2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

MUNICH, GERMANY
OCTOBER 14-15, 2025

Next-Gen Verification with Python: Driving Hardware Tests with Pytest and Cocotb

Tymoteusz Błażejczyk, Abhiroop Bhowmik, Ronahi Halitoglu,
Shrinivas Naik, Reinier Bodemeijer

Qblox B.V

accellera SYSTEMS INITIATIVE

History and change of verif infrastructure (reinier)

Hard selling points (shri) - Done but can add more points to it

Analytic info about verif eng shortage (ronahi)

Slide number for each slide - Done

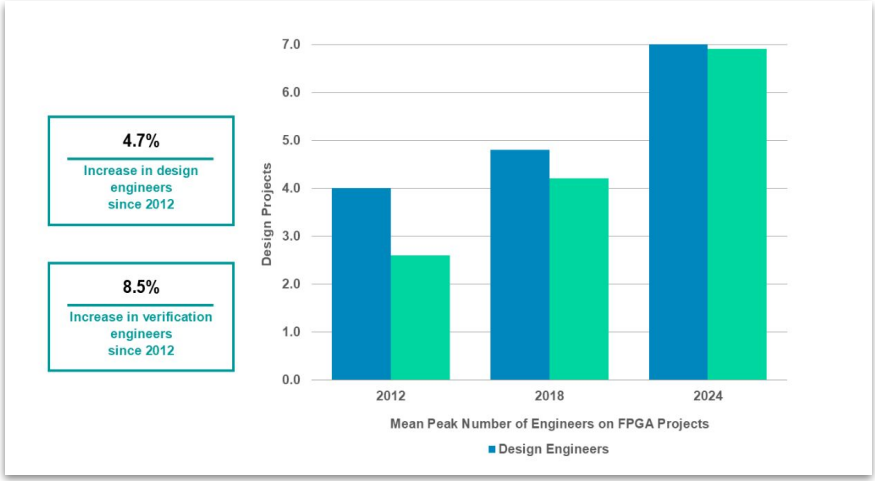
Less text (all)

More support material (images) (abhi)

Time for Slides + Time for demo + time for questions

Verification trends

- Difficulty in getting experienced SV/UVM verification engineers
- Difficulty onboarding to custom verification environment
- Wish to migrate smoothly to something easier?
- Something scalable and open source

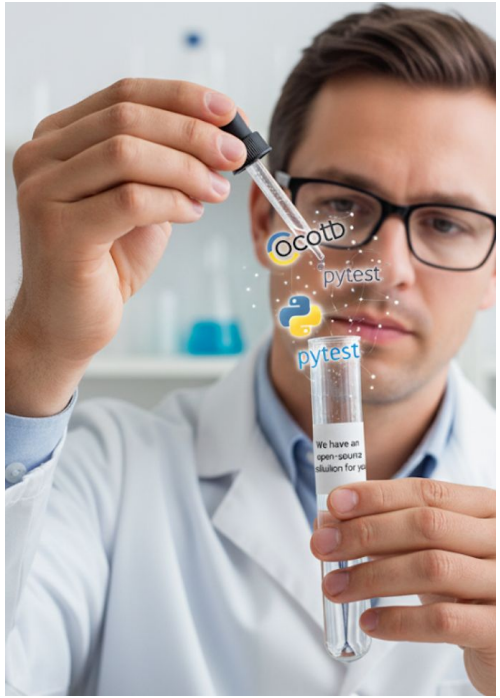


Source : “2024 Siemens EDA and Wilson Research Group FPGA functional verification trend report,” White Paper, Siemens EDA, 2024

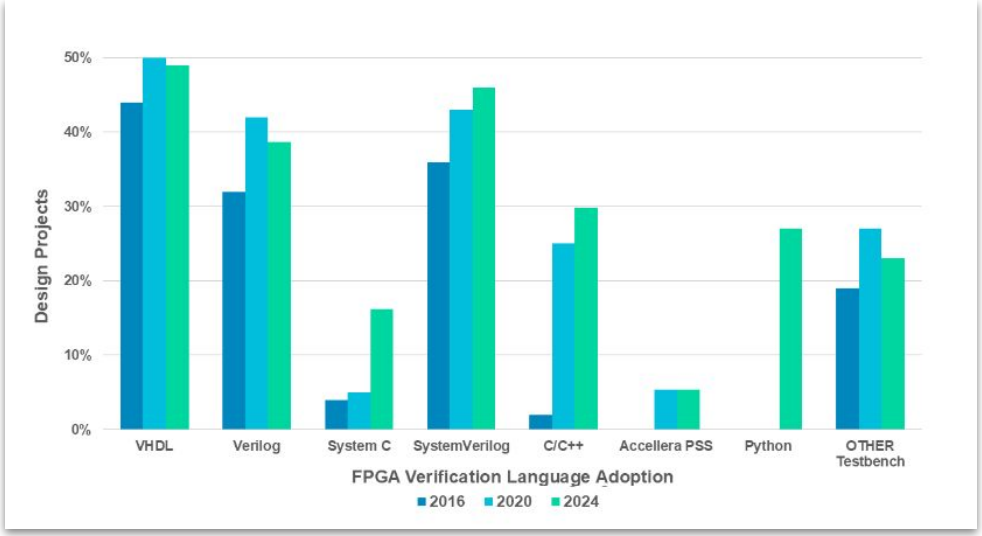
Verification trends

- Difficulty in getting experienced SV/UVM verification engineers
- Difficulty onboarding to custom verification environment
- Wish to migrate smoothly to something easier?
- Something scalable and open source

We have an open-source solution!



Python adoption?



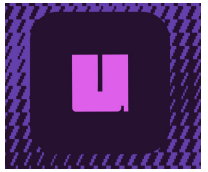
Source : "2024 Siemens EDA and Wilson Research Group FPGA functional verification trend report," White Paper, Siemens EDA, 2024

Rapid adoption of python in testbenches in recent years

Why Python?

- Productivity boost
  - Easy to learn
  - Faster reading, writing and debug
- Ecosystem
  - Huge collection of libraries, tools, frameworks
- Development
  - Code auto-completion integrated with editor (Jedi)

Why Python?



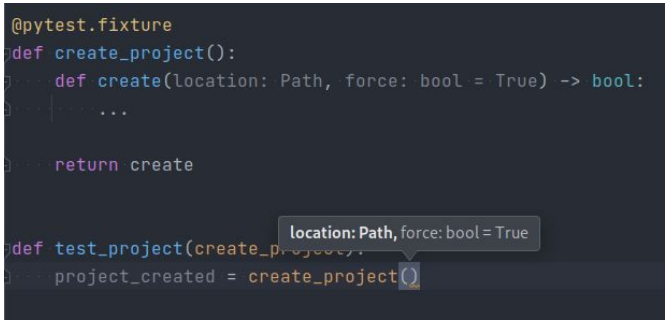
- Modern development ecosystem:
  - Dependency management (pip, uv)
  - Documentation generation (Sphinx)
  - Test frameworks (pytest)
  - Code quality: formatters, linters , auto-completion

WRITE LESS, VERIFY MORE !

Why Pytest?

```
tests/test_basic_example.py::test_square[1] PASSED
tests/test_basic_example.py::test_square[2] PASSED
tests/test_basic_example.py::test_square[3] PASSED

================================================== 3 passed in 0.01s ======
```

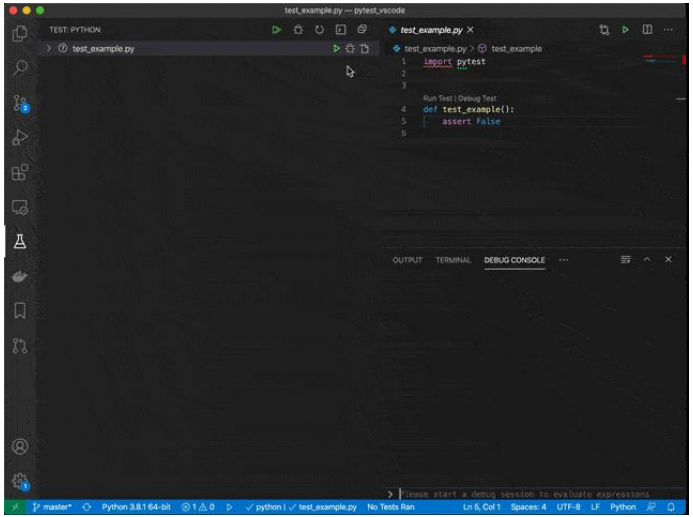
- Test framework
  - **Parametrization**: One test, multiple runs, reduce code redundancy
  - **Fixtures**: set up test prerequisites, reuse, teardown
  - **Parallelization**: distribute test across cores



- Test management (discovery, running, filtering)
  - Easy test discovery with **“`test_`”** functions (no explicit adding of tests)
  - Filter tests using strings, regex

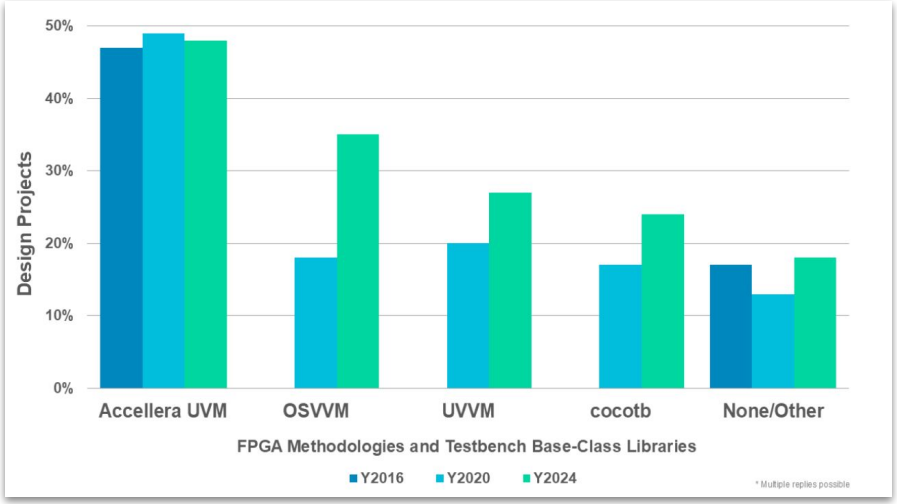
Why Pytest?

- Concise syntax of writing tests, easy to use
- Very well documented with good community support
- Easy to extend with plugin framework:
  - Rich free collection of existing external plugins (~1641)
  - Suit yourself, write own plugin! (integration)
- Integration with IDE and text editors (running tests)



Why Cocotb?

- Using python and pytest with all benefits in HDL verification targeting FPGA and ASIC
- Portability - running the same tests with different HDL simulators (open-source and proprietary)
- Open-source, maturity, community, well documented



Source : "2024 Siemens EDA and Wilson Research Group FPGA functional verification trend report," White Paper, Siemens EDA, 2024

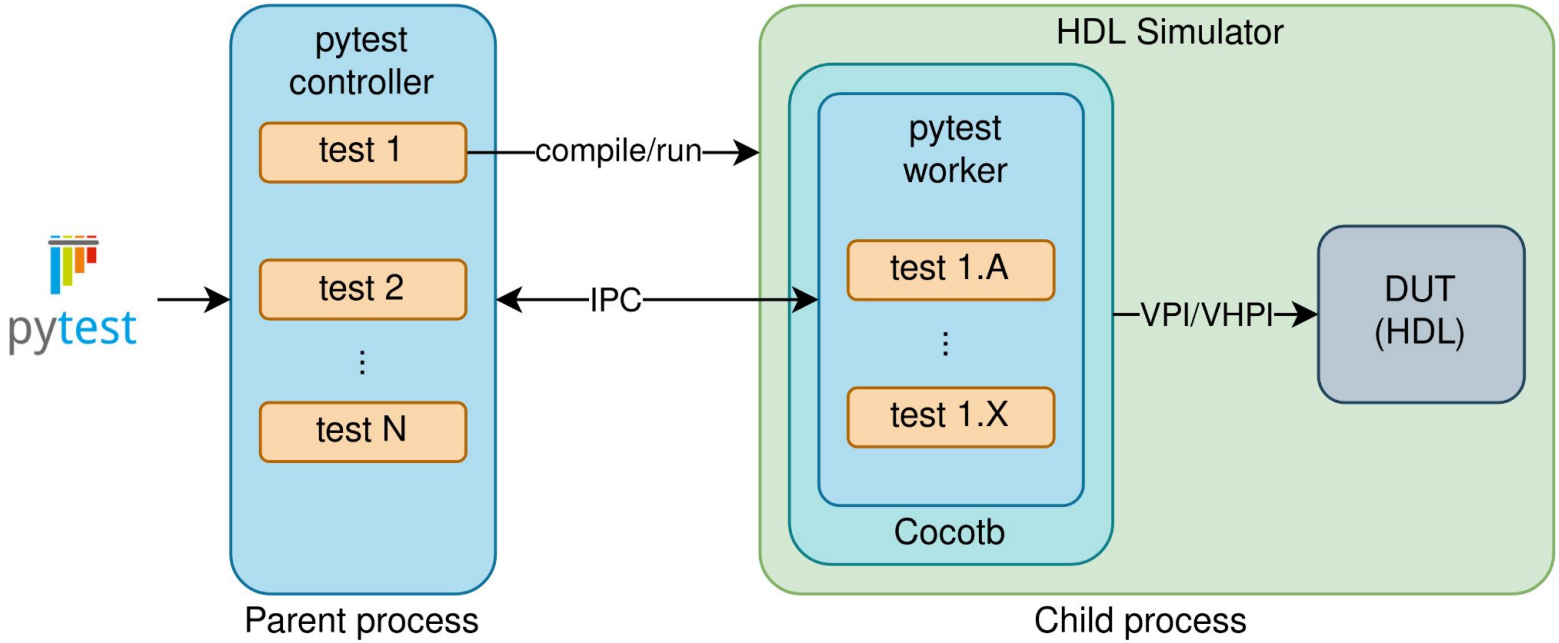
Cons? Any?

- Work in Progress

Cons? Any?

- Runtime performance
  - Slower (without HDL simulator optimization)
  - Example boost - Clock can be generated on HDL side.
- Debugging
  - Two separate debuggers (Python, HDL simulator)
  - Dynamic part of Python may not be aligned with HDL code

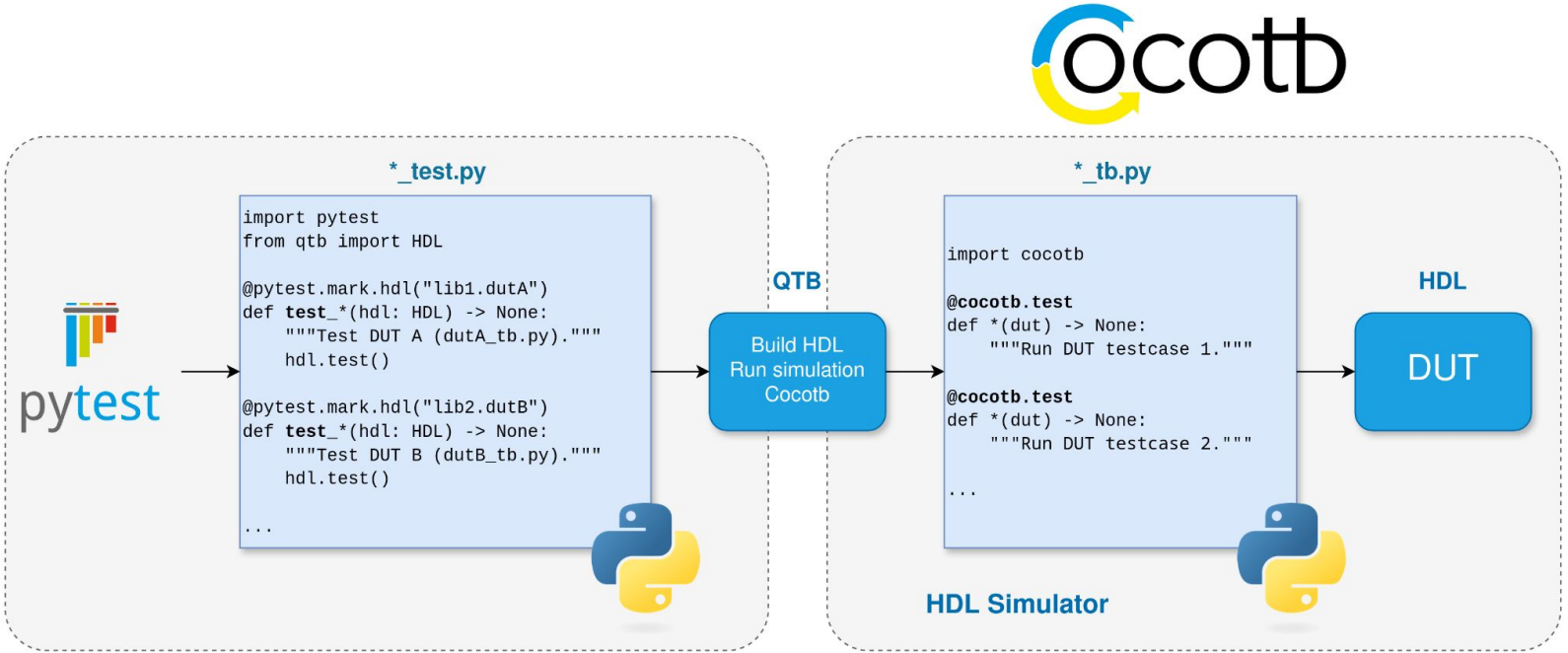
Pytest <-> Cocotb Architecture Overview



Pytest <-> Cocotb Architecture Overview

- Similar architecture to pytest-xdist
- Pytest functions called on pytest **controller** side can compile HDL project and run HDL simulator as separate subprocess
- Pytest functions called on pytest **worker** side will run from HDL simulator to verify DUT using Cocotb
- From user and pytest perspective, controller and worker are **transparent**
- Distinguished using pytest markers (**pytest.mark.***)

Pytest <-> Cocotb Architecture Overview

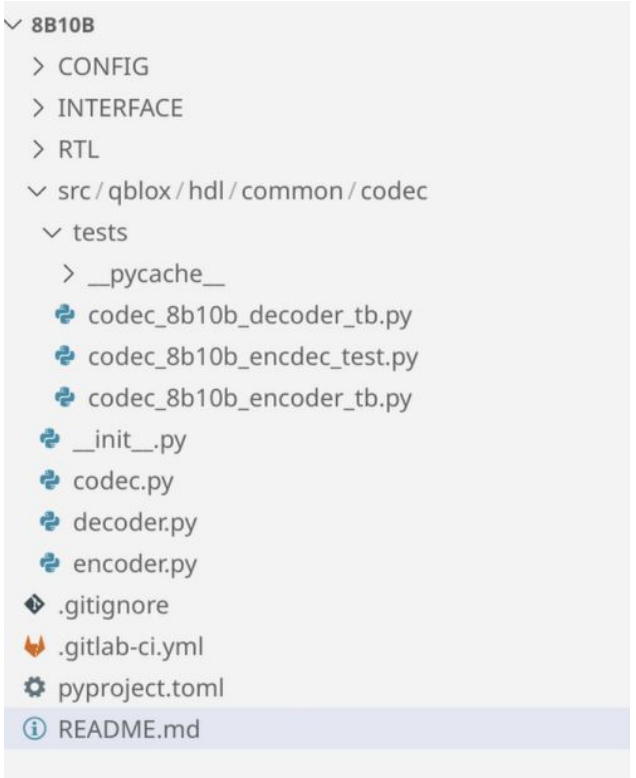


Hands-On: Cocotb and Pytest in Action

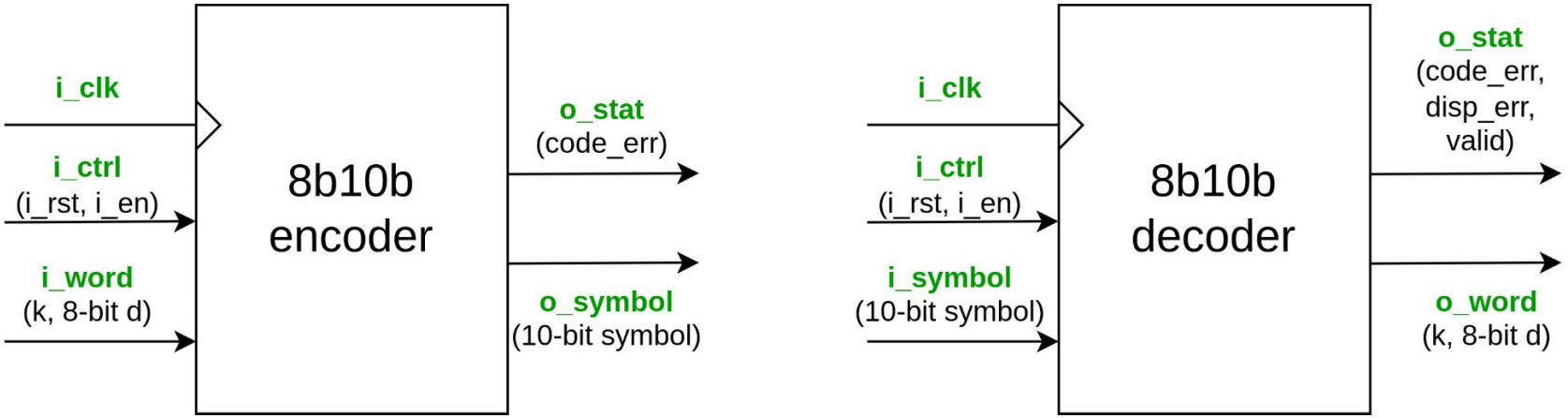
- **Agenda**
  - Will look at simple RTL designs -
    - 8b10b encoder/decoder
    - Pseudo-random binary sequence (PRBS) generator
  - Python project layout and writing a **cocotb** testbench, running with **pytest** framework
  - Test discovery, filtering, parametrization, plug-in capabilities, command line arguments
  - View reports, coverage results
  - Pytest in validation, integration with IDEs (VSCode)
  - Conclusion

Project layout, folder structure

- Working out-of-box with Python project in editable mode (development mode)
- Using python namespace packages (different HDL modules) into single common namespace:
  - `qblox/hdl/`
- Combining and merging distributed HDL modules from different HDL libraries into:
  - `qblox/hdl/<library>/<module>/`

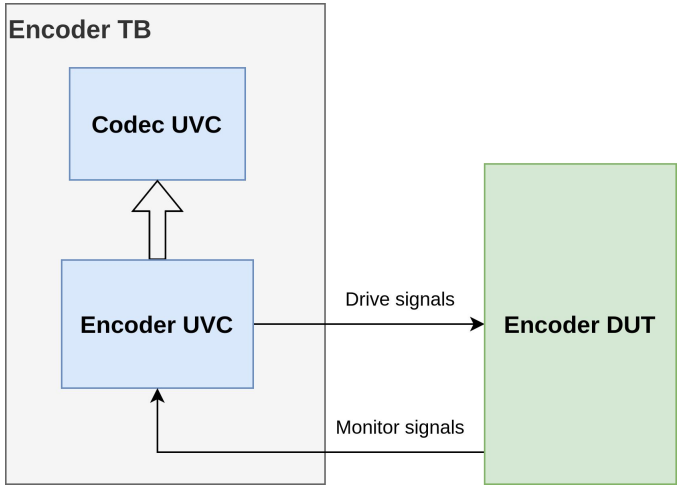


Design under Test (DUT) - 8b10b encoder/decoder



8b10b testbench structure

- First step in writing the testbench -
  - Universal Verification Component (UVCs) to create helper functions to drive and monitor
- `__init__.py` file initializes the top level package (required to make Python treat directories containing this file as packages)
- `codec.py` contains the base UVC (used in encoder and decoder TBs)



8b10b Codec UVCs

```
"""QVC for codec"""

from .encoder import Encoder
from .decoder import Decoder

# https://docs.astral.sh/ruff/rules/unused-import/
# https://docs.python.org/3/tutorial/modules.html#importing-from-a-package
__all__ = ["Encoder", "Decoder"]
```

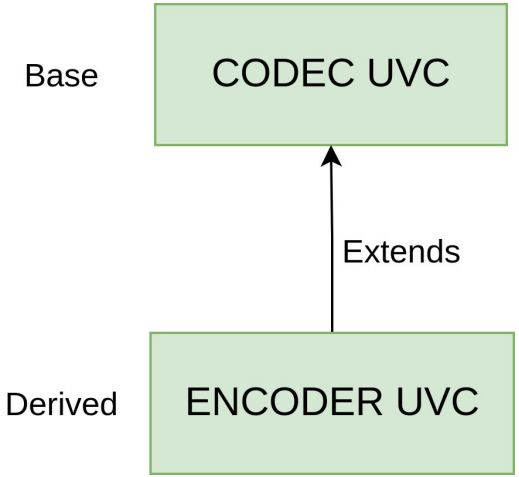
- `__init__.py` code defines a list named `__all__`
  - it is taken to be the list of module names that should be imported when from `package import *`
- In this example, does not contain any other code
  - Could also contain the UVC for the toplevel DUT

8b10b Codec UVCs

- `__init__ (self, dut)`: constructor function to pass DUT instance handle to UVC class
- **`input`** and **`output`** to drive and monitor DUT input/output signals
  - Set to None, can be customized according to Encoder/Decoder UVC.
- Can use standard python libraries like **encdec8b10b**

```python
class Codec:

    async def send(self, data: Union[int, list[int]]) -> None:
        """Send data to codec."""
        if not isinstance(data, Sequence):
            data = [data]

        while data:
            self.input.value = data[0]
            data = data[1:]
            await RisingEdge(self.dut.i_clk)

    async def receive(self, count: int) -> list[int]:
        """Receive data from codec."""
        data = count * [0]
        index = 0

        while index < count:
            await RisingEdge(self.dut.i_clk)
            data[index] = self.output.value.integer
            index += 1

        return data

    def encode(self, data_in, running_disparity=0, ctrl=0):
        """Encode 8b value"""
        # Returns disparity and encoded value
        return EncDec8B10B.enc_8b10b(
            data_in=data_in, running_disparity=running_disparity, ctrl=ctrl
        )

    def decode(self, data_in):
        """Decode 10b value"""
        # Returns control and decoded value
        return EncDec8B10B.dec_8b10b(data_in=data_in)
```

8b10b Encoder UVC




```python
class Encoder(Codec):
    """Codec 8b/10b encoder."""

    def __init__(self, dut, active: bool = False):
        """Create new instance of 8b/10b encoder.

        Args:
            initialize: If True, initialize all inputs of encoder. By default is False.
        """
        super().__init__(dut)

        self.input = dut.i_word.d
        self.output = dut.o_symbol

        if active:
            self.initialize()

    def initialize(self):
        """Initialize all inputs of encoder."""
        super().initialize()
        self.dut.i_word.k.value = 0
        self.dut.i_word.d.value = 0
```

8b10b testbench

- Test defined using `@cocotb.test` decorator
- Instantiate UVC, generate clock, drive reset sequence
- Use UVC functions or directly drive DUT signals in TB

```python
@cocotb.test()
async def simple_transfer_encoder(dut, count: int = 1024):
    """Test 8b/10b encoder using stream of bytes"""

    # Generate clock in the background
    generate_clock(clock=dut.i_clk)

    # Encoder QVC initialization
    encoder = Encoder(dut, active=True)

    # Reset encoder DUT
    await encoder.reset()

    # Enable encoder DUT
    encoder.enable = True

    # Running disparity
    disp = 0

    # Generate data of 0..255 + random bytes. This will allow to increase code coverage hits
    expected = list(range(256)) + [random.randint(0, 255) for _ in range(count)]
    expected = expected[:count]
    random.shuffle(expected)

    # Generate reference data for encoder output
    encoded_expected = len(expected) * [0]

    for index, value in enumerate(expected):
        disp, encoded_expected[index] = encoder.encode(value, disp)
```

8b10b testbench

- Test defined using `@cocotb.test` decorator
- Instantiate UVC, generate clock, drive reset sequence
- Use UVC functions or directly drive DUT signals in TB
- Finish test with checkers/assertions

```python
async def simple_transfer_encoder(dut, count: int = 1024):

    # One clock cycle delay between input and output of encoder, hence one extra receive
    tasks = (
        cocotb.start_soon(encoder.send(expected)),
        cocotb.start_soon(encoder.receive(len(expected) + 1)),
    )

    # Wait for tasks to finish
    await Combine(*tasks)
    encoded_captured = tasks[1].result()[1:]

    # Assert captured encoder output with expected encoded data
    assert (
        encoded_expected == encoded_captured
    ), "Captured encoded data is not equal to expected encoded data"
```

Checkers/assertions

```python
async def checker(self):
    """Checkers for module"""

    while True:
        await FallingEdge(self.dut.i_clk)

        # Write model for calculating expected outputs
        exp_output = 0x145

        # Compare against DUT values
        assert self.dut.o_status.value == exp_output, "Status not matching"
```

- Can define checkers inside the UVC
  - Async function running till test end
  - Can write cycle accurate models and check signals at every clock cycle using assertions
- Can also define assertions/checkers directly in the TB

Writing pytest for testbench (fixtures, markers)

- Usage of **HDL** fixture in our pytest plugin (**QTB**)
- Contains information about HDL environment
  - HDL source files, libraries, HDL define macros, information about HDL simulator
- Usage of markers to provide metadata information to tests

```python
"""Pytest for 8b10b encoder decoder"""

import pytest
from qtb import HDL


@pytest.mark.hdl("common_lib.codec_8b10b_encoder")
def test_encoder(hdl: HDL):
    """Test 8b10b encoder DUT"""
    hdl.test()


@pytest.mark.hdl("common_lib.codec_8b10b_decoder")
def test_decoder(hdl: HDL):
    """Test 8b10b decoder DUT"""
    hdl.test()
```

Writing pytest for testbench

- Our pytest plugin (**QTB**) provides own predefined list of markers
  - `@pytest.mark.hdl()` allows to pass additional metadata to the QTB fixture **hdl**
- `@pytest.mark.parametrize(argnames, argvalues)`
  - Defined args used to parameterize test function.

```python
"""Pytest for 8b10b encoder decoder"""

import pytest
from qtb import HDL


@pytest.mark.hdl("common_lib.codec_8b10b_encoder")
def test_encoder(hdl: HDL):
    """Test 8b10b encoder DUT"""
    hdl.test()


@pytest.mark.hdl("common_lib.codec_8b10b_decoder")
def test_decoder(hdl: HDL):
    """Test 8b10b decoder DUT"""
    hdl.test()
```

Tests discovery

- The pytest framework will automatically discover all tests based on Python file names and Python function names.
- Python **file** name, by default, **MUST** start from the `test_*` prefix or end with the `*_test.py` suffix
  - `codec_8b10b_encdec_test.py` - Pytest that will build HDL and run Cocotb testbench
  - `codec_8b10b_encoder_tb.py` - Cocotb testbench executed from HDL simulator
- Python **function** must start with `test_*` prefix

Tests discovery

- To collect all tests
  - **`pytest --collect-only`**
  - **`pytest --co`** (shorthand)
- **`pytest`** searches for tests everywhere in current working directory
  - Unless defined in **`pyproject.toml`** or using cmdline args

```
$ pytest --collect-only

Setup Python virtual environment

Load HDL design from dflow
============================================================================================
platform linux -- Python 3.12.5, pytest-8.4.1, pluggy-1.6.0
rootdir: /home/abhiroop/qblox/projects/standalone/fpga/HDL_Design/common_lib/8b10b
configfile: pyproject.toml
testpaths: src/qblox/hdl/common/codec/tests
plugins: qtb-0.2.3
collected 12 items

<Dir 8b10b>
  <Dir src>
    <Dir qblox>
      <Dir hdl>
        <Dir common>
          <Package codec>
            <Dir tests>
              <Module codec_8b10b_encdec_test.py>
                <Runner test_encoder>
                  <Testbench codec_8b10b_encoder_tb.py>
                    <Test simple_transfer_encoder>
                    <Test simple_transfer_encoder[count:1]>
                    <Test simple_transfer_encoder[count:16]>
                    <Test simple_transfer_encoder[count:1024]>
                    <Test simple_transfer_encoder[count:2048]>
                    <Test simple_transfer_encoder[count:4096]>
                <Runner test_decoder>
                  <Testbench codec_8b10b_decoder_tb.py>
                    <Test simple_transfer_decoder>
                    <Test simple_transfer_decoder[count:1]>
                    <Test simple_transfer_decoder[count:16]>
                    <Test simple_transfer_decoder[count:1024]>
                    <Test simple_transfer_decoder[count:2048]>
                    <Test simple_transfer_decoder[count:4096]>
```

Test filtering and selection

- Include or exclude tests based on their names and markers
  - Using `-k EXPRESSION` cmdline option. By default, pytest utility will run all tests.
- To run specific test(s) that will match provided expression:
  - `pytest -k <test-name>`
- With `or` statement:
  - `pytest -k '<test-name-1> or <test-name-2>'`
- With `and` and `not` statements:
  - `pytest -k '<test-name-1> and not <test-name-2>'`
- Use the `--collect-only` command line option to list all available names to be used later with `-k`:
  - `pytest --collect-only --quiet`

Test filtering and selection

- `pytest -s -k 'test_encoder and simple_transfer_encoder[count:1]'`

```
Test: src/qblox/hdl/common/codec/tests/codec_8b10b_encoder_tb.py::simple_transfer_encoder[count:1]
     0.00ns INFO     cocotb.regression                  running simple_transfer_encoder_001 (1/1)
                                                          Automatically generated test

                                                          count: 1
    20.00ns INFO     cocotb.regression                  simple_transfer_encoder_001 passed
.
Summary
    20.00ns INFO     cocotb.regression                  ********************************************************************************************************
                                                        ** TEST                                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                                                        ********************************************************************************************************
                                                        ** codec_8b10b_encoder_tb.simple_transfer_encoder_001   PASS          20.00           0.00      13148.95  **
                                                        ********************************************************************************************************
                                                        ** TESTS=1 PASS=1 FAIL=0 SKIP=0                                20.00           0.16        122.06  **
                                                        ********************************************************************************************************
```

Test logging

- pytest framework can colorize logs
  - add various colorful sections to increase readability
  - lot of options to configure Python logging from configuration file or command line.
- By default, pytest hides standard output from tests
  - It will show it only for failed tests. To disable it:

```
pytest --capture=no
pytest -s (shorthand)
```

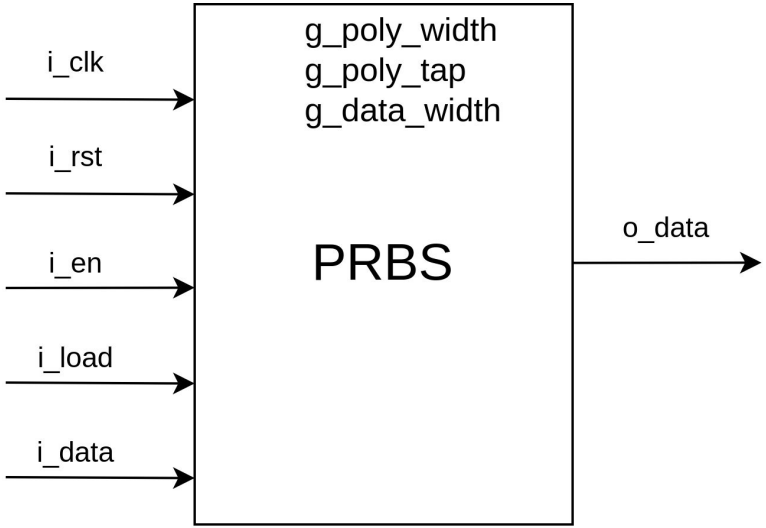
Test logging

```
**************************************************************************************************************
** TEST                                                STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
**************************************************************************************************************
** codec_8b10b_decoder_tb.simple_transfer_decoder       PASS        4112.00           0.17      24299.91  **
** codec_8b10b_decoder_tb.simple_transfer_decoder_001   PASS          24.00           0.00      16845.30  **
** codec_8b10b_decoder_tb.simple_transfer_decoder_002   PASS          84.00           0.00      23980.79  **
** codec_8b10b_decoder_tb.simple_transfer_decoder_003   PASS        4116.00           0.17      24858.22  **
** codec_8b10b_decoder_tb.simple_transfer_decoder_004   PASS        8212.00           0.33      24733.75  **
** codec_8b10b_decoder_tb.simple_transfer_decoder_005   PASS       16404.00           0.68      24156.70  **
**************************************************************************************************************
** TESTS=6 PASS=6 FAIL=0 SKIP=0                                   32952.01           1.59      20765.16  **
**************************************************************************************************************
```

- Pytest log with standard output enabled
- Log files stored in **`sim`** directory, using standard python logging
- Each test result generated under name of TB combined with test name

Test parameterization

- Let’s take another example design - Pseudo Random Binary Sequence(**PRBS**) generator
- Usage of LFSR, based on standard polynomial (defined by generics **`g_poly_width`**, **`g_poly_tap`**), with parallel output (based on **`g_data_width`**)



Test parameterization

- Running different generics with the same test using pytest parametrize marker

```python
# Parametrizing different core widths and taps
@pytest.mark.hdl("common_lib.prbs")
@pytest.mark.parametrize(
    "g_poly_width,g_poly_tap", [(9, 5), (15, 14), (23, 18), (31, 28)]
)
def test_prbs_parametrized(hdl: HDL, g_poly_width, g_poly_tap):
    """Test PRBS with different cores"""
    hdl["g_poly_width"] = g_poly_width
    hdl["g_poly_tap"] = g_poly_tap
    hdl.test()
```

Pytest command line arguments

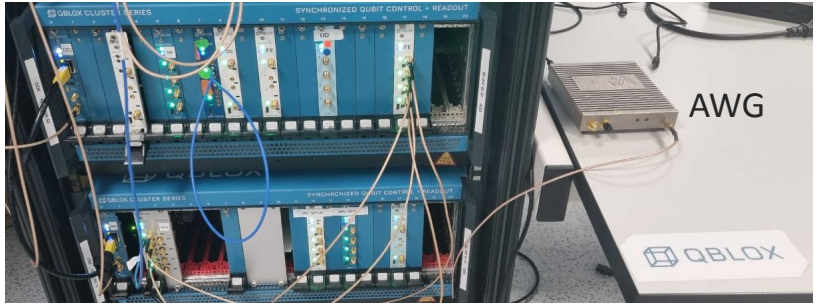
- Additional CLI arguments can be added using pytest_addoption
- Pytest is using standard Python built-in argparse module
- Only long names with double dashes -- are supported
- They will be part of pytest --help output
- They can be used in fixtures and tests
- Examples:
  - **`--gui`** to start HDL simulator with GUI
  - **`--hdl-coverage`** to enable HDL code coverage

Pytest plugin capabilities - pyloops

- Pytest plugin example for loops: https://pypi.org/project/pytest-loop
- Use the `--loop` command line option: `pytest --loop=10 test_file.py`
- Decorator:

```
import pytest
@pytest.mark.test_creation
@pytest.mark.loop(25 if PYLOOPS_ENABLED else 0)
def test_scope_acq_inp_cal(my_dut, awg):
    Pass
```

Pytest for validation



Validation v/s verification tests:

- Validation relies on availability of test equipment. Simulation resources are always present
- Functional checks performed in validation regression, similar to verification
- Stress-test a design until its limits. (temperature cycling, corner cases)
- Real-time tests

The Need for Dynamic Test Control in validation:

- Dynamic control over test variables and parameters
- "Knobs" for specific thresholds, edge cases, or critical events - pinpoint bugs

Pytest empowering validation

Parameterized Fixtures (`pytest.fixture(params=...)`): Running test with diverse input data or configurations

```python
@pytest.fixture (scope="session")
def my_dut():
    return My_qblox_module(IP = "101.101.101.101", slot = "2")

@pytest.fixture(scope="session")
def signal_generator():
    """Setup signal generator, then close."""

    print("\n--- Setting up signal generator connection---")
    signal_gen = set_sig_gen.SignalGeneratorDriver('101.101.101.101', 2020)
    signal_gen.set_power_state(0)

    yield signal_gen  # The object is given to the test.

    print("--- Tearing down signal generator connection ---")
    signal_gen.set_power_state(0)
    signal_gen.close()  # Cleanup, close connection, etc.
```

SETUP

YIELD

TEARDOWN

Pytest empowering validation

- **Pytest Hooks:** Allows reading external setup files (JSON/TOML).
- **Indirect Parametrization:** Similar to verification
- **Interactive Mode (via Custom Hooks):** Facilitates real-time debugging and immediate feedback.
  - Example: A custom hook can automatically drop the tester into a Python Debugger.

VS code integration with pytest

Benefits of Integration:

- Run tests in the editor.
- Breakpoints, step through code, inspect variables.
- Visual summary of test results.
- Test-driven development workflow.

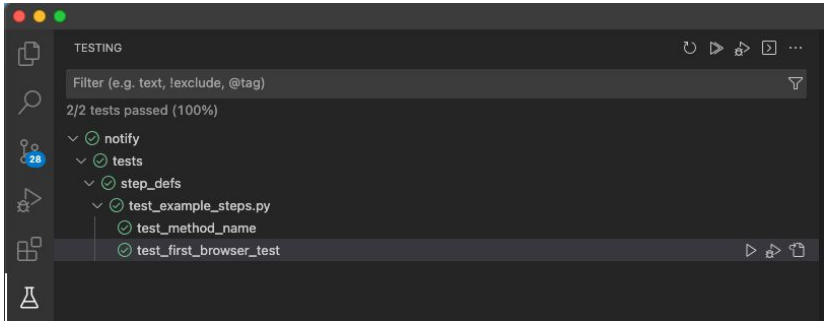
Easy to enable pytest:

- `pip install pytest`
- Open Command Palette: Ctrl+Shift+P
- Search for Python: Configure Tests
- Select pytest as your test framework.
- Specify Test Folder

VS code integration with pytest

Test Explorer View:

- Click the "Testing" icon in activity bar.
- See all your tests listed.
- Run test or multiple tests with a single click.
- "Debug Test" icon to start debugging.



Conclusion

- Pytest framework - unified testing approach
- Standardized but still customizable
- No need to reinvent the wheel
- Future work - pytest framework, plugins can be standardized to be used with HDL testing, and made open-source

Thank you

Questions?