



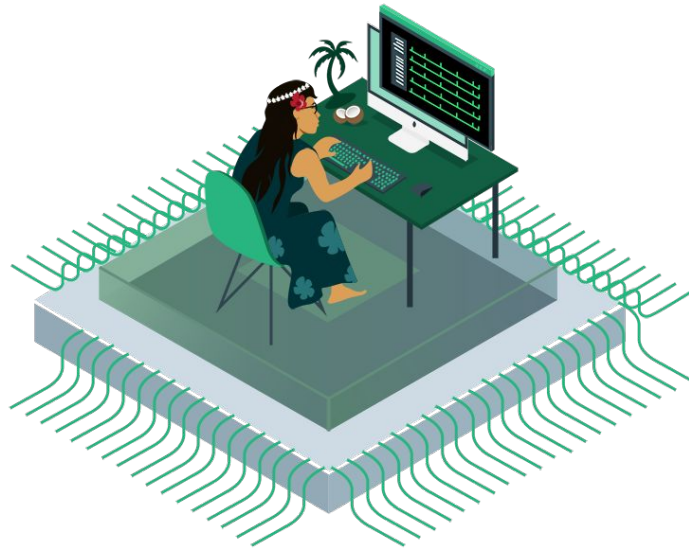
cocotb 2.0: Modernize your testbenches for even more productivity

Philipp Wagner

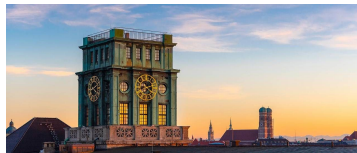
Kaleb Barrett

DVClub Europe

Oct 8, 2024



About Philipp



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

Why cocotb 2.0?

- Needed API breaking changes
 - Replacing footguns
 - Refining public API
 - General code cleanup
- Break as little backwards compatibility as possible
 - **NOT** like Python 2 to Python 3

Glow Up

A [Glow](#) Up is a mental, physical, and an emotional [transformation](#) for the better. Glow [Ups](#) can be both natural or planned. As well as being gradual and permanent, or fast and temporary.

She has really had a [glow up](#) since [the last time I saw](#) her!

by [Werewolf_Girl](#) April 15, 2020

 932

 83

 FLAG

Get the **Glow Up** mug.



Project Automation

Python Runner

What is it?

- Designed to build most HDL designs
- Runs cocotb simulations
- Replacement for Makefiles
- Based on `cocotb-test`

Why prefer Python?

- Python > Make language
- Better cross-platform support
- pytest ecosystem

```
1
2  def test_alu():
3      sim = runner.Icarus()
4      sim.build(
5          sources=[
6              "src/adder.sv",
7              "src/multiplier.sv",
8              "src/alu.sv",
9          ],
10         includes=[
11             "src/"
12         ],
13         defines={
14             "SIM": "1",
15         },
16     )
17     sim.test(
18         test_module="alu_tests",
19         waves=True,
20     )
21
```

Python Runner

Python Runner

```
1
2 SIM := icarus
3
4 ✓ VERILOG_SOURCES := \
5   | src/adder.sv \
6   | src/multiplier.sv \
7   | src/alu.sv
8 COMPILE_ARGS += -DSIM=1 # turn on simulation-only code
9 COMPILE_ARGS += -Isrc/
10
11 MODULE := alu_tests
12 WAVES ?= 1
13
14 include $(shell cocotb-config --makefiles)/Makefile.sim
15
```

Makefile

```
1
2 ✓ def test_alu():
3     sim = runner.Icarus()
4     sim.build(
5         sources=[
6             "src/adder.sv",
7             "src/multiplier.sv",
8             "src/alu.sv",
9         ],
10        includes=[
11            "src/"
12        ],
13        defines={
14            "SIM": "1",
15        },
16    )
17 ✓ sim.test(
18     test_module="alu_tests",
19     waves=True,
20 )
21
```

Python Runner

Python Runner

```
3
4 @pytest.mark.parametrize("depth", [16, 64, 1024, 4096])
5 @pytest.mark.parametrize("width", [1, 8, 64, 512])
6 @pytest.mark.parametrize("valid_fraction", [0.5, 1.0])
7 @pytest.mark.parametrize("ready_fraction", [0.5, 1.0])
8 def test_fifo(depth: int, width: int, valid_fraction: float, ready_fraction: float) -> None:
9
10     sim = Questa()
11     sim.build(
12         build_dir=f"build_{depth}_{width}_{valid_fraction}_{ready_fraction}",
13         parameters={
14             "DEPTH": depth,
15             "WIDTH": width,
16         },
17         sources=["fifo.vhd", "fifo_top.sv"], # mixed language build
18         args=[VHDL("-2008")], # VHDL-only build args
19     )
20     sim.test(
21         test_module="fifo_tests",
22         plusargs=[
23             f"+valid_fraction={valid_fraction}", f"+ready_fraction={ready_fraction}"
24         ]
25     )
26
```



Test Discovery and Generation

Test Discovery Changes

What changed?

- `cocotb.test` decorator no longer returns Test objects
- Tests are placed in special attribute in module where they are defined

Why?

- Allows reusing test definitions
- Import test into different modules without adding the test to the module

Potential Incompatibilities

- Custom test generation utilities

```
1
2  @pyuvvm.test()
3  class TestA(uvm_test):
4      def build_phase(self):
5          ...
6
7      async def run_phase(self):
8          ...
9
10
11  @pyuvvm.test()
12  class TestB(TestA):
13      async def run_phase(self):
14          ... # different test, same structure
15
```

cocotb.parametrize()

What is it?

- Write test once, generate many related tests
- Replacement for TestFactory
- Inspired by `pytest.mark.parametrize`

Why replace TestFactory?

- Lists options near test function parameters
- Generates better tests names
- Familiar to pytest users

```
2
3 @cocotb.test
4 @cocotb.parametrize(
5     valid_fraction=[0.5, 1.0],
6     ready_fraction=[0.5, 1.0],
7 )
8 async def test_fifo(
9     dut,
10     valid_fraction: float,
11     ready_fraction: float,
12 ) -> None:
13     ...
14
```

cocotb.parametrize()

```
3
4 @cocotb.test
5 async def test_fifo(
6     dut,
7     valid_fraction: float,
8     ready_fraction: float
9 ) -> None:
10     ...
11
12 tf = TestFactory(test_fifo)
13 tf.add_option("valid_fraction", [0.5, 1.0])
14 tf.add_option("ready_fraction", [0.5, 1.0])
15 tf.generate_tests()
16
```

```
*****
** TEST                                STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
*****
** example_tests.test_fifo_001         PASS      0.00      0.00      0.00 **
** example_tests.test_fifo_002         PASS      0.00      0.00      9.00 **
** example_tests.test_fifo_003         PASS      0.00      0.00     12.45 **
** example_tests.test_fifo_004         PASS      0.00      0.00     13.03 **
*****
** TESTS=4 PASS=4 FAIL=0 SKIP=0        0.00      0.00      0.70 **
*****
```

TestFactory

cocotb.parametrize()

```
*****
** TEST                                STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
*****
** example_tests.test_fifo/valid_fraction=0.5/ready_fraction=0.5  PASS      0.00      0.00      0.00 **
** example_tests.test_fifo/valid_fraction=0.5/ready_fraction=1.0  PASS      0.00      0.00      8.47 **
** example_tests.test_fifo/valid_fraction=1.0/ready_fraction=0.5  PASS      0.00      0.00     16.98 **
** example_tests.test_fifo/valid_fraction=1.0/ready_fraction=1.0  PASS      0.00      0.00     19.42 **
*****
** TESTS=4 PASS=4 FAIL=0 SKIP=0        0.00      0.02      0.13 **
*****
```

```
2
3 @cocotb.test
4 @cocotb.parametrize(
5     valid_fraction=[0.5, 1.0],
6     ready_fraction=[0.5, 1.0],
7 )
8 async def test_fifo(
9     dut,
10     valid_fraction: float,
11     ready_fraction: float,
12 ) -> None:
13     ...
14
```

COCOTB_TEST_FILTER

What?

- A regular expression filter for selecting tests
- Replacement for TESTCASE Makefile variable or testcase argument to `Runner.test()`

Why replace current methods?

- More expressive than current methods
- Works well with `cocotb.parametrize`
- Regular expressions are well understood

```

1
2 @cocotb.test
3 @cocotb.parametrize(
4     error_rate=[0.0, 0.01, 0.8],
5     poll_rate=[50, 1000, 20000],
6     resync_rate=[0, 100],
7 )
8 async def test_external_device(
9     dut,
10     error_rate: float,
11     poll_rate: int,
12     resync_rate: int
13 ) -> None:
14     ...
15

```

Run all parametrized tests

COCOTB_TEST_FILTER

```

(venv) ~/cocotb_examples
> make COCOTB_TEST_FILTER=test_external_device

```

TESTCASE

```

(venv) ~/cocotb_examples
> make TESTCASE=test_external_device/error_rate=0.0/poll_rate=50/resync_rate=0,example_tests.test_external_
e=0.0/poll_rate=20000/resync_rate=0,example_tests.test_external_device/error_rate=0.0/poll_rate=20000/resyn
e_tests.test_external_device/error_rate=0.01/poll_rate=1000/resync_rate=100,example_tests.test_external_dev
=0.8/poll_rate=50/resync_rate=100,example_tests.test_external_device/error_rate=0.8/poll_rate=1000/resync_r

```

```

1
2 @cocotb.test
3 @cocotb.parametrize(
4     error_rate=[0.0, 0.01, 0.8],
5     poll_rate=[50, 1000, 20000],
6     resync_rate=[0, 100],
7 )
8 async def test_external_device(
9     dut,
10     error_rate: float,
11     poll_rate: int,
12     resync_rate: int
13 ) -> None:
14     ...
15

```

Run only tests where poll_rate == 50

```

(venv) ~/cocotb_examples
> make COCOTB_TEST_FILTER="test_external_device.*poll_rate=50"

```

```

*****
** TEST                                STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
*****
** example_tests.test_external_device/error_rate=0.0/poll_rate=50/resync_rate=0  PASS      0.00      0.00      0.00 **
** example_tests.test_external_device/error_rate=0.0/poll_rate=50/resync_rate=100 PASS      0.00      0.00     10.16 **
** example_tests.test_external_device/error_rate=0.01/poll_rate=50/resync_rate=0  PASS      0.00      0.00     16.91 **
** example_tests.test_external_device/error_rate=0.01/poll_rate=50/resync_rate=100 PASS      0.00      0.00     19.24 **
** example_tests.test_external_device/error_rate=0.8/poll_rate=50/resync_rate=0  PASS      0.00      0.00     19.33 **
** example_tests.test_external_device/error_rate=0.8/poll_rate=50/resync_rate=100 PASS      0.00      0.00     19.24 **
*****
** TESTS=6 PASS=6 FAIL=0 SKIP=0                                0.01      0.01      0.49 **
*****

```



Modeling Types

New Modeling Types

What are the new modeling types?

- Set of types to deal with common HDL values
 - Logic: 9-value logic scalar type
 - Range: right-bound-inclusive integer range
 - Array: `list`-like, arbitrarily-indexed, immutably-sized
 - LogicArray: Array of Logic + bitwise ops
- Replacement for `BinaryValue`

Why replace `BinaryValue`?

- Bugs
- Fundamentally assumes 0/1 values
- No indexing/slicing

Array

- Like Python list
- Immutable size
- Uses Range to describe arbitrary indexing

```
1
2 # Deduced Range
3 >>> Array([1, "2", None, (1,)])
4 Array([1, "2", None, (1,)], Range(0, "to", 3))
5
6
7 # Explicit Range
8 >>> a = Array("1234", Range(56, "downto", 53))
9 >> a
10 Array(["1", "2", "3", "4"], Range(56, "downto", 53))
11
12
13 # Uses arbitrary indexing
14 >>> a[56]
15 "1"
16 >>> a[53]
17 "4"
18 >>> a[56:55]
19 Array(["1", "2"], Range(56, "downto", 55))
20 >>> a[53] = 170
21 >> a
22 Array(["1", "2", "3", 170], Range(56, "downto", 53))
23
24
25 # Works like list
26 >>> len(a)
27 4
28 >>> for elem in a:
29 ...     print (elem)
30 "1"
31 "2"
32 "3"
33 170
34
```

LogicArray

- Similar to Array
- Constructs values into Logic
- Supports bitwise operations
- Convertible to/from int, str, bytes, sequences of Logic

```
1
2 # Automatic conversion to Logic
3 >>> l = LogicArray([0, 1, 1, 0])
4 >>> l[3:1] = "100"
5 >>> l
6 LogicArray("1000", Range(3, "downto", 0))
7
8
9 # Bitwise operations
10 >>> l & LogicArray("01XZ")
11 LogicArray("00XX", Range(3, "downto", 0))
12 >>> l | LogicArray("0H0H")
13 LogicArray("1101", Range(3, "downto", 0))
14 >>> ~l
15 LogicArray("0111", Range(3, "downto", 0))
16
17
18 # Conversion to/from integers
19 >>> LogicArray.from_signed(127, 8)
20 LogicArray("01111111", Range(7, "downto", 0))
21 >>> LogicArray("1000").to_unsigned()
22 8
23 >>> LogicArray("1000").to_signed()
24 -8
25
26
27 # Conversion to/from bytes
28 >>> LogicArray.from_bytes(b"\x0F\xF0", byteorder="big")
29 LogicArray("0000111111110000", Range(15, "downto", 0))
30 >>> LogicArray("01010101").to_bytes(byteorder="little")
31 b"U"
32
```

Incompatibilities

- `binaryRepresentation` removed
 - Use `from_signed`, `from_unsigned`, `to_signed`, `to_unsigned`
- `bigEndian` removed
 - Use `from_bytes`, `to_bytes`, takes `byteorder`
- Arithmetic operations removed
 - Convert to `int` first
 - Future work: Signed and Unsigned modeling types
- `COCOTB_RESOLVE_X` removed
 - Equality between ints and Xs no longer `ValueError`
 - Use `is_resolvable` to check for Xs before converting to int or bytes



Handles

Value Type Changes

What changed?

- Signal values are now LogicArray, not BinaryValue
- Array signal value are now Array, not list

Compatibilities

- Equality with existing types
- `.binstr, str()`
- `.signed_integer, .integer, int()`
- `.buff`
- `.is_resolvable`
- `bool()`, conditionals

```
1
2 @cocotb.test
3 async def test_thing(dut) -> None:
4     # .binstr, str()
5     while dut.rst.value.binstr != "0":
6         await Edge(dut.rst)
7
8     # int(), .integer, .signed_integer
9     PIPE_DEPTH = int(dut.PIPE_DEPTH.value)
10
11     dut.input.value = 1
12     for i in range(PIPE_DEPTH):
13         # equality with int
14         assert dut.output.value == 0
15         await RisingEdge(dut.clk)
16
17     # equality with str
18     assert dut.output.value == "1"
19
```

Value Type Changes

Incompatibilities

- No more BinaryValue
- Indexing on arrayed signals uses HDL indexes

```
1
2 @cocotb.test
3 async def test_thing(dut) -> None:
4     #
5     # logic [7:0] array_signal [3:0];
6     #
7
8     # Works
9     assert dut.array_signal.value == [0, 1, 2, 3]
10
11    # Breaks
12    array_value = dut.array_signal.value
13    assert array_value[0] == 0 # Fails, actual is 3
14
```

New [] Indexing Syntax

What?

- Get child objects in design hierarchy
- Alternative to dot syntax

Why?

- Dot syntax limited to Python identifiers
- May conflict with Python object attributes

When to use over dot syntax?

- Extended identifiers
- Name collisions

```
1
2 @cocotb.test
3 async def test_thing(dut) -> None:
4
5     # Works interchangeably with dot syntax
6     dut["my_module"].my_signal
7
8     # Python attribute '_log' shadows child signal
9     dut.my_module["_log"]
10
11     # Valid signal name is not valid Python
12     dut.my_module["my$signal"]
13     dut.my_module["\\my extended identifier!\\"]
14
```

Other Handle *Incompatibilities*

- Handle classes refactored and renamed
- Support for indexing into packed objects removed
 - Wasn't consistently supported
 - Edge triggers weren't consistent
 - Setting value wasn't consistent
 - Index values instead
 - Or use unpacked objects

```
2
3 @cocotb.test
4 async def test_thing(dut) -> None:
5
6     async def drive_inputs() -> None:
7         try:
8             while True:
9                 await drive()
10        finally:
11            # Want this to run if the driver is cancelled
12            drive_idle()
13
14    # Drive inputs concurrently
15    drive_task = cocotb.start_soon(drive_inputs())
16
17    # Check 100 outputs
18    for _ in range(100):
19        await check()
20
21    # Cancel task to stop driving input
22    drive_task.cancel()
23
24    ... # test continues
25
```




Concurrency

cocotb.start_soon()

What is it?

- Way to run multiple independent threads of execution concurrently
- Replacement for `cocotb.fork()`

Why replace `cocotb.fork()`?

- Schedule re-entrancy causing bugs
- Inconsistent handling of Exceptions

```
1
2 @cocotb.test
3 async def test_dff(dut) -> None:
4
5     async def drive_inputs() -> None:
6         dut.input.value = 0
7         while True:
8             await RisingEdge(dut.clk)
9             dut.input.value = int(dut.input.value) + 1
10
11     # Drive input concurrently to output checking
12     cocotb.start_soon(drive_inputs())
13
14     # Check outputs
15     for _ in range(100):
16         prev_value = int(dut.output.value)
17         await RisingEdge(dut.clk)
18         assert dut.output.value == prev_value + 1
19
```

cocotb.start_soon()

Incompatibilities

- Runs *soon*, not immediately
- In those cases use `cocotb.start()`

Read more about the differences at
<https://fossi-foundation.org/blog/2021-10-20-cocotb-1-6-0>.

```
1
2 @cocotb.test
3 async def test_thing(dut) -> None:
4     driver_running = False
5
6     async def drive_inputs() -> None:
7         nonlocal driver_running
8         driver_running = True
9
10    # Fails, 'drive_inputs' hasn't started.
11    cocotb.start_soon(drive_inputs())
12    assert driver_running
13
14    # Passes, Forces 'drive_inputs' to start.
15    await cocotb.start(drive_inputs())
16    assert driver_running
17
```

Task.cancel()

What is it?

- Stops a Task from running
- Task.cancel() replaces Task.kill()
- Inspired by asyncio and trio

Why replace Task.kill()?

- Scheduler re-entrancy causes bugs
- Doesn't allow Tasks to "clean up"

How does it work?

- Schedules CancelledError to be thrown into cancelled Tasks
- Use context manager and try-finally blocks

```
2
3 @cocotb.test
4 async def test_thing(dut) -> None:
5
6     async def drive_inputs() -> None:
7         try:
8             while True:
9                 await drive()
10            finally:
11                # Want this to run if the driver is cancelled
12                drive_idle()
13
14        # Drive inputs concurrently
15        drive_task = cocotb.start_soon(drive_inputs())
16
17        # Check 100 outputs
18        for _ in range(100):
19            await check()
20
21        # Cancel task to stop driving input
22        drive_task.cancel()
23
24        ... # test continues
25
```

TaskManager

coming soon

What is it for?

- Structuring *asymmetric* concurrency
- Inspired by SV's fork/join, trio's nurseries, asyncio's TaskGroup
- Easy to use correctly
- Hard to use incorrectly

When to use it?

- When your test or components need to do multiple things at the same time.
- Use `cocotb.start_soon()` to run independent components (Drivers, Monitors, etc.)

```
3
4 @cocotb.test
5 async def test_thing(dut) -> None:
6
7     async with TaskManager() as tm:
8
9         @tm.fork
10        async def drive_input() -> None:
11            for _ in range(100):
12                await drive()
13
14        @tm.fork
15        async def check_output() -> None:
16            for _ in range(100):
17                await check()
18
19        ... # joins all when context ends
20
```

TaskManager

- Create child Tasks
 - `@fork` decorator
 - `start_soon`, `start`
- Implicit “Join all” if context ends
- `await` Tasks
- Cancel Tasks
- Catch Exceptions in Tasks

coming soon

```
4
5 @cocotb.test
6 async def test_thing(dut) -> None:
7     async with TaskManager() as tm:
8
9         @tm.fork
10        async def drive_input() -> None:
11            while True:
12                await drive()
13
14        # start with start_soon
15        drive_task = tm.start_soon(drive_input())
16
17        # start with fork decorator
18        @tm.fork
19        async def check_output() -> None:
20            for _ in range(100):
21                await check()
22
23        try:
24            # wait for child Task to finish
25            await check_output
26        except Exception:
27            # handle Exception from child Task
28            ...
29
30        # cancel child Task
31        drive_task.cancel()
32
33        # all tasks are done by now
34
```

TaskManager

- Use ExceptionGroups
- Use `except*` syntax from Python 3.11

coming soon

```
3
4 @cocotb.test
5 async def test_thing(dut) -> None:
6     try:
7         async with TaskManager() as tm:
8
9             @tm.fork
10             async def drive_input() -> None:
11                 # stuff
12                 raise ValueError # oops
13
14             @tm.fork
15             async def check_output() -> None:
16                 # various things are occurring
17                 raise IndexError # oops
18
19         except* ValueError:
20             # handle ValueError from 'drive_input'
21             # or any other child Task
22             ...
23
24         except* IndexError:
25             # handle IndexError from 'check_output'
26             # or any other child Task
27             ...
28
```

TaskManager

- Canceling recursively cancels children Tasks
- Prevents leaking Tasks

coming soon

```
3
4 @cocotb.test
5 async def test_thing(dut) -> None:
6
7     async def do_stuff() -> None:
8
9         async with TaskManager() as tm:
10
11             @tm.fork
12             async def drive_input() -> None:
13                 try:
14                     ...
15                 finally:
16                     # TaskManager cancels child Tasks
17                     # when it gets CancelledError
18                     print("Cleaned up!")
19
20             @tm.fork
21             async def check_output() -> None:
22                 ...
23
24     # do_stuff in Task so we can cancel it
25     do_stuff_task = cocotb.start_soon(do_stuff())
26
27     # Cancel Task with running TaskManager
28     do_stuff_task.cancel()
29
```




Additional Changes

Public API Guarantees

- Properly define Python public API using Python idioms
- Can make guarantees about stability of API
- Semantic Versioning

The screenshot shows the PyPI page for cocotb 1.9.1. The header is blue with the text 'cocotb 1.9.1' and a green 'Latest version' badge. Below the header is a grey bar with the text 'cocotb is a coroutine based cosimulation library for writing VHDL and Verilog testbenches in Python.' The main content area is white and contains a navigation sidebar on the left and a release history section on the right. The sidebar includes links to 'Project description', 'Release history', and 'Download files'. It also lists 'Verified details' (What is this?), 'Maintainers' (fossi-foundation-escrow, imphil, themperck), 'Unverified details' (These details have not been verified by PyPI), and 'Project links' (Homepage, Bug Tracker, Documentation, Source Code). The release history section shows a vertical timeline of versions: 1.9.1 (Aug 29, 2024), 1.9.0 (Jul 15, 2024), 1.9.0rc2 (PRE-RELEASE, Jul 10, 2024), 1.9.0rc1 (PRE-RELEASE, Jul 8, 2024), 1.8.1 (Oct 6, 2023), and 1.8.0 (Jun 15, 2023). The 1.9.1 version is marked as 'THIS VERSION'.

cocotb 1.9.1 ✓ Latest version

`pip install cocotb`

Released: Aug 29, 2024

cocotb is a coroutine based cosimulation library for writing VHDL and Verilog testbenches in Python.

Navigation

- Project description
- Release history**
- Download files

Verified details *(What is this?)*
These details have been verified by PyPI

Maintainers

- fossi-foundation-escrow
- imphil
- themperck

Unverified details
These details have *not* been verified by PyPI

Project links

- Homepage
- Bug Tracker
- Documentation
- Source Code

Release history [Release notifications](#) [RSS feed](#)

1.9.1 THIS VERSION
Aug 29, 2024

1.9.0
Jul 15, 2024

1.9.0rc2 PRE-RELEASE
Jul 10, 2024

1.9.0rc1 PRE-RELEASE
Jul 8, 2024

1.8.1
Oct 6, 2023


1.8.0
Jun 15, 2023

Meta

Documentation

- API reference documentation complete
- If it's documented, it's part of the public API
- Guaranteed by running pydocstyle in CI





latest

Search docs

Quickstart Guide

Installation

TUTORIALS

More Examples

HOW-TO GUIDES

Writing Testbenches

Building HDL and Running Tests

Coroutines and Tasks

Triggers

Extending existing build flows

Rotating Log Files

Writing cocotb extensions

KEY TOPICS

Installing the Development Version

Troubleshooting

REFERENCE

Build options and Environment Variables

Python Code Library Reference

Python Test Runner

Writing and Generating Tests

Interacting with the Simulator

Triggers

Test Utilities

Simulation Time Utilities

Logging

Simulation Object Handles

Miscellaneous

Implementation Details

GPI Library Reference

Simulator Support

Pinmux

return type: `Callable[[...], Coroutine[Any, Any, Result]]`

Changed in version 2.0: Renamed from `external`. No longer implemented as a type. The `log` attribute is no longer available.

cocotb.resume(func) [\[source\]](#)

Converts a coroutine function into a blocking function.

This allows a [coroutine function](#) that awaits cocotb triggers to be called from a [blocking function](#) converted by `cocotb.bridge()`. This completes the bridge through non-`async` code.

When a converted coroutine function is called the current function blocks until the converted function exits.

Results of the converted function are returned from the function call.

Parameters: `func` (`Callable[[...], Coroutine[Any, Any, Result]]`) – The [coroutine function](#) to convert into a [blocking function](#).

Returns: `func` as a [blocking function](#).

Raises: [RuntimeError](#) – If the function that is returned is subsequently called from a thread that was not started with `cocotb.bridge`.

Return type: `Callable[[...], Result]`

Changed in version 2.0: Renamed from `function`. No longer implemented as a type. The `log` attribute is no longer available.

HDL Datatypes

These are a set of datatypes that model the behavior of common HDL datatypes.

Added in version 1.6.0.

class cocotb.types.Logic(value=None) [\[source\]](#)

Model of a 9-value (`U`, `X`, `0`, `1`, `Z`, `W`, `L`, `H`, `-`) datatype commonly seen in VHDL.

This is modeled after VHDL's `std_logic` type. (System)Verilog's 4-value `logic` type only utilizes `X`, `0`, `1`, and `Z` values.

`Logic` can be converted to and from `int`, `str`, and `bool`. The list of values convertible to `Logic` includes `"u"`, `"x"`, `"0"`, `"1"`, `"z"`, `"w"`, `"l"`, `"h"`, `"-"`, `0`, `1`, `True`, and `False`.

```
>>> Logic("X")
Logic('X')
>>> Logic(True)
Logic('1')
>>> Logic(1)
Logic('1')

>>> Logic() # default value
Logic('X')

>>> str(Logic("Z"))
'Z'
>>> bool(Logic(0))
False
>>> int(Logic(1))
1
```

35

Typing

- Check your cocotb tests with type checkers
- Better suggestions in editors
- Guaranteed by running mypy in CI

```
29 class DataValidMonitor:
30
31     def start(self) -> None:
32         """Start monitor"""
33         if self._coro is not None:
34             raise RuntimeError("Monitor already started")
35         self._coro = cocotb.start_soon(self._run())
36
37     def stop(self) -> None:
38         """Stop monitor"""
39         if self._coro is None:
40             raise RuntimeError("Monitor never started")
41         self._coro
42         self._coro
43
44     async def _run(self) -> None:
45         while True:
46             await R
47             if self._has_started:
48                 await join
49                 con
50                 self._va
51                 log
52
53     def _sample(self, _add_done_callback) -> None:
54         """
55         Samples the data signals and builds a transaction object
56
57         Return value is what is stored in queue. Meant to be overridden by the user.
58         """
59         return {name: handle.value for name, handle in self._datas.items()}
60
61 class MatrixMultiplierTester:
62     """
63     Basic checker of a matrix multiplier (testcase)
64     """
```

Migrating to cocotb 2.0

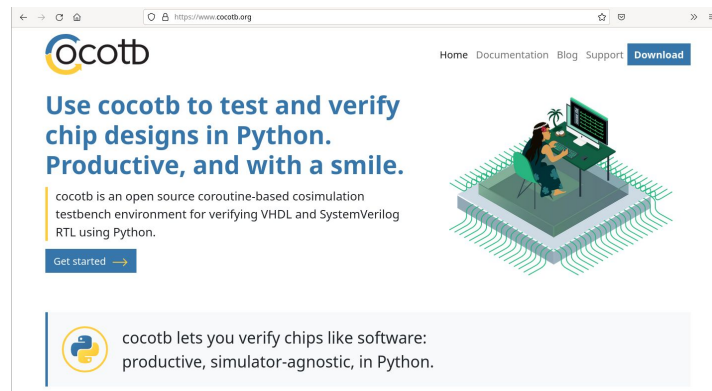
1. Upgrade to the latest cocotb 1.x.
2. Resolve all deprecation warnings.
3. Upgrade to cocotb 2.0 (or a current master version)
4. Fix remaining issues.

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 preferred 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.
```

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.



www.cocotb.org

GitHub: [cocotb](https://github.com/cocotb)

LinkedIn: [cocotb](https://www.linkedin.com/company/cocotb)

Twitter/X: [@cocotbnews](https://twitter.com/cocotbnews)