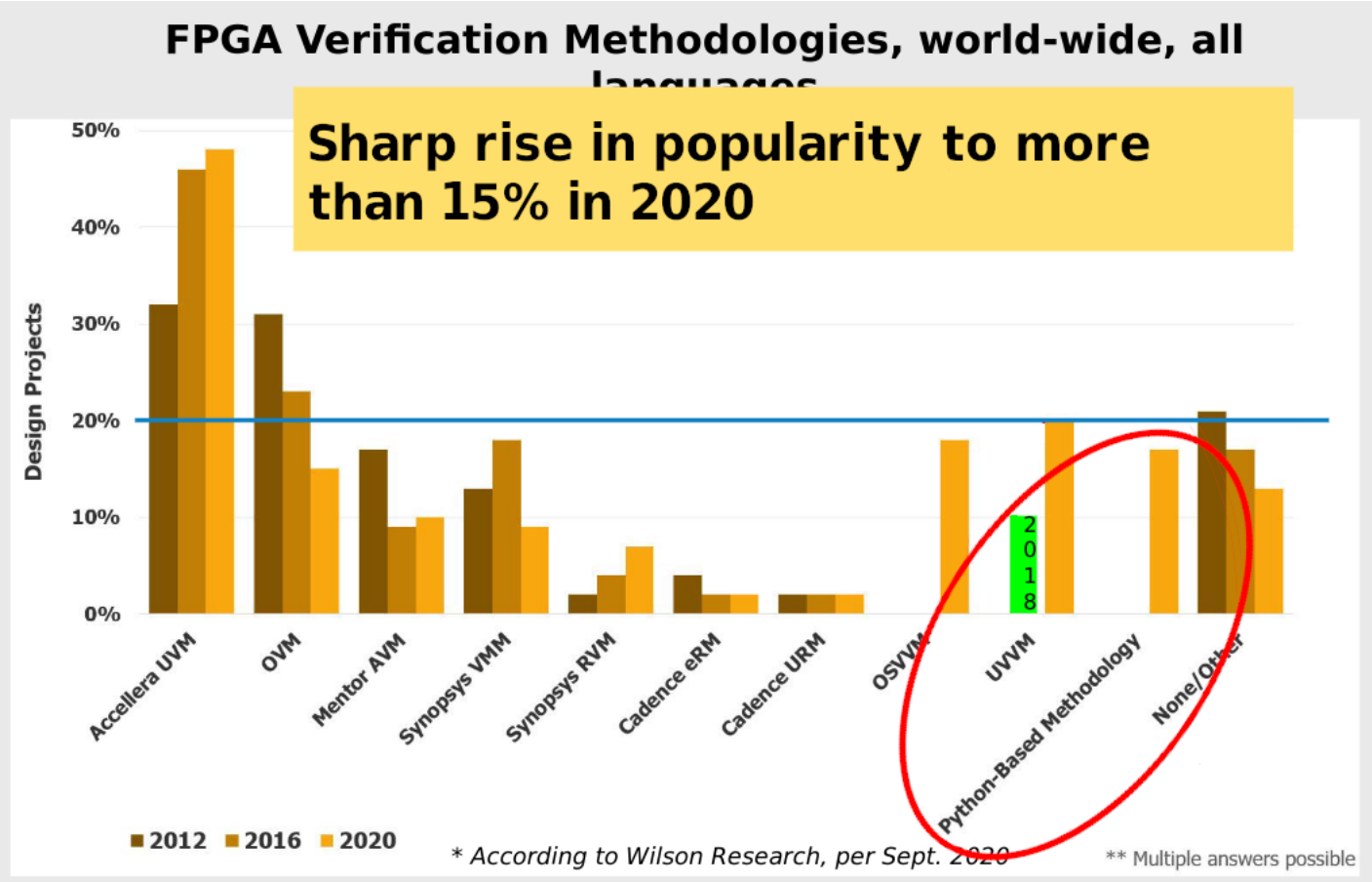
Workshop: cocotb and cocotb-test

Maximilian Babeluk

Nov 16^{th} 2022

Verification frameworks

- Python based verification is increasing in popularity
- Python is easy to learn and used in many areas
- Extremely wide range of python packages



Espen Tallaksen: UVVM (Universal VHDL Verification Methodology), TWEPP 2022 (modified)

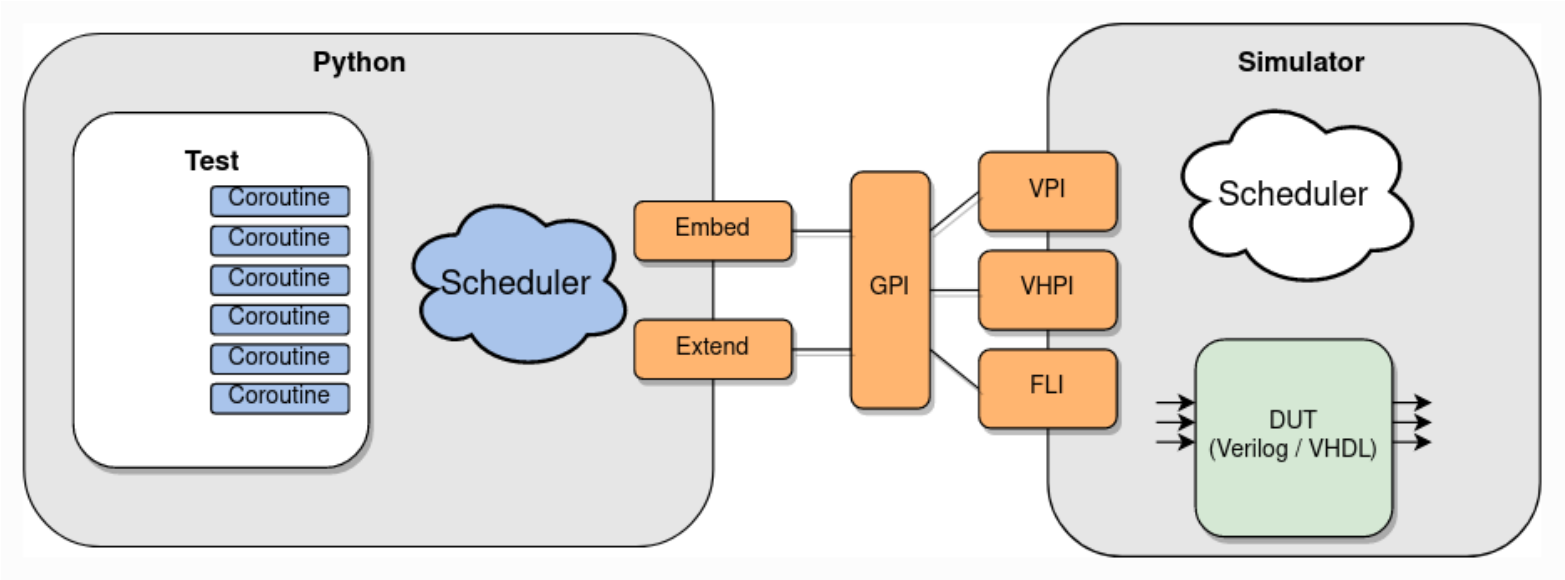
What is cocotb?

cocotb is a COroutine based COsimulation TestBench environment for verifying VHDL and SystemVerilog RTL using Python.

Cocotb Documentation.

- HDL code is only the real design
- Testbench is written in python
- cocotb provides an interface between the python code and the HDL simulator
- This means: a HDL simulator is still necessary
- python code provides test stimulus and/or verifies the results

What is cocotb?



Cocotb documentation.

- python code is organzed in coroutines
- multiple coroutines can run at the same time
- can send commands and wait for triggers (eg RisingEdge, Timer)

```
async def set_bcid_coroutine(dut):
    await RisingEdge(dut.Clk)
    for time in range(500):
        dut.BcidIn.value = time & (1<<dut.BCID_WIDTH.value)-1
        #                          ^ truncate python integer to vector size
        #   ^ name of the signal to set
        await RisingEdge(dut.Clk)  # wait for one clock cycle
```

Components of Coroutines

- Function declaration, can use parameters:

```
async def name_of_coroutine(dut, other_parameters):
```

- Trigger to 'wait' for rising edge:

```
await RisingEdge(dut.SignalNameOfModule)
```

- Seting and reading signals of module
- Casting value to int will have some impact:
  - Generally safer to have a real integer than some cocotb-object
  - X and Z values will throw an exception when casted to int

```
dut.SignalName.value = 0x12
python_variable_name = dut.SignalName.value
python_variable_name_int = int(dut.SignalName.value)
```

- Asserting values (Always boolean):

```
assert value < 0x12
assert dut.SignalName.value == 0
```

Components of Coroutines (2)

- One coroutine is the 'main' one, only that one is executed
- This is identified usind the `@cocotb.test()` decorator

```
1 @cocotb.test()
2 async def bcid_counter(dut):
3   ...
```

- To execute another coroutine in parallel use:

```
1 cocotb.fork(name_of_coroutine(dut, other_parameters=5))
```

- Special builtin coroutine for clocks:

```
1 cocotb.fork(Clock(dut.Clk, 50, units='ns').start())
```

- Coroutines should never contain the word 'test'

The cocotb-test package

- Typical cocotb workflow uses makefiles
- cocotb-test allows the use of the python testing framework (pytest)
- Powerful, yet very convenient to use
- Short and readable output when all tests succeed

```
mx@tp:~/Schule … st-examples/tests master ± pytest
============================= test session starts ==============================
platform linux -- Python 3.8.10, pytest-4.6.9, py-1.8.1, pluggy-0.13.0
rootdir: /mnt/Daten/Nosync/Schule/cocotb-test-examples/tests
plugins: forked-1.4.0, xdist-2.5.0, cocotb-test-0.2.1
collected 30 items

unit/test_bcid_counter.py ..............                                 [ 50%]
unit/test_bcid_counter_seperated.py ..............                       [100%]

========================== 30 passed in 27.95 seconds ==========================
θ78° mx@tp:~/Schule … st-examples/tests master 28s ±
```

Simulator invocation with cocotb-test

- One function is necessary to launch cocotb and the simulator
- Needs to start with 'test', in a file with a name that contains 'test'
- Automatically discovered by pytest
- Contains the HDL files, toplevel verilog module, and filename of the python coroutine

```
1 def test_bcid_counter():
2   run(verilog_sources=[os.path.join(tests_dir, "../hdl/bcid_counter.sv")],
3     sim_build=os.path.join(output_dir, 'sim_build'),
4     toplevel="bcid_counter",
5     module="test_bcid_counter")
```

Simulator invocation (full)

- Verilog parameters and definitions can be passed to the simulator
- Environment variables can be passed to the main coroutine
- Additional path for shared python modules possible

```
1  def test_bcid_counter():
2    # parameters to the top level module
3    params = {"BCID_WIDTH": 8}
4    # environemnt variables to pass to the test coroutine
5    test_parameters = {"ENVVAR1":   str(12),
6                       "ENVVAR2":   '0x43', }
7    # definitions passed to the simulator
8    defs = ["PARAMETER1", "PARAMETER2"]
9
10   run(verilog_sources=[os.path.join(tests_dir, "../hdl/bcid_counter.sv")],
11     includes=[os.path.join(tests_dir, "../hdl")],
12     python_search=[os.path.join(tests_dir, "../util")],
13     parameters=params,
14     extra_env=test_parameters,
15     defines=defs,
16     sim_build=os.path.join(output_dir, 'sim_build'),
17     toplevel="bcid_counter",
18     module="test_bcid_counter")
```

Test variations

- Tests can be parametrized
- Such tests are run multiple times, each time with a different parameter
- Single parameter: 0, 1 and 3

```
1 @pytest.mark.parametrize("prescaler", [0, 1, 3])
2 def test_bcid_counter(prescaler):
3   ...
```

- Multiple parameters: all combiantions (cross product)

```
1 @pytest.mark.parametrize("prescaler", [0, 1, 3])
2 @pytest.mark.parametrize("latency", [0, 1, 2, 100, 255])
3 def test_bcid_counter(prescaler, latency):
4   ...
```

- Multiple parameters: certain combiantions

```
1 @pytest.mark.parametrize("prescaler,latency", [(0, 0),(1, 10),(2, 30)])
2 def test_bcid_counter(prescaler, latency):
3   ...
```

Usage of pytest

- Run all tests from current directory including subdirectories:

```
cd tests/
export SIM=xcelium    # only for xcelium as simulator
pytest
```

- Run all tests from one python file:

```
pytest unit/test_bcid_counter.py
```

- To list names of all possible tests run:

```
pytest --collect-only -q
pytest unit/test_bcid_counter.py --collect-only -q
```

- Run particular test (good for debugging):

```
pytest unit/test_bcid_counter.py::test[2-1]
```

Summary

- Cocotb is an interface for python with digital simulators
- HDL code is simulated with the simulator of choice
- The inputs/outputs of the DUT are driven by cocotb coroutines
- One coroutine is the entry point for each test
- A pytest function invokes the simulator and the entry point coroutine

Questions

Questions?

Exercise

- Github repo contains a simple counter design: https://github.com/maxbab1/cocotb-test-examples.git
- Try to run all tests, one file, one test
- Modify the HDL code to include one additional input pin for the counting direction (up/down, only in 'master mode')
- Include a small test for the up/down mode

Links

- https://www.cocotb.org/
- https://docs.cocotb.org/en/stable/index.html
- https://pypi.org/project/cocotb-test/
- https://indico.cern.ch/event/1127562/contributions/4954530/attachments/2512843/4319511/TWEPP-2022_UVVM.pptm