Cocotb: a Python-based digital logic verification framework

Ben Rosser

University of Pennsylvania

December 11, 2018

Introduction

- Cocotb is a library for digital logic verification in Python.
  - **Co**routine **co**simulation **t**est**b**ench.
  - Provides Python interface to control standard RTL simulators (Cadence, Questa, VCS, etc.)
  - Offers an alternative to using Verilog/SystemVerilog/VHDL framework for verification.
- At Penn, we chose to use cocotb instead of UVM for verification last summer.
- This talk will cover:
  - What cocotb is, what its name means, and how it works.
  - What are the potential advantages of cocotb over UVM or a traditional testbench?
  - What has Penn's experience been like using cocotb?

Introduction: Disclaimer

- Disclaimer: **I am not a UVM expert!**
- My background is in physics and computer science, not hardware or firmware design.
- During my first year at Penn, I spent some time trying to learn UVM, but we decided to try out cocotb soon afterward.
- I can talk about what made us want an alternative to UVM, but I cannot provide a perfect comparison between the two.

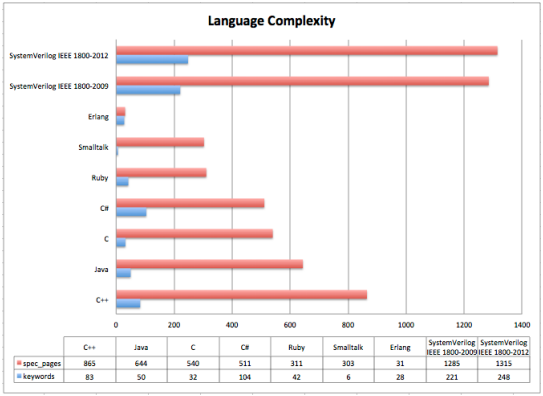
What is Cocotb?

Approaches to Verification

- Why not just use Verilog or VHDL for verification?
  - Hardware description languages are great for designing hardware or firmware.
  - But hardware design and verification are different problems.
  - Using the same language for both might not be optimal.
  - Verification testbenches are **software**, not hardware.
  - Higher level language concepts (like OOP) are useful when writing complex testbenches.
- Two possible ways to improve the situation:
  - Add higher level programming features to a hardware description language.
  - Use an existing general purpose language for verification.
- SystemVerilog is the first approach: simulation-only OOP language features.
- UVM (Universal Verification Methodology) libraries written in SystemVerilog.

Verification using SystemVerilog

- But as a result, SystemVerilog is a very complicated language.
- The SystemVerilog spec is over a thousand pages!
- Language has 221 keywords; compare to C++'s 83.
- Powerful, but takes a while to learn.
- Grad students unlikely to have prior SystemVerilog experience.

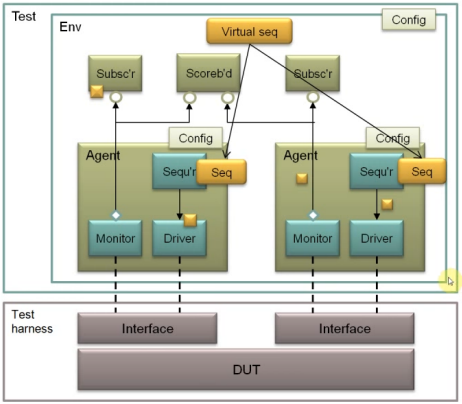


	C++	Java	C	C#	Ruby	Smalltalk	Erlang	SystemVerilog IEEE 1800-2009	SystemVerilog IEEE 1800-2012
spec_pages	865	644	540	511	311	303	31	1285	1315
keywords	83	50	32	104	42	6	28	221	248

`http://www.fivecomputers.com/language-specification-length.html`

Verification using UVM

- UVM has similar complexity issues.
- There are over 300 classes in UVM.
- Lots of ways to do the same thing.
- Again, very powerful, but very difficult to get started.
- My personal experience:
  - Followed online tutorials for a few weeks, but barely scratched the surface.
  - Found official documentation a bit lacking.
  - Tried using Doulos resources on UVM, but this only took me so far.



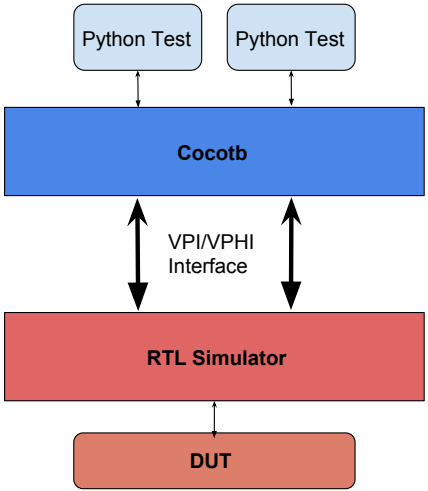
UVM class hierarchy, taken from tutorial:
`https://www.youtube.com/watch?v=NlUii8N-VXc`

Verification using Python

- The SV/UVM approach is powerful, but complicated.
- cocotb's developers, Chris Higgs and Stuart Hodgson, tried a different approach:
  - Keep the hardware description languages for what they're good at– design!
  - Use a high-level, general purpose language for developing testbenches.
  - Object oriented programming is much more natural in general purpose languages!
- They picked **Python** as their language of choice:
  - Python is simple (only 23 keywords) and easy to learn, but very powerful.
  - Python has a large standard library and a huge ecosystem; lots of existing libraries.
  - Python is well documented and popular: lots of resources online.
- For Penn's purposes, Python was good for another reason:
  - Verification tasks being given to graduate students (e.g. as qualification tasks).
  - All ATLAS grad students and postdocs should know at least some Python!

Cocotb: Basic Architecture

- How does cocotb work?
- Design under test (DUT) runs in standard simulator.
- cocotb provides interface between simulator and Python.
- Uses Verilog Procedural Interface (VPI) or VHDL Procedural Interface (VHPI).
- Python testbench code can:
  - Reach into DUT hierarchy and change values.
  - Wait for simulation time to pass.
  - Wait for a rising or falling edge of a signal.



Basic Example - RTL and Python

Let's walk through an example of cocotb, for a simple MUX design.

```
// 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!")
```

Basic Example - Python Annotated

- A few key points in the testbench code:
- `@cocotb.test()` decorator declares a function as a test.
- The variable `dut` represents the hierarchy.
- `dut.L0_i <= 0` is shorthand to assign to a RTL variable (`dut.L0_i.value = 0`).
- `yield Timer(1, "ns")` waits 1 ns for the simulator to advance.
- `raise TestFailure` fails the test if the MUX is not working.

```python
# 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!")
```

Basic Example - Makefile

- Cocotb projects each need a Makefile to specify which files to include in simulation. Below is a Makefile for the above example.

```
SIM ?= ius

MODULE = mux_tester
TOPLEVEL = example_mux
TOPLEVEL_LANG ?= verilog

EXTRA_ARGS =

VERILOG_SOURCES = ../rtl/example_mux.v
VHDL_SOURCES =

include $(COCOTB)/makefiles/Makefile.inc
include $(COCOTB)/makefiles/Makefile.sim
```

- `MODULE`, `TOPLEVEL` control which Python, RTL module to instantiate.
- `TOPLEVEL_LANG` can be Verilog or VHDL.
- `EXTRA_ARGS` allows extra arguments to be passed to simulator.
- `SIM` sets which simulator to use; ius is Cadence.
- `VERILOG_SOURCES` and `VHDL_SOURCES`: RTL files to include.

Running the Example

How do you actually run this? Just type `make`! The simulator will start and run the tests.

```
     0.00ns INFO     Running on ncsim(64) version 15.20-s046
     0.00ns INFO     Python interpreter initialised and cocotb loaded!
     0.00ns INFO     Running tests with Cocotb v1.0.1 from /tape/cad/cocotb/cocotb-20171128
     0.00ns INFO     Seeding Python random module with 1544025098
     0.00ns INFO     Found test mux_tester.mux_test
     0.00ns INFO     Running test 1/1: mux_test
     0.00ns INFO     Starting test: "mux_test"
                     Description: None
     3.00ns INFO     Test Passed: mux_test
     3.00ns INFO     Passed 1 tests (0 skipped)
     3.00ns INFO     *******************************************************************************
                     ** TEST                 PASS/FAIL  SIM TIME(NS)  REAL TIME(S)  RATIO(NS/S) **
                     *******************************************************************************
                     ** mux_tester.mux_test    PASS            3.00          0.00      1005.35  **
                     *******************************************************************************

     3.00ns INFO     ********************************************************************************
                     **                              ERRORS : 0                                    **
                     ********************************************************************************
                     **                            SIM TIME : 3.00 NS                              **
                     **                           REAL TIME : 0.01 S                               **
                     **                     SIM / REAL TIME : 335.53 NS/S                          **
                     ********************************************************************************

     3.00ns INFO     Shutting down...
Simulation complete via $finish(1) at time 3 NS + 0
ncsim> exit
```

Cosimulation: Triggers

- Design and testbench simulated independently: this is **cosimulation**.
- Communication through VPI/VHPI interfaces, represented by cocotb "triggers".
- When the Python code is executing, **simulation time is not advancing**.
- When a trigger is yielded, the testbench waits until the triggered condition is satisfied before resuming execution.
- Available triggers include:
  - `Timer(time, unit)`: waits for a certain amount of simulation time to pass.
  - `Edge(signal)`: waits for a signal to change state (rising or falling edge).
  - `RisingEdge(signal)`: waits for the rising edge of a signal.
  - `FallingEdge(signal)`: waits for the falling edge of a signal.
  - `ClockCycles(signal, num)`: waits for some number of clocks (transitions from 0 to 1).

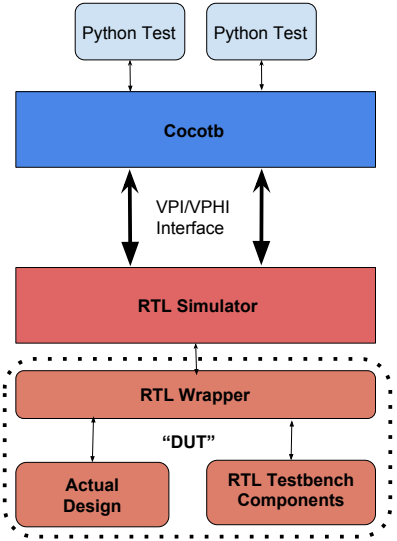
Modifying the Hierarchy

- Since Python and RTL are co-simulated, easy to reach into the hierarchy.
- The Python testbench can read or change the value of any internal signal.
- Makes simulation of single event upsets very simple!
- Example shows how value of internal signal could be read (and changed).

```
import cocotb
from cocotb.triggers import RisingEdge

@cocotb.test()
def test(dut):
    yield RisingEdge(dut.clk)

    # Access value of internal signal.
    current = int(dut.submodule.important.value)

    # Change it, see what happens!
    dut.submodule.important <= (not current)
    yield RisingEdge(dut.clk)
```

Including RTL in a Testbench

- Important point: it is still possible to have RTL components of testbench!
- Simply write a Verilog or VHDL "wrapper" top-level:
  - Instantiate the actual design being tested, along with other components for testing.
  - Modify `TOPLEVEL` in Makefile to point at the wrapper.
- Must use trigger interface to communicate– not possible to directly call procedures.
- But still useful for low-level pieces of testing, or assertions, or to include existing code.



Post-Synthesis Simulations

- Cocotb can be used for **post-synthesis simulations** too!
- Can use the wrapper approach to load timing constraints (SDF) files on demand.
- Example: post-synthesis simulation can be launched with `make SIM_MODE=synthesis`.
- Makefile has been modified to check this variable when deciding what to do.

```
module Wrapper();
  initial begin
    `ifdef USE_SYN_SDF
        $sdf_annotate("timing.sdf",
                      Wrapper.DUT,
                      "",
                      "sdf.log");
    `endif
  end
  // Declare DUT.
endmodule
```

```
# Set "TOPLEVEL" to point at the wrapper.
TOPLEVEL = Wrapper
VERILOG_SOURCES = Wrapper.v

ifeq ($(SIM_MODE), synthesis)
    VERILOG_SOURCES += synthesis_netlist.v
    EXTRA_ARGS += -define USE_SYN_SDF
else
    # Do normal RTL simulation
    # Add RTL files to VERILOG_SOURCES.
endif
```

Coroutines

- Cocotb uses a cooperative multitasking architecture.
- Tests can call other methods and functions, just like normal Python.
- If those methods want to consume simulation time, they must be **coroutines**.

- In cocotb, coroutines are just functions that obey two properties.
- Decorated using the `@cocotb.coroutine` decorator.
- Contain at least one `yield` statement, yielding another coroutine or trigger.

```
import cocotb
from cocotb.triggers import RisingEdge

@cocotb.coroutine
def test_helper(dut):
    dut.member <= 1
    yield RisingEdge(dut.clk)

@cocotb.test()
def test(dut):
    yield test_helper(dut)
```

Forking Coroutines

- Coroutines can be yielded, but they can also be **forked** to run in parallel.
- This allows the creation of something like a Verilog always block.
- Key to creating complex testbenches: start up monitors, run them in the background.

```
module tb;

    wire clk;

    always @(posedge clk) begin
        // Do something.
    end

endmodule
```

```
import cocotb
from cocotb.triggers import RisingEdge

@cocotb.coroutine
def always_block(dut):
    while True:
        yield RisingEdge(dut.clk)
        # Do something.

@cocotb.test()
def test(dut):
    # Start clock.
    thread = cocotb.fork(always_block(dut))
```

Joining Forked Coroutines

- Unlike always blocks, it is possible to join a forked coroutine.
- Calling `.join()` returns a trigger that can be yielded.
- Will wait until the coroutine finishes executing.
- Also possible to kill a coroutine immediately by calling `.kill()`.

```
import cocotb
from cocotb.triggers import RisingEdge, Timer

@cocotb.coroutine
def always_block(dut):
    while True:
        yield RisingEdge(dut.clk)
        # Do something.

@cocotb.test()
def test(dut):
    # Start clock.
    thread = cocotb.fork(always_block(dut))
    yield thread.join()
```

Yielding Multiple Triggers

- Example on the previous slide will not actually terminate.
- But we can fix that!
- Yield a list of triggers (or coroutines) instead of just one.
- Testbench will wait until one of the triggers fires.
- This example will now continue after 100 ns of simulation time.

```
import cocotb
from cocotb.triggers import RisingEdge, Timer

@cocotb.coroutine
def always_block(dut):
    while True:
        yield RisingEdge(dut.clk)
        # Do something.

@cocotb.test()
def test(dut):
    # Start clock.
    thread = cocotb.fork(always_block(dut))
    yield [thread.join(), Timer(100, "ns")]
```

Communicating with Coroutines

- For building complex testbenches: necessary to pass information between forked coroutines.
- A couple different ways to do this: can use the `Event()` trigger:
  - A coroutine can yield `event.wait()` to block until another coroutine calls `event.set()`.
  - Data can be passed between coroutines by setting `event.data`.
- A simpler way: **use classes**:
  - Functions in classes can be made coroutines and forked.
  - The class will be accessible from both the main and the forked coroutine.
- Combining these techniques: can create advanced testbench components like drivers, monitors.

Coroutines and Classes

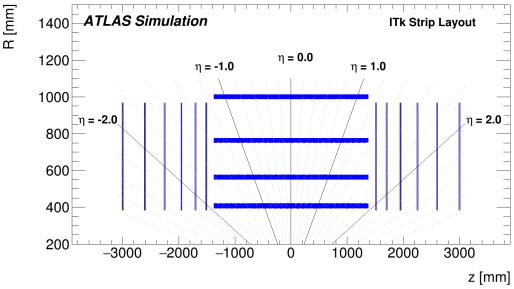
- Here's a very simple example of how to build a driver using coroutines.
- Uses a Python class (`SimpleDriver`).
- Once the `drive` function starts, on every clock it sets a port on the DUT equal to `self.value`.
- This flag can then be set from the test, outside the coroutine.
- Contrived example; a more sophisticated driver could implement a serial protocol, or have a built-in queue for commands.
- Example of how to build up more complex testbenches!

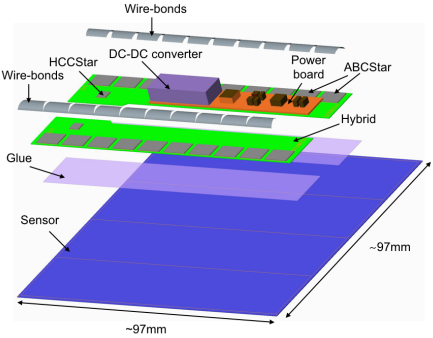
```
import cocotb
from cocotb.triggers import RisingEdge, FallingEdge

class SimpleDriver:

    def __init__(self, dut):
        self.dut = dut
        self.value = 0

    @cocotb.coroutine
    def drive(self):
        while True:
            yield RisingEdge(self.dut.clk)
            self.dut.data <= self.value

@cocotb.test()
def test(dut):
    driver = SimpleDriver(dut)
    cocotb.fork(driver.drive())
    yield FallingEdge(dut.clk)
    driver.value = 1
```

Penn Experience

Aside: The ITk Strips Project

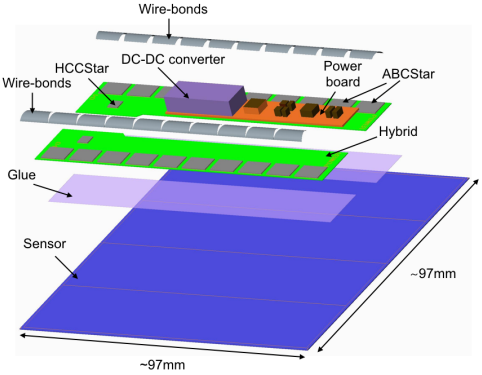
- Penn's engineering group is involved with the ATLAS ITk Strips upgrade project:
  - Brand-new inner tracker for the HL-LHC, scheduled to be installed in 2026.
  - The ITk Strips TDR (public): `https://cds.cern.ch/record/2257755`
- We are helping design, build, and test the front-end readout electronics.
- Each strips module will have several custom ASICs for readout, control, and monitoring.





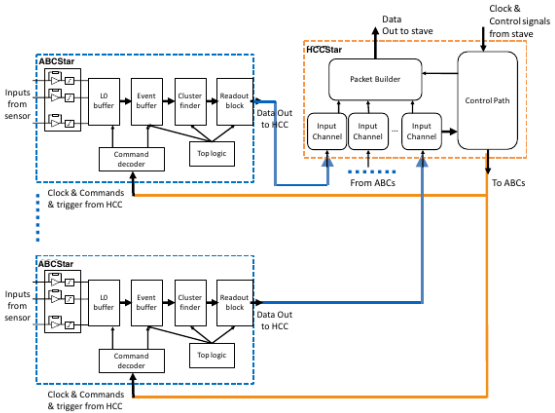
ITk Strips ASICs

- Three ASICs being designed:
- **ABCStar** (ATLAS Binary Chip): reads in hits from the sensors and clusters them. Responds to readout commands.
- **HCCStar** (Hybrid Control Chip): controls a group of ABCStars (a "hybrid") and combines their outputs into a single data stream.
- **AMACv2** (Autonomous Monitor And Control): one per module, sits on the power board.
- Penn heavily involved in the last two.



ITk Strips: HCCStar and ABCStar

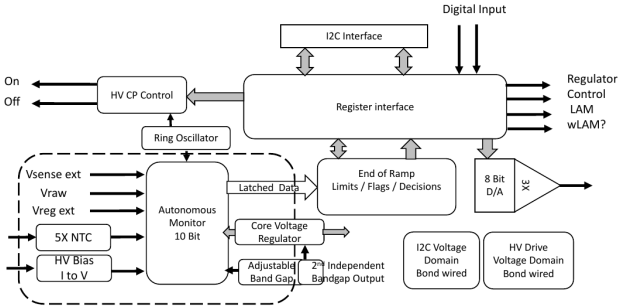
- Rough block diagrams of the hybrid, containing HCCStar and two ABCStars.
- Other groups responsible for verification of the ABCStar.
- Penn responsible for verification of the HCCStar.
- Additional verification goal: simulate both chips together.



"Star network" architecture; ABCs talk directly to HCC.

ITk Strips: AMACv2

- Penn responsible for design and verification of the AMAC as well.
- Rough block diagram of the AMACv1; Penn began work on new AMACv2 in August 2017.
- Many more analog blocks than the HCCStar.



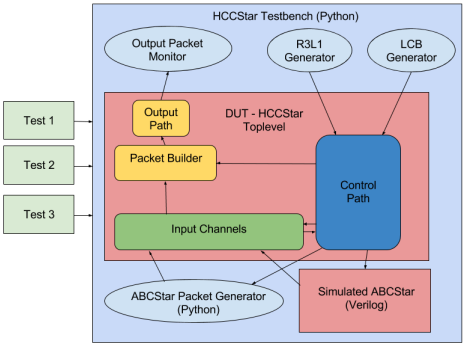
ITk Verification - Using Cocotb

- We began HCCStar verification in the spring of 2017.
- Considered what approach to adopt for verification:
  - Wanted to use something more sophisticated than a traditional Verilog testbench.
  - Considered UVM: found it very powerful, but difficult to get started.
  - Decided to look for alternatives before embracing UVM.
- Bill Ashmanskas and I discovered cocotb! Seemed to be what we were looking for:
  - Easier to get students and postdocs involved, due to familiarity with Python.
  - Much simpler to get started in comparison to UVM.
  - Supported the Cadence Incisive simulator (which we were using for this project).

Initial ITk Verification

- Began by writing simple unit tests for existing HCCStar blocks; slowly built up more complex testbench structures over the summer of 2017.
- Used Python classes for testbench components, like monitors and drivers.
- Used libraries (bitarray and bitstring) for manipulating fixed length bit vectors.
- Used numpy and scipy libraries to generate random data.
- Created Python models of data flow to check the ASIC against the specification.
- Used a similar approach to do AMACv2 verification in fall of 2017.
- Once the AMAC design was submitted in mid-October, work resumed on the HCCStar.

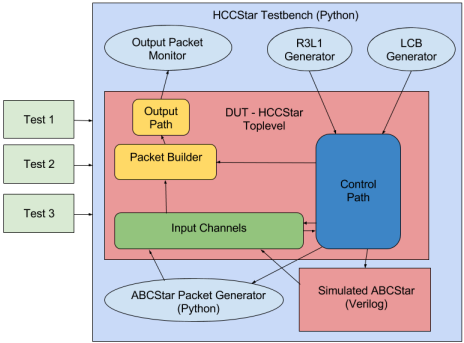
Complete HCCStar Verification



- Rough sketch of HCC Testbench.
- Light blue blocks written in Python.

- By early 2018, had built up complete HCCStar testbench.
- Wrote Python implementation of communications protocols.
- Wrote Python model of ABCStar to produce physics data in response to commands.
- All tests made to be self-checking; mismatches between expected, actual outputs automatically flagged.

Hybrid Verification

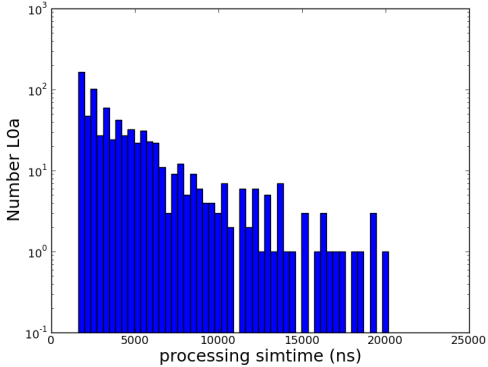


- Important goal: simulate HCCStar with RTL version of ABCStar!
- Designed testbench so that Python or RTL ABCStars could be used.
- Ended up with more complex makefile to handle multiple cases:
  - Different make targets to start different types of simulations.
  - Could switch between hybrid and standalone top-level block.
  - Could also switch between hybrid and standalone testbench.

- We found bugs that the ABCStar verification (SystemVerilog without UVM) missed!

Hybrid Simulations

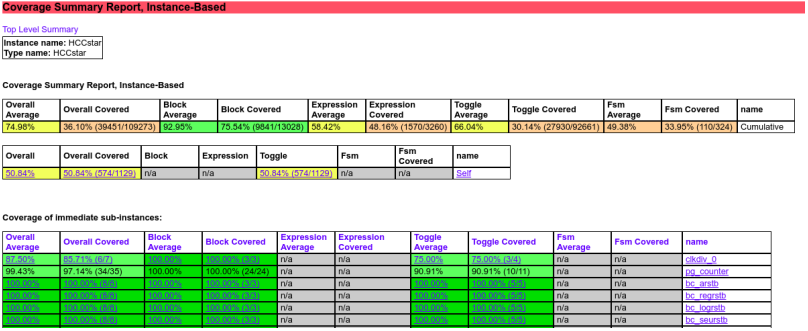
- Hybrid verification also included simulations of realistic operating conditions.
- Trigger rate, expected occupancy, and runtime could all be configured.
- Simulations were sometimes ran for several days– caught several rare problems!
- Used matplotlib library to produce performance plots from inside the testbench.



- Example plot from a run of the hybrid testbench.
- Shows how long it takes for the system to respond to a readout request.

Continuous Integration and Code Coverage

- Ran all our tests nightly using Continuous Integration (CI) system.
- cocotb test results stored in XML file; wrote script to process this.
- Also generated coverage reports; Coverage data (and monitoring plots!) available online: `http://www.hep.upenn.edu/asic_CI/`



Coverage Summary Report, Instance-Based

Top Level Summary

Instance name: HCCstar
Type name: HCCstar

Coverage Summary Report, Instance-Based

Overall Average	Overall Covered	Block Average	Block Covered	Expression Average	Expression Covered	Toggle Average	Toggle Covered	Fsm Average	Fsm Covered	name
74.96%	36.10% (39451/109273)	92.95%	75.54% (9841/13028)	58.42%	48.16% (1570/3260)	66.04%	30.14% (27930/92661)	49.38%	33.95% (110/324)	Cumulative

Overall	Overall Covered	Block	Expression	Toggle	Fsm	Fsm Covered	name
50.84%	50.84% (574/1129)	n/a	n/a	50.84% (574/1129)	n/a	n/a	Self

Coverage of immediate sub-instances:

Overall Average	Overall Covered	Block Average	Block Covered	Expression Average	Expression Covered	Toggle Average	Toggle Covered	Fsm Average	Fsm Covered	name
87.50%	85.71% (6/7)	100.00%	100.00% (3/3)	n/a	n/a	75.00%	75.00% (3/4)	n/a	n/a	clkdiv_0
99.43%	97.14% (34/35)	100.00%	100.00% (24/24)	n/a	n/a	90.91%	90.91% (10/11)	n/a	n/a	pg_counter
100.00%	100.00% (4/4)	100.00%	100.00% (3/3)	n/a	n/a	100.00%	100.00% (1/1)	n/a	n/a	bc_arstb
100.00%	100.00% (5/5)	100.00%	100.00% (3/3)	n/a	n/a	100.00%	100.00% (2/2)	n/a	n/a	bc_regrstb
100.00%	100.00% (5/5)	100.00%	100.00% (3/3)	n/a	n/a	100.00%	100.00% (2/2)	n/a	n/a	bc_logrstb
100.00%	100.00% (5/5)	100.00%	100.00% (3/3)	n/a	n/a	100.00%	100.00% (2/2)	n/a	n/a	bc_seurstb

ITk Verification - Overview

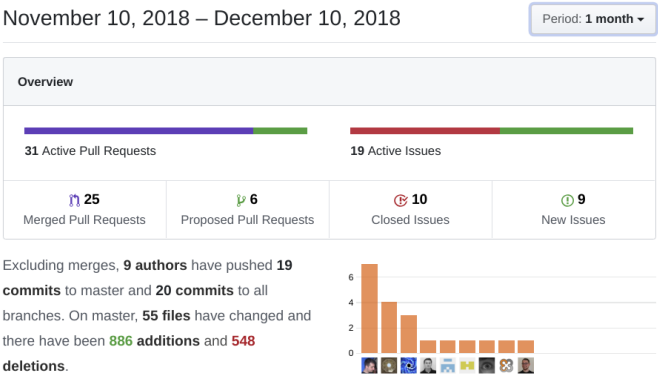
- Overall, the cocotb approach to verification was very successful.
- Easy for myself, postdoc Jeff Dandoy, and grad student Joe Mullin to get involved.
- Wrote many tests for the HCCStar, hybrid, and AMAC simulations since last fall:
  - **84** tests for the HCCStar.
  - **32** tests for the hybrid (HCCStar and ABCStar).
  - **30** tests for the AMAC.
- Many critical issues were found and fixed!
- According to JIRA, **65 tickets** relevant to the HCCStar of varying severity have been resolved. (More bugs were reported in person or via email, so this is a lower bound).
- Continuous integration helped us catch lots of problems as they happened.

Cocotb - Upstream Issues

- Were there any problems with the cocotb approach to verification?
- Biggest obstacle we ran into: low upstream activity.
- The cocotb community has been growing, but development had stagnated.
- We quickly ended up depending on a few unofficial patches to fix bugs.
- Some newer features were missing from cocotb documentation; we sometimes had to look at the source code to learn things.
- Were able to make things work, but concerning for long-term health of the project.
- Good news: this has been improving over the last few months!

New Community Activity

- Several new maintainers appointed.
- One of them is an ATLAS member! Tomasz Hemperek, from the University of Bonn.
- New community development guidelines agreed upon.
- A lot of recent activity working through the backlog of issues and patches.



https://github.com/potentialventures/cocotb/pulse/monthly

Other Problems and Solutions

- A couple of other problems we ran into, and how we worked around them:
- The simulator crashed when we tried to load pyROOT from the testbench:
  - Seems to be a problem to do with Cadence + ROOT; worked for other simulators.
  - Ended up just using numpy/scipy/matplotlib instead.
- How can you pass configuration to the testbench?
  - We ended up with a large Makefile– around 200 lines long!
  - Heavy use of environment variables to pass configuration to make and to tests.
  - Several different make targets for different simulation setups (RTL, post-synthesis, post-PNR for standalone HCC and hybrid) testbenches).

Conclusion

Conclusion

- Cocotb is a powerful tool for verification in a high-level programming language:
  - More powerful than a traditional Verilog testbench.
  - Easier to get started with than a SystemVerilog or UVM testbench.
- Cosimulation approach means that RTL simulator still used under the hood:
  - Testbenches can contain a mixture of Python and RTL.
  - Cocotb testbenches can be used for post-synthesis simulations as well.
- Penn succcessfully used cocotb for verifying ASICs for the ITk Strips upgrade project.
- We are planning to continue using cocotb for future projects!

Thanks!

- Thank you for your attention!
- People at Penn group who have worked on the verification effort:

- HEP Instrumentation Group:
  - Bill Ashmanskas
  - Paul Keener
  - Adrian Nikolica
- Graduate students:
  - Ben Rosser
  - Joe Mullin
- Postdocs:
  - Jeff Dandoy

Backup

Cocotb Resources

- Cocotb project repository: `https://github.com/potentialventures/cocotb`
- Official cocotb documentation: `https://cocotb.readthedocs.io/en/latest/`
- Mailing list: `https://lists.librecores.org/listinfo/cocotb`
- Other talks given about cocotb:
  `https://github.com/potentialventures/cocotb/wiki/Further-Resources`
- Lots of examples can be found in the documentation and repository!