An(other) Introduction to Cocotb

Orconf 2018

Luke Darnell
Broadcom

Overview

- Cocotb is an RTL simulator plugin that allows engineers to verify their designs using python.
- It is open source with the source available on github.
- Presentation used within Broadcom to introduce engineers to cocotb
- Accompanying examples used in this presentation available on github

Why

python:

- great language that is constantly improving
- amazingly productive
- huge library of existing code
- great online resources
- extremely popular
- fun
- same code, different environments (simulation, silicon,..)

Why not

UVM/SystemVerilog:

- my own experience has been uncompelling
- simple things were hard
- macro hell
- compiled
- simulation only

People love cocotb

“As far as I'm concerned it's the greatest thing that's ever happened to digital design. I've been doing this for 15 years and it's the first time that some 'framework' actually solves the problem of 'DV' elegantly.”

David Lamb, Broadcom Engineer

Quickstart

Cocotb comes with its own set of testcases and examples.

Very easy to run these and play around with the code quickly.

To run the entire test suite:

```
git clone https://github.com/potentialventures/cocotb.git
cd cocotb
make
```

And to run the more in depth examples:

```
cd cocotb/examples
make
```

Examples in this introduction

All the examples in this introduction are available on github.

To run the examples:

```
git clone https://github.com/lukedarnell/cocotb.git
cd cocotb/tests/test_cases/orconf2018
make
```

Top level verilog for all tutorial example is:

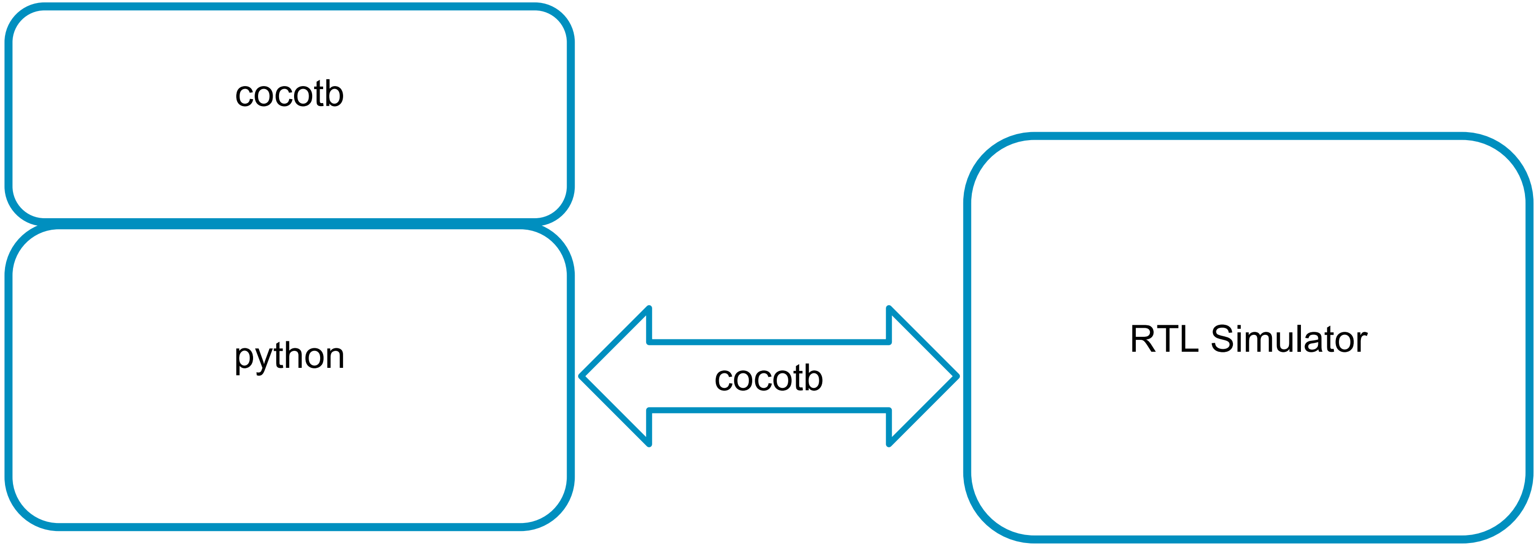
`designs/sample_module/sample_module.v`

To run an individual testcase in the orconf2018 directory (my_first_test in this example):

```
make TESTCASE=my_first_test
```

What is cocotb?

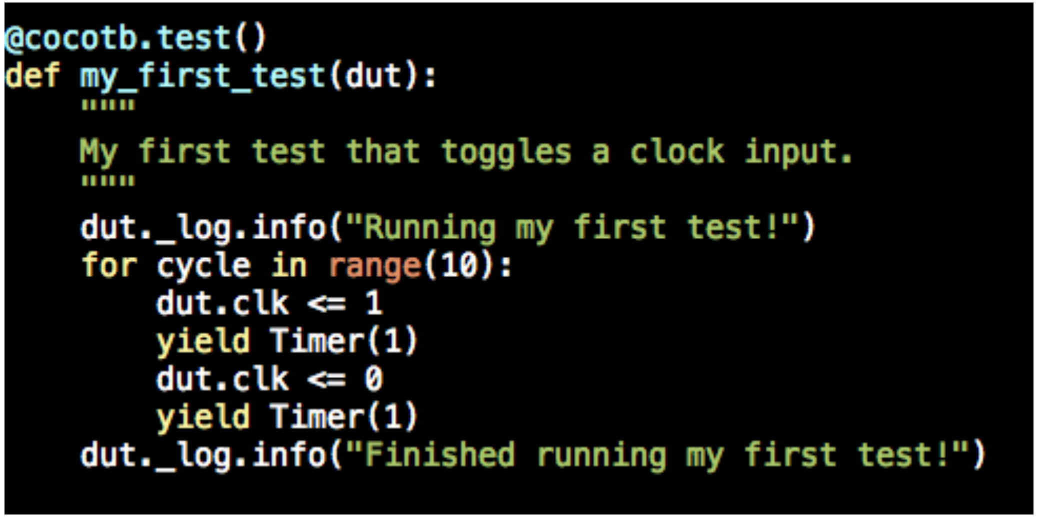
1. Plugs python into an rtl simulator
2. Extends python with knowledge of discrete time synchronous digital simulation



Creating a test

```
make TESTCASE=my_first_test
```

Tests are written in Python. Assuming we have a toplevel port called clk we could create a test file containing the following:



```
@cocotb.test()
def my_first_test(dut):
    """
    My first test that toggles a clock input.
    """
    dut._log.info("Running my first test!")
    for cycle in range(10):
        dut.clk <= 1
        yield Timer(1)
        dut.clk <= 0
        yield Timer(1)
    dut._log.info("Finished running my first test!")
```

This will drive a square wave clock onto the clk port of the toplevel.

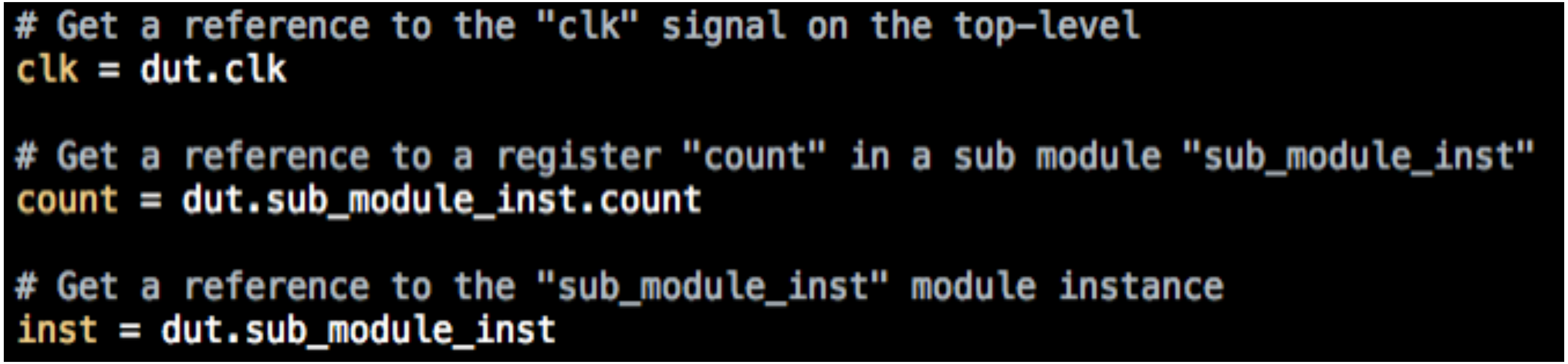
The toplevel module, **dut**, gets passed into the test function.

We'll explain each element of this test as we go through this presentation.

Accessing the design

```
make TESTCASE=accessing_the_design
```

- top-level simulator instance passed to cocotb test as first argurment (dut)
- instance signals can be accessed using the “dot” notation
- same mechanism can be used to access signals inside the design.



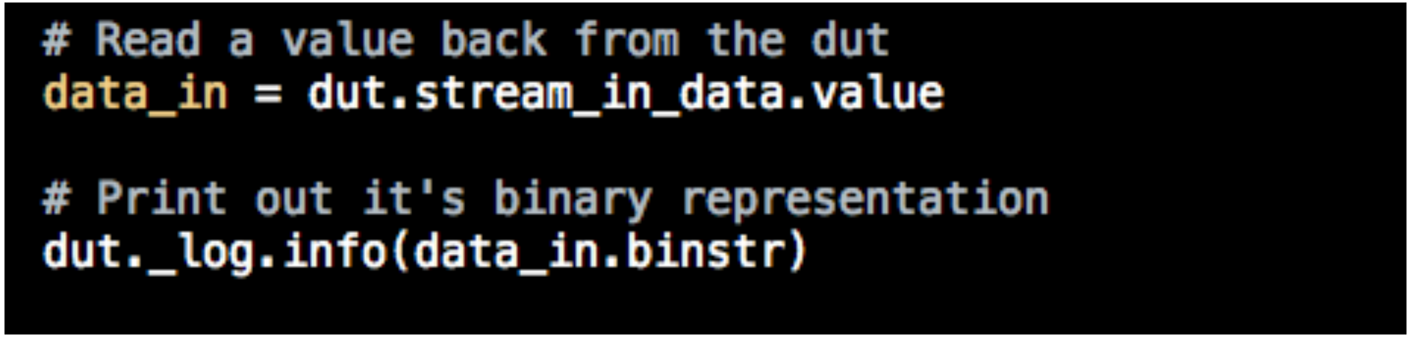
```
# Get a reference to the "clk" signal on the top-level
clk = dut.clk

# Get a reference to a register "count" in a sub module "sub_module_inst"
count = dut.sub_module_inst.count

# Get a reference to the "sub_module_inst" module instance
inst = dut.sub_module_inst
```

Accessing the design

- NOTE: This only gets a **reference (or handle)** to the signals, not their underlying value.
- This is one of the powerful features of cocotb, you can pass these signal references around as objects within python.
- Can also do the same for modules.

Reading values from signals

- .value property of a handle object will return a **`BinaryValue`** object.
- **`BinaryValue`** class is the way cocotb handles signal values coming from the simulator
- It provides nice features for handling binary values – Little/Big Endian, get/set bits, truncation, unsigned/2's comp/signed magnitude



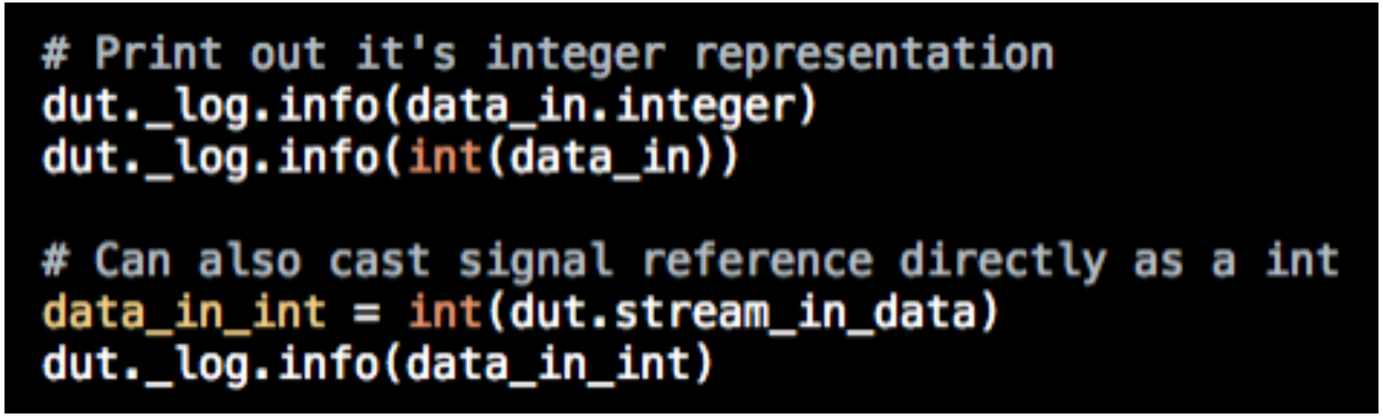
```
# Read a value back from the dut
data_in = dut.stream_in_data.value

# Print out it's binary representation
dut._log.info(data_in.binstr)
```

make TESTCASE=reading_values

Reading values from signals

- access the integer representation by either the .integer property.
- Or by casting to an int.
- Can also get the integer representation by directly casting the signal reference.

make TESTCASE=reading_values

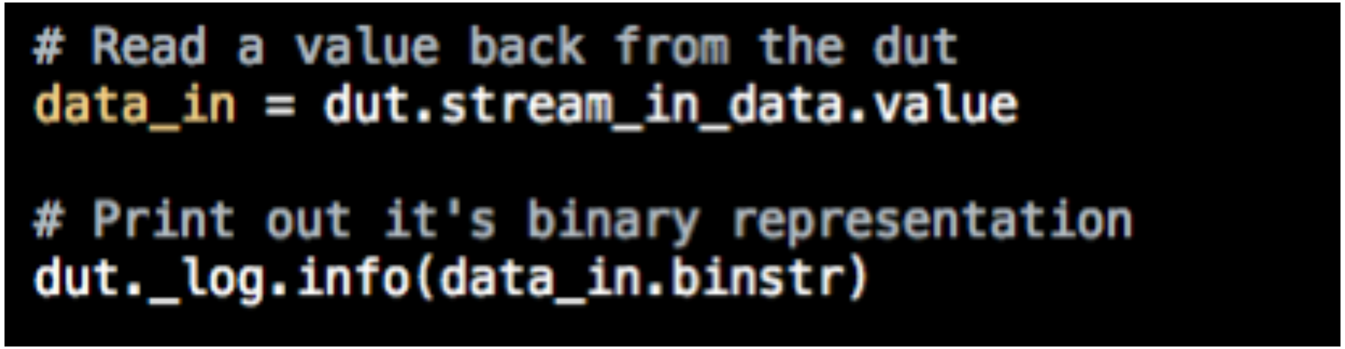


```
# Print out it's integer representation
dut._log.info(data_in.integer)
dut._log.info(int(data_in))

# Can also cast signal reference directly as a int
data_in_int = int(dut.stream_in_data)
dut._log.info(data_in_int)
```

Reading x/z values

make TESTCASE=reading_xz_values

- still access the .value property of a handle object to get a **`BinaryValue`** object.
- unresolved and undriven bits are preserved and can be accessed using the binstr attribute.



```
# Read a value back from the dut
data_in = dut.stream_in_data.value

# Print out it's binary representation
dut._log.info(data_in.binstr)
```

Reading x/z values

- Casting **`BinaryValue`** objects containing x/z values will raise **`ValueError`** by default
- Can control x/z casting behaviour with `COCOTB_RESOLVE_X` environment variable
- `COCOTB_RESOLVE_X=ZEROS`
- `COCOTB_RESOLVE_X=ONES`
- `COCOTB_RESOLVE_X=RANDOM`

```
make TESTCASE=reading_xz_values
make TESTCASE=reading_xz_values COCOTB_RESOLV_X=ZEROS
make TESTCASE=reading_xz_values COCOTB_RESOLV_X=ONES
make TESTCASE=reading_xz_values COCOTB_RESOLV_X=RANDOM
```

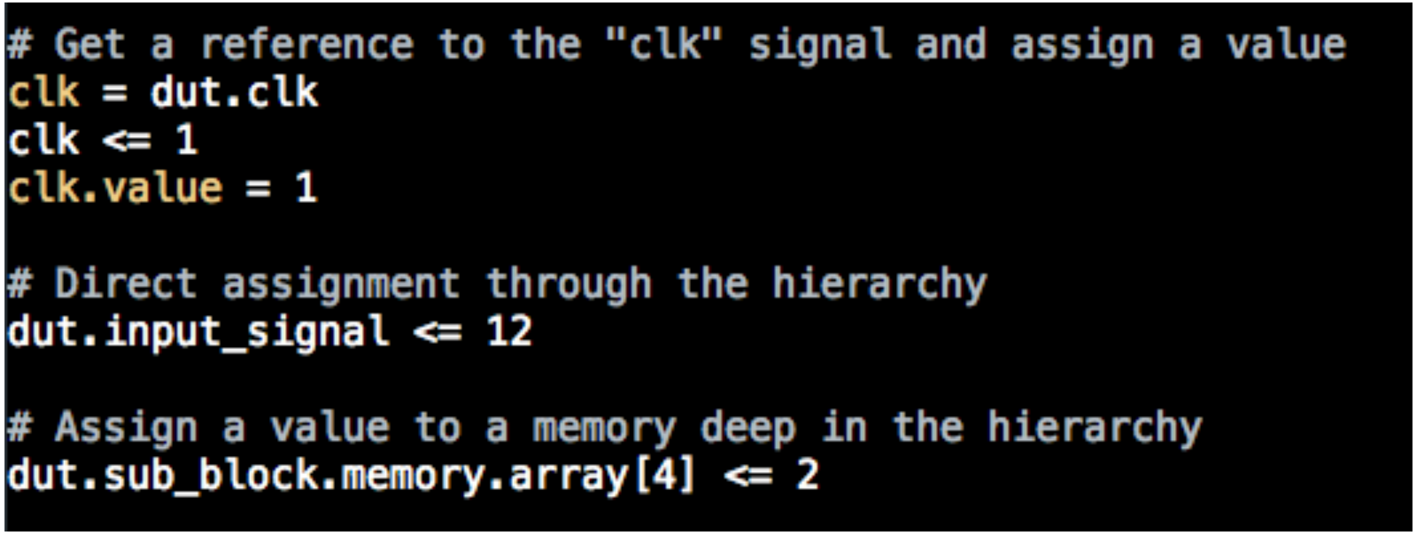
BROADCOM®

Assigning values to signals

make TESTCASE=assigning_values

Values assigned to signals in the simulator using either:

1. less than equal operator (hdl style)
2. .value property of a handle object

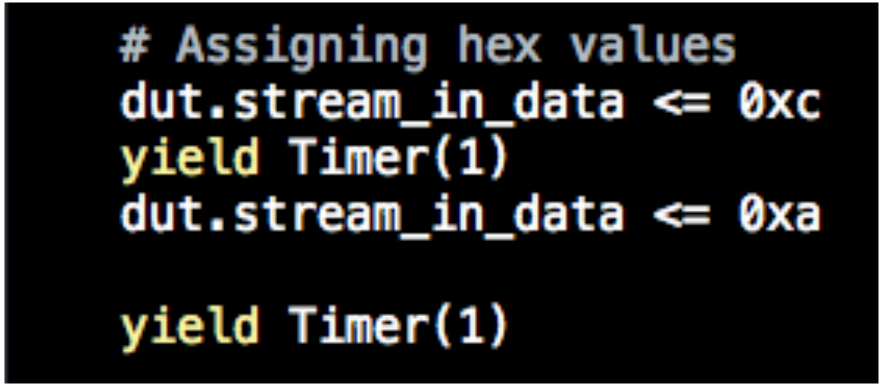


```
# Get a reference to the "clk" signal and assign a value
clk = dut.clk
clk <= 1
clk.value = 1

# Direct assignment through the hierarchy
dut.input_signal <= 12

# Assign a value to a memory deep in the hierarchy
dut.sub_block.memory.array[4] <= 2
```

NOTE: array assignment not working with icarus

Assigning hex values

Since python has hex number representation builtin, assigning hex values to signals is exactly the same as assigning integers.



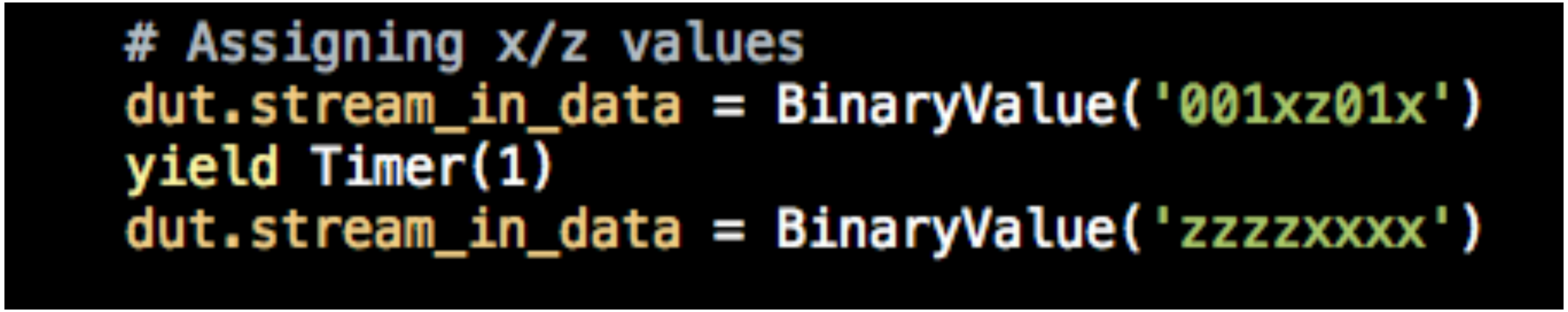
```
# Assigning hex values
dut.stream_in_data <= 0xc
yield Timer(1)
dut.stream_in_data <= 0xa

yield Timer(1)
```

make TESTCASE=assigning_hex_values

Assigning x/z values

python has no understanding of x and z values like an rtl simulator does.

cocotb implements the reading and assignment of x/z values through the **`BinaryValue`** class.



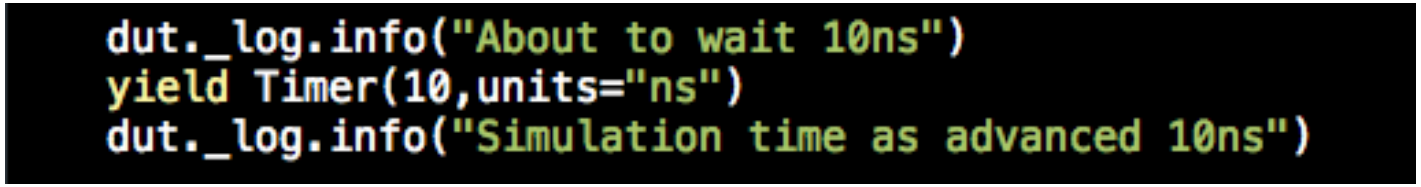
```
# Assigning x/z values
dut.stream_in_data = BinaryValue('001xz01x')
yield Timer(1)
dut.stream_in_data = BinaryValue('zzzzxxxx')
```

```
make TESTCASE=assigning_xz_values
```

yield - waiting on the simulator

Cocotb uses the python keyword **`yield`** to a pass control of execution back to the simulator and simulation time can advance.

Typically we **`yield`** on a **`Trigger`** object which indicates to the simulator some event which will cause the thread of execution to be passed back to cocotb. For example:



```
dut._log.info("About to wait 10ns")
yield Timer(10,units="ns")
dut._log.info("Simulation time as advanced 10ns")
```

make TESTCASE=yielding_to_the_simulator

yield, decorators, generators, coroutines, ????????

cocotb makes use of some of the less common python programming features.

You don’t need to know how these work to use cocotb.

So don’t worry if you’re not a python guru, cocotb is still for you!

And if you do want to know what all this stuff is,
check out the various tutorials by David Beazley.

BROADCOM

Triggers

Cocotb can pass control back to the simulator using several `Trigger` objects.

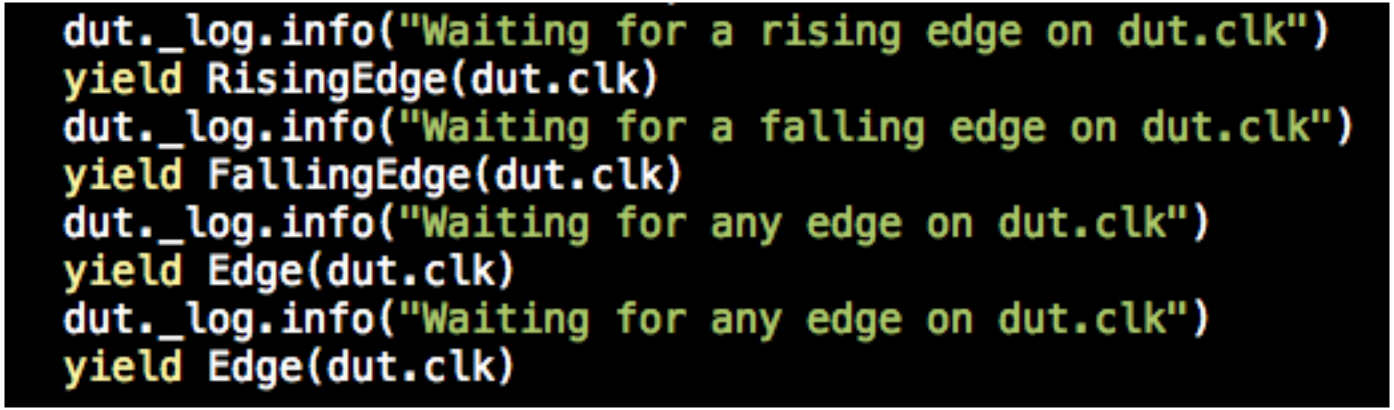
`Timer`:
cocotb execution will resume when the specified time period expires.
Argument is simulation steps, with optional units argument (“s”, “ms”, “us”, “ns”, etc…)

`Edge`:
cocotb execution will resume when an edge occurs on the provided signal.

`RisingEdge`:
cocotb execution will resume when a rising edge occurs on the provided signal.

`FallingEdge`:
cocotb execution will resume when a falling edge occurs on the provided signal.

Triggers



```
dut._log.info("Waiting for a rising edge on dut.clk")
yield RisingEdge(dut.clk)
dut._log.info("Waiting for a falling edge on dut.clk")
yield FallingEdge(dut.clk)
dut._log.info("Waiting for any edge on dut.clk")
yield Edge(dut.clk)
dut._log.info("Waiting for any edge on dut.clk")
yield Edge(dut.clk)
```

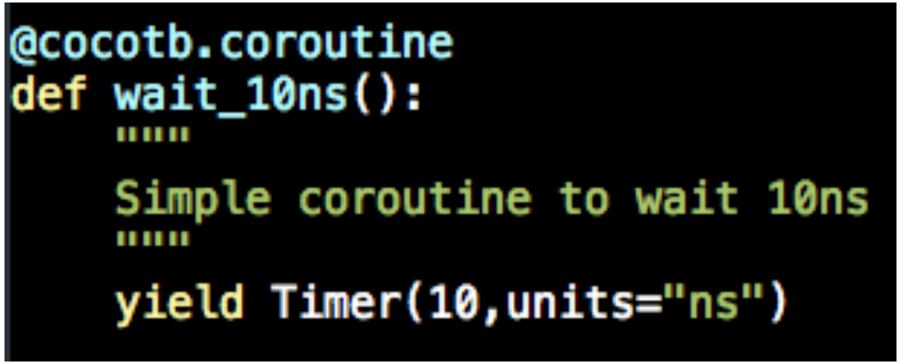
make TESTCASE=simulator_triggers

BROADCOM

Coroutines

So far we've shown that cocotb can pass control of execution back to the simulator by yield'ing to simulator Triggers. But how do we pass control of execution to another function within cocotb?

Coroutines.

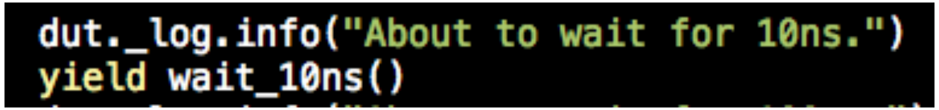


```
@cocotb.coroutine
def wait_10ns():
    """
    Simple coroutine to wait 10ns
    """
    yield Timer(10,units="ns")
```

make TESTCASE=yielding_to_coroutines

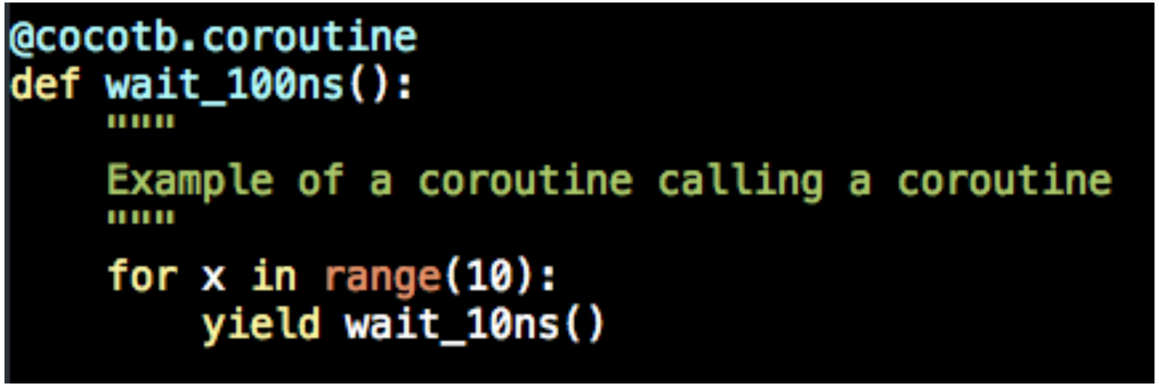
Coroutines

Just like Triggers we yield control of execution over to coroutines:



```
dut._log.info("About to wait for 10ns.")
yield wait_10ns()
```

Coroutines can also call other coroutines:



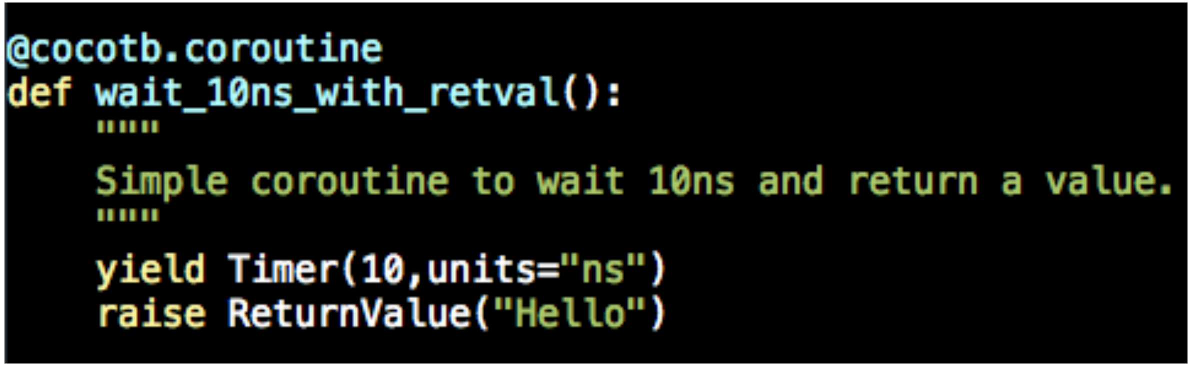
```
@cocotb.coroutine
def wait_100ns():
    """
    Example of a coroutine calling a coroutine
    """
    for x in range(10):
        yield wait_10ns()
```

make TESTCASE=yielding_to_coroutines

Coroutines

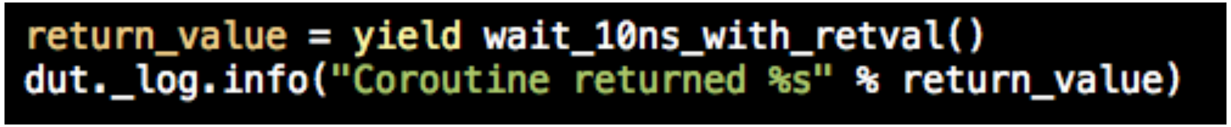
make TESTCASE=yielding_to_coroutines

Coroutines can also return values.
They do this by “raising” a **`ReturnValue`** object:



```
@cocotb.coroutine
def wait_10ns_with_retval():
    """
    Simple coroutine to wait 10ns and return a value.
    """
    yield Timer(10,units="ns")
    raise ReturnValue("Hello")
```

And to call a coroutine that returns a value:



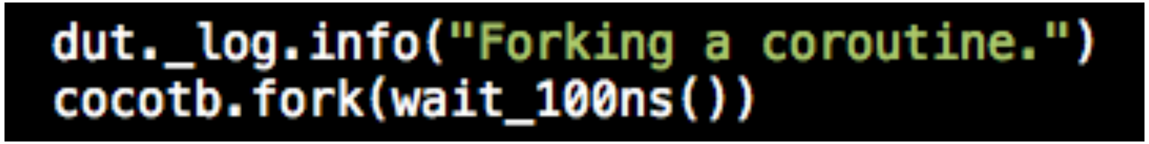
```
return_value = yield wait_10ns_with_retval()
dut._log.info("Coroutine returned %s" % return_value)
```

BROADCOM

forking and joining

It’s possible to fork coroutines in cocotb.

This creates another thread of execution for the forked coroutine, while letting the calling thread continue it’s own thread of execution.

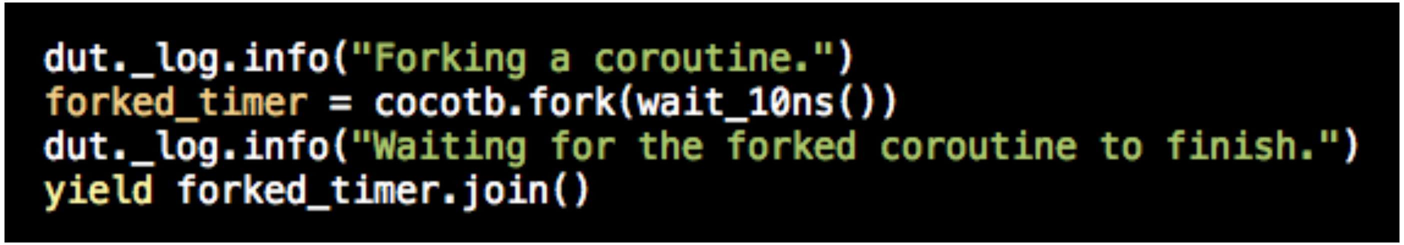


```
dut._log.info("Forking a coroutine.")
cocotb.fork(wait_100ns())
```

```
make TESTCASE=forking_and_joining
```

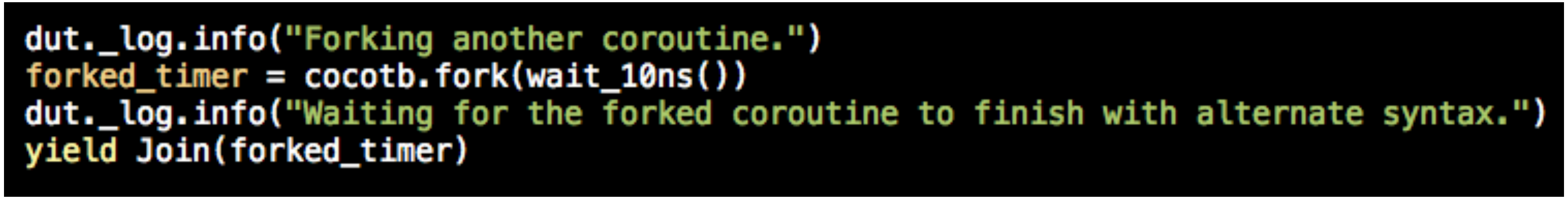
forking and joining

If we save a reference to the forked coroutine, it's then possible to subsequently wait for the forked coroutine to finish by joining.



```
dut._log.info("Forking a coroutine.")
forked_timer = cocotb.fork(wait_10ns())
dut._log.info("Waiting for the forked coroutine to finish.")
yield forked_timer.join()
```

Can also use the **Join** function:



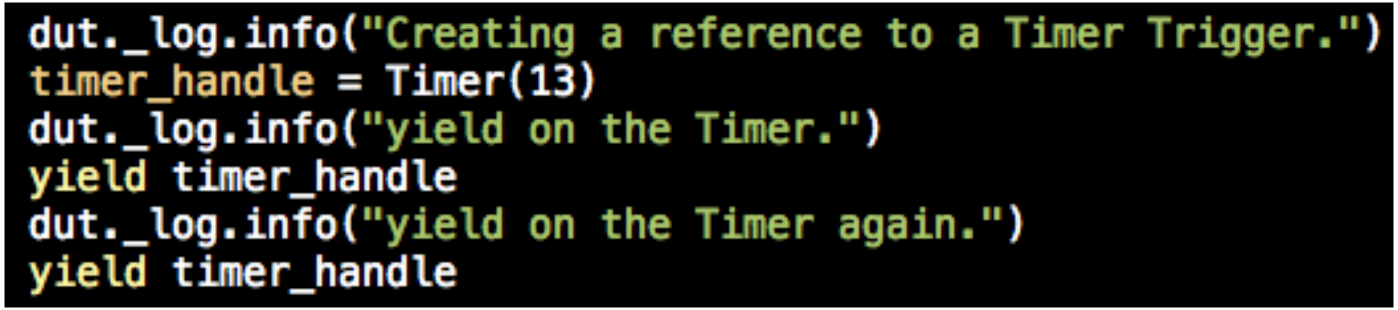
```
dut._log.info("Forking another coroutine.")
forked_timer = cocotb.fork(wait_10ns())
dut._log.info("Waiting for the forked coroutine to finish with alternate syntax.")
yield Join(forked_timer)
```

make TESTCASE=forking_and_joining

Advanced yield'ing

cocotb has a few nice features that give it some interesting capabilities.

We can create **Trigger** objects and yield on them multiple times:

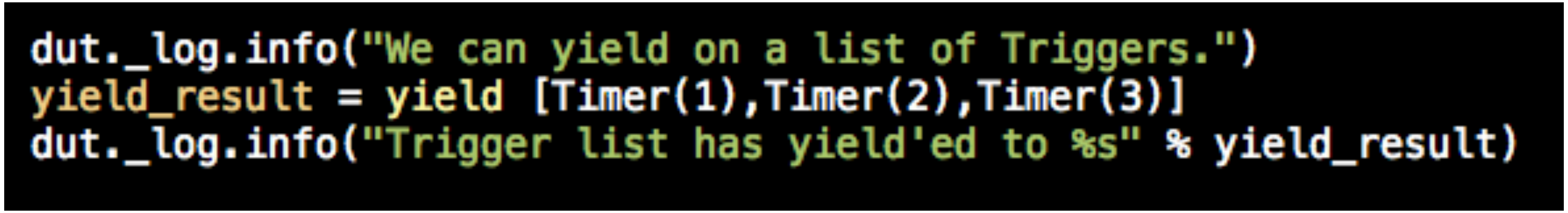


```
dut._log.info("Creating a reference to a Timer Trigger.")
timer_handle = Timer(13)
dut._log.info("yield on the Timer.")
yield timer_handle
dut._log.info("yield on the Timer again.")
yield timer_handle
```

make TESTCASE=advanced_yielding

Advanced yield'ing

Can yield to a list of Triggers, which returns the Trigger that fired first.

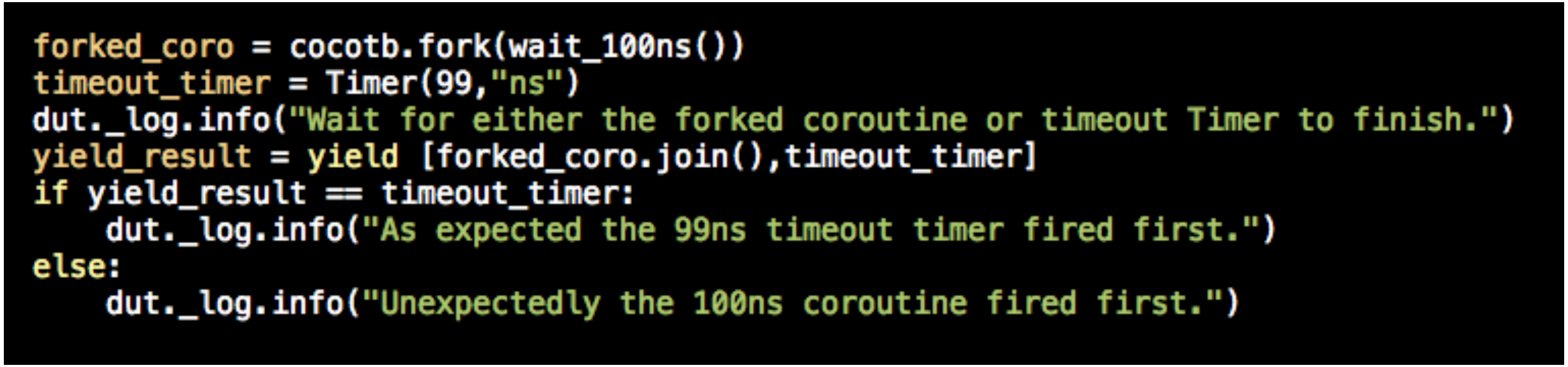


```
dut._log.info("We can yield on a list of Triggers.")
yield_result = yield [Timer(1),Timer(2),Timer(3)]
dut._log.info("Trigger list has yield'ed to %s" % yield_result)
```

make TESTCASE=advanced_yielding

Advanced yield’ing

Can yield to a mix of Triggers and forked coroutines, checking which returns first.



```
forked_coro = cocotb.fork(wait_100ns())
timeout_timer = Timer(99,"ns")
dut._log.info("Wait for either the forked coroutine or timeout Timer to finish.")
yield_result = yield [forked_coro.join(),timeout_timer]
if yield_result == timeout_timer:
    dut._log.info("As expected the 99ns timeout timer fired first.")
else:
    dut._log.info("Unexpectedly the 100ns coroutine fired first.")
```

subtlety: if the forked_coro in the above example returns first, we get the value returned

make TESTCASE=advanced_yielding

tests

The `@test` decorator tells cocotb that the following function is a test.

Cocotb will select which test/s to run based on the `MODULE` and/or `TESTCASE` commandline arguments.

`MODULE` is the python module in which to search for `@test`'s

`TESTCASE` is the `@test` we want to run.

If `TESTCASE` isn't specified, cocotb will run all the `@test`'s it can find (sequentially) based on it's search path.

The search path is defined by the normal python search path (PYTHONPATH) and also the MODULE commandline switch.

cocotb generates a junit xml file (results.xml) showing the results of all the tests it runs.

```
make MODULE=tests_module
```

tests

A cocotb test can indicate success by Raising **TestSuccess**

And indicate failure by Raising **TestFailure**

If neither is raised the test is assumed to have passed.

TestSuccess and TestFailure can both optionally accept a message.

```
raise TestSuccess
```

```
raise TestFailure
```

```
raise TestSuccess("This test is passing!")
```

```
raise TestFailure("This test is failing!")
```

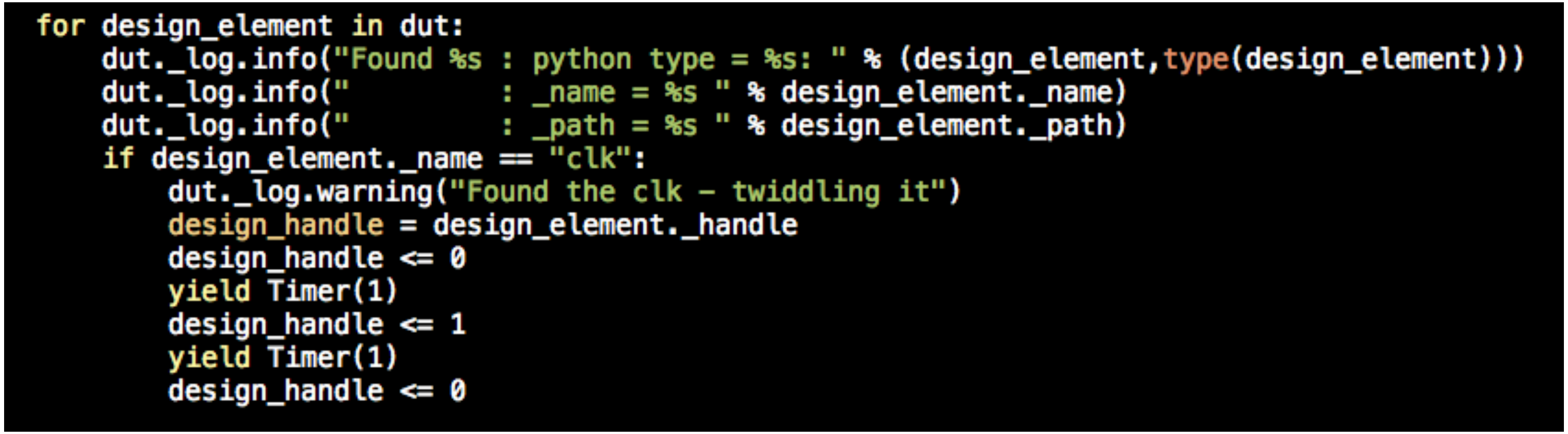
```
make MODULE=tests_module
```

BROADCOM

exploring the design hierarchy

Cocotb allows code to iterate over the design hierarchy.

Makes connecting buses and interfaces very easy.



```
for design_element in dut:
    dut._log.info("Found %s : python type = %s: " % (design_element,type(design_element)))
    dut._log.info("           : _name = %s " % design_element._name)
    dut._log.info("           : _path = %s " % design_element._path)
    if design_element._name == "clk":
        dut._log.warning("Found the clk - twiddling it")
        design_handle = design_element._handle
        design_handle <= 0
        yield Timer(1)
        design_handle <= 1
        yield Timer(1)
        design_handle <= 0
```

make TESTCASE=exploring_design_hierarchy

Handy cocotb Environment Variables

`COCOTB_LOG_LEVEL` – controls internal log messaging from cocotb

`RANDOM_SEED` – seed the python random module

`COCOTB_ANSI_OUTPUT` – controls coloured log messaging

Thanks!

To run the examples:

```
git clone https://github.com/lukedarnell/cocotb.git
cd cocotb/tests/test_cases/orconf2018
make
```