Cocotb Presentation

Introduction to Cocotb

Timothée Charrier
timothee.charrier@elsys-design.com

Elsys Design
Advans Group

February 4, 2025

Agenda

1. Introduction
2. Basic Example with Combinational Logic
3. Writing Testbenches
4. Example with Clocked Logic

1. Introduction

1.1 Introduction

References

[1] Robin Müller. *Python based FPGA verification using CocoTB*. Tech. rep. University of Applied Sciences and Arts Northwestern Switzerland, July 2024. URL: https://github.com/m47812/CocoTb_Example/tree/main.

[2] FOSSi Foundation. *CocoTB*. Version 2.0.0. Oct. 26, 2024. URL: https://github.com/cocotb/cocotb.

[3] Ben Rosser. *Cocotb: a Python-based digital logic verification framework*. Tech. rep. University of Pennsylvania, Dec. 1028. URL: https://indico.cern.ch/event/776422/attachments/1769690/2874927/cocotb_talk.pdf.

1.2 Introduction

What is Cocotb?

Definition

- **CO**routine based **CO**-simulation **T**est**B**ench (COCOTB) is an open-source Python library for digital design verification.
- Enables writing testbenches in Python as an alternative to traditional testbenches in VHDL, Verilog, or SystemVerilog.
- Provides a high-level, Pythonic interface to interact with and validate RTL designs.
- Allows seamless simulation of VHDL and Verilog designs entirely in Python.
- Similar in philosophy to Universal Verification Methodology (UVM), but leverages the simplicity and power of Python.
- Encourages reusable, modular, and scalable verification code.

1.3 Introduction

Verification in Microelectronics

Key Points

- Verification is a crucial step in the design of digital systems.
- Ensures that the design meets the specifications.
- Verification is a time-consuming task.
- Verification is a complex task.

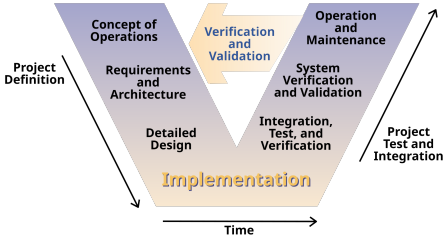


Figure: V-Cycle in Microelectronics

1.4 Introduction

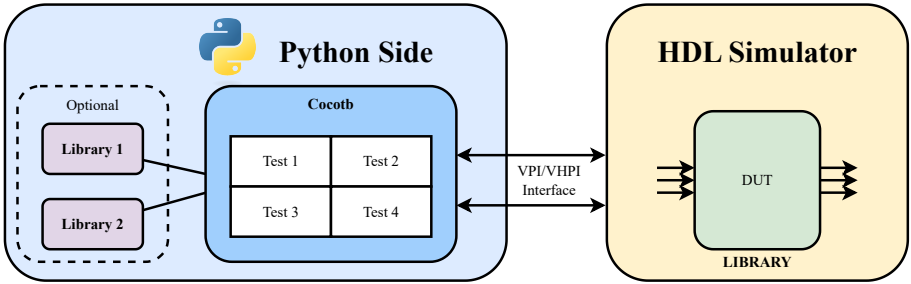
Existing Solutions

Tools and Frameworks for Verification

- VHDL/Verilog/SystemVerilog testbenches.
- Universal Verification Methodology (UVM).
- Open Source VHDL Verification Methodology (OSVVM).
- C++ or SystemC-based testbenches.
- Python-based testbenches.

1.5 Introduction

Cocotb Architecture Overview (adapted from [1] and [2])



- **VPI/VHPI**: **V**erilog/**VH**DL **P**rocedural **I**nterface
- **DUT**: **D**esign **U**nder **T**est

Key Points

- The simulator runs the RTL design.
- Uses the VHDL Procedural Interface (VHPI) or Verilog Procedural Interface (VPI) interfaces.

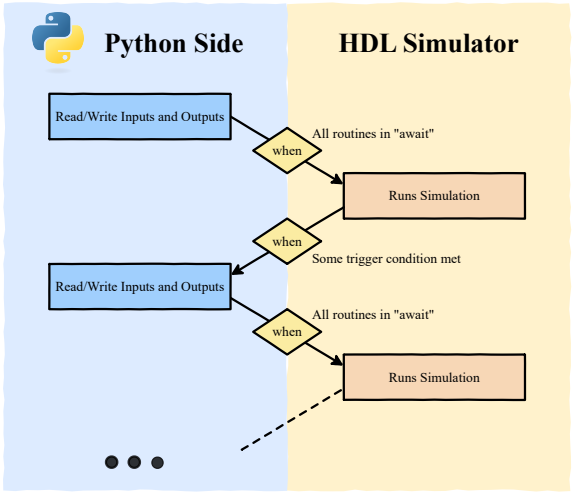
1.6 Introduction

Python and Simulator Interaction (adapted from [1])

How it Works

Ping Pong control between Python Testbench and the simulator.

(image source)



1.7 Installation

Requirements

Requirements

- Python 3.6 or later.
- GNU Make 3 or later.
- VHDL/Verilog simulator (e.g. GHDL, NVC, Verilator, ModelSim, Cadence, Synopsys, etc.).

Note that this presentation is based on Cocotb version 2.0.0-alpha (commit `3b9fee3`). See the Simulator Support page for more information.

1.8 Installation

Good Practices

Recommendations

- Use a Python virtual environment:

```
$ python3 -m venv .venv
$ source .venv/bin/activate
```

- Use a linter for any code you write:
  - Ruff, Black, Flake8, etc. for Python.
  - vhdl-ls for VHDL code, Verible for Verilog/SystemVerilog code.

2. Basic Example with Combinational Logic

2.1 Basic Example with Combinational Logic

HDL Design and Python Testbench

Verilog Adder (adder.sv)

```
`timescale 1ns/1ps

module adder #(
    parameter integer DATA_WIDTH = 4
  ) (
    input  logic unsigned [DATA_WIDTH-1:0] X,
    input  logic unsigned [DATA_WIDTH-1:0] Y,
    output logic unsigned [DATA_WIDTH:0]   SUM
  );

  assign SUM = X + Y;

endmodule
```

Python Testbench (test_adder.py)

```
from random import randint

import cocotb
from cocotb.triggers import Timer


@cocotb.test()
async def test_adder_10_random_values(dut):
    dut.X.value = 0
    dut.Y.value = 0

    await Timer(2, units="ns")
    assert dut.SUM.value == 0, "Error: 0 + 0 != 0"

    # Get generic value for DATA_WIDTH
    DATA_WIDTH = int(dut.DATA_WIDTH.value)

    for i in range(10):
        X = randint(0, 2**DATA_WIDTH - 1)
        Y = randint(0, 2**DATA_WIDTH - 1)
        dut.X.value = X
        dut.Y.value = Y

        await Timer(2, units="ns")
        assert dut.SUM.value == X + Y, f"Error: {X} + {Y} != {X + Y}"
```

2.2 Basic Example with Combinational Logic

Some Explanations on the Testbench

- `@cocotb.test()` is a decorator to mark a function as a test.
- `async` is used to define a coroutine.
- `.value = some_value` is used to assign a value to a signal.
- `.value` is used to read the value of a signal.
- `await` is used to wait for a coroutine to finish.
- `assert` is used to test a condition.

Python Testbench (test_adder.py)

```python
from random import randint

import cocotb
from cocotb.triggers import Timer


@cocotb.test()
async def test_adder_10_random_values(dut):
    dut.X.value = 0
    dut.Y.value = 0

    await Timer(2, units="ns")
    assert dut.SUM.value == 0, "Error: 0 + 0 != 0"

    # Get generic value for DATA_WIDTH
    DATA_WIDTH = int(dut.DATA_WIDTH.value)

    for i in range(10):
        X = randint(0, 2**DATA_WIDTH - 1)
        Y = randint(0, 2**DATA_WIDTH - 1)
        dut.X.value = X
        dut.Y.value = Y

        await Timer(2, units="ns")
        assert dut.SUM.value == X + Y, f"Error: {X} + {Y} != {X + Y}"
```

2.3 How to invoke the test (1/3)?

Using the Makefile

Makefile invoking the adder test

```
# Makefile for simulating Verilog code with cocotb and Verilator

# Directory containing Verilog source files
SRC_DIR             := ../../../rtl/example/

# Source Files
VERILOG_SOURCES     := $(SRC_DIR)/adder.sv

# Cocotb and Verilator configuration
SIM                 := verilator
TOPLEVEL_LANG       := verilog
TOPLEVEL            := adder
COCOTB_TEST_MODULES := test_adder
MODULE              := $(COCOTB_TEST_MODULES)
EXTRA_ARGS          += --trace --trace-fst --trace-structs -GDATA_WIDTH=8

# Calling cocotb
PWD                 := $(shell pwd)
export PYTHONPATH   := $(PWD)/../model:$(PYTHONPATH)
include $(shell cocotb-config --makefiles)/Makefile.sim

clean::
  rm -rf results.xml __pycache__ sim_build dump.fst
```

- `VERILOG_SOURCES` is the list of Verilog files to compile.
- `SIM` is the simulator to use.
- `TOPLEVEL_LANG` is the language of the top-level module.
- `TOPLEVEL` is the name of the top-level module.
- `COCOTB_TEST_MODULE` and `MODULE` are the name of the python test module.
- `EXTRA_ARGS` are the extra arguments to pass to the simulator (generics, trace, etc.).

2.4 How to invoke the test (2/3)?

Using the Python test runner

Python Runner Part 1

```python
from cocotb_tools.runner import Runner, get_runner


def test_counter_runner() -> None:
    """Function Invoked by the test runner to execute the tests."""
    # Define the simulator to use
    default_simulator: str = "verilator"

    # Build Arguments
    build_args: list[str] = [
        "--trace",
        "--trace-fst",
        "--trace-structs",
    ]

    # Define LIB_RTL
    library = "LIB_RTL"

    # Define rtl_path
    rtl_path: Path = (Path(__file__).parent.parent.parent.parent /
    ↪  "rtl/").resolve()

    # Define the sources
    sources: list[str] = [f"{rtl_path}/example/adder.sv"]
```

Python Runner Part 2

```python
# Top-level HDL entity
entity: str = "adder"

# Generics Configuration
parameters: dict[str, int] = {"DATA_WIDTH": 8}

try:
    # Get simulator name from environment
    simulator: str = os.environ.get("SIM", default=default_simulator
    ↪  )

    # Initialize the test runner
    runner: Runner = get_runner(simulator_name=simulator)

    # Build HDL sources
    runner.build(
        build_dir="sim_build",
        build_args=build_args,
        clean=True,
        hdl_library=library,
        hdl_toplevel=entity,
        parameters=parameters,
        sources=sources,
        waves=True,
    )
```

2.5 How to invoke the test (3/3)?

Using the Python test runner (continued)

Python Runner Part 3

```
        # Run tests
        runner.test(
            build_dir="sim_build",
            hdl_toplevel=entity,
            hdl_toplevel_library=library,
            test_module=f"test_{entity}",
            waves=True,
        )

        # Log path to waveform file
        sys.stdout.write(
            f"Waveform file: {(Path('sim_build') / f'dump_{entity}.fst').resolve()}\n",
        )

    except Exception as e:
        error_message: str = f"Failed in {__file__} with error: {e}"
        raise RuntimeError(error_message) from e


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

2.6 Build Flows Comparison

Makefile vs. Python Test Runner

Makefile Flow

- Widely used and well-documented.
- Involves writing a Makefile and executing `make` in the terminal.
- Conceptually similar to the Python test runner but requires additional tools and setup.

This flow is slowly being deprecated in favor of the Python Test Runner.

Python Test Runner Flow

- A newer and experimental approach.
- Allows running tests directly using Python scripts or `pytest`.
- Integrates seamlessly with Python testing frameworks such as `pytest`.
- Requires fewer tools (only Python and a simulator).
- Simplifies CI/CD pipeline integration by using `pytest` instead of navigating directories and executing `make`.

2.7 Running the Test

Output of the Adder Test

Commands to Run the Test

Method	Command
Makefile	`make`
Python Test Runner	`python test_adder.py` or `pytest test_adder.py`

Terminal Output of the Adder Test

```
     0.00ns INFO     cocotb                   Running on Verilator version 5.031 devel
     0.00ns INFO     cocotb                   Seeding Python random module with 1731421538
     0.00ns INFO     cocotb.regression        Found test test_adder.test_adder_10_random_values
     0.00ns INFO     cocotb.regression        running test_adder (1/1)
    22.00ns INFO     cocotb.regression        test_adder passed
    22.00ns INFO     cocotb.regression        ****************************************************************************************************
                                              ** TEST                                 STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                                              ****************************************************************************************************
                                              ** test_adder.test_adder_10_random_values   PASS          22.00           0.00      27744.70  **
                                              ****************************************************************************************************
                                              ** TESTS=1 PASS=1 FAIL=0 SKIP=0                          22.00           0.04        550.57  **
                                              ****************************************************************************************************

- :0: Verilog $finish
INFO: Results file: /path/to/project/sim_build/results.xml
Waveform file: /path/to/project/sim_build/dump.fst
```

2.8 Waveform Debugging Overview

Generate and view waveforms for debugging:

Enabling Waveform Generation

Method	Configuration
Makefile	`WAVES := 1`
Python Runner	Add `waves=True` to `runner.build` and `runner.test`

Simulator-Specific FST Options

Simulator	Options
Verilator	`--trace --trace-fst --trace-structs`
NVC	`--wave=wave_dump.fst`

Use tools like GTKWave or Surfer to view waveforms if using Open Source simulators.

3. Writing Testbenches

3.1 Time Management

Simulation Triggers (Part 1/2)

Trigger Overview

- **Independent Simulation**: The simulator runs the design and the testbench concurrently.
- **Communication**: Interaction occurs through VPI/VHPI interfaces, managed via cocotb *triggers*.
- **Execution and Time**:
  - During Python execution, simulation time is paused.
  - The testbench halts at a trigger, waiting for the specified condition to be met before continuing.
- To use a trigger, a coroutine should `await` it.

3.2 Time Management

Simulation Triggers (Part 2/2)

Most Common Triggers

Trigger	Description
`Edge(signal)`	Waits for any edge transition of the signal
`RisingEdge(signal)`	Waits for a rising edge of the signal
`FallingEdge(signal)`	Waits for a falling edge of the signal
`ClockCycles(signal, num_cycles, rising=True)`	Waits for a number of clock cycles
`Timer(time, units)`	Waits for a certain amount of time

Trigger Usage Examples

```
await RisingEdge(dut.clock)       # Wait for next clock edge
await ClockCycles(dut.clock, 5)   # Wait for 5 clock cycles
await Edge(dut.s_axis_tready)     # Wait for any edge on tready
await Timer(100, units='ns')      # Wait for 100 nanoseconds
```

3.3 Concurrent and Sequential Execution

Coroutines in Python (Part 1/2)

Coroutines Overview

- **Coroutines** are functions that can pause and resume execution.
- **Concurrency**:
  - Multiple coroutines can run concurrently.
  - Use `await` to pause a coroutine until a trigger is satisfied.
  - Use `cocotb.start` or `cocotb.start_soon` to run a coroutine concurrently, allowing the current coroutine to continue executing.

3.4 Concurrent and Sequential Execution

Coroutines in Python (2/2)

Following example from [2] shows how to run a coroutine concurrently:

Example of Concurrent and Sequential Testbench

```python
# A coroutine
async def reset_dut(reset_n, duration_ns):
    reset_n.value = 0
    await Timer(duration_ns, units="ns")
    reset_n.value = 1
    reset_n._log.debug("Reset complete")

@cocotb.test()
async def parallel_example(dut):
    reset_n = dut.reset

    # Execution will block until reset_dut has completed
    await reset_dut(reset_n, 500)
    dut._log.debug("After reset")

    # Run reset_dut concurrently
    reset_thread = cocotb.start_soon(reset_dut(reset_n, duration_ns=500))

    # This timer will complete before the timer in the concurrently executing "reset_thread"
    await Timer(250, units="ns")
    dut._log.debug("During reset (reset_n = %s)" % reset_n.value)

    # Wait for the other thread to complete
    await reset_thread
    dut._log.debug("After reset")
```

3.5 Number Representation

How to correctly assign values to signals

Number Representation

- Cocotb allows assigning both signed and unsigned values to signals using a Python int.
- The range of values is determined by the width of the signal:

$$-2^{(Nbits-1)} \leq \text{value} \leq 2^{Nbits} - 1$$

- Assigning out-of-range values raises an `OverflowError`.
- Example with our adder and `DATA_WIDTH = 4`:

Assigning Values to a Signal

```
dut.X.value = 15  # valid
dut.X.value = -4  # valid (in range for 4 bits, even if X is unsigned)
dut.X.value = 16  # raises OverflowError
dut.X.value = -9  # raises OverflowError
```

3.6 Reading Values from Signals

Accessing and Interpreting Values

Value Types

- Access values in the DUT using the `value` property of a handle object.
- The Python type of a value depends on the handle's HDL type:
  - `LogicArray` for arrays of logic and subtypes (e.g., `std_logic`, `std_logic_vector` in VHDL, `logic`, `bit`, `bit_vector` in SystemVerilog).
  - `Array` for arrays and subtypes.
  - `int` for integer nets and constants.
  - `float` for floating point nets and constants.
  - `bool` for boolean nets and constants.
  - `bytes` for string nets and constants.
- For constrained or unconstrained arrays, cocotb creates an `Array` (list like) of `LogicArray` objects.

3.7 Finding elements in the design

Finding Elements

- To find elements of the DUT (for example, instances, signals, or constants) at a certain hierarchy level, we can use the `dir()` function.
- Here is the output of `dir()` for the DUT in the adder example:

Terminal Output of `dir(dut)` for the Adder

```
['DATA_WIDTH', 'SUM', 'X', 'Y', '_len', '_log', '_name',
 ...,
 '_path', '_type', 'get_definition_file', 'get_definition_name']
```

- We can then access the elements directly using the names in the list (e.g., `dut.SUM.value`), even internal signals, instances or constants.

3.8 Forcing and Freezing Signals

Changing Signal Values (example from [2])

Forcing and Freezing Signals

Cocotb provides a way to force and freeze signals to specific values.

Examples of Forcing and Freezing Signals

```
# Deposit action
dut.my_signal.value = 12
dut.my_signal.value = Deposit(12)  # equivalent syntax

# Force action
dut.my_signal.value = Force(12)    # my_signal stays 12 until released

# Release action
dut.my_signal.value = Release()    # Reverts any force/freeze assignments

# Freeze action
dut.my_signal.value = Freeze()
↪  # my_signal stays at current value until released
```

3.9 Logging and Debugging

The logging library

Logging Levels

- Cocotb uses the Python `logging` library to manage logging.
- The logging level for cocotb logs is set based on the `COCOTB_LOG_LEVEL` environment variable.
- The default logging level is `INFO`.
- The logging levels are, in order of increasing severity:
  - `DEBUG`
  - `INFO`
  - `WARNING`
  - `ERROR`
  - `CRITICAL`
- It is very useful to anything from debugging to tracing the simulation (e.g. `dut._log.info(f"DATA_WIDTH={dut.DATA_WIDTH.value})"`).

3.10 Using an HDL Library

Custom Libraries with a Makefile

HDL Libraries

- Use the `TOPLEVEL_LIBRARY` variable to specify the library name.
- Most simulators will try to automatically determine the compilation order. However, it is recommended to specify the order of compilation explicitly:

Specifying the Compilation Order with Multiple Files

```
VERILOG_SOURCES := $(SRC_DIR)/ascon_pkg.sv                          \
                   $(SRC_DIR)/add_layer/add_layer.sv                \
                   $(SRC_DIR)/substitution_layer/sbox.sv            \
                   $(SRC_DIR)/substitution_layer/substitution_layer.sv \
                   $(SRC_DIR)/diffusion_layer/diffusion_layer.sv    \
                   $(SRC_DIR)/xor/xor_begin.sv                      \
                   $(SRC_DIR)/xor/xor_end.sv                        \
                   $(SRC_DIR)/permutation/permutation.sv            \
                   $(SRC_DIR)/fsm/ascon_fsm.sv                      \
                   $(SRC_DIR)/ascon/ascon.sv
```

3.11 Using an HDL Library

Custom Libraries with a Python Test Runner

HDL Libraries

- Use `hdl_library=LIBRARY_NAME` in the `runner.build()` and `runner.test()` methods.
- For example, to compile the ascon module, use:

Specifying the Compilation Order with Multiple Files

```
sources: list[str] = [
    f"{rtl_path}/ascon_pkg.sv",
    f"{rtl_path}/addition_layer/addition_layer.sv",
    f"{rtl_path}/substitution_layer/sbox.sv",
    f"{rtl_path}/substitution_layer/substitution_layer.sv",
    f"{rtl_path}/diffusion_layer/diffusion_layer.sv",
    f"{rtl_path}/xor/xor_begin.sv",
    f"{rtl_path}/xor/xor_end.sv",
    f"{rtl_path}/permutation/permutation.sv",
    f"{rtl_path}/fsm/ascon_fsm.sv",
    f"{rtl_path}/ascon/ascon.sv"]
```

4. Example with Clocked Logic

4.1 Testing a Clocked Design

An HDL Counter

Verilog Adder (counter.sv)

```
`timescale 1ns / 1ps

module counter #(
    parameter integer DATA_WIDTH = 8,                       // Counter width
    parameter integer COUNT_FROM = 0,                       // Initial value
    parameter integer COUNT_TO   = 2 ** (DATA_WIDTH - 1),   // Terminal value
    parameter integer STEP       = 1                        // Increment step
) (
    input  logic                   clock,         // Clock signal
    input  logic                   reset_n,       // Asynchronous reset (active low)
    input  logic                   count_enable,  // Enable signal
    output logic [DATA_WIDTH-1:0] count           // Counter output
);

    // Sequential logic
    always @(posedge clock or negedge reset_n) begin
        if (!reset_n) count <= COUNT_FROM[DATA_WIDTH-1:0];
        else if (count_enable) begin
            if (count >= COUNT_TO[DATA_WIDTH-1:0]) count <= COUNT_FROM[DATA_WIDTH-1:0];
            else count <= count + STEP[DATA_WIDTH-1:0];
        end
    end
endmodule
```

4.2 Defining reusable functions

Common Operations for Clocked Designs

Common Operations

To maintain clean and reusable test code, we define standard functions for common operations:

Function	Description
`setup_clock`	Configures the clock signal with specified period and duty cycle
`reset_dut`	Performs asynchronous reset sequence of the design
`sys_enable_dut`	Controls the system enable signal for the DUT
`initialize_dut`	Sets up initial conditions and default signal values
`toggle_signal`	Changes signal state from 0 to 1 or vice versa

Note: These functions are not exhaustive and can be extended to suit the design requirements.

4.3 Breakdown of the basic functions

Function: setup_clock

Python Function (setup_clock) - Part 1

```python
async def setup_clock(
    dut: cocotb.handle.HierarchyObject,
    period_ns: int = 10,
    *,
    verbose: bool = True,
) -> None:
    """
    Initialize and start the clock for the DUT.

    Parameters
    ----------
    dut : cocotb.handle.HierarchyObject
        The Device Under Test (DUT).
    period_ns : int
        Clock period in nanoseconds (default is 10).
    verbose : bool, optional
        If True, logs the clock operation (default is True).

    """
```

Python Function (setup_clock) - Part 2

```python
try:
    clock = Clock(signal=dut.clock, period=period_ns, units="ns")
    await cocotb.start(clock.start(start_high=False))

    if not verbose:
        return

    dut._log.info(f"Clock started with period {period_ns} ns.")

except Exception as e:
    error_message: str = (
        f"Failed in setup_clock with error: {e}",
        "Hint: DUT might not have a clock signal.",
    )
    raise RuntimeError(error_message) from e
```

4.4 Breakdown of the basic functions

Function: reset_dut

Python Function (reset_dut) - Part 1

```python
async def reset_dut(
    dut: cocotb.handle.HierarchyObject,
    num_cycles: int = 5,
    *,
    reset_high: int = 0,
    verbose: bool = True,
) -> None:
    """
    Reset the DUT.

    This function assumes the reset signal is active low.
    It asserts the reset signal for 'num_cycles' and then deasserts it.

    Parameters
    ----------
    reset_high : int, optional
        Indicates if the reset signal is active high (1) or active low (0).
        By default, the reset signal is active low (0).

    ... skipped for slide ...


    """
    if reset_high not in [0, 1]:
        error_message: str = (
            f"Invalid reset_high value: {reset_high}",
            "Hint: reset_high should be 0 or 1.",
        )
        raise ValueError(error_message)
```

Python Function (reset_dut) - Part 2

```python
try:
    if reset_high == 0:
        dut.reset_n.value = 0
    else:
        dut.reset_h.value = 1

    await ClockCycles(signal=dut.clock, num_cycles=num_cycles)

    if reset_high == 0:
        dut.reset_n.value = 1
    else:
        dut.reset_h.value = 0

    await ClockCycles(signal=dut.clock, num_cycles=2)

    if not verbose:
        return

    dut._log.info(
        f"DUT reset for {num_cycles} cycles with reset_high={reset_high}."
        ↪ ,
    )

except Exception as e:
    error_message: str = (
        f"Failed in reset_dut with error: {e}",
        "Hint: DUT might not have reset_n or reset_h port.",
    )
    raise RuntimeError(error_message) from e
```

4.5 Breakdown of the basic functions

Function: sys_enable_dut

Python Function (sys_enable_dut)

```python
async def sys_enable_dut(
    dut: cocotb.handle.HierarchyObject,
    *,
    verbose: bool = True,
) -> None:
    """
    Enable the DUT.

    Parameters
    ----------
    dut : SimHandleBase
        The device under test.
    verbose : bool, optional
        If True, logs the enable operation (default is True).

    """
    try:
        dut.i_sys_enable.value = 1
        await RisingEdge(signal=dut.clock)

        if not verbose:
            return

        dut._log.info("DUT enabled.")

    except Exception as e:
        error_message: str = (
            f"Failed in sys_enable_dut with error: {e}",
            "Hint: DUT might not have i_sys_enable port or clock signal.",
        )
        raise RuntimeError(error_message) from e
```

4.6 Breakdown of the basic functions

Function: initialize_dut

Python Function (initialize_dut) - Part 1

```python
async def initialize_dut(
    dut: cocotb.handle.HierarchyObject,
    inputs: dict,
    outputs: dict,
    *,
    clock_period_ns: int = 10,
    reset_high: int = 0,
    verbose: bool = True,
) -> None:
    """
    Initialize the DUT with default values.

    Parameters
    ----------
    dut : SimHandleBase
        The device under test (DUT).
    inputs : dict
        A dictionary containing the input names and values.
    outputs : dict
        A dictionary containing the output names and expected values.

    ... skipped for slide ...

    Usage
    -----

    >>> inputs = {"i_data": 0, "i_valid": 0}
    >>> outputs = {"o_data": 0, "o_valid": 0}
    >>> await initialize_dut(dut, inputs, outputs)
    """
    try:
```

Python Function (initialize_dut) - Part 2

```python
        # Setup the clock
        await setup_clock(dut=dut, period_ns=clock_period_ns, verbose=verbose)

        # Reset the DUT
        await reset_dut(dut=dut, reset_high=reset_high, verbose=verbose)

        # Set the input values
        for key, value in inputs.items():
            getattr(dut, key).value = value

        # Wait a few clock cycles
        await ClockCycles(signal=dut.clock, num_cycles=5)

        # Check the output values
        for key, value in outputs.items():
            assert getattr(dut, key).value == value, f"Output {key}
            ↪   is incorrect"

        # Check if i_sys_enable is present
        if hasattr(dut, "i_sys_enable"):
            await sys_enable_dut(dut=dut, verbose=verbose)
            await ClockCycles(signal=dut.clock, num_cycles=5)

        if not verbose:
            return

        dut._log.info("DUT initialized successfully.")

    except Exception as e:
        error_message: str = f"Failed in initialize_dut with error: {e}"
        raise RuntimeError(error_message) from e
```

4.7 Breakdown of the basic functions

Function: toggle_signal

Python Function (toggle_signal) - Part 1

```python
async def toggle_signal(
    dut: cocotb.handle.HierarchyObject,
    signal_dict: dict,
    *,
    verbose: bool = True,
) -> None:
    """
    Toggle a signal between high and low values.

    Parameters
    ----------
    dut : SimHandleBase
        The device under test (DUT).
    signal_dict : dict
        A dictionary containing the signal name and value.
        If the value is 1, the signal is toggled to 0;
        otherwise, it is toggled to 1.
    verbose : bool, optional
        If True, logs the signal toggling operation (default is True).

    Usage
    -----

    >>> signal_dict = {"i_valid": 0, "i_ready": 0}
    >>> await toggle_signal(dut, signal_dict)

    """
```

Python Function (toggle_signal) - Part 2

```python
try:
    for key, value in signal_dict.items():
        getattr(dut, key).value = value
        await RisingEdge(signal=dut.clock)

        if value == 1:
            getattr(dut, key).value = 0
        else:
            getattr(dut, key).value = 1

        await RisingEdge(signal=dut.clock)

    if not verbose:
        return

    dut._log.info("Signal toggled successfully.")

except Exception as e:
    error_message: str = (
        f"Failed to toggle signal {key}.",
        f"Error: {e}",
    )
    raise RuntimeError(error_message) from e
```

4.8 Breakdown of the basic functions

Function: get_generics

Another useful function

If a design has generics, we can define a function to get the generics from the design:

Python Function (get_generics) for counter.sv

```python
def get_generics(dut: cocotb.handle.HierarchyObject) -> dict:
    """
    Retrieve the generic parameters from the DUT.

    Parameters
    ----------
    dut : cocotb.handle.HierarchyObject
        The device under test (DUT).

    Returns
    -------
    dict
        A dictionary containing the generic parameters.

    """
    return {
        "DATA_WIDTH": int(dut.DATA_WIDTH.value),
        "COUNT_FROM": int(dut.COUNT_FROM.value),
        "COUNT_TO": int(dut.COUNT_TO.value),
        "STEP": int(dut.STEP.value),
    }
```

4.9 Modeling the HDL Counter

Counter Model

Counter Model

- The `CounterModel` class is a simple model of the counter.
- It is used to verify the behavior of the DUT.
- The model is used to compare the output of the DUT with the expected output.

This way, we can verify the correctness of the DUT. The class feeds the DUT with random inputs to compute an expected output, and finally compares the module output to the expected for correctness.
With more complex designs, the model could monitor a streaming data/valid bus, sample the bus when a transaction occurs, and compare the output to the expected output.

4.10 Tying it all together

Complete Testbench Implementation

Testbench Organization

- All utility functions are consolidated in `src/bench/cocotb_utils.py`
- Functions are designed to be modular and reusable across different testbenches
- Easy integration with both simple and complex designs

Complete Example

A full implementation example is available in this GitHub repository:

- Basic counter example: `src/rtl/example/`
- Associated testbenches: `src/bench/example/`
- Advanced implementation: Ascon-128 cryptographic core, using all the aspects discussed in this presentation.

Timothée Charrier

Aix-en-Provence, France, February 4, 2025

timothee.charrier@elsys-design.com