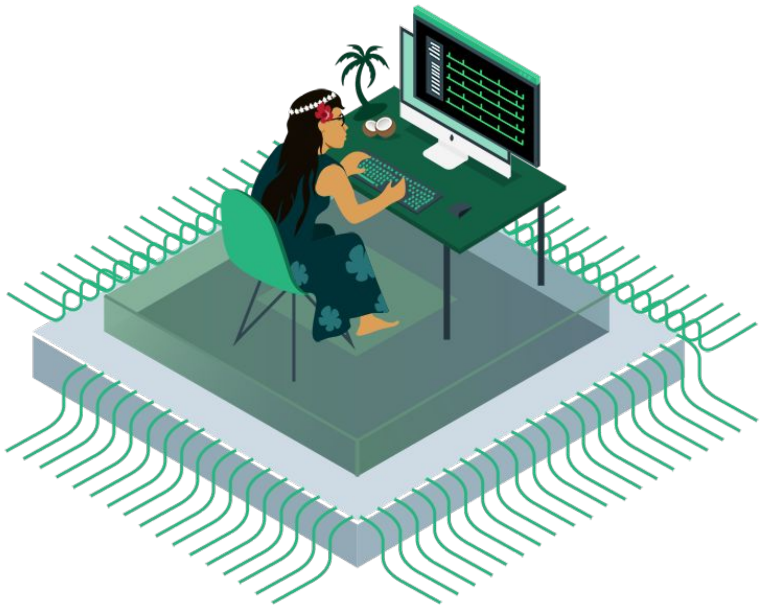
cocotb 2.0 How to get the best out of the new major version of the Python-based testbench framework

Holger Horbach, Philipp Wagner

DVCon Europe, 2024-10-15, Munich



Let’s talk about your verification experience.

Philipp and Holger

phw@ibm.com
philipp@fossi-foundation.org
@imphil on GitHub
@MrImphil@mastodon.social

holle@de.ibm.com
(and nowhere else)

In this talk

- A step-by-step introduction to cocotb.
- What’s new in cocotb 2.0?
- How IBM is using cocotb to turbo-charge its chip development.

Introducing cocotb

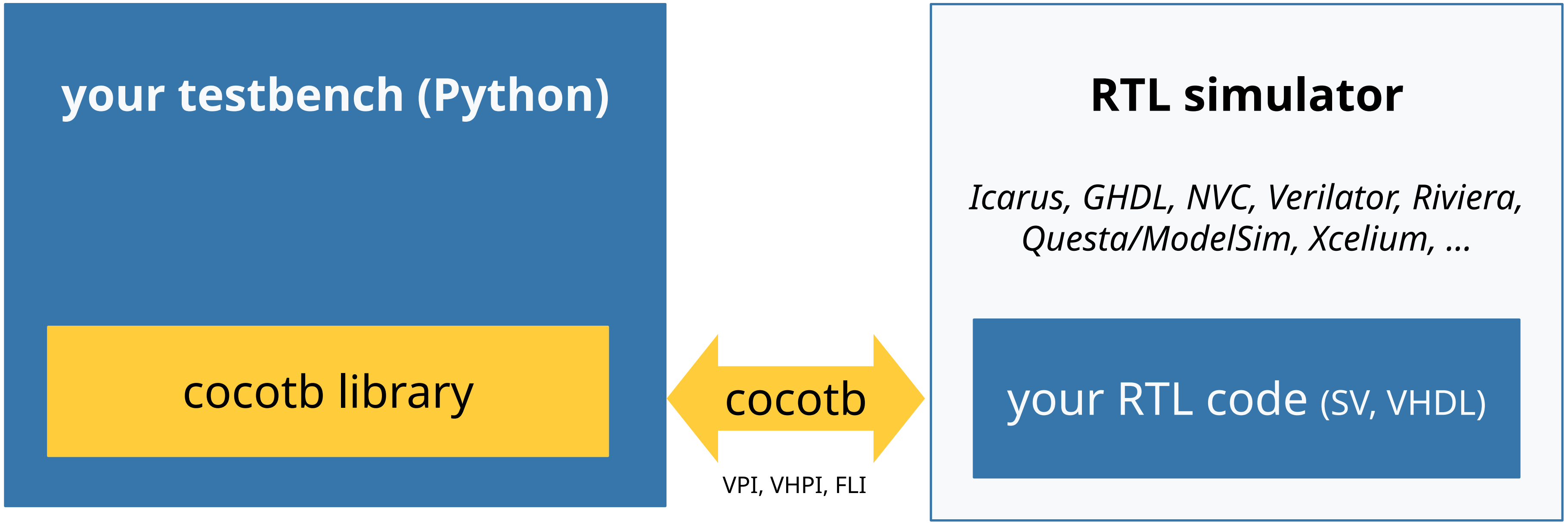
Verification is software.

With cocotb, you write verification code like software.
In Python.

What is cocotb?

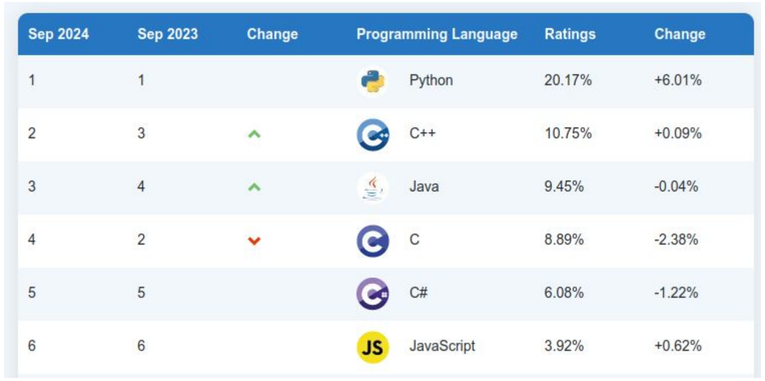
- A RTL simulator plugin
- A Python library for writing synchronous logic

cocotb overview



Why Python for verification?

- **Productive** to write, easy to read
- **Easy to interface** with
- Huge existing **ecosystem**
- **Popular language**: easy to find engineers



Sep 2024	Sep 2023	Change	Programming Language	Ratings	Change
1	1		Python	20.17%	+6.01%
2	3	▲	C++	10.75%	+0.09%
3	4	▲	Java	9.45%	-0.04%
4	2	▼	C	8.89%	-2.38%
5	5		C#	6.08%	-1.22%
6	6		JavaScript	3.92%	+0.62%

TIOBE Index, Sept 2024 (https://www.tiobe.com/tiobe-index/)

cocotb is open source

- You are free to use and modify at will.
- Developed, maintained, and supported by volunteers. Be nice!
- You can contribute. Just open a pull request on GitHub.
- You should contribute. The only guaranteed way to get improvements into cocotb.
- The FOSSi Foundation provides a legal and administrative home for the project. Sponsor them if you can.

Install cocotb

Install cocotb

```
$ # Install cocotb and (optional, but highly recommended) pytest
$ # Use a development version until cocotb 2.0 is released.
$ pip3 install git+https://github.com/cocotb/cocotb@master
$ pip3 install pytest

$ # Check the installation
$ cocotb-config --version
2.0.0.dev0+5d0f4f76
```

It's fresh! This tutorial uses a pre-release version of cocotb 2.0. Interfaces and naming conventions are not yet finalized. Please check with an up-to-date cocotb documentation when you use this material in the future!

Good to know:

- cocotb works with Windows, Linux, and Mac. Linux is strongly recommended.
- You also need a working Python installation and a simulator.

Our first cocotb test

Our first DUT: An adder

```
// adder.sv

`timescale 1ns/1ps

module adder #(
 parameter integer DataWidth = 4
) (
 input  logic [DataWidth-1:0] a_i,
 input  logic [DataWidth-1:0] b_i,
 output logic [DataWidth:0]   x_o
);

 assign x_o = a_i + b_i;
endmodule
```

A directed test for the adder

```
# test_adder_1.py

import cocotb
from cocotb.triggers import Timer


@cocotb.test
async def adder_basic_test(dut):
   """Test for 5 + 10"""

   dut.a_i.value = 5
   dut.b_i.value = 10

   await Timer(2, units="ns")

   assert dut.x_o.value.to_unsigned() == 14 # really?
```

A closer look at design access

Let’s look at `dut.x_o.value.to_unsigned()`:

`dut`

The *root handle* (the VHDL/Verilog toplevel). Passed as first argument to any `@cooctb.test` function. “dut” stands for “device under test”.

`dut.x_o`

A *handle* (pointer) to a top-level signal named `x_o` (could be a port or an internal signal).

`dut.x_o.value`

A `LogicArray` instance representing the value of signal `x_o`.

`dut.x_o.value.to_unsigned()`

A Python integer representing the value of the `x_o` as unsigned integer.

A Python script to run Icarus Verilog

```python
# run_test_adder_1.py

from cocotb_tools.runner import Icarus


def test_adder_1():
   """Simulate the adder with Icarus Verilog"""

   sim = Icarus()
   sim.build(
       sources=["adder.sv"],
       hdl_toplevel="adder",
       always=True,
   )
   sim.test(hdl_toplevel="adder", test_module="test_adder_1")


if __name__ == "__main__":
   test_adder_1()
```

And go! Oops.

```
❯ python3 run_test_adder_1.py
    -.--ns INFO     gpi                                ..mbed/gpi_embed.cpp:108  in set_program_name_in_venv        Using Python virtual environ
ment interpreter at /home/philipp/src/cocotb-tutorial/.direnv/python-3.11/bin/python
    -.--ns INFO     gpi                                ../gpi/GpiCommon.cpp:101  in gpi_print_registered_impl       VPI registered
    0.00ns INFO     cocotb                             Running on Icarus Verilog version 12.0 (stable)
    0.00ns INFO     cocotb                             Running tests with cocotb v2.0.0.dev0+5d0f4f76 from /path/to/site-packages/cocotb
    0.00ns INFO     cocotb                             Seeding Python random module with 1726854133
    0.00ns INFO     cocotb.regression                  running test_adder_1.adder_basic_test (1/1)
                                                           Test for 5 + 10
    2.00ns INFO     cocotb.regression                  test_adder_1.adder_basic_test failed
                                                       Traceback (most recent call last):
                                                         File "/path/to/01.adder/test_adder_1.py", line 16, in add
er_basic_test
                                                           assert dut.x_o.value.to_unsigned() == 14 # really?
                                                           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                                                       AssertionError: assert 15 == 14
                                                        +  where 15 = <bound method LogicArray.to_unsigned of LogicArray('01111', Range(4, 'down
to', 0))>()
                                                        +    where <bound method LogicArray.to_unsigned of LogicArray('01111', Range(4, 'downto'
, 0))> = LogicArray('01111', Range(4, 'downto', 0)).to_unsigned
                                                        +      where LogicArray('01111', Range(4, 'downto', 0)) = LogicObject(adder.x_o).value
                                                        +        where LogicObject(adder.x_o) = HierarchyObject(adder).x_o
    2.00ns INFO     cocotb.regression                  **************************************************************************************
                                                       ** TEST                             STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                                                       **************************************************************************************
                                                       ** test_adder_1.adder_basic_test   FAIL           2.00           0.00       1705.69  **
                                                       **************************************************************************************
                                                       ** TESTS=1 PASS=0 FAIL=1 SKIP=0                   2.00           0.01        229.25  **
                                                       **************************************************************************************
```

Recap: our first cocotb test

What did we learn?

- A test is a `async` Python function decorated with `@cocotb.test`.
- Access any signal in the design through the `dut` root handle.
- Make time pass by using `await Timer()`.
- Use `assert` for checking.
- Invoke the simulator to run the test through a Python script using `cocotb_tools.runner`.

What's new in cocotb 2.0?

- The Python data type of `dut.my_signal.value` changed from `BinaryValue` to `LogicArray`. The interfaces are similar, but you need to be more explicit at times (e.g., when casting to an integer).
- The `yield` syntax is gone.
- `raise TestFailure` is gone.
- Makefiles continue to be provided, but give the new Python runner a try!

Basic randomization and multi-simulator support

Configure the test runner

```python
# test_adder_2.py

import random
import cocotb
from cocotb.triggers import Timer


@cocotb.test
async def adder_randomised_test(dut):
   """Test for adding 2 random numbers multiple times"""

   for _ in range(10):
       a = random.getrandbits(4)
       b = random.getrandbits(4)

       dut.a_i.value = a
       dut.b_i.value = b

       await Timer(2, units="ns")

       assert dut.x_o.value.to_unsigned() == a + b
```

Run the test

```python
# run_test_adder_2.py

import os
from cocotb_tools.runner import get_runner


def test_adder_2():
   """Simulate the adder with the simulator specified in SIM."""

   # Example: Set build and test arguments for Icarus Verilog only.
   build_test_args = []
   if sim == "icarus":
       build_test_args = ["-g2012"]

   sim = get_runner(os.getenv("SIM", "icarus"))
   sim.build(
       sources=["adder.sv"],
       hdl_toplevel="adder",
       always=True,
       build_args=build_test_args,
   )
   sim.test(
       hdl_toplevel="adder", test_module="test_adder_2", test_args=build_test_args
   )


if __name__ == "__main__":
   test_adder_2()
```

Recap: randomize, and multi-simulator

What did we learn?

- Use Python's random module to access randomness.
- Set the environment variable `COCOTB_RANDOM_SEED` to control the seed for reproducible reruns.
- cocotb's runner provides a unified interface to all supported simulators.

What's new in cocotb 2.0?

- The environment variable `RANDOM_SEED` got renamed.

Randomize Verilog parameters (and defines) or VHDL generics

Run the test

```python
# test_adder_3.py

import random

import cocotb
from cocotb.triggers import Timer


@cocotb.test
async def adder_randomised_test(dut):
   """Test for adding two random numbers multiple times"""

   data_width = dut.DataWidth.value.to_unsigned()

   for _ in range(10):
       a = random.getrandbits(data_width)
       b = random.getrandbits(data_width)

       dut.a_i.value = a
       dut.b_i.value = b

       await Timer(2, units="ns")

       assert dut.x_o.value.to_unsigned() == a + b
```

Run the test

```python
# run_test_adder_3.py

import os
import pytest
from cocotb_tools.runner import get_runner


@pytest.mark.parametrize("data_width", [4, 8, 16, 32, 64])
def test_adder_3(data_width: int) -> None:
   """Simulate the adder with the simulator specified in SIM."""

   sim = get_runner(os.getenv("SIM", "icarus"))
   sim.build(
       sources=["adder.sv"],
       hdl_toplevel="adder",
       always=True,
       parameters={"DATA_WIDTH": data_width},
   )
   sim.test(hdl_toplevel="adder", test_module="test_adder_3")
```

Run the test

```
❯ pytest -v run_test_adder_3.py
=================================== test session starts ====================================
platform linux -- Python 3.11.9, pytest-7.4.3, pluggy-1.3.0 -- /path/to/bin/python3
cachedir: .pytest_cache
rootdir: /path/to/01.adder
plugins: xdist-3.3.1
collected 5 items

run_test_adder_3.py::test_adder_3[4] PASSED                                           [ 20%]
run_test_adder_3.py::test_adder_3[8] PASSED                                           [ 40%]
run_test_adder_3.py::test_adder_3[16] PASSED                                          [ 60%]
run_test_adder_3.py::test_adder_3[32] PASSED                                          [ 80%]
run_test_adder_3.py::test_adder_3[64] PASSED                                          [100%]

==================================== 5 passed in 1.79s =====================================
```

Recap: simulation randomization

What did we learn?

- Read VHDL generics and Verilog parameters just like any signal in the design.
- Set VHDL generics and Verilog parameters and defines when invoking the simulation through the runner.
- Start multiple simulation runs with different parameters easily by calling the runner through pytest.

What's new in cocotb 2.0?

- The combination of runner + pytest is super powerful!

Introducing coroutines

fifo.sv: Device under Test (DUT)

```
// fifo.sv

`timescale 1ns / 1ps

module fifo #(
 parameter Width = 8,
 parameter Depth = 32
) (
 input logic                clk,
 input logic                rst,

 input logic [(Width-1):0]  din,
 input logic                wr_en,
 output logic               full,

 output logic [(Width-1):0] dout,
 input logic                rd_en,
 output logic               empty
);
// implementation omitted.
endmodule
```

Our first coroutine: create a clock

```
async def create_clock(clock):
    """Coroutine to create a clock with a 1 ns period."""
    while True:
        clock.value = 1
        await Timer(0.5, units="ns")
        clock.value = 0
        await Timer(0.5, units="ns")
```

Pro tip: There are much better ways to generat clocks. Use `Clock()` if you want to generate the clock in Python. If you care about performance even more: create the clock in VHDL or Verilog (e.g., in a toplevel testbench wrapper).

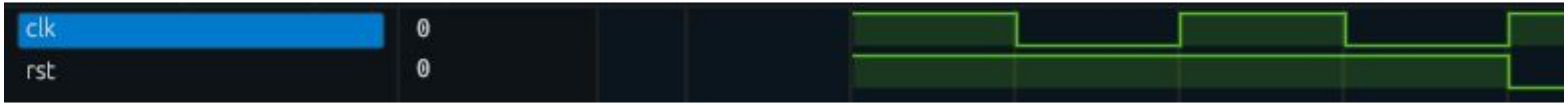
Initialize the DUT

```python
@cocotb.test
async def test fifo manual(dut):
   # clock with 1 ns period
   cocotb.start_soon(create_clock(dut.clk))

   # Better: use the cocotb library function:
   # cocotb.start_soon(Clock(dut.clk, 1, units="ns").start())

   # reset (2 cycles)
   dut.din.value = 0
   dut.wr en.value = 0
   dut.rd_en.value = 0

   dut.rst.value = 1
   await ClockCycles(dut.clk, 2)
   dut.rst.value = 0
```



Test a single FIFO write/read

```
# async def test_fifo_manual(dut) continued from previous slide
   # Write a single word into the FIFO
   dut.din.value = 100
   dut.wr_en.value = 1
   await RisingEdge(dut.clk)
   dut.wr_en.value = 0

   if dut.empty.value != 0:
       await FallingEdge(dut.empty)

   dut.rd_en.value = 1
   await RisingEdge(dut.clk)
   dut.rd_en.value = 0

   # one cycle latency
   await RisingEdge(dut.clk)

   expected_output = 100

   assert dut.dout.value.to_unsigned() == expected_output
```

Simple read and write agents

```python
@cocotb.test
async def test_fifo_random(dut):
   """A randomized test for a FIFO"""

   # Initialize.
   cocotb.start_soon(Clock(dut.clk, 1, "ns").start())
   await init_dut(dut)

   # Start read and write tasks.
   write_task = cocotb.start_soon(write_fifo(dut))
   read_task = cocotb.start_soon(read_fifo(dut))

   # Wait for read/write to finish. Read only finishes if all required data
   # has been obtained, i.e. it implicitly waits for write as well.
   await read_task

   # Check that all data has been read.
   assert not _fifo_data
```

The FIFO read coroutine

```
async def read fifo(dut):
   fifo rdcnt = 0
   while True:
       if dut.empty.value:
           await Edge(dut.clk)
       else:
           dut.rd en.value = 1
           fifo_rdcnt += 1

           await RisingEdge(dut.clk)

           dut.rd en.value = 0
           await RisingEdge(dut.clk)

           data_read = dut.dout.value

           data expected =  fifo data.pop(0)
           assert data_read == data_expected

       if fifo rdcnt >= TEST_ITEM_CNT:
           return
```

Recap: introducing coroutines

What did we learn?

- Coroutines are modeling concurrency.
- cocotb is single-threaded. Either the simulator runs, or a coroutine until it hits the next `await` statement.
- Use `cocotb.start_soon()` to schedule a coroutine to run concurrently. Use `await cocotb.start()` in special cases.
- You can create a clock in Python with `Clock()`.

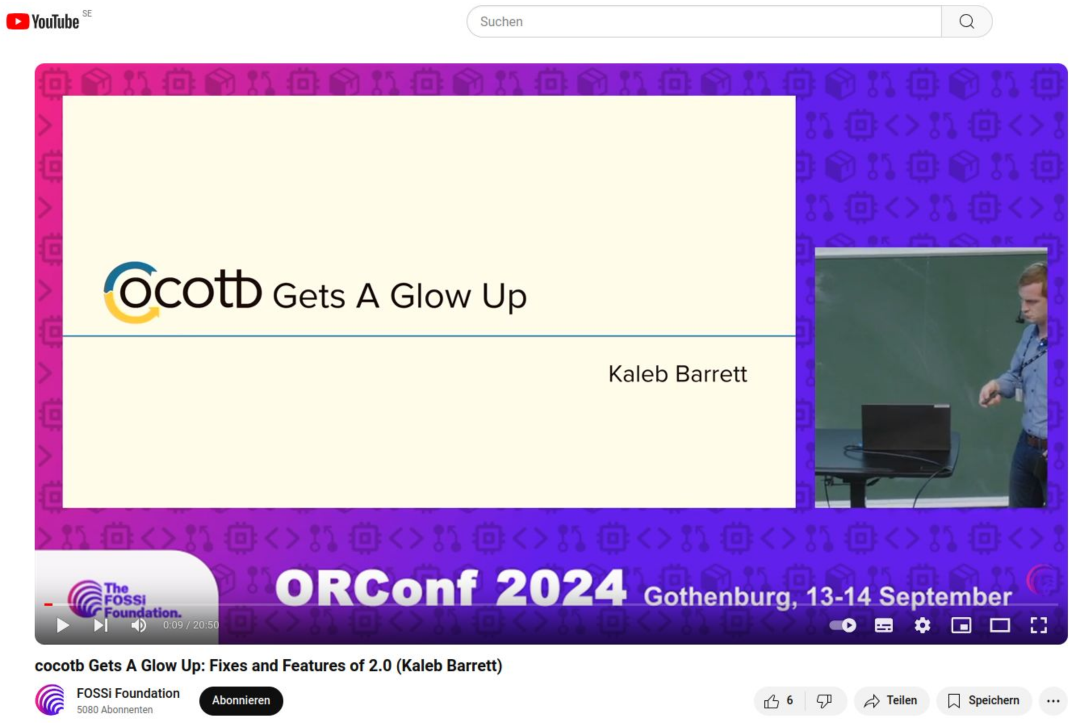
What's new in cocotb 2.0?

- No more `cocotb.fork()`.
- Use `my_task.cancel()` instead of `my_task.kill()`
- TaskManager will be added (API design not fully finalized yet).
- `Clock()` is now faster than before. Use it instead of hand-rolling your clock coroutine!

More about cocotb 2.0

Where do I learn more about cocotb 2.0?

- Kaleb Barrett. cocotb Gets A Glow Up: Fixes and Features of 2.0. ORConf 2024. Slides and video available at https://orconf.org
- In-Progress release notes at https://docs.cocotb.org/en/latest/release_notes.html



cocotb Gets A Glow Up: Fixes and Features of 2.0 (Kaleb Barrett)

Migrating to cocotb 2.0

1. Upgrade to the latest cocotb 1.x.
2. Resolve all deprecation warnings.
3. Upgrade to cocotb 2.0 (or a current master version)
4. Fix remaining issues.

A migration guide will be released together with cocotb 2.0. Tell us your migration experience by opening a GitHub Discussion item at https://github.com/cocotb/cocotb/discussions.

```
/path/to/02.fifo/test_fifo.py:111: DeprecationWarning:
Using `task` directly is prefered to`task.join()` in all
situations where the latter could be used.`
 await read_thread.join()
 2025.00ns DEBUG    cocotb.fifo
WRITE: Wait for 1 clock cycles
 2025.00ns DEBUG    cocotb.fifo
WRITE: wait
 2025.00ns DEBUG    cocotb.fifo
READ: Wait for 0 clock cycles
/path/to/site-packages/cocotb/types/logic_array.py:807:
FutureWarning: The behavior of bool casts and using
LogicArray in conditionals may change in the future. Use
explicit comparisons (e.g. `LogicArray() == "11"`) instead
 warnings.warn(
 2025.00ns DEBUG    cocotb.fifo
READ: FIFO empty, not reading
 2025.00ns DEBUG    cocotb.fifo
READ: Wait for 3 clock cycles
 3025.00ns DEBUG    cocotb.fifo
WRITE: wait for falling clock edge
 4025.00ns DEBUG    cocotb.fifo
WRITE: Wrote word 1 to FIFO, value 0x0
/path/to/02.fifo/fifo_test_common.py:134:
DeprecationWarning: `.integer` property is deprecated. Use
`value.to_unsigned()` instead.
```

Taking things further

More cocotb resources

cocotb is intentionally slim; it’s a robust, unopinionated base for Python-based verification.

The cocotb ecosystem provides a choices of

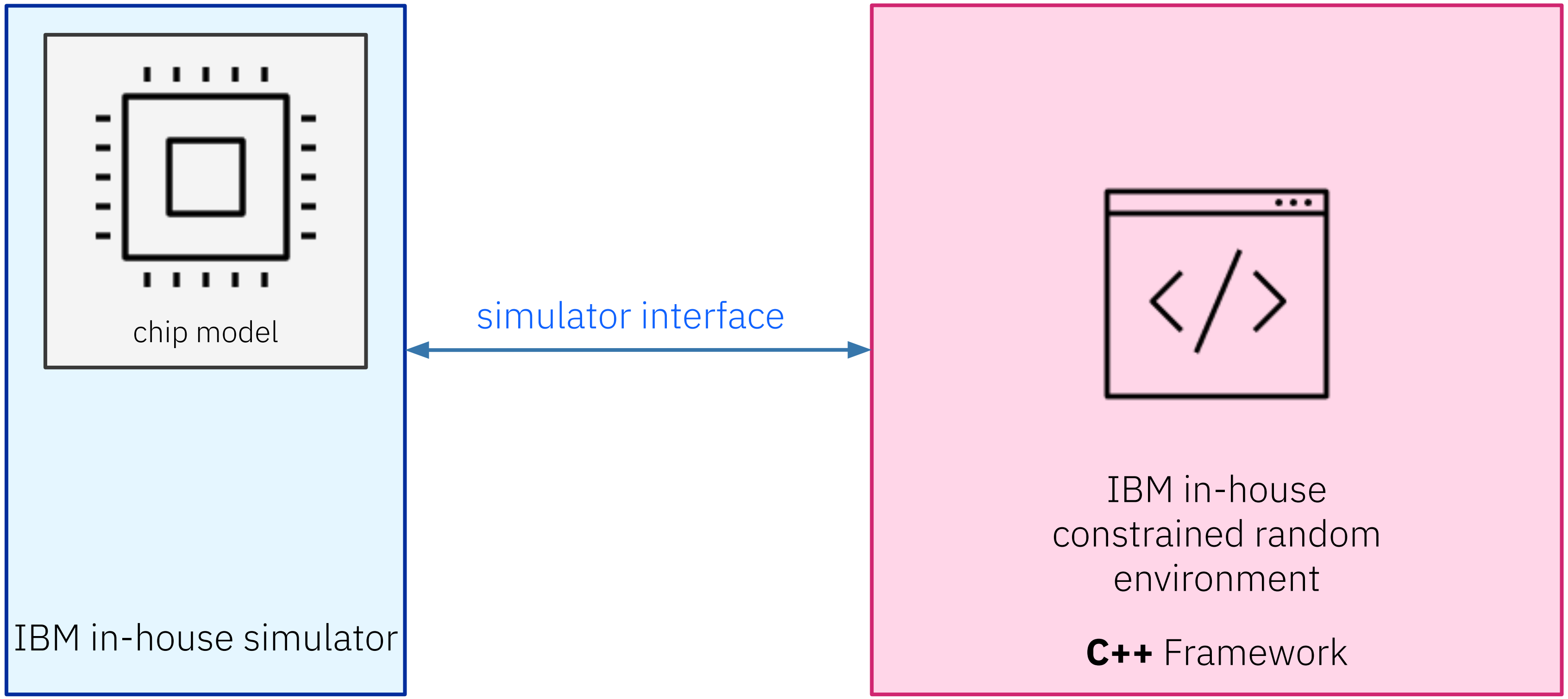
- Methodology
- Constraint random
- Functional coverage
- Verification IP (VIP), e.g., bus adapters

Have a look at https://github.com/cocotb/cocotb/wiki/Further-Resources for inspiration!

Let's hear more from cocotb users.

Today: cocotb x IBM

cocotb x IBM : C++-based random verification at IBM



testbench framework

Why change what we have?

It's too complicated!

* any designer sim user ever

Are you guys not using any industry-standard tools?

* New hires

I read about this great Python lib and want to play around with it in my verif env!

* Innovation driver

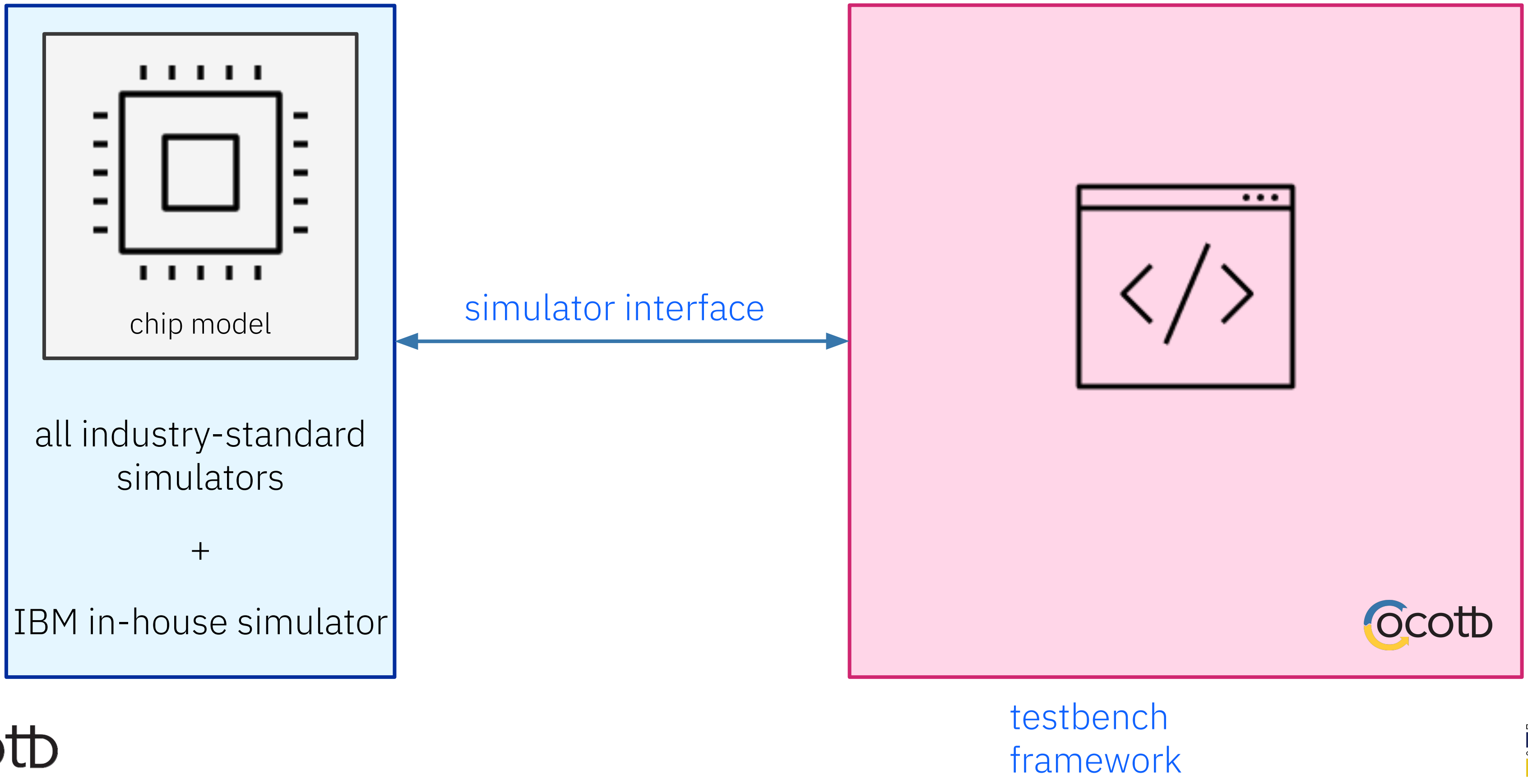
* anyone

I just want to get this quick test going!

* verif engineer + logic designer

We're not speaking the same language!

cocotb x IBM: open source verification with Python at IBM



cocotb x IBM

Designer Simulation Enablement

Scenario

New hire logic designer without prior exposure to in-house IBM verification methodology

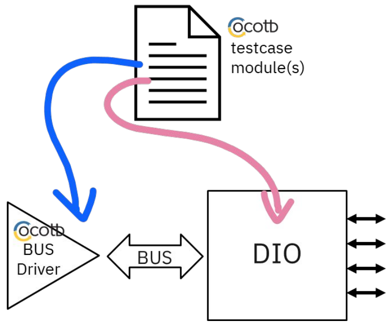
… with a desire to pre-verify newly written logic by himself before "handing over" to verification team

… spending 1 hour for initial cocotb sandbox set up

modular multi-project designer simulation env

First cocotb simulation interface driver available 24h later
(coded w/o external help)

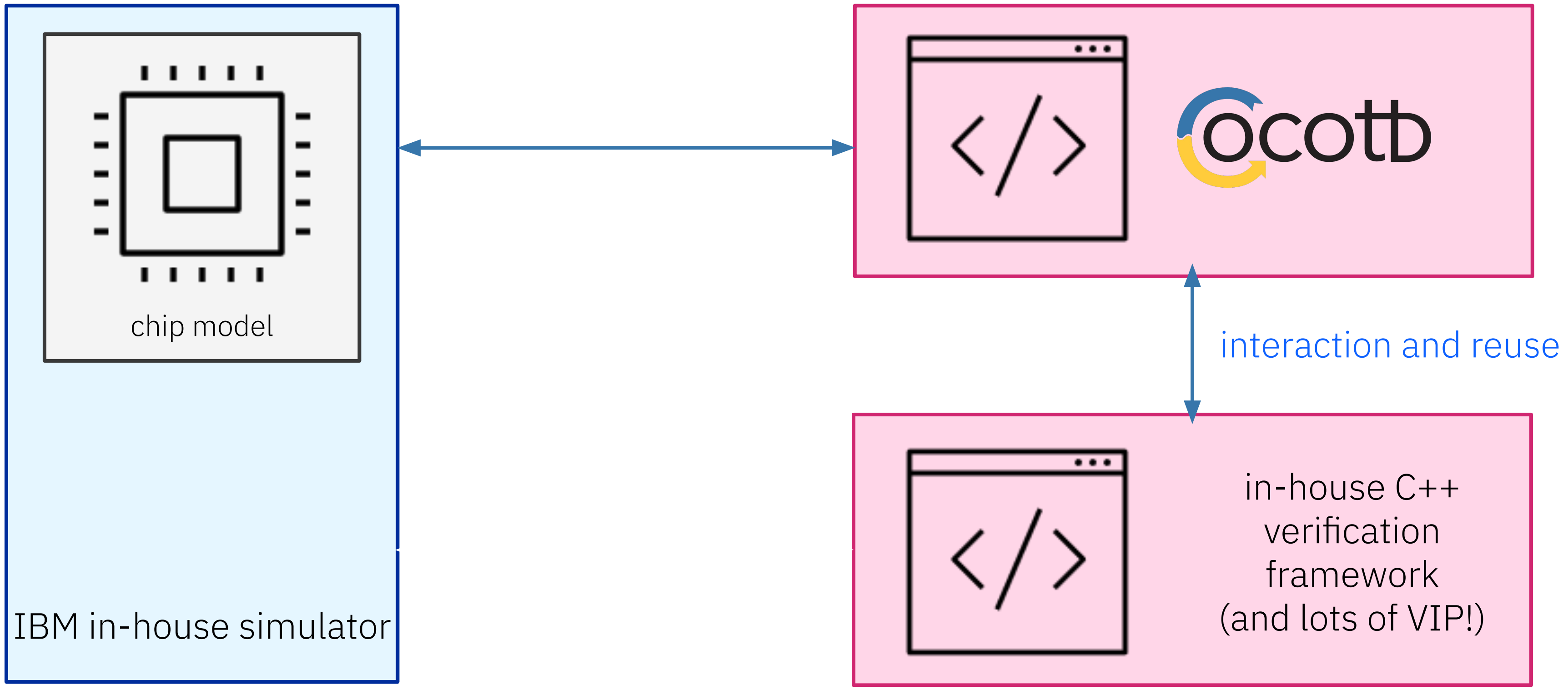
- adding capability for writing low-level directed testcases to trigger very specific scenarios designer cares for
- allowing repro of a field fail



Environment quickly extended to cover additional interfaces now owned and run by multiple team members

Environment completely integrated into IBM Hardware Design tools flow and now integral part of CI pipeline

cocotb x IBM : open source verification with Python at IBM



cocotb x IBM : connect to existing C++ VIPs

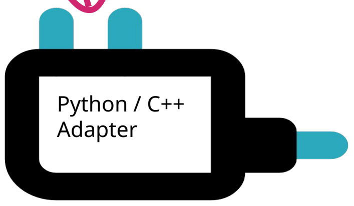
MODULE=tilelink_write.py,tilelink_adapter.pyx

tilelink_write.**py**

```python
import cocotb
from tilelink_support import tl

@cocotb.test()
async def test_tilelink_write(dut):
    """Simple test of a basic TL MMIO write"""
    reg = 0x42
    data = 0xdeadbeef
    await tl.register_write(reg, data)
```

Cython



cocotb

tilelink_vip_adapter.**pyx**

```python
import cocotb

cdef enum SIM_STATUS:
    cDone = 0, cBusy = 1

cdef extern from "util/TileLinkCmd.h" namespace "TileLink":
    cdef enum eTileLinkCmd:
        cGet, cAccessAckData, cPutFullData, cPutPartialData, cAccessAck
    cdef enum eTileLinkChannel:
        cA, cB, cC, cD, cE, cINVALIDCH
    cdef cppclass TileLinkCmd:
        TileLinkCmd( eTileLinkCmd, uint64_t addr, uint32_t size)

cdef extern from "driver/TileLinkRequestDriver.h":
    cdef cppclass TileLinkRequestDriver:
        void addCmd(TileLinkCmd)
        SIM_STATUS status()

cdef class TileLinkRequest:
    cdef TileLinkRequestDriver * c_drv

    def __cinit__(self):
        self.c_drv = get_obj_ptr[TileLinkRequestDriver]("TileLinkRequestDriver", "BRG")

    async def register_write(self, addr : uint64, data : uint64):
        cdef TileLinkCmd cmd = TileLinkCmd(cPutFullData, address, 3) # 2^3 bytes
        self.c_drv.addCmd(cmd)
        while (self.c_drv.status() != cDone): await cocotb.Timer(1)

tl = TileLinkRequest()
```

in-house C++ verification framework
(and lots of VIP!)ds

cocotb

2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE MUNICH, GERMANY OCTOBER 15-16, 2024 cocotb x IBM

Design + Verification Collaboration

Scenario

Experienced logic designer for a new unit

… wanting to get a head start on unit verification before availability of verif engineer (later in project)

… with existing verification IP (interface drivers)

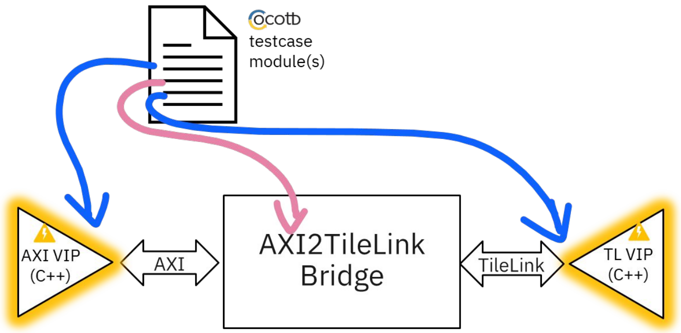
… spending 1 day w/ verif engineer for initial IP integration effort

Fully regressionable sim env w/ verif IP

First developed and run by designer to create directed specialized scenarios

Later picked up by verif engineer and extended to allow random scenarios and functional coverage for regression

Full reuse of verification IP and seamless integration into environment



Easy collaboration between designer and verif engineer using common language: Python!

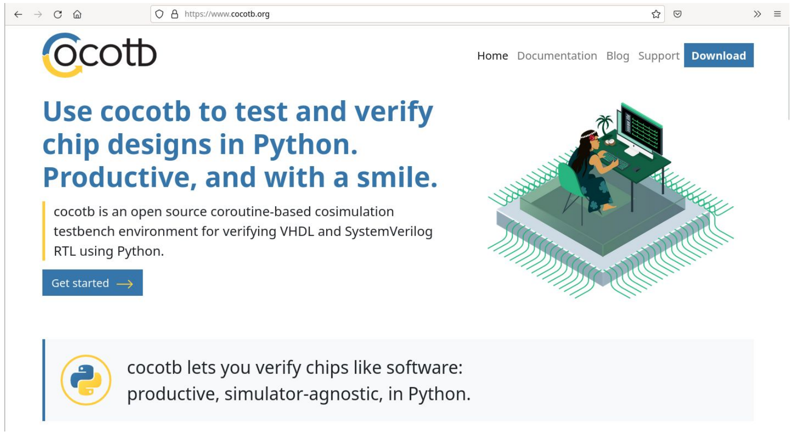
cocotb

cocotb x IBM : building a verif tools dream team

	C++ state-of-the-art	cocotb	both
Encourage reuse	✓	✓	✓
Tailored for IBM	✓		✓
Native code for performance sensitive testbenches	✓		✓
Good collaboration			
… between verification teams	✓	✓	✓
… between logic and verification		✓	✓
Easy-to-learn testbench language		✓	✓
Used beyond IBM		✓	✓
Easy to contribute to		✓	✓
Easy to integrate third-party modules		✓	✓

Remember

- Verification is software.
- cocotb 2.0 is around the corner. Start testing it now!
- cocotb is not just verification in Python – it can turbo-charge your chip development.



Use cocotb to test and verify chip designs in Python. Productive, and with a smile.

cocotb is an open source coroutine-based cosimulation testbench environment for verifying VHDL and SystemVerilog RTL using Python.

cocotb lets you verify chips like software: productive, simulator-agnostic, in Python.

www.cocotb.org
GitHub: cocotb
LinkedIn: cocotb
Twitter/X: @cocotbnews