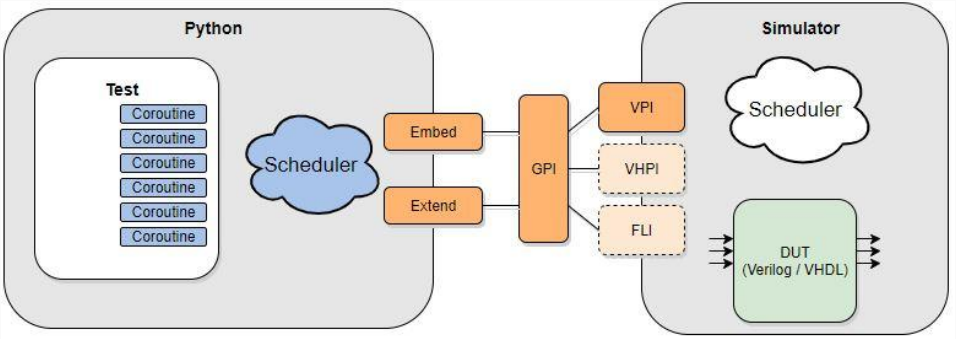
CocoTB : python based verification for the RTL projects

V. Shebalin

December 1, 2022

What Cocotb is?



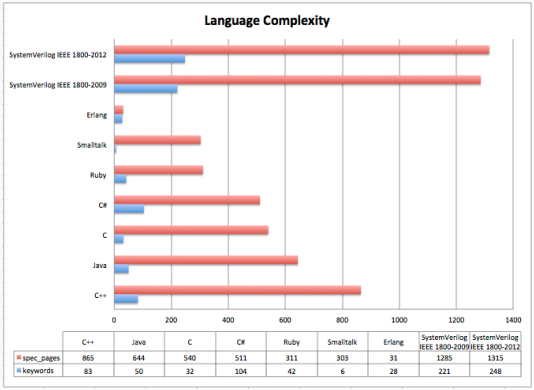
- `https://www.cocotb.org`
- A RTL simulator plugin / a python library for writing synchronous logic.
- Cocotb is an open source CO-routine-based CO-simulation TestBench environment for verifying VHDL and Verilog TRL using Python.
- Provides Python interface to control standard RTL simulators (Modelsim/Questa, Cadence, etc.)
- Hardware design and verification are different tasks. Verification testbenches are **software** not hardware!

Motivation for python

- **VHDL** lacks built-in verification tools. OSVVM offers a lot of tools for constrained random testing, functional coverage, message filtering, scoreboards and so on, but it's still VHDL.
- **Verilog/SystemVerilog** – object oriented language with a lot of verification oriented classes and functions, but it's a pretty difficult language to learn.

Why python?

- Easy to learn
- Huge existing ecosystem
- Interpreted
- Popular language
- Simulation is a software (not hardware), python is good for making software.



	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

Some of HEP specifics

- Physics group – they know what kind of output they need from the electronics, know physics, don't know technical details, don't know HDL, most likely know python
- Electronics group – they know all details of the electronics, don't have proficiency in data analysis

Physics group

- Know what they need the electronics for.
- Don't know the details of the digital design.
- Have the realistic data for the electronics input signals (MC simulation, experimental data samples).
- Have the tools to analyze the data electronics should provide.
- Most likely know python.

Electronics group

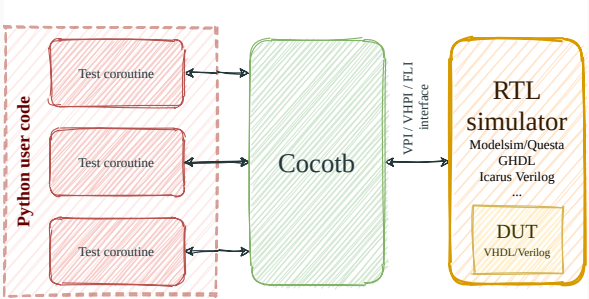
- Probably don't know all the aspects of the electronics application.
- Know everything about details of the logic implementation.
- Have their oscilloscopes and pulse generators for the basic testing.
- Most likely know python.

There are verification engineers in big companies and big experiments, but smaller groups usually have very limited manpower. Cocotb makes it possible to bring more people into firmware verification: students, software experts, physicists.

Cocotb workflow

Cocotb main features

- Design under test (DUT) runs in a standard simulator.
- cocotb provides interface between simulator and Python (compiled C++ library called GPI).
- Simulator is controlled by VPI, VHPI, or FLI interfaces.
- Python testbench allows to: access DUT hierarchy, wait for simulator time to advance, trigger on different conditions



Python basis

- **async functions** – not called directly, but *awaited* instead
- **coroutines** – cooperative multitasking routines.
- **decorators** – some kind of a wrapper function

Prior to python 3.5 cocotb used generators/yield to emulate concurrent processing.

Cocotb basic example

dff.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity dff is
port(
  clk: in std_logic;
  d: in std_logic;
  q: out std_logic);
end dff;

architecture behavioral of dff is
begin
  process (clk) begin
    if rising_edge(clk) then
      q <= d;
    end if;
  end process;
end behavioral;
```

test_dff.py

```python
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import RisingEdge

@cocotb.test()
async def test_dff(dut):
    # Create a 10us period clock on port clk
    clock = Clock(dut.clk, 10, units="us")
    # Start the clock
    cocotb.start_soon(clock.start())

    dut.d.value = 0
    await RisingEdge(dut.clk)
    dut.d.value = 1
    await RisingEdge(dut.clk)
    assert dut.q.value == 1, f"Error!"
```

Compare it to the VHDL testbench!

https://github.com/vsheb/cocotb_practice – my non-professional examples I did for practice

Makefiles

Minimal makefile example

```
VHDL_SOURCES = dff.vhd
TOPLEVEL := dff
MODULE   := test_dff

include $(shell cocotb-config --makefiles)/Makefile.sim
```

Run it

```
pip3 install cocotb # install cocotb if you haven't done it yet
make # run cocotb
```

Output

```
**********************************************************************************************
** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
**********************************************************************************************
** test_dff.test_dff             PASS       15000.00           0.00   15167444.55  **
**********************************************************************************************
** TESTS=1 PASS=1 FAIL=0 SKIP=0             15000.00           0.20      74682.24  **
**********************************************************************************************
```

View the waveform

Makefile

```
VHDL_SOURCES = dff.vhd
TOPLEVEL := dff
MODULE   := test_dff
## Save waveforms
WAVES    := 1

include $(shell cocotb-config --makefiles)/Makefile.sim
```

View waveform (you can use any supported simulator, not only *gtkwave*, obviously)

```
gtkwave dff.vcd
```

Makefile pros and cons

- Allows integration of verification into the building process.
- An extra file to take care of.

As an alternative to makefiles there is the cocotb-test package which makes it possible to run cocotb tests as a standalone program.
https://github.com/themperek/cocotb-test

cocotb-test provides standard python unit testing capabilities for cocotb

- allow the look and feel of Python unit testing (see `pytest.org`)
- remove the need for Makefiles (includes Makefile compatibility mode)
- allow easy customization of simulation flow
- allow to use pytest-xdist or pytest-parallel for parallel runs

Installation

```
pip install cocotb-test
```

cocotb-test example

import cocotb-test

```
from cocotb_test.simulator import run
```

Run the test without invoking make

```
def run_test_dff():
  run(
    vhdl_sources  = ['dff.vhd'], # the list of vhdl sources to be included
    toplevel      = 'dff',       # DUT
    module        = 'test_dff1', # python module which contains coroutines to run
    toplevel_lang = 'vhdl',
    sim_args      = ['--vcd=dff.vcd'], # this one is specific to ghdl simulator
    waves         = 1)
if __name__ == "__main__" :
  run_test_dff()
```

Among other things cocotb-test allows for running parametrized tests.

Accessing the hierarchy

Cocotb makes it possible to access the internal signal by it's hierarchy using the simulator interface even if the language (i.e. VHDL-93) doesn't allow to do so.

Usage examples

```
# access the current value of the signal of interest
val = int(dut.module.submodule.signal.value)
# change the current value of the signal
dut.module.submodule.signal.value = 1
# force a handle to a given value
dut.module.submodule.signal.value = Force(1)
# make a handle keep its current value
dut.module.submodule.signal.value = Freeze()
# stop the effects of a previously applied force/freeze action
dut.module.submodule.signal.value = Release()
```

Hierarchy access to the internal signals gives a very flexible way to simulate single event upsets.

Coroutines

- Any function which imply advancing of the simulation time must be **coroutine**.
- Cocotb coroutines emulate synchronous **process**(VHDL) or **always**(Verilog) blocks.
- Can run in parallel.
- A key thing to create complex testbenches : different coroutines running in parallel may serve as **drivers** and **monitors**.

```python
@coroutine
async def dff_driver(dut):
  dut.d.value = 0
  for _ in range(100) :
    await RisingEdge(dut.clk)
    dut.d.value = 1
    await RisingEdge(dut.clk)
    dut.d.value = 0
    await ClockCycles(dut.clk, 10)
```

```python
@coroutine
async def dff_monitor(dut):
  while True:
    await RisingEdge(dut.clk)
    if dut.d.value == 1 :
      await RisingEdge(dut.clk)
      assert dut.q.value == 1, f"Error!"
```

In older examples you may see *@cocotb.coroutine* decorator which is deprecated, also *yield* keyword has been replaced with *await* starting from version 1.4.0 (2020).

Starting coroutines

```python
@cocotb.test()
async def test_dff(dut):
  # start clock
  clk = Clock(dut.clk, 10, units="ns")
  cocotb.start_soon(clk.start())

  # start monitoring
  mon = cocotb.start_soon(dff_monitor(dut))
  # start driver
  drv = cocotb.start_soon(dff_driver(dut))

  # wait the driver coroutine to finish
  await Join(drv)
```

Methods to start a coroutine

- **fork()** – (deprecated) schedules and executes the new coroutine immediately, no other pending tasks are run.
- **start()** – schedules the new coroutine to be executed concurrently, then yields control to allow the new task (and any other pending tasks) to run, before resuming the calling task.
- **start_soon()** – schedules the new coroutine for future execution, after the calling task yields control.

Triggers

Triggers are used to indicate when the cocotb scheduler should resume coroutine execution. In other words the execution of the python code has been stopped and simulation time advances until trigger condition is met. To use a trigger, a coroutine should **await** it.

Edge(signal)	Fires on any value change of signal.
RisingEdge(signal)	Fires on the rising edge of signal.
FallingEdge(signal)	Fires on the falling edge of signal.
ClockCycles(signal,num)	Fires after *num* transactions of signal from 0 to 1.
Timer(time)	Fires after the specific simulation time period elapsed.
Join(coroutine)	Fires when a task completes.
Combine(*triggers)	Fires when all triggers have fired.
First(*triggers)	Fires when the first trigger fires.
Event()	Event to permit synchronization between coroutines.

Communication between coroutines

There are two basic approaches to pass the information between coroutines:

- Using the Event() trigger, the data in transferred by setting event.data.
- Using the python **classes**. Class methods may be cocotb coroutines and at the same time have access to the class data objects.

Classes as testbenches

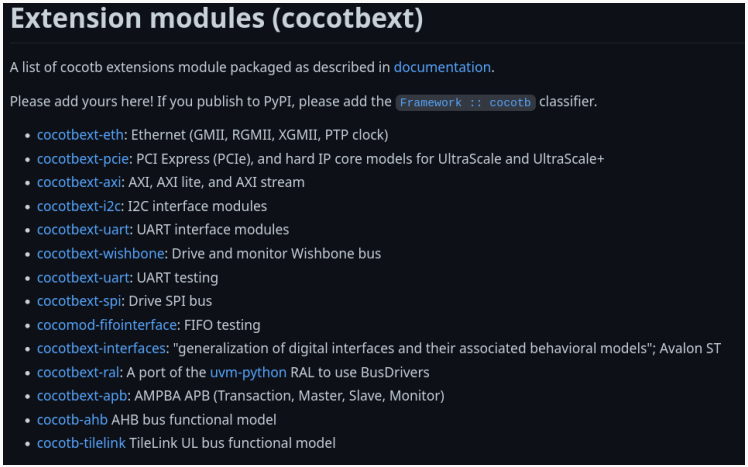
```
1  import cocotb
2  from cocotb.triggers import RisingEdge, FallingEdge
3
4  class TestBench :
5    def __init__(self,dut):
6    self.dut = dut
7    self.expected_data = [] # this will be accessible by both monitor and driver
8
9    @coroutine
10   async def drive(self,dut):
11     # drive DUT inputs with randomized/preset values
12     # set the data expected at output based on what we set at input
13
14   @coroutine
15   async def monitor(self,dut):
16     # monitor DUT outputs and compare to the expected values
17     # compare actual outputs to the expected ones
```

Cocotb extensions

cocotb-bus – testbenching tools and reusable bus interfaces

cocotb-coverage – an extension for functional coverage and constrained randomization.

A complete list of extension modules may be found here.



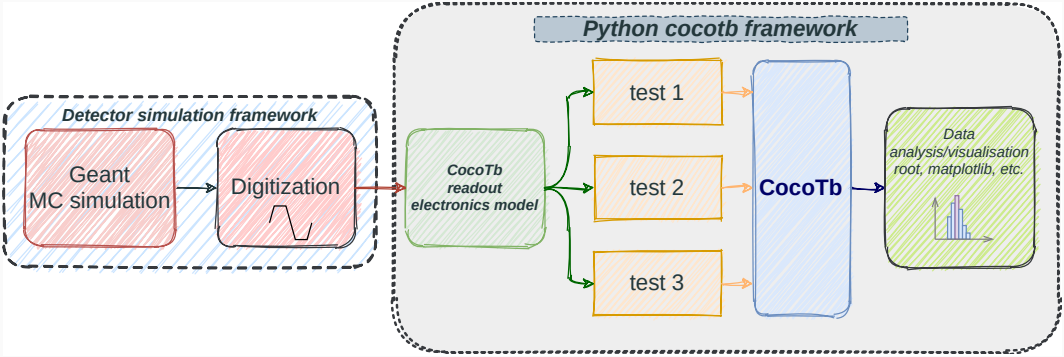
Extension modules (cocotbext)

A list of cocotb extensions module packaged as described in documentation.

Please add yours here! If you publish to PyPI, please add the `Framework :: cocotb` classifier.

- cocotbext-eth: Ethernet (GMII, RGMII, XGMII, PTP clock)
- cocotbext-pcie: PCI Express (PCIe), and hard IP core models for UltraScale and UltraScale+
- cocotbext-axi: AXI, AXI lite, and AXI stream
- cocotbext-i2c: I2C interface modules
- cocotbext-uart: UART interface modules
- cocotbext-wishbone: Drive and monitor Wishbone bus
- cocotbext-uart: UART testing
- cocotbext-spi: Drive SPI bus
- cocomod-fifointerface: FIFO testing
- cocotbext-interfaces: "generalization of digital interfaces and their associated behavioral models"; Avalon ST
- cocotbext-ral: A port of the uvm-python RAL to use BusDrivers
- cocotbext-apb: AMPBA APB (Transaction, Master, Slave, Monitor)
- cocotb-ahb AHB bus functional model
- cocotb-tilelink TileLink UL bus functional model

Integration with other tools



- Cocotb testbenches are python scripts, so they can use all the power of python.
- It's much easier to make functional models of the devices (e.g. ADC, DRS4, custom chips, etc.) in python than in HDL.
- Rapid prototyping of logic for hardware components which don't exist yet.
- Visualization and data analysis with matplotlib, CERN Root or other tools.

(to be continued...)

Further reading

The cocotb community collects all materials on cocotb available on internet:
https://github.com/cocotb/cocotb/wiki/Further-Resources
There is a complete list of extension modules, lots of links to custom libraries and examples, youtube lectures and streams, etc.

More topics might be covered

- More complex examples from real projects.
- Basics of verification theory: constrained random testing, functional coverage, etc.
- Testbenching tools for reusable bus interfaces : cocotb-bus.
- Other open-source tools for FPGA development and verification: ghdl, Yosys, X-Ray project, OSVWM.