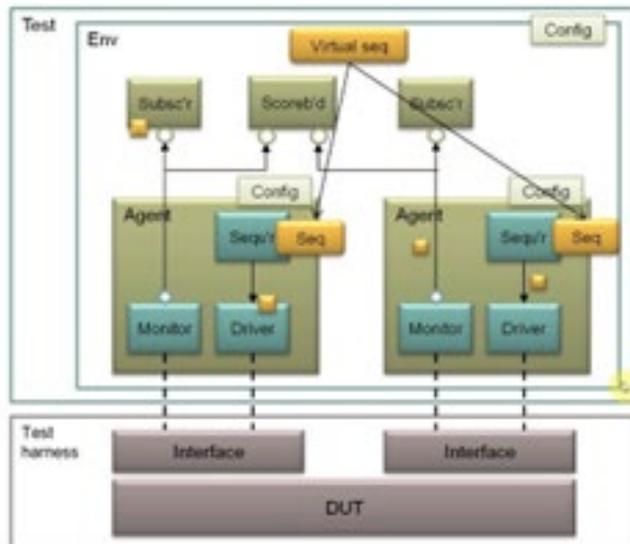


Why CocoTB?

Post Docs and Grad Students will likely come in knowing at least a little Python, but no Verilog or VHDL

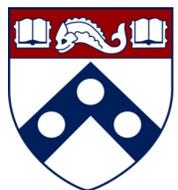
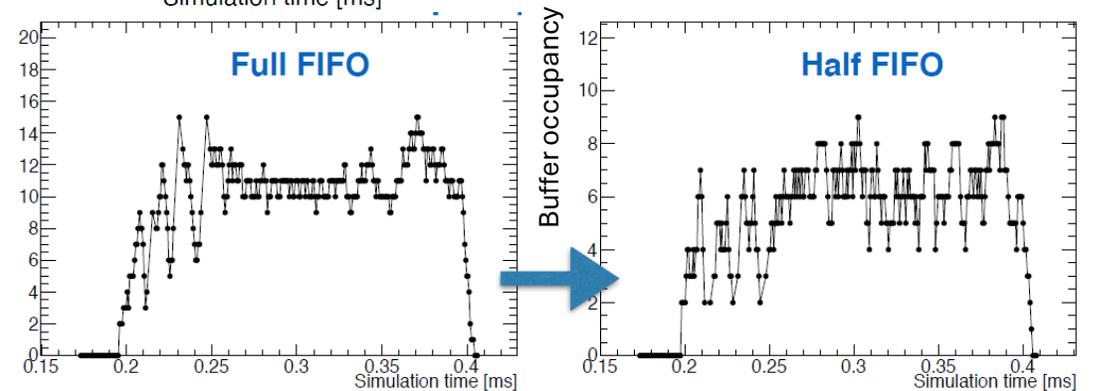
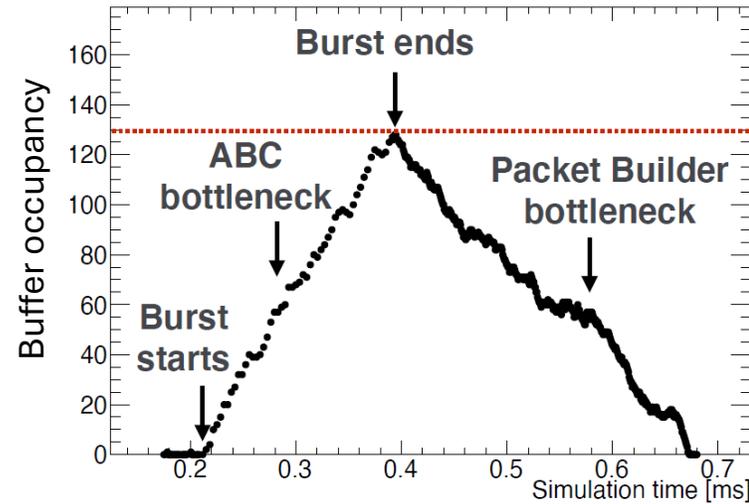
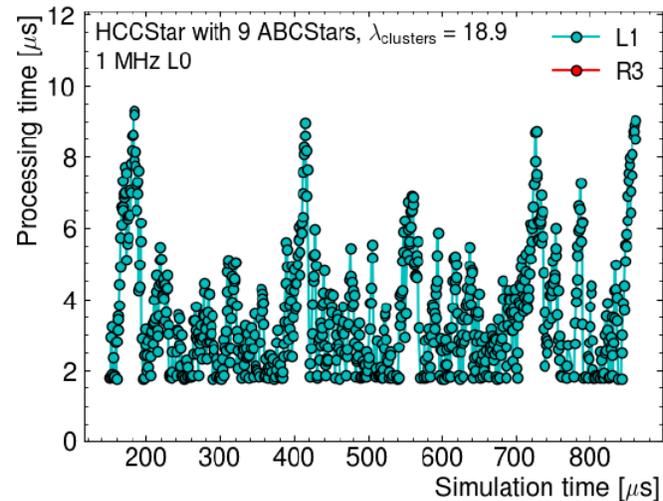
Being able to immediately focus on the job at hand rather than learning a new, somewhat arcane language is a big win

UVM is **complicated!**



Why CocoTB (2)?

Python comes with a rich ecosystem of libraries that greatly simplify common tasks like making plots, eg, matplotlib



Single Event Effects (SEE)

Single Event Effects are disturbances in circuits caused by particles traversing components (devices or wires) of the circuit

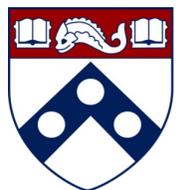
There are two kinds of SEE we are mainly concerned about:

1) Single Event Upset (SEU)

The particle causes a storage element, eg, flip-flop, to spuriously change state, eg $0 \rightarrow 1$ or vice versa

2) Single Event Transient (SET)

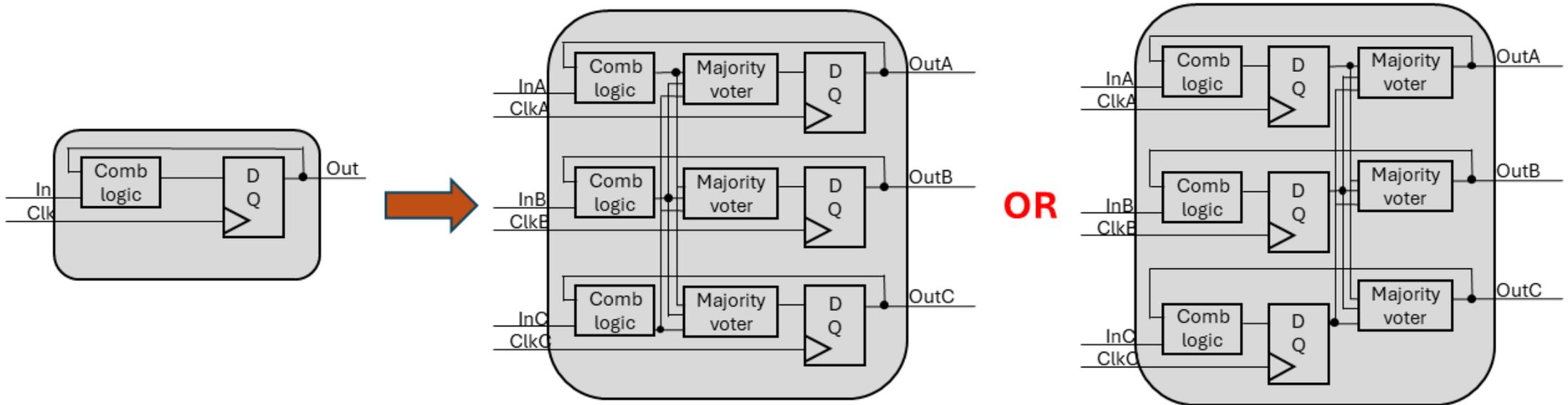
The particle causes a wire to briefly change state, frequently by traversing the driver of the preceding logic cell



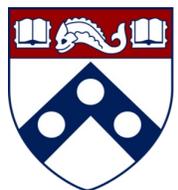
SEE Mitigation

The most common mitigation is Triple Modular Redundancy (TMR) – make three copies of the logic and add voters

Need three voters to prevent the voter itself to be a single point of failure



We have used both styles in our designs

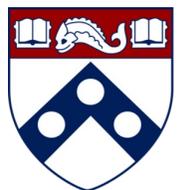


Generating TMR

By writing the RTL in Verilog, using specific conventions, the TMRG tool(*) will automatically generate TMR Verilog

A side effect of using TMRG, the triplicated flip-flops have a predictable name, eg, fooA, fooB, fooC

Non-triplicated flip-flops can easily be identified



* <http://stacks.iop.org/1748-0221/12/i=01/a=C01082>

Generating lists of “interesting” objects

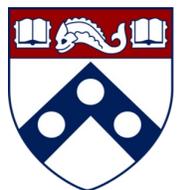
List of sequential elements was generated from the synthesis script:

```
vfind /designs/HCCstarTMR -instance *seq/*
```

HCCStar had over 45,000 flip-flops

List of wires was generated with a script using pyVerilog and removing irrelevant nets, eg UNCONNECTEDs

HCCStar had almost 160,000 wires



SEE Verification

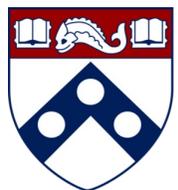
Two styles of SEE Verification were performed

- 1) Fast – demonstrating voting corrections worked
- 2) Thorough – look for incorrect behavior

The Fast Verification would generate random SEUs and verify the A,B,C versions of the flip-flops matched on the next clock

The Thorough Verification would exercise the functionality of the ASIC, while randomly injecting SEUs and (artificially long) SETs looking for incorrect behavior

Which may come **much** later after the SEE injection



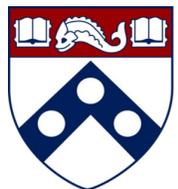
SEE Verification Results

A small number of failures were seen

Mostly false positives

A few real design errors

Easily correctable

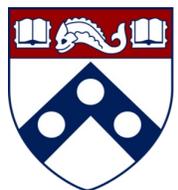


Conclusions

CocoTB is a powerful verification framework that simplifies the work required

CocoTB leverages what Grad Students and Post Docs already know to allow them to make real contributions quickly

TMR generation can be relatively easy and can be verified within the CocoTB framework



References

CocoTB: <https://www.cocotb.org/>

[Seminar](#) by Ben Rosser

TMRG:

gitlab: [TMRG](#)

Paper: <http://stacks.iop.org/1748-0221/12/i=01/a=C01082>

