University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering

member of
swissuniversities

Python based FPGA verification using CocoTB

ISE Microelectronics Group

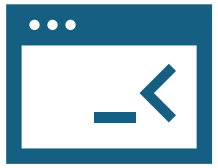
Robin Müller

1 July 2024

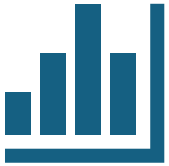
What we will cover today

1. 

- Introduction
- Concepts
- Setup

2. 

- Example

3. 

- Discussion
- Who uses CocoTB?
- My & others experice

Concepts

1.

Overview: What is CocoTB?

What it is:

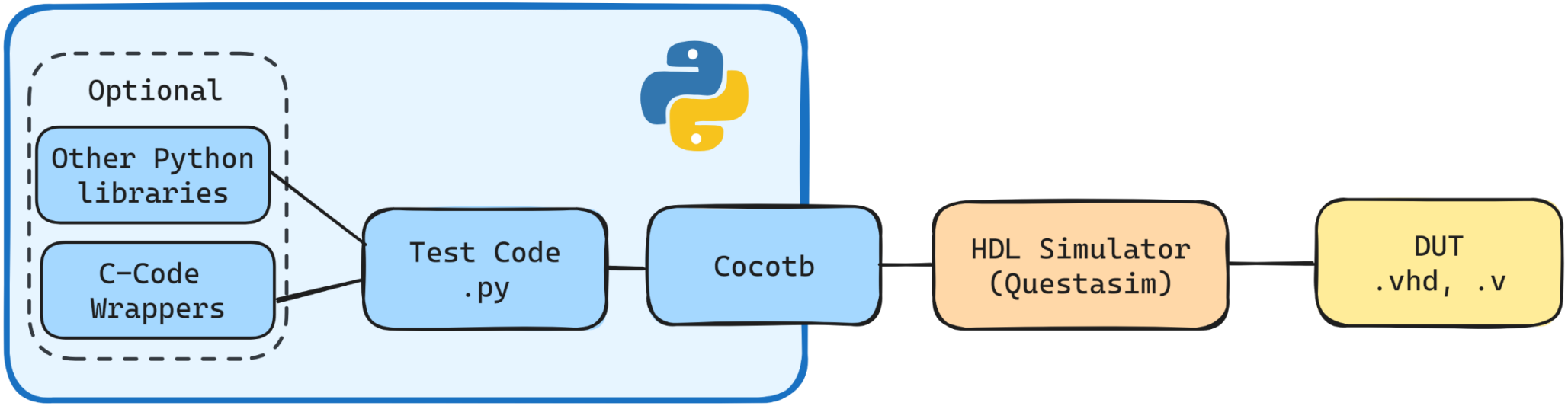
- A Python library that allows to write testbenches for VHDL and Verilog designs fully in Python.
- Similar in philosophy to other frameworks like UVM but in Python
- Open Source

What it is not:

- A replacement for HDL simulators (e.g Modelsim/Questasim)
- Simulator specific
- Used for RTL design (not HLS)

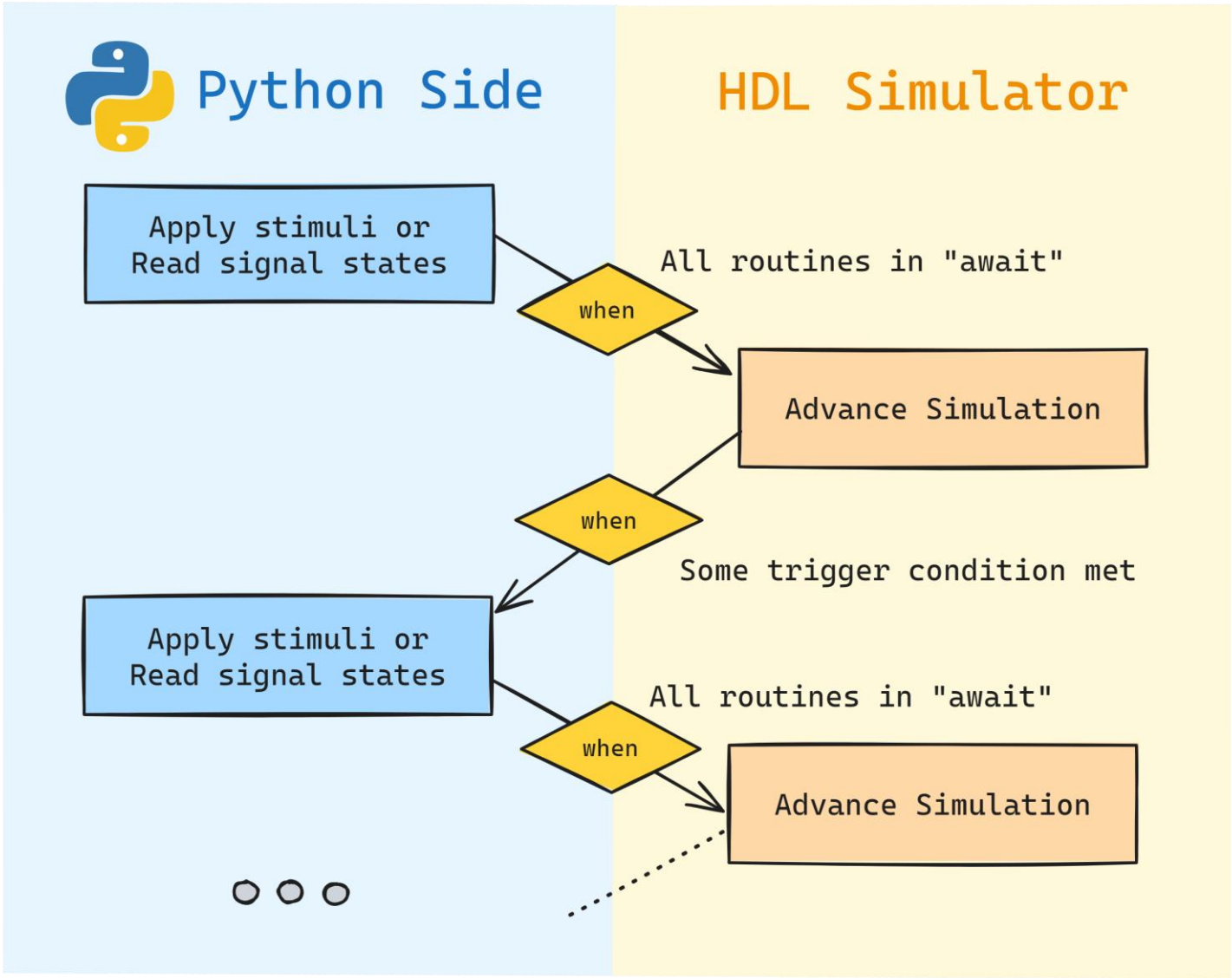
How it works

CocoTB connects to the API of the simulator and lets you apply/read values from/to signals and advance simulation time



Simulator-Python Interaction

«Ping Pong» control between simulator and python test code



Setup

What we need:

- A HDL Simulator (e.g. Questasim)
- Python installed
- CocoTB

Installation:

```
pip install cocotb
```

Check you can run:

```
cocotb-config
```

GitHub

CocoTB Repo

Important Code Constructs

Simulator Time Interaction

```
from cocotb.triggers import …
```

Await statements need to be in async functions

«await» passes control to Simulator until condition is met

```
async def some_function():
    #Previous Code
    await someTrigger
    #Following Code
```

Most important functions

Wait for rising/falling edge of a signal (typ. clk)

```
await RisingEdge(SIGNAL)
await FallingEdge(SIGNAL)
```

Wait for some number of «clock cycles»[1]

```
await ClockCycles(SIGNAL, NB_CYCLES, rising=True)
```

Wait for some time

```
await Timer(DELAY, units='ns')
```

[1] Wait for `NB_CYCLES` of edges `SIGNAL`

Important Code Constructs

Coroutines & Parallelism

«start_soon» returns the task not the return value of the function.
To obtain return values use:

```
task = cocotb.start_soon(some_function())
returned_value = await task
```

Launches a new coroutine

or

```
returned_value = await some_function()
```

Passes control

How we can run multiple functions in «parallel»[1]

«start_soon» passes control to other tasks at the next blocking condition (i.e. some await statement)

```
async def some_function_A():
    #Do Stuff

async def some_function_B():
    #Do Stuff
    return some_value

async def main_function():
    #Run two functions in parallel
    task_1 = cocotb.start_soon(some_function_A())
    task_2 = cocotb.start_soon(some_function_B())
    #Wait for both to finish
    await task_1
    returned_value = await task_2
```

[1]Parallel execution in this context refers to independent execution and interaction with the simulator. Parallel regarding simulator time **not** physical time (i.e. no multithreading). Only quasi-concurrent execution.

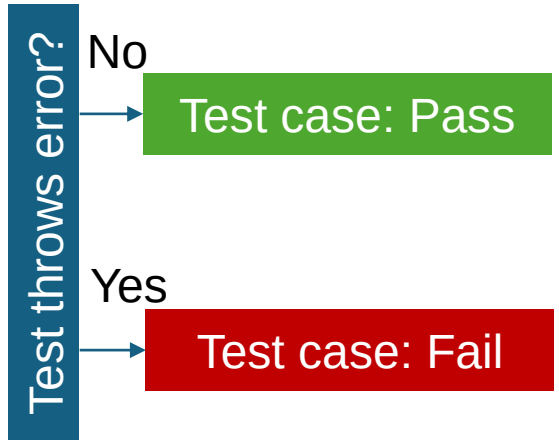
Important Code Constructs

Defining Tests & Pass/Fail Criteria

The passed «dut» argument lets you acces your DUT

Defining test cases

```
#Define different test ca
@cocotb.test()
async def my_test_case_A(dut):
    #entry point for test case A

#Can include additional parameters
@cocotb.test(timeout_time=1000, timeout_unit='ns')
async def my_test_case_B(dut):
    #entry point for test case B
```



Assert results are correct

```
assert result_value == reference_value, "Some Error Message"
```

Important Code Constructs

Set/Read signals & generating clocks

modify/read signal values

```
#Read Signal
value = dut.my_signal.value
#Write Signal
dut.my_signal.value = value
```

There is nothing inherently special in how clocks are treated compared to other signals. This is equivalent to writing:

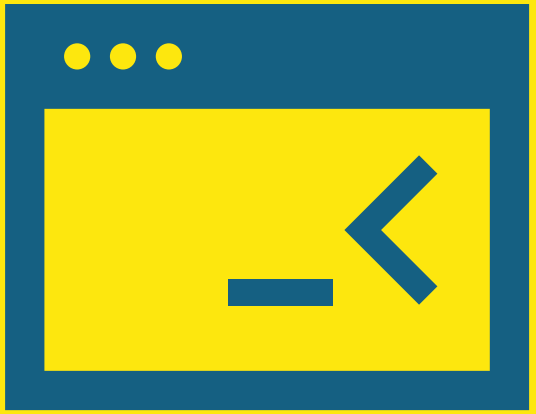
Clock generation

```
from cocotb.clock import Clock
#Create a 400MHz clk --> 2.5 ns period
clk_400MHz = Clock(dut.clk_signal, 2.5, units='ns’)
clk_task = cocotb.start_soon(clk_400MHz.start())
```

```
async def clock_function(signal, period, time_unit):
    while True:
        signal.value = 0
        await Timer(period / 2, time_unit)
        signal.value = 1
        await Timer(period / 2, time_unit)

clk_task = cocotb.start_soon(clock_function(dut.clk_signal, 2.5, 'ns'))
```

Example

2.

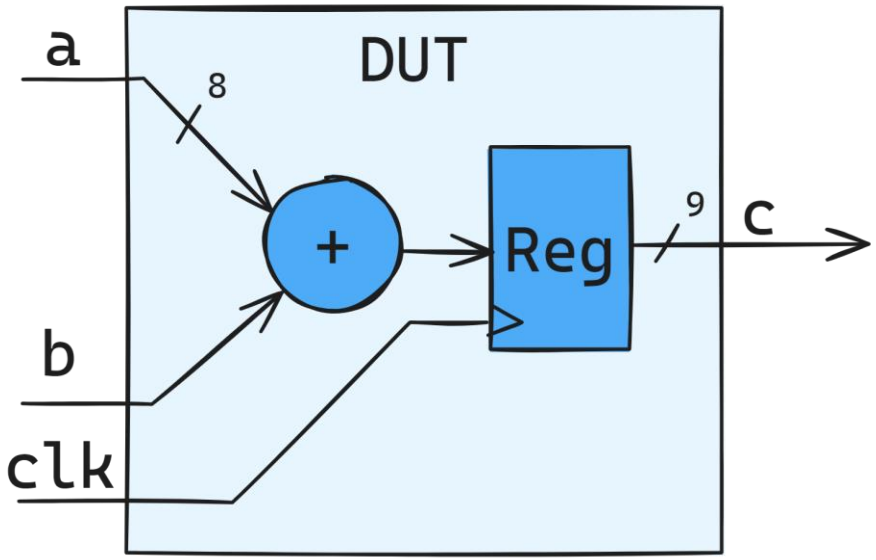


A simple example

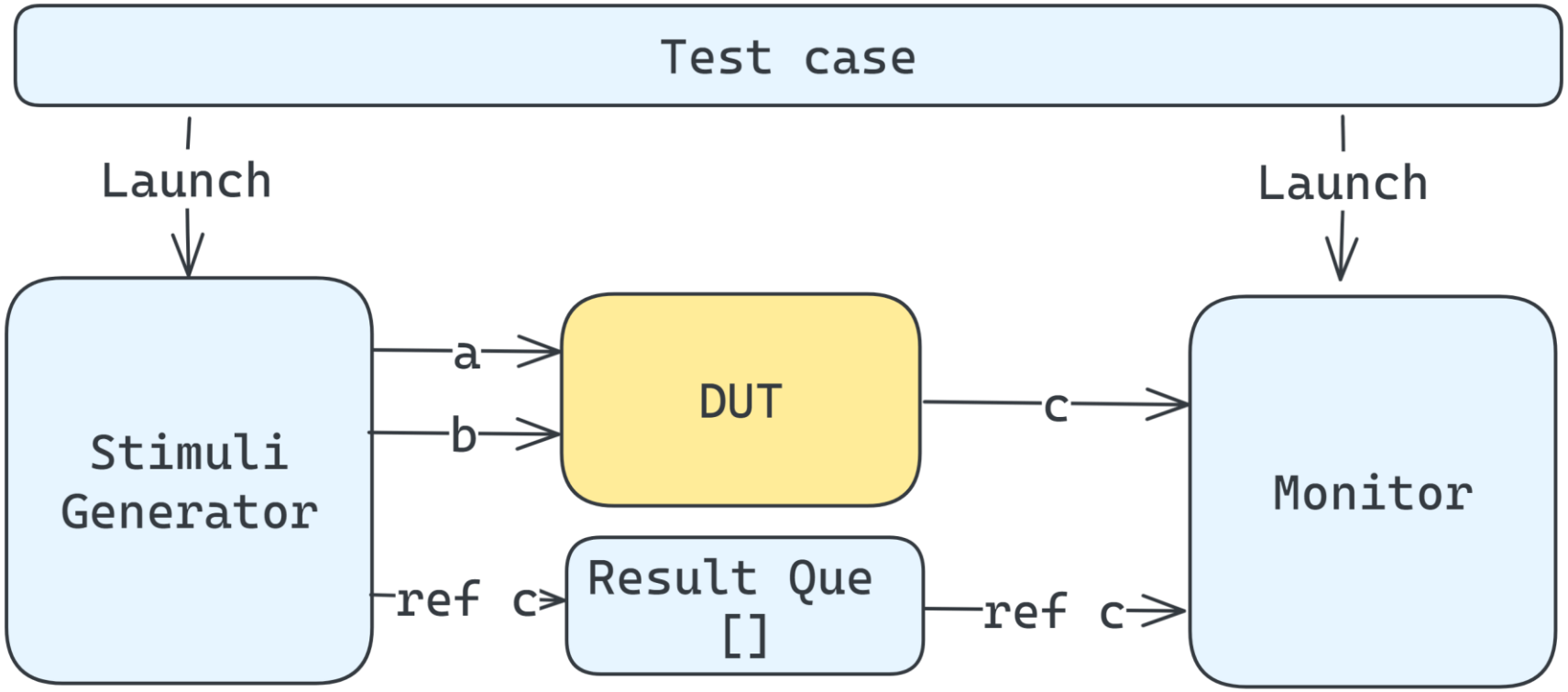
-Test for a simple VHDL 8-bit unsigned adder

- Simulate all 256^2 = 65’536 input combinations

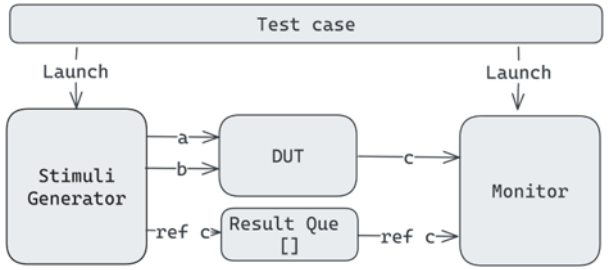
```
entity simple_adder is
    generic ( g_nb_bits : natural := 8 );
    port (
        clk : in STD_LOGIC;
        a : in   STD_LOGIC_VECTOR (g_nb_bits - 1 downto 0);
        b : in   STD_LOGIC_VECTOR (g_nb_bits - 1 downto 0);
        c : out  STD_LOGIC_VECTOR (g_nb_bits downto 0)
    );
end simple_adder;
```



A simple test structure

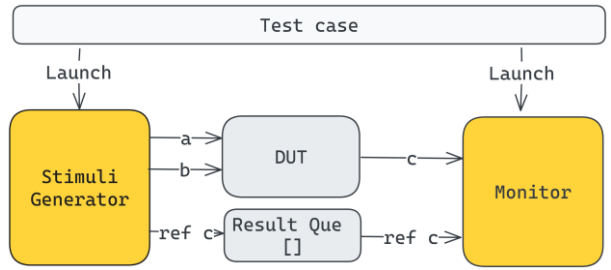


Test structure in code



Test structure in code

Stimuli Generator & Monitor



Stimuli Generator

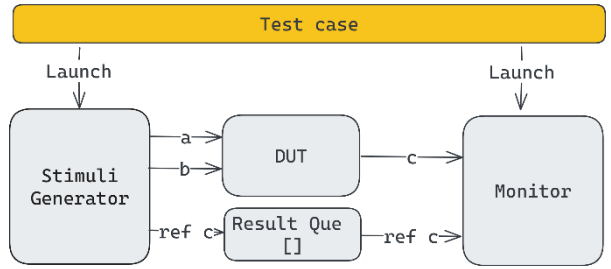
```python
async def stimuli(dut, result_que):
    for a in range(256):
        for b in range(256):
            dut.a.value = a
            dut.b.value = b
            result_que.append(ref_result_gen(a, b))
            await FallingEdge(dut.clk)

def ref_result_gen(a, b):
    return a + b
```

Monitor

```python
async def monitor(dut, LATENCY, result_que):
    await ClockCycles(dut.clk, LATENCY, rising=False)
    for _ in range(256*256):
        ref_value = result_que.pop(0)
        result_value = int(dut.c.value)
        assert ref_value == result_value, \
            f"Expected {ref_value} but got {result_value}"
        await FallingEdge(dut.clk)
```

Test structure in Code

Test Case



Test Case

```
@cocotb.test(timeout_time=10, timeout_unit='ms')
async def exhaustive_adder_test(dut):
    #Start Clock
    clk_200MHz = Clock(dut.clk, 5, units="ns")
    clk_task = cocotb.start_soon(clk_200MHz.start())
    await FallingEdge(dut.clk) #Wait for first falling edge
    #Start stimuli and monitor
    result_que = []
    monitor_task = cocotb.start_soon(monitor(dut, DUT_LATENCY, result_que))
    stimuli_task = cocotb.start_soon(stimuli(dut, result_que))
    await monitor_task, stimuli_task
```

How do we run this?

CocoTB supports two build flows to run tests:

Makfile based:
You write a Makefile and execute “make” in your shell

- Standard flow you will find in most examples

Python runner based:
You run your test directly from python

- Newer experimental feature
- You will find fewer examples (but I’ll show you how)

My Opinion:
I strongly prefer the Python based flow although it’s not as common. Reasons are:

- Allows for easy integration with other frameworks like Pytest to further automate test runs
- You need one less tool (simply Python + Simulator) especially on Windows systems it’s more convenient
- It seems to take less time to initialize (you can re-run tests more quickly)

Build Flow Example

With Makefiles

Run by:

```
cd <<My Makefile Directory>>
make
```

Example Makefile

```
# Your desired build settings
SIM ?= questa
TOPLEVEL_LANG ?= vhdl

# Add all vhdl source files
VHDL_SOURCES += $(PWD)/simple_adder.vhd

TOPLEVEL = simple_adder #Name of the top level VHDL entity
MODULE = simple_adder #Name of the python file with the test cases
(simple_adder.py)

# include cocotb's make rules to take care of the simulator setup
include $(shell cocotb-config --makefiles)/Makefile.sim
```

Build Flow Example

With Python Runner

Run by:
Executing «test_adder()»

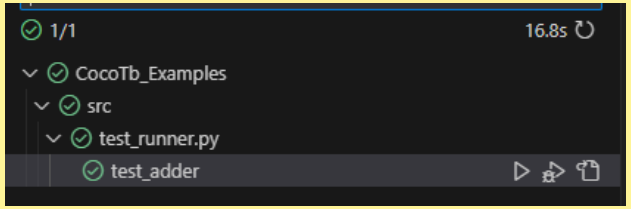
Example Python Runner

```python
import os
from cocotb.runner import get_runner

def test_adder():
    # Base directory where test_runner.py is located
    base_dir = os.path.dirname(os.path.abspath(__file__))

    # Construct absolute paths
    vhdl_sources = [os.path.join(base_dir, "simple_adder.vhd")]  # Absolute path to VHDL sources
    build_dir = os.path.join(base_dir, "sim_build")  # Absolute path to build directory

    runner = get_runner("questa")  # Specify your simulator

    # Compile
    runner.build(
        vhdl_sources=vhdl_sources,
        hdl_toplevel="simple_adder",
        build_dir=build_dir
    )

    # Run Test
    runner.test(
        hdl_toplevel="simple_adder",  # Name of the top level entity
        test_dir=base_dir,  # Directory of the test
        test_module="simple_adder",  # Test python file (needs to be located in test_dir)
        hdl_toplevel_library="./sim_build/top",  # Specifies where the build results are located relative to test_dir
        test_args=['-t', '1ps']  # Sets Simulator timescale from 'ns' to 'ps'
    )
```

The Results you get

By Default:
Results.xml containing [Testcase Name, PASS/FAIL, Sim Time, Physical Exec Time, …]

Pytest Feedback:



Python or compiler error message if failed

Note: You may sometimes need to delete the existing sim_build folder

Makfile flow add line:
`WAVES=1`

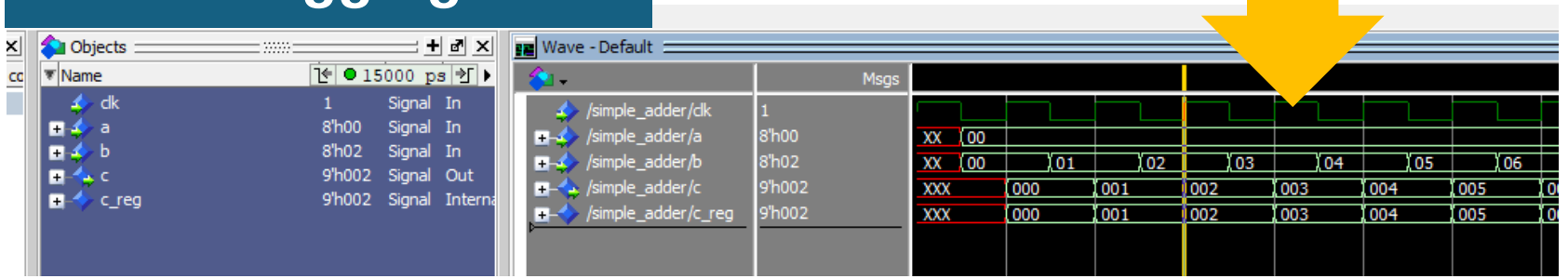
Python runner flow:
`runner.test(waves=True,…)`

Run Simulation

Generates a «vsim.wlf» file that you can view with questa

Open with:
`vsim -view vsim.wlf`
Or
open the file with Questa on Windows

What if I want to see the waveforms for debugging?



Demo Time

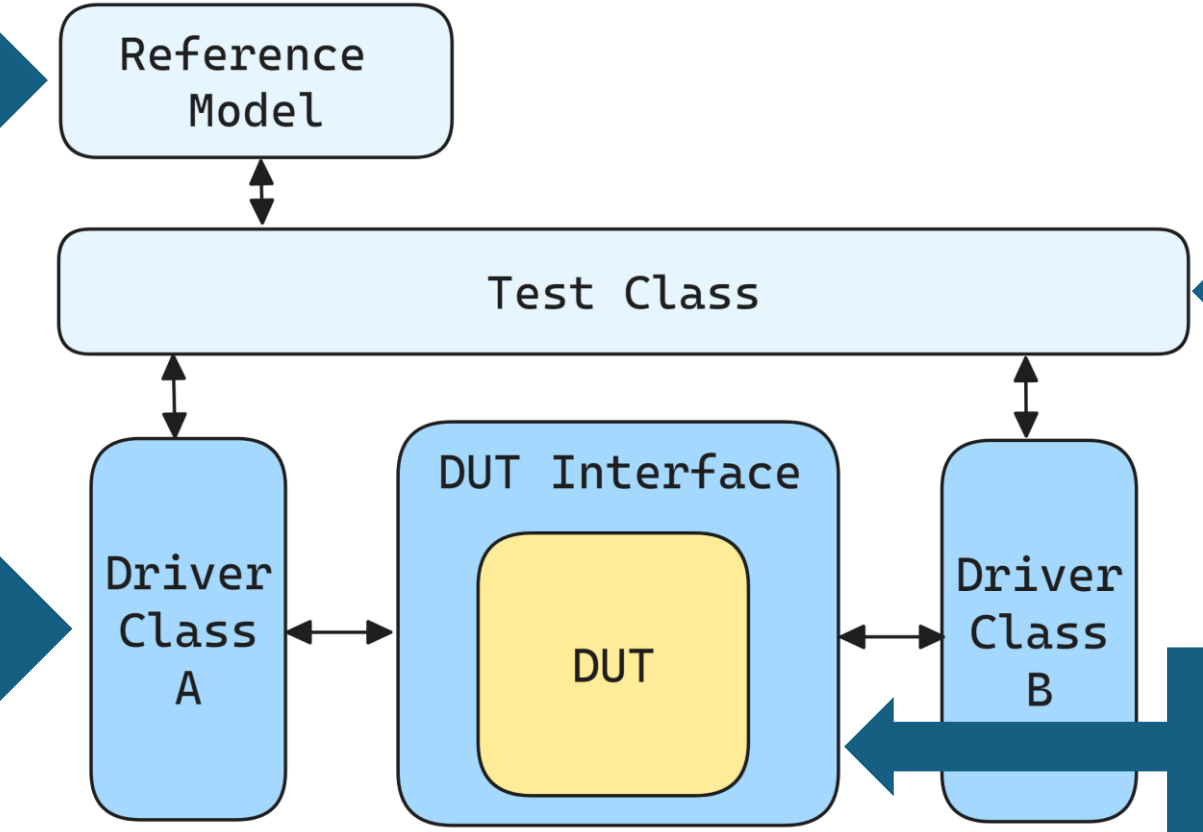
pytest

\+ cocotb +

My preferred test structure

The example was very simple almost any verification tool would do the job. What about verification of more complex modules?

Software model of the desired operation of the DUT as a reference



Test class in which the desired test is implemented at high level

Driver class that implements and abstracts the low level protocol (e.g. AXI, Avalon, …)

Simple interface that explicitly wraps top level entity to python

Discussion

3.

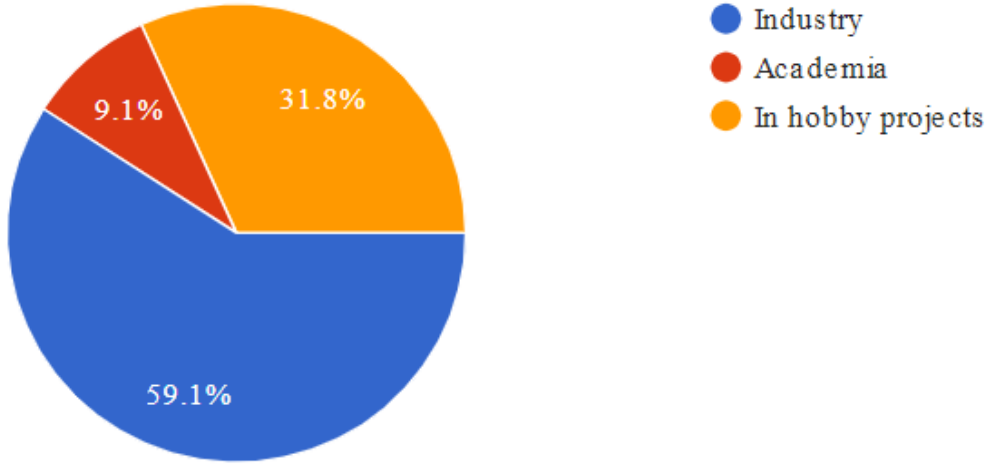
To use or not to use…?

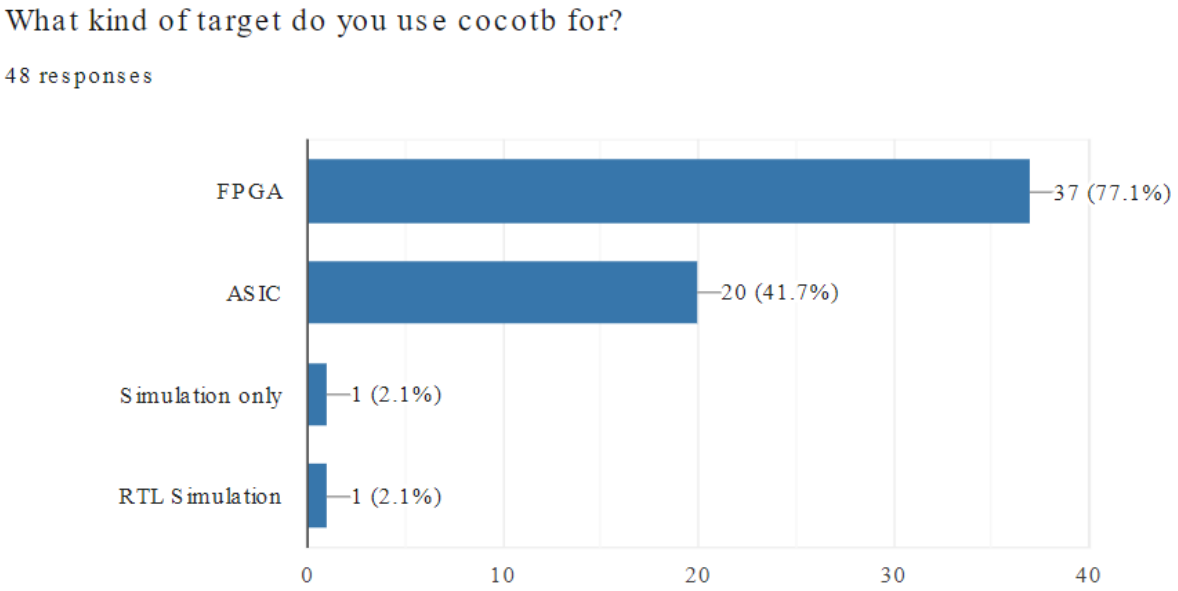
In my opinion the advantages of using CocoTB:

- Python is much less verbose than many other languages (especially VHDL). Using Python lets you write more test functionality faster.
- Creating tests at a high abstraction level is relatively easy
- It's easy to integrate tests with other libraries to enable (among others) automation or advanced result visualization (e.g. pytest, matplotlib, …)
- Python is somewhat easier to get into than for example OSVVM, also most new bachelor students come with prior python knowledge
- The community is active in contributing useful functionality like drivers for common interfaces (e.g. AXI), fixing bugs and providing documentation

What others say [1]

User Demographic



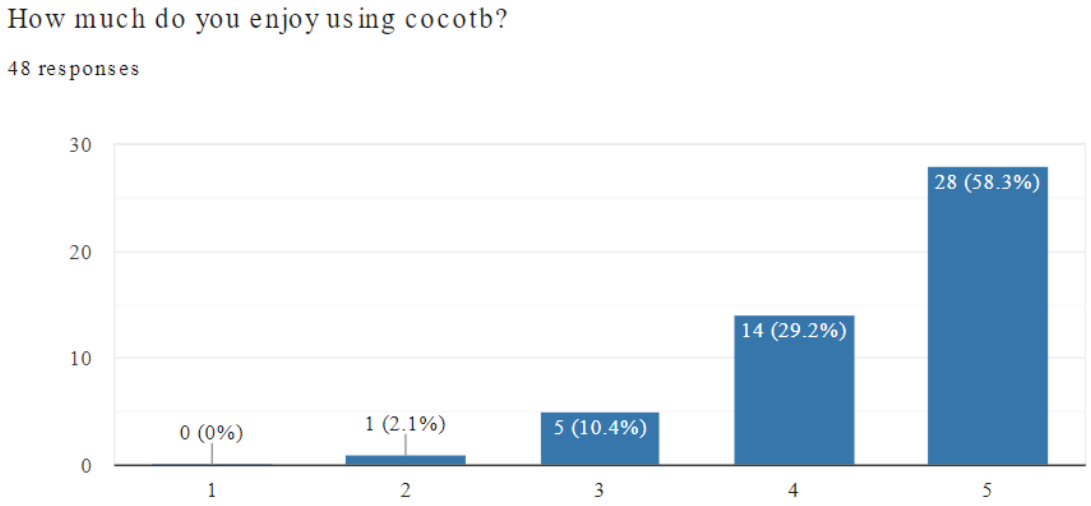


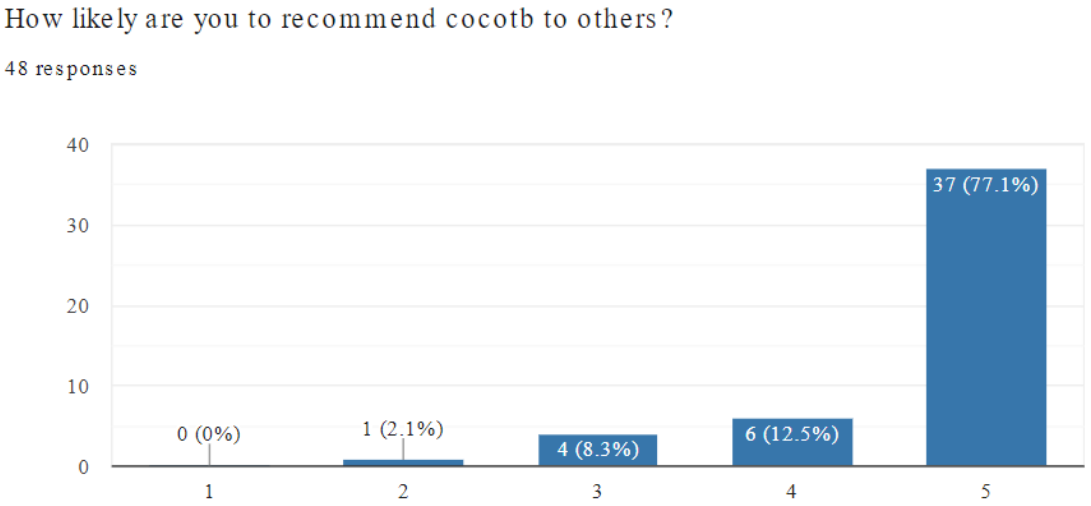
User Background: mostly users from industry (not just an academic or hobbyist tool)

[1] Cocotb user survey 2023: https://www.cocotb.org/2023/06/17/user-survey-2023.html

What others say [1]

User Satisfaction & Feedback





When asked about the pain points of these other verification approaches, cocotb users repeatedly mentioned:
- the complexity (and learning curve) of UVM
- poor CI/test framework integration
- the amount of boilerplate code and effort required until the actual writing of the test can start

[1] Cocotb user survey 2023: https://www.cocotb.org/2023/06/17/user-survey-2023.html

My Recommendation

When to use:

- FPGA projects in general
- RTL design and quickly want to simulate something or debug my design
- When productivity (time) is important
- Great for students doing P5/P6 (flat learning curve, prior knowledge of students and time budgets)

When I would not (yet) use it:

- Functional safety
- ASIC projects (with some caution) due to the impact of uncaught errors in designs. Maybe not the best place to start using a new tool.

Thanks

Today's example can be found on:

GitHub

https://github.com/m47812/CocoTb_Example

Delayed questions or support:

robin.mueller@fhnw.ch

Now open for questions and discussion