SIMPLE YET POWERFUL: OPEN-SOURCE HDL SIMULATION WITH COCOTB

HAYDER AL-HAKEEM

WÄRTSILÄ FINLAND OY

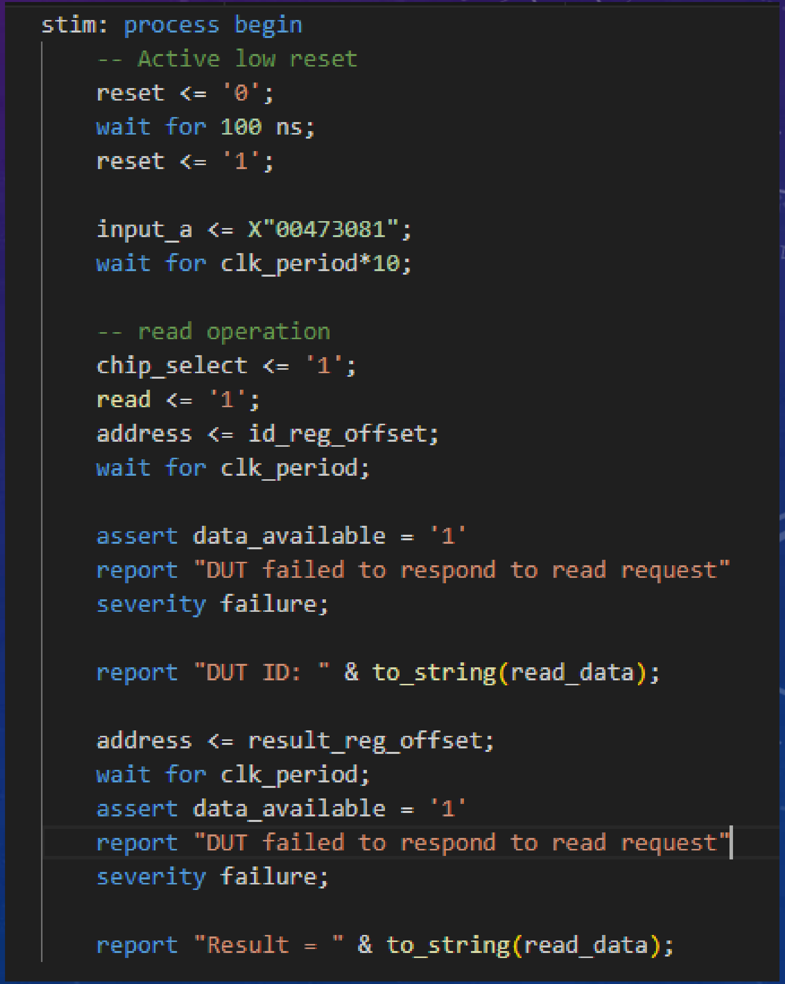
DVCLUB EUROPE | 08 OCT 2024

PRESENTATION OBJECTIVES

- Why and when is python a good choice for simulating HDL?
- How to integrate a free and open source verification environment for simulating HDL with Python?
- DEMO: Simulating a DUT with AXI-Lite bus using CocoTB

GENERAL PURPOSE PROGRAMMING LANGUAGES FOR VERIFICATION

- Verification is the process to ensure that the implementation fulfils the technical requirements and specifications.
- HDL testbenches script stimulus and verify outputs.
- They make heavy use of software-like language features which are not synthesizable.
- HDLs have been extended with features to support verification but they are still not as powerful and flexible as general-purpose programming languages.



```vhdl
stim: process begin
    -- Active low reset
    reset <= '0';
    wait for 100 ns;
    reset <= '1';

    input_a <= X"00473081";
    wait for clk_period*10;

    -- read operation
    chip_select <= '1';
    read <= '1';
    address <= id_reg_offset;
    wait for clk_period;

    assert data_available = '1'
    report "DUT failed to respond to read request"
    severity failure;

    report "DUT ID: " & to_string(read_data);

    address <= result_reg_offset;
    wait for clk_period;
    assert data_available = '1'
    report "DUT failed to respond to read request"
    severity failure;

    report "Result = " & to_string(read_data);
```

BENEFITS

- The main reference should be the complete and non-ambiguous requirements which both the designer and verification engineer interpret the same way.
  - Writing the tests without relying on the HDL implementation code avoids repeating the same mistakes.
- HDL will be tested in similar scenarios to how it will be used by drivers and applications code.

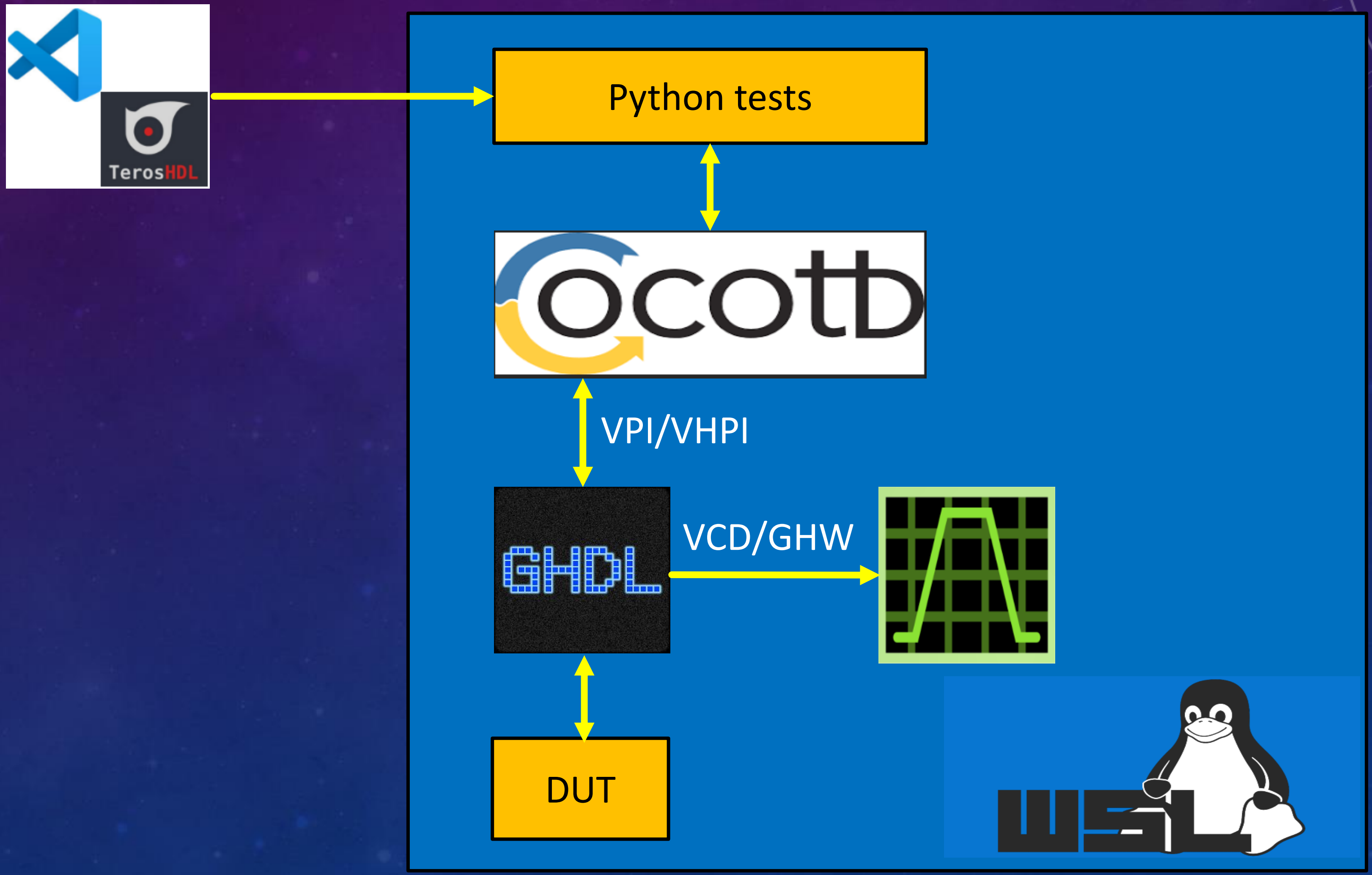
WHY PYTHON AND COCOTB?

- Python:
  - Has the largest package libraries to simulate any CPU/software use case.
  - Is popular and easy, invites driver developers to join the verification effort.
- Cocotb:
  - Enables Python code to access ports and internal signals in the DUT.
  - There is a wealth of BFMs in the Cocotb-bus repository.
  - It works well with most commercial and free simulators and is in active development.
    - Vivado Xsim is still not supported yet.

DRAWBACK

- No visibility for testbench signals in the simulation trace.
- Strong language but still catching up with methodology
  - pyuvm and uvm-python are not widely used like UVM or OSVVM/UVVM
- SystemVerilog can be faster depending on the simulation case:
  - A study of UVM compared to pyuvm: “The execution times of the Python testbench on commercial simulators were 8 to 21 times longer than those of the SystemVerilog testbench in tests with AHB-Lite write operations and random stimulus.” [reference]
- However
  - The more expensive cost is the developer’s time which python saves
  - Simulation execution time for our work was not an issue as all our tests finish within few minutes.
  - Certain parts of computationally-heavy testbenches can be accelerated by writing C extensions to python

FREE AND OPEN SOURCE VERIFICATION ENVIRONMENT AROUND COCOTB



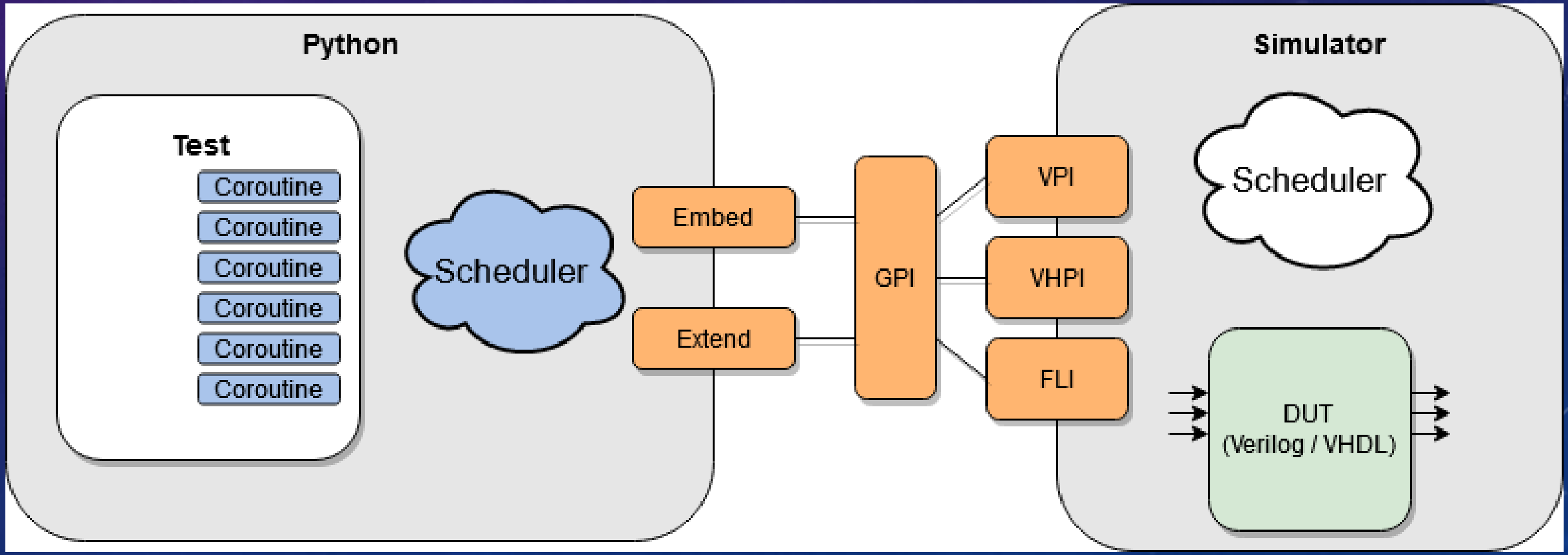
SETTING UP THE TOOLS

- WSL is free and strongly integrated with Windows, it offers better performance and easier files and network sharing with the host compared to virtual machines. It works seamlessy with MS VSCode and has lots of extensions to support HDL and python development
- For the simulator and wave viewer all that is needed is:
  - # apt install ghdl-gcc libghdl<same_simulator_version>
  - # apt install gtkwave
- To install Cocotb:
  - # pip install Cocotb cocotb-bus cocotb-coverage
- Teros HDL: One stop shop to install all the extensions you need!
  - Error checking via the simulator or a language server & Style checking.
  - Auto-generate Testbench, Instace, templates ..etc
  - Code Formatter, schematic and state machine viewer, documentation generators and more
  - # pip install teroshdl
  - # then install VSCode extension.

WHAT IS COCOTB

- “Cocotb is a COroutine based COsimulation TestBench environment for verifying VHDL and SystemVerilog RTL using Python.”

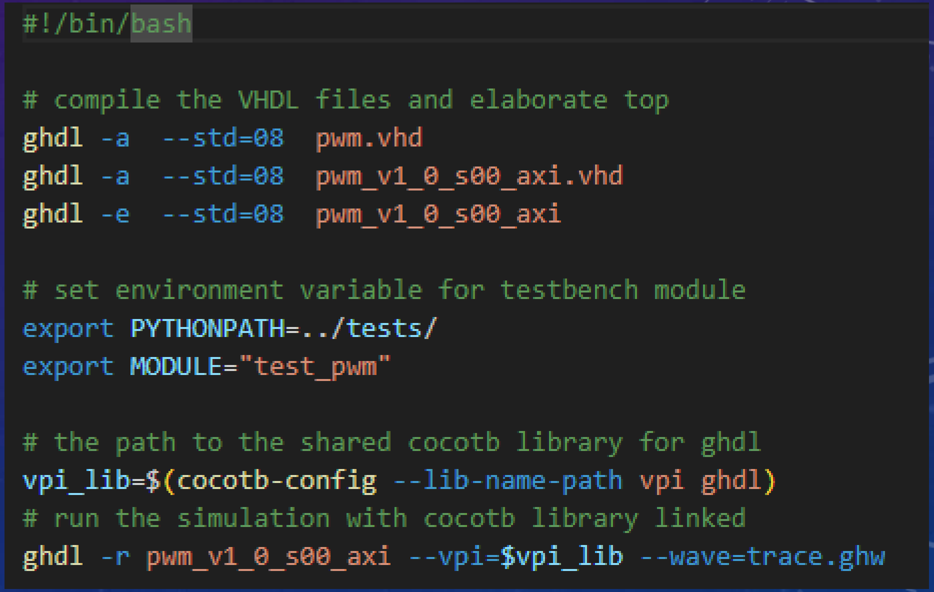
- It will allow the HDL code to bind with python code using the VPI or VHPI implementation of the simulator.
- What do we get? full access to the DUT VHDL ports and internal signals from python



HOW TO RUN A COCOTB SIMULATION WITH BASH

Full Details in webinar: “Ways to run Cocotb: makefiles, cocotb-test, or your custom setup”

- Bash script calling the simulator directly
  - Full control over build and simulation
  - Switching simulators isn’t easy
- Steps:
  - Build your files and elaborate top entity with the simulator as usual
  - Export the python testbench path and module name
  - Get the file path for the Cocotb shared library for your simulator
  - Call the simulator with *--vpi* pointing to the Cocotb shared library



```
#!/bin/bash

# compile the VHDL files and elaborate top
ghdl -a  --std=08  pwm.vhd
ghdl -a  --std=08  pwm_v1_0_s00_axi.vhd
ghdl -e  --std=08  pwm_v1_0_s00_axi

# set environment variable for testbench module
export PYTHONPATH=../tests/
export MODULE="test_pwm"

# the path to the shared cocotb library for ghdl
vpi_lib=$(cocotb-config --lib-name-path vpi ghdl)
# run the simulation with cocotb library linked
ghdl -r pwm_v1_0_s00_axi --vpi=$vpi_lib --wave=trace.ghw
```

HOW TO RUN A COCOTB SIMULATION WITH TEST RUNNERS

- Makefile
  - Portable between simulators
  - Difficult syntax

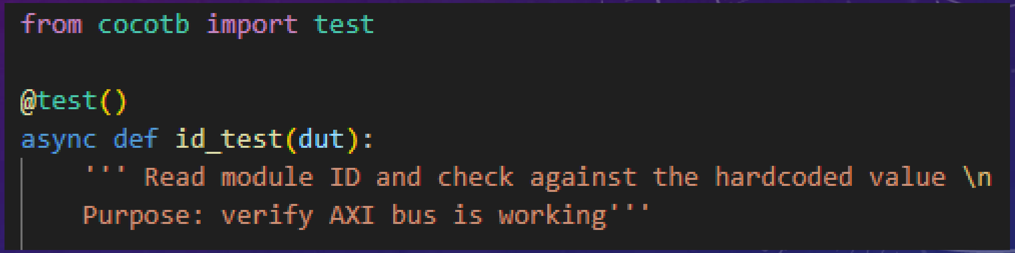
```
SIM=riviera
TOPLEVEL_LANG=verilog

PWD=$(shell pwd)

export PYTHONPATH := $(PWD)/../tests:$(PWD)/../model:$(PYTHONPATH)

VERILOG_SOURCES := $(PWD)/../hdl/adder.sv
TOPLEVEL := adder
MODULE := test_adder

include $(shell cocotb-config --makefiles)/Makefile.sim
```

- Python runner
  - Recommended by Cocotb
  - Leverage the power of python to setup the simulation and post-process the results

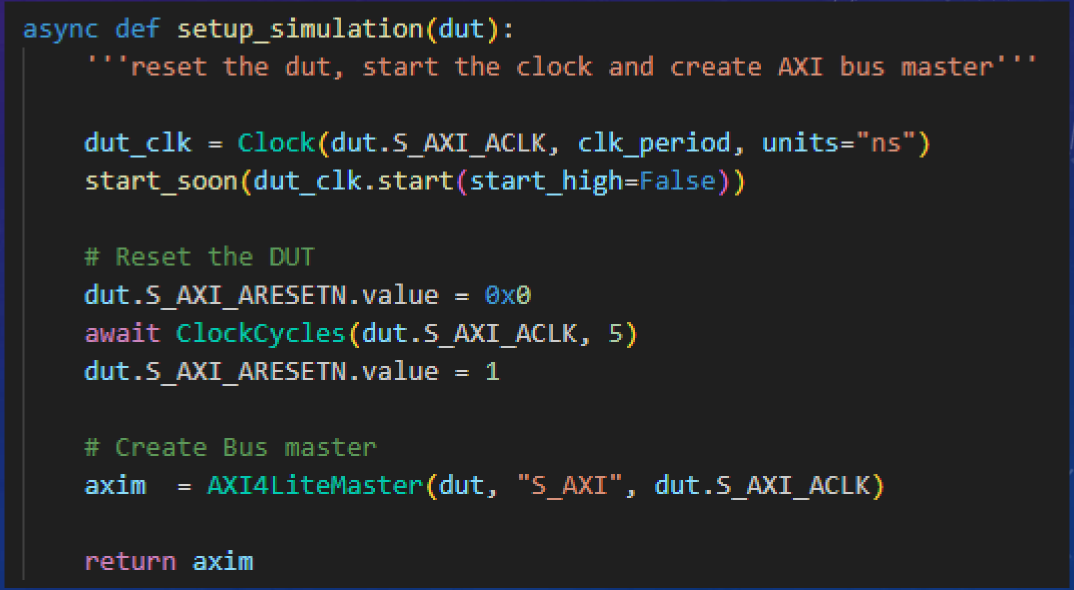
```
def run_adder_test():
    run(
        verilog_sources=verilog_sources, # sources
        toplevel="adder",                # top level HDL
        module="test_adder"              # name of cocotb test module
    )

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

BASICS OF A COCOTB TESTBENCH

- dut:
  - An object pointing to the top-level entity instance
  - All ports and internal signals of the DUT are accessible using the dot operator
- @cocotb.test:
  - A decorator to mark a Callable which returns a Coroutine as a test.
  - Cocotb will automatically pick up and run tests present in the python module.
- Async def
  - Used to declare a “coroutine”; an asynchronous function
  - We can think of it like the VHDL processes/ Verilog always block running concurrently inside the testbench



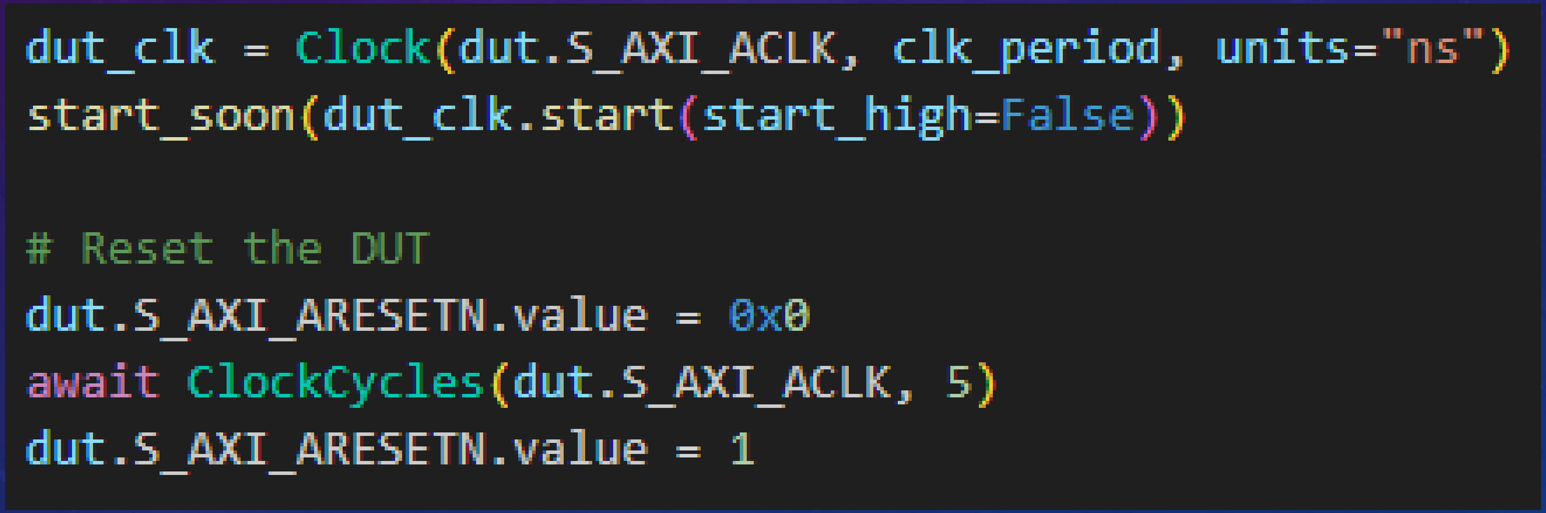
```python
from cocotb import test

@test()
async def id_test(dut):
    ''' Read module ID and check against the hardcoded value \n
    Purpose: verify AXI bus is working'''
```



```python
async def setup_simulation(dut):
    '''reset the dut, start the clock and create AXI bus master'''

    dut_clk = Clock(dut.S_AXI_ACLK, clk_period, units="ns")
    start_soon(dut_clk.start(start_high=False))

    # Reset the DUT
    dut.S_AXI_ARESETN.value = 0x0
    await ClockCycles(dut.S_AXI_ACLK, 5)
    dut.S_AXI_ARESETN.value = 1

    # Create Bus master
    axim  = AXI4LiteMaster(dut, "S_AXI", dut.S_AXI_ACLK)

    return axim
```

BASICS OF A COCOTB TESTBENCH

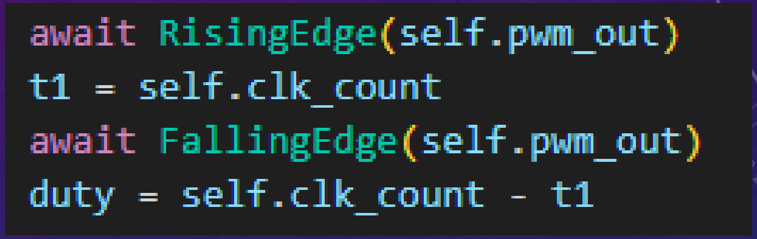
- await & start_soon:
  - An **await** will run an async coroutine and wait for it to complete.
  - The called coroutine "blocks" the execution of the current coroutine.
  - **start_soon()** runs the coroutine concurrently, allowing the current coroutine to continue executing.



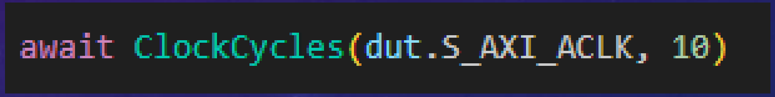
```
dut_clk = Clock(dut.S_AXI_ACLK, clk_period, units="ns")
start_soon(dut_clk.start(start_high=False))

# Reset the DUT
dut.S_AXI_ARESETN.value = 0x0
await ClockCycles(dut.S_AXI_ACLK, 5)
dut.S_AXI_ARESETN.value = 1
```

TRIGGERS

- Common triggers:
  - RisingEdge(), FallingEdge()
  - ClockCycles()
  - ReadOnly(), ReadWrite(): if you need a specific simulation delta cycle phase

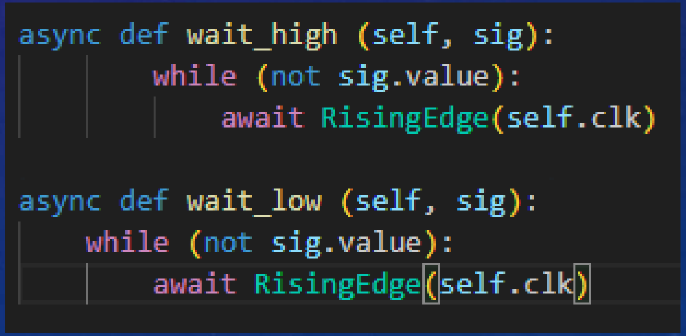


```
await RisingEdge(self.pwm_out)
t1 = self.clk_count
await FallingEdge(self.pwm_out)
duty = self.clk_count - t1
```



```
await ClockCycles(dut.S_AXI_ACLK, 10)
```

- Not built-in:
  - Test if signal is stable or certain value



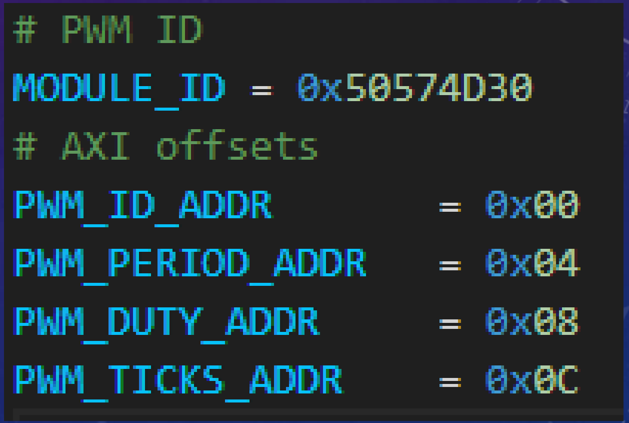
```
async def wait_high (self, sig):
        while (not sig.value):
            await RisingEdge(self.clk)

async def wait_low (self, sig):
    while (not sig.value):
        await RisingEdge(self.clk)
```

DEMO OBJECTIVES

- How to structure a Cocotb simulation project.
- How to write a python test runner.
- How to perform transaction-based tests with Cocotb-bus AXI-Lite BFM
- How to write a concurrent checker for a DUT output

DUT DESCRIPTION

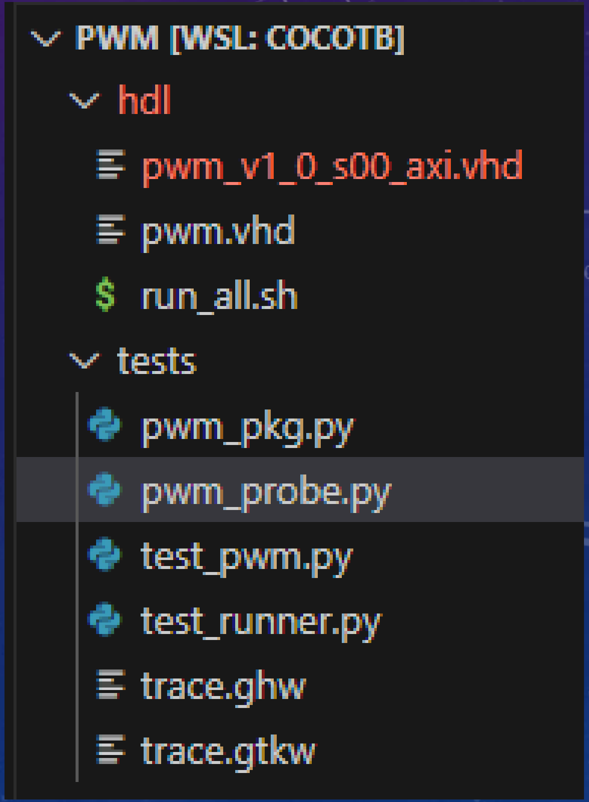
- One output pin (PWM)
- AXI-Lite Bus interface
- Four registers
  - Hard-Coded ID hex value, RO
  - Adjustable PWM period and duty-cycle (in clock ticks) R/W
  - Counts the total number of PWM ticks since last reset. R/O



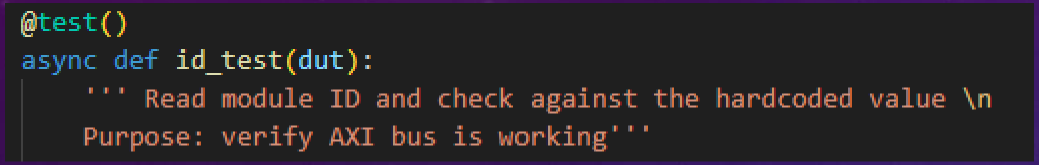
```
# PWM ID
MODULE_ID = 0x50574D30
# AXI offsets
PWM_ID_ADDR         = 0x00
PWM_PERIOD_ADDR     = 0x04
PWM_DUTY_ADDR       = 0x08
PWM_TICKS_ADDR      = 0x0C
```

PROJECT STRUCTURE

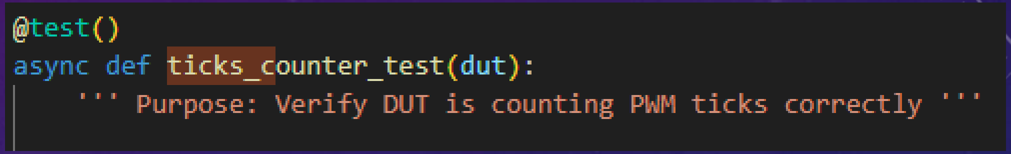
File/folder	Description
pwm.vhd	The PWM generator
pwm_v1_0_s00_axi.vhd	Vivado generated 4 registers AXI4-Lite slave interface
sim_build	The default folder where cocotb places compiled HDL entities
pwm_pkg.py	Register offsets, constants and reusable coroutines
test_runner.py	An instance of cocotb test runner with required simulator configurations to setup the simulation
test_pwm.py	A file containing the tests only (separating the tests from the architecture and setup)
pwm_probe.py	Concurrent checker class that measures PWM duty and cycle and the number of PWM cycles since reset



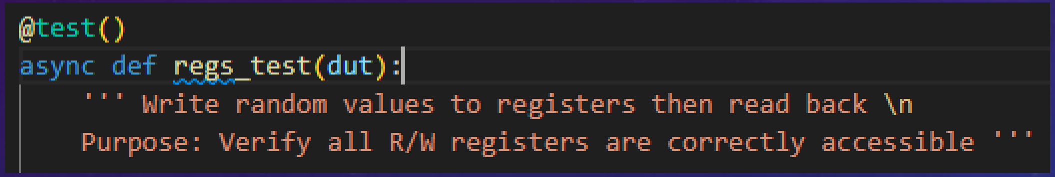
TESTS DESCRIPTION



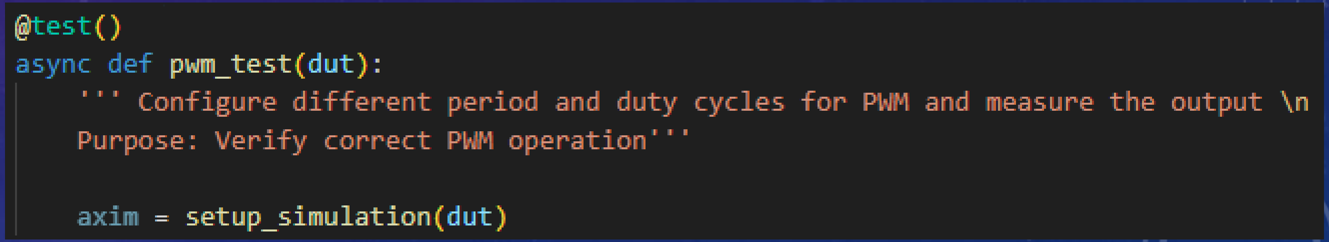
```python
@test()
async def id_test(dut):
    ''' Read module ID and check against the hardcoded value \n
    Purpose: verify AXI bus is working'''
```



```python
@test()
async def ticks_counter_test(dut):
    ''' Purpose: Verify DUT is counting PWM ticks correctly '''
```



```python
@test()
async def regs_test(dut):
    ''' Write random values to registers then read back \n
    Purpose: Verify all R/W registers are correctly accessible '''
```



```python
@test()
async def pwm_test(dut):
    ''' Configure different period and duty cycles for PWM and measure the output \n
    Purpose: Verify correct PWM operation'''

    axim = setup_simulation(dut)
```

THANK YOU FOR LISTENING!

- Live DEMO
- Q&A