

2024 IEEE NPSS Real Time Conference

Workshop on open-source tools for FPGA and ASIC design

Introduction to Cocotb

Marc-André Tétrault, Eng., PhD.

20/04/2024

The logo consists of the characters '3!T' in a bold, white, sans-serif font. The '3' is a simple numeral, the '!' is a standard exclamation point, and the 'T' is a simple capital letter. The logo is positioned in the top-left corner of the slide, partially overlapping a decorative green abstract graphic that resembles a network or data stream.

https://github.com/mtetrault/RT2024_CocotbWorkshopFiles.git

Please « git clone » these files inside the virtual box machine

Includes a pdf file with step-by-step lab instructions.

RT2024 Workshop

Coding block

- 1- Cocotb
- 2- Surf

Hardware block

- 1- FABulous eFPGAs
- 2- Open-source ASIC Design (openLANE, Skywater, Caravel)

Cocotb

Coroutine cosimulation testbench

Overview in 2018 by **Benjamin John Rosser**

<https://indico.cern.ch/event/776422/>

Uses Python instead of HDL/tailored language

- Python widely used by students and scientists
- Access to Python packages
- Object Oriented Programming support
- ...

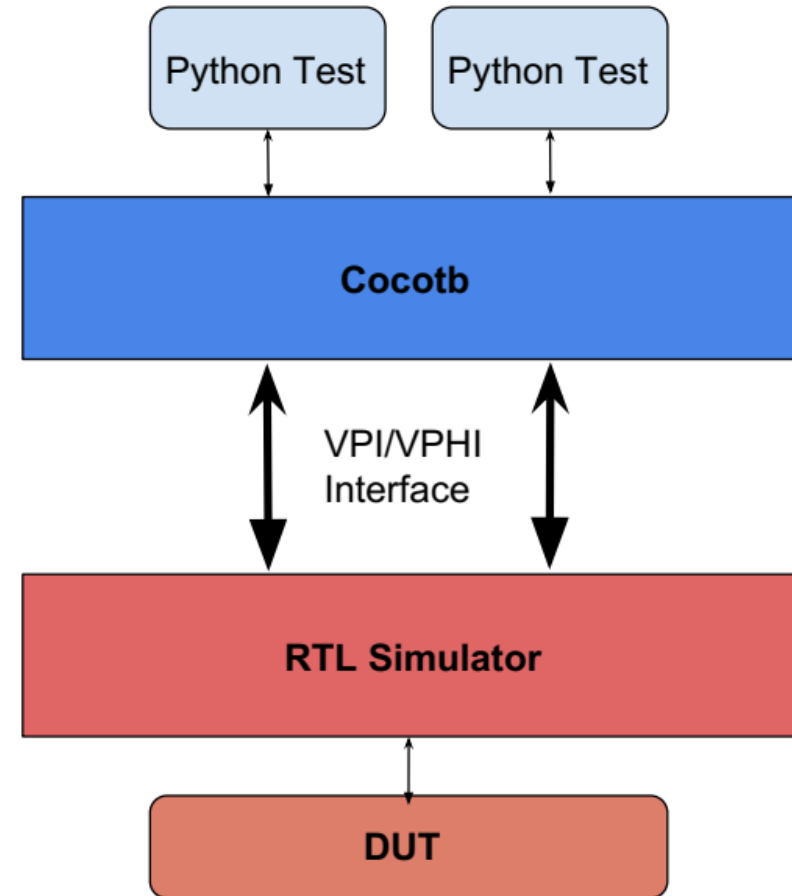
Cocotb

Many updates to the package/syntax since 2018

- Changed Makefile approach to pure Python
- Changed Python keywords to manage asynchronous behaviour
- New features merged as community resources allow
- Open-source

Cocotb – Link with simulator

- Simulator compiles HDL
- Simulator is main thread
- Simulator imports Python code
- Flow control is swapped between simulator and Python code.



From <https://indico.cern.ch/event/776422/>

Supported simulators

Commercial

Incisive/Xcellium
Questa/Modelsim
VCS (Synopsys)
Xsim (Vivado)

Open-source

Verilator
GHDL
Icarus Verilog
...

More at https://en.wikipedia.org/wiki/List_of_HDL_simulators

Opensource vs commercial simulators

Commercial

FPGA editions free
Mixed language
Bug support
Advanced features (SVA)
Includes Waveform viewer

Large designs might lock down performance

Open-source

Single language
No design size limit
Community support
External waveform viewer
Some HDL features not supported

Survey – Participant experience?

How often have you:

- Used a Linux and command line terminal?
- Used Python?
- Used git (to download examples)?
- Used HDL simulators (Vivado, Modelsim, etc)?
- Used VHDL or Verilog?

Cocotb - Workshop

Workshop relies on basic experience

Goal: provide a first contact with the Cocotb module and use flow.

Uses VHDL, with the GHDL simulator

Uses GtkWave to view waveforms

Basic text editors (nano, vim, gedit) for file edition

VSCodium for interactive debugging

3!T

Lab 1

First contact

Lab 1

Objectives

- Launch a cocotb simulation.
- Add command line arguments to the underlying simulator (ghdl in this case).
- View waveforms using gtkwave.
- Learn from error messages

Lab 1 - Design

Simple adder from the official cocotb git repository (v1.8)

`cocotb/example/adder`

Three files:

- Simple HDL Adder
- Python model (addition)
- Cocotb test and runner
 - * first part is testbench
 - * second part is runner/simulator call

Lab 1 - Design

Changes and simplifications

- Removed simulator configurability: GHDL simulator
- Removed language configurability : VHDL only
- Simplified options
- Added comments

Lab 1 - Design

```
0.00ns INFO cocotb.regression Found test test_adder_solution.adder_randomised_test
0.00ns INFO cocotb.regression running adder_randomised_test (1/1)
Test for adding 2 random numbers multiple times
./../src/openieee/v93/numeric_std-body.vhdl:398:9:@0ms:(assertion warning): NUMERIC_STD."+": non logical value detected
20.00ns INFO cocotb.regression adder_randomised_test passed
20.00ns INFO cocotb.regression
*****
** TEST                               STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
*****
** test_adder_solution.adder_randomised_test  PASS  20.00         0.01         2972.15 **
*****
** TESTS=1 PASS=1 FAIL=0 SKIP=0          20.00         1.52         13.14 **
*****
```

Lab 1 - GtkWave

The screenshot shows the GtkWave application window. The title bar indicates the file path: `/home/mtetrault/Documents/Documents/Repos/RT2024_CocotbWorkshop/Lab01/solution/sim_build/Lab01_waveforms.vcd`. The menu bar includes File, Edit, Search, Time, Markers, View, and Help. The toolbar contains icons for file operations and navigation, with a search icon circled in red. The main interface is divided into three panels:

- Left Panel:** A tree view showing the project structure. The `adder` component is selected and circled in red.
- Middle Panel:** A list of signals. The signals `a[3:0]`, `b[3:0]`, and `x[4:0]` are listed, with `x[4:0]` selected and circled in red.
- Right Panel:** A waveform display showing a table of signal values over time. The time scale is set to 10 ns. The table contains the following data:

Signal	1	F	C	E	4	7	C	A	2
E	1	F	C	E	4	7	C	A	2
B	4	A	5	6	5	C	6	5	
x	19	05	19	11	14	09	0C	18	10

At the bottom of the window, there is a Filter input field and buttons for Append, Insert, and Replace.

3!T

Lab 2

Customizing a cocotb template

Lab 2

Objectives

- Write your first customized Cocotb runner.
- Write your first Cocotb test.
- Automate verification (confirm design function without waveforms)

Lab 2 - Design

Square root arithmetic core from

<https://vhdlguru.blogspot.com/2020/12/synthesizable-clocked-square-root.html>

Two files:

- Square root arithmetic core (provided)
- Cocotb test and runner
 - * first part is testbench (edit second)
 - * second part is runner/simulator call (edit first)

Lab 2 - Design

Interface: clk, reset, input interface, output interface

Square Root Core, 32-bit integer input

clk

reset

arg_valid

sqrt_valid

arg(31:0)

sqrt_res(15:0)

Lab 2 - Work

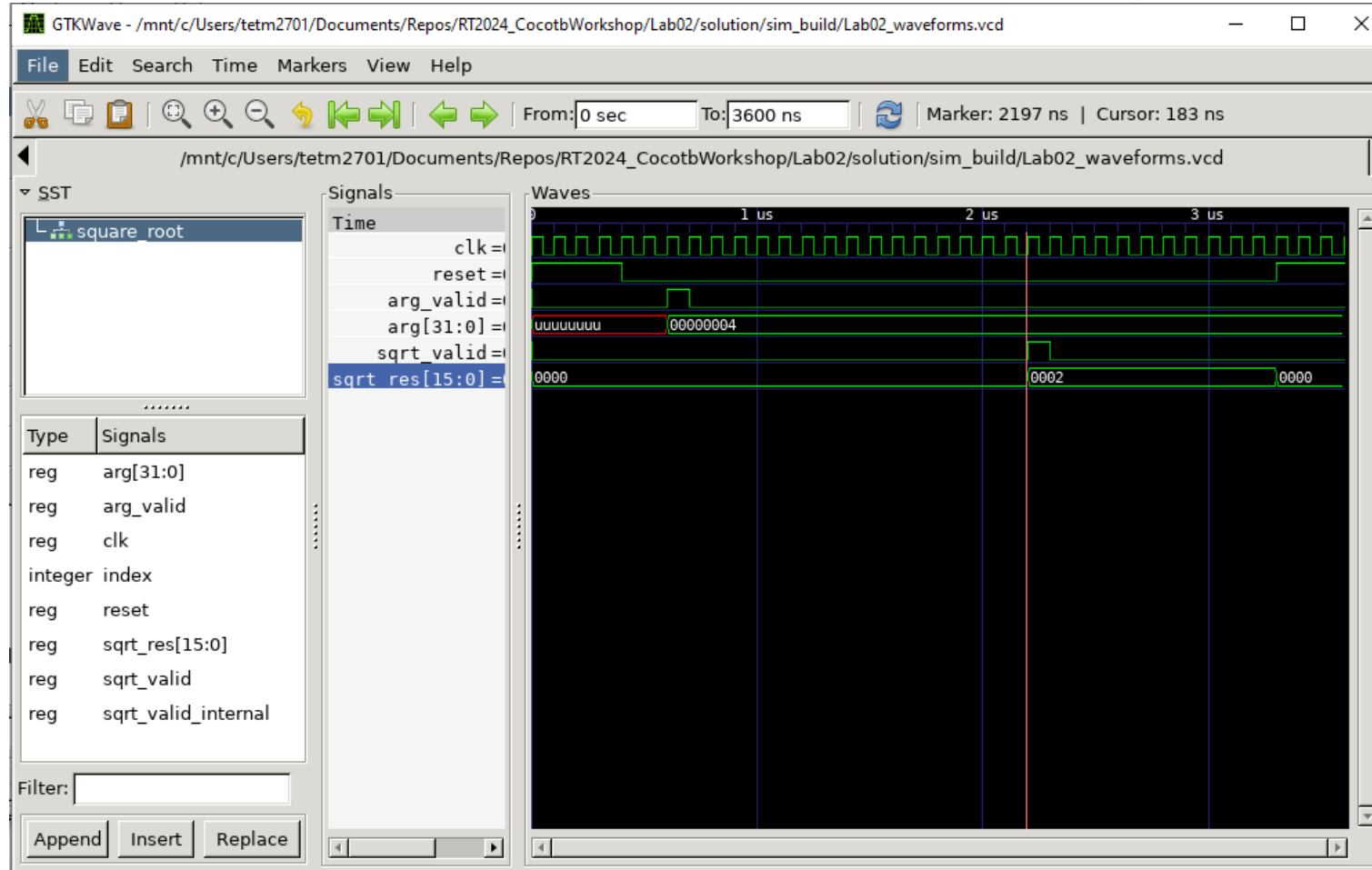
- Modify the runner to match the provided design
- Learn Cocotb relevant Python keywords
- Add very simple test for arithmetic core

Note: Python “async” and “await” keywords (yield in older versions)

Use Snippets File!

- Faster than google 😊

Lab 2 – Expected outcome



Lab 2 – GtkWave Tip

The screenshot shows the GtkWave application window. The title bar reads "GTKWave - /mnt/c/Users/tetm2701/Documents/Repos/RT2024_CocotbWorkshop/Lab02/solution/sim_build/Lab02_waveforms.vcd". The menu bar includes File, Edit, Search, Time, Markers, View, and Help. The 'File' menu is open, listing various actions such as "Open New Window", "Open New Tab", "Reload Waveform", "Export", "Close", "Print To File", "Grab To File", "Read Save File", "Write Save File", "Write Save File As", "Read Sim Logfile", "Read Verilog Stemsfile", "Read Tcl Script File", and "Quit". The "Reload Waveform" option is highlighted with a red circle. The main window displays a digital logic simulation waveform with signals like clk, reset, valid, and data buses. The time scale is set to 1 us, 2 us, and 3 us. The waveform shows a clock signal (clk) and a reset signal (reset). The valid signal (valid) is shown as a pulse. The data bus [31:0] is shown as a sequence of bits, and the data bus [15:0] is shown as a sequence of bits.

3!T

Lab 3

Interactive debugging

Lab 3 - Debugging

Error messages may come from

- Starting the simulator (like in lab 1)
- HDL compile errors
- Python syntax (during execution)
- Testbench assertion failures

Using « print » functions is very inefficient

Lab 3 – Why not directly from a GUI?

With Cocotb, the simulator calls Python.

Need to add a hook in Cocotb tests, where the IDE can connect.

Supported in PyCharm Pro, but not PyCharm Community

<https://blog.patfarley.org/pages/cocotb-pycharm.html>

Supported in VSCode/VSCodium (free)

Lab 3 - Objectives

- Use an IDE to graphically debug a cocotb test.
 - Configure the cocotb test to support debug.
 - Configure VSCodium to attach to the cocotb test.
 - Add break points in the cocotb test.
 - Inspect variables and dut signals within the IDE.

Lab 3 – Test project

Copy of solution from Lab 2, already provided

```
clk
```

```
reset
```

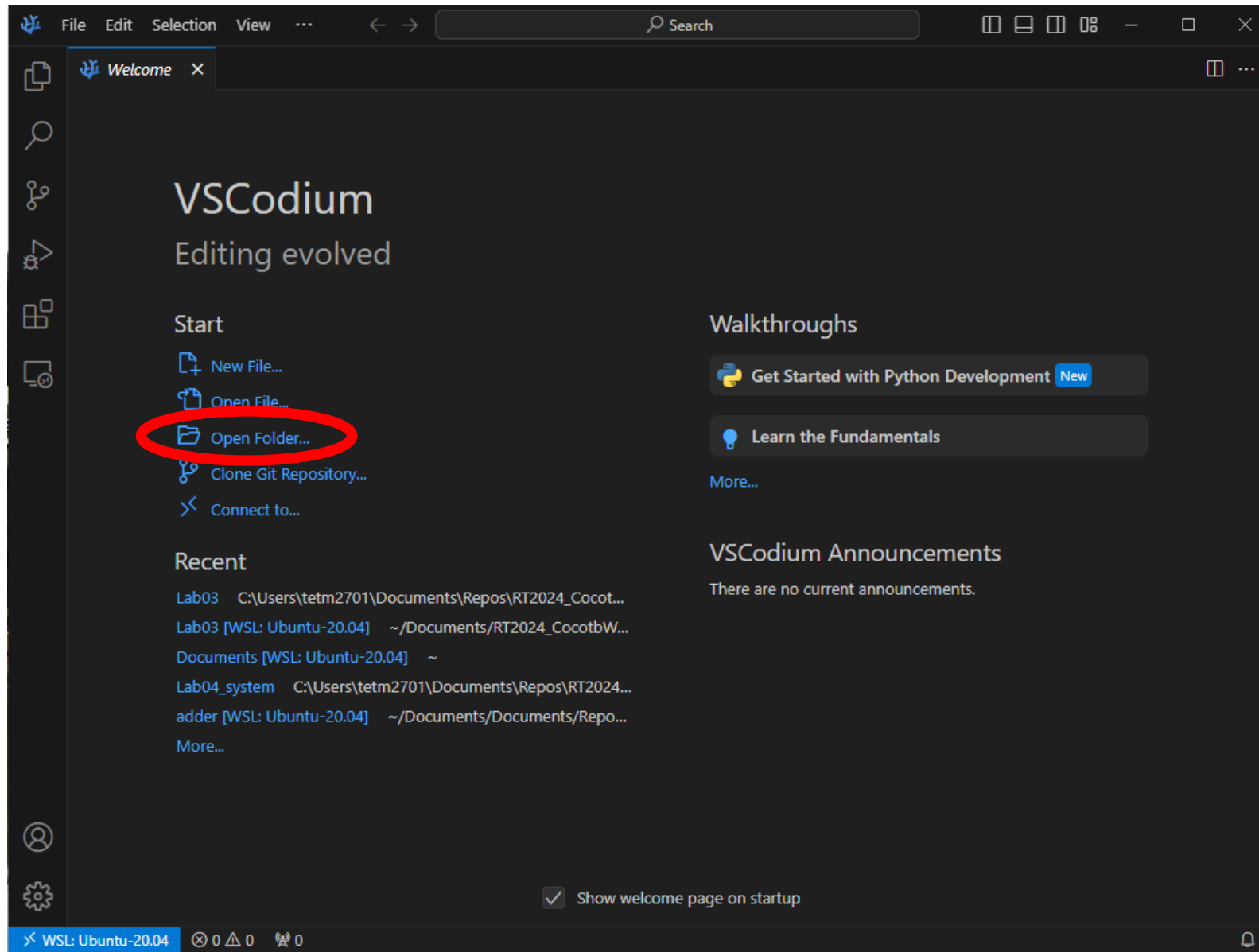
```
arg_valid
```

```
sqrt_valid
```

```
arg(31:0)
```

```
sqrt_res(15:0)
```

Lab 3 – Visual steps



Lab 3 – Visual steps

The screenshot shows the Visual Studio Code (VS Code) interface. The 'Run and Debug' menu is open, showing the 'Python Debugger' option, which is circled in red. Below the menu, there are two red circles: one around the 'Run and Debug' button and another around the text 'to customize Run and Debug create a launch.json file.' The main editor displays Python code for a cocotb test. The terminal window at the bottom shows the command prompt.

to customize Run and Debug create a launch.json file.

Show all automatic debug configurations.

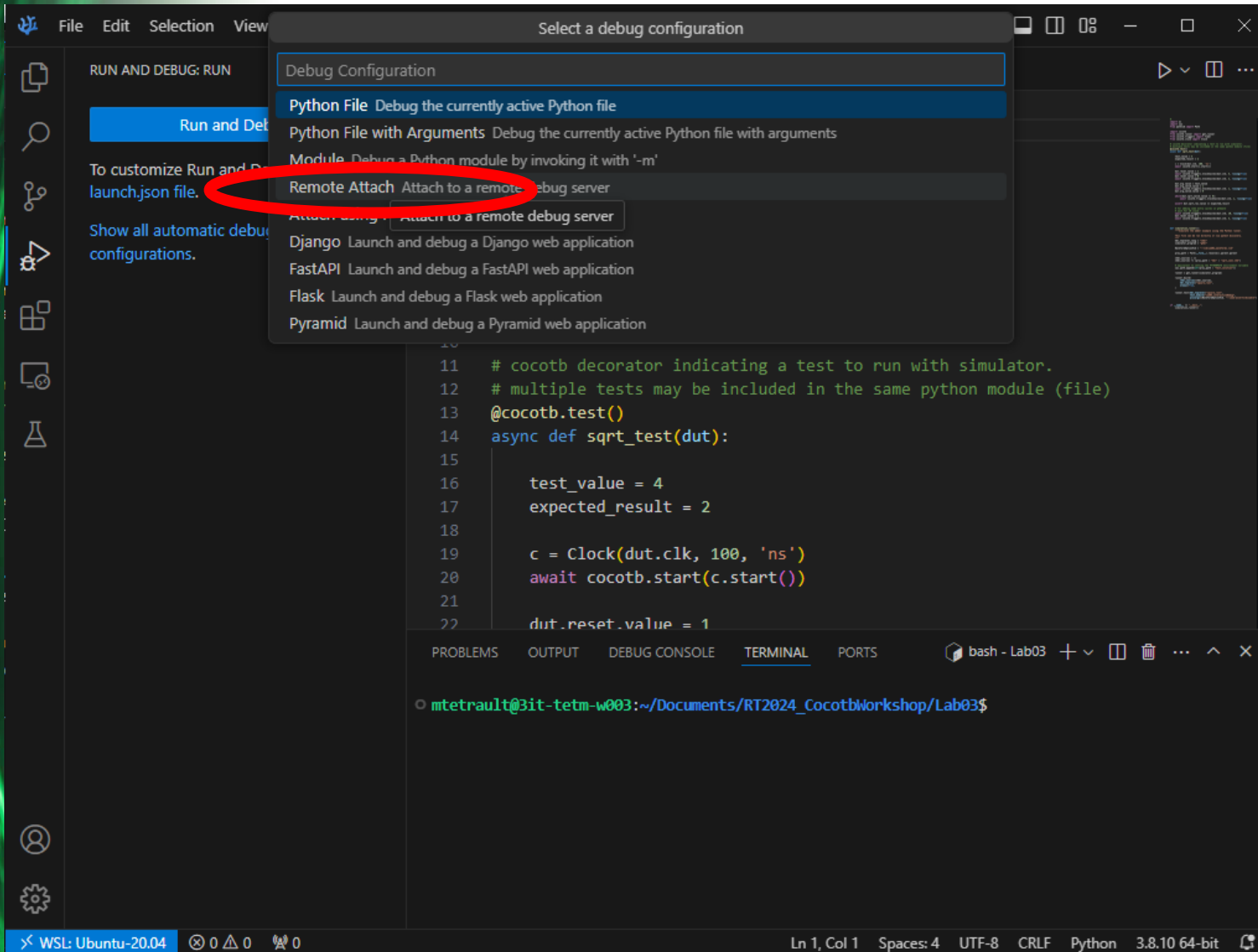
```
1 #
2 import os
3 import sys
4 from pathlib import Path
5
6 import cocotb
7 from cocotb.runner import get_runner
8 from cocotb.triggers import Timer
9 from cocotb.clock import Clock
10
11 # cocotb decorator indicating a test to run with simulator.
12 # multiple tests may be included in the same python module (file)
13 @cocotb.test()
14 async def sqrt_test(dut):
15
16     test_value = 4
17     expected_result = 2
18
19     c = Clock(dut.clk, 100, 'ns')
20     await cocotb.start(c.start())
21
22     dut.reset_value = 1
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS bash - Lab03

mtetrault@Bit-tetm-w003:~/Documents/RT2024_CocotbWorkshop/Lab03\$

WSL: Ubuntu-20.04 0 0 0 Ln 1, Col 1 Spaces: 4 UTF-8 CRLF Python 3.8.10 64-bit

Lab 3 – Visual steps



And then press
« enter » twice for the
server name and port

- localhost
- 5678

Lab 3 – Visual steps

The screenshot displays the Visual Studio Code interface with a Python launch configuration file named `launch.json` open. The `launch.json` file contains the following configuration:

```
1 // Use IntelliSense to learn about possible attributes.
2 // Hover to view descriptions of existing attributes.
3 // For more information, visit: https://go.microsoft.com/fwlink/?l
4 "version": "0.2.0",
5 "configurations": [
6   {
7     "name": "Python Debugger: Remote Attach",
8     "type": "debugpy",
9     "request": "attach",
10    "connect": {
11      "host": "localhost",
12      "port": 5678
13    },
14    "pathMappings": [
15      {
16        "localRoot": "${workspaceFolder}",
17        "remoteRoot": "."
18      }
19    ]
20  }
21 ]
```

The interface also shows the `DEBUG CONSOLE` and `TERMINAL` panels. The terminal window shows the command prompt for a bash shell in a WSL environment:

```
mtetrault@3it-tetm-w003:~/Documents/RT2024_Cocotbworkshop/Lab03$
```

The `DEBUG CONSOLE` panel is currently empty. The `TERMINAL` panel shows the current directory and the prompt.

Lab 3 – Visual steps

The screenshot displays a Python IDE interface for debugging. The main editor shows a Python script with the following code:

```
14 # multiple tests may be included in the same python module (file)
15 @cocotb.test()
16 async def sqrt_test(dut):
17     debugpy.listen(5678)
18     debugpy.wait_for_client()
19     debugpy.breakpoint()
20
21     test_value = 4
22     expected_result = 2
23
24     c = Clock(dut.clk, 100, 'ns')
25     await cocotb.start(c.start())
26
27     dut.reset.value = 1
28     await cocotb.triggers.ClockCycles(dut.clk, 5, rising=True)
29     dut.reset.value = 0
30     await cocotb.triggers.ClockCycles(dut.clk, 2, rising=True)
31
32     dut.arg.value = test_value
33     dut.arg_valid.value = 1
34     await cocotb.triggers.ClockCycles(dut.clk, 1, rising=True)
35     dut.arg_valid.value = 0
36
```

The IDE interface includes several panels:

- Variables:** Shows local variables like `dut: HierarchyObject(square_root)` and global variables.
- WATCH:** A panel for monitoring variable values during execution.
- CALL STACK:** Shows the current execution context, including `sqrt_test` and `Lab03_interactiveDebug...`.
- BREAKPOINTS:** Lists active breakpoints, such as `Lab03_interactiveDebug_solutio...` at line 24 and 27.
- DEBUG CONSOLE:** A panel for viewing debug output and messages.

Red circles highlight the debugger controls: one circle is around the top toolbar (run, step, etc.) and another is around the breakpoint markers in the left margin of the code editor.

At the bottom of the IDE, the status bar shows: `WSL: Ubuntu-20.04`, `0` errors, `0` warnings, `0` messages, `Python Debugger: Remote Attach (Lab03)`, `Ln 21, Col 1`, `Spaces: 4`, `Python`, `3.8.10 64-bit`.

3!T

Lab 4

Code reuse 1 - functions and drivers

Lab 4

Custom designs → custom testbench

Some parts are standard, like busses. Some are simple (UART), others more detailed (PCIe).

They might already exist somewhere in another project...?

Lab 4 – Cocotb extensions

Mainly bus/communication protocols

Many available through python/pip3: ethernet, spi, uart, ...

<https://pypi.org/search/?q=cocotbext>

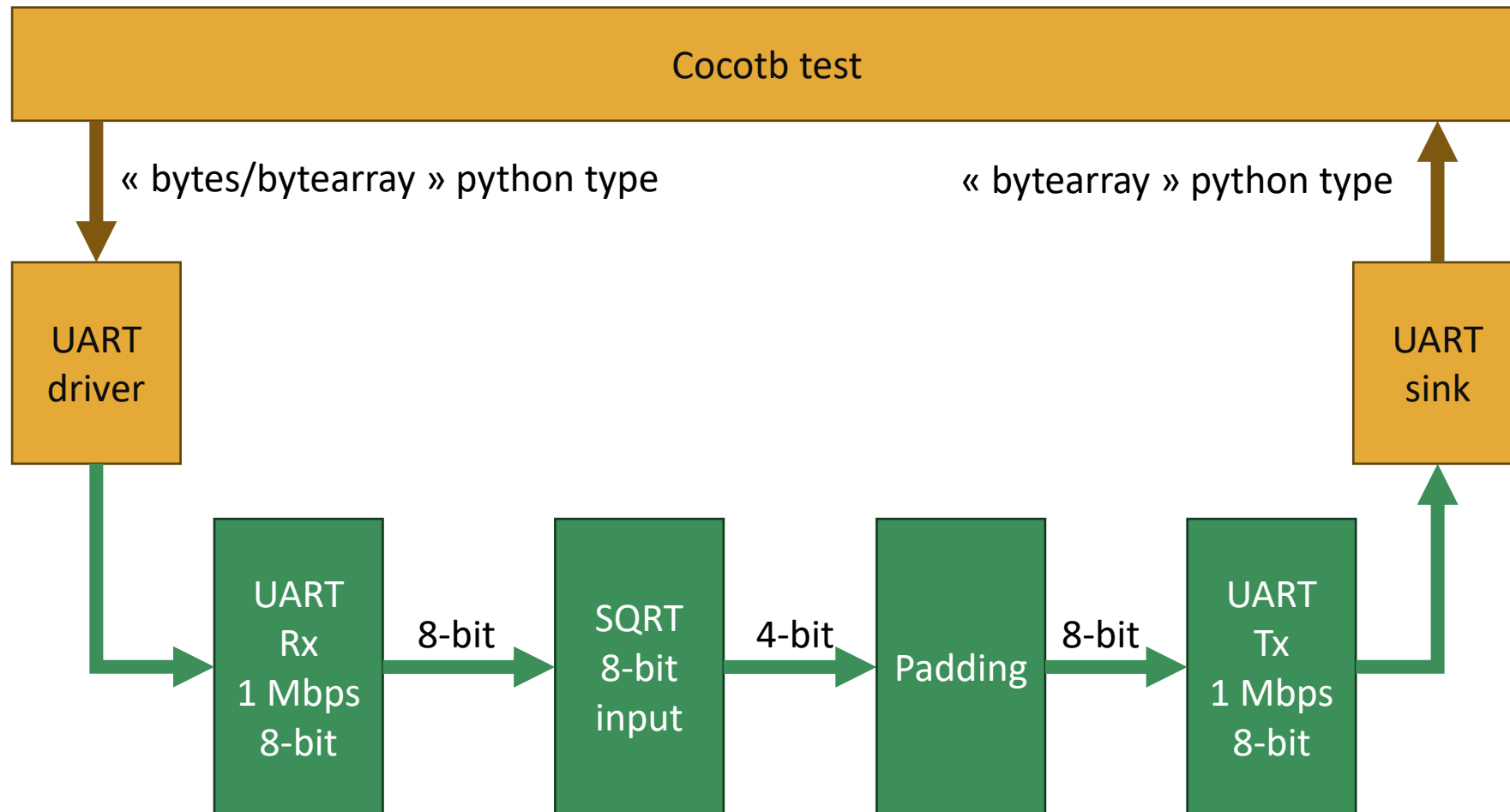
Others available from private repositories: ahb, ...

SURF workshop will use the AXI cocotb extension

Lab 4 - Objectives

- Encapsulate reusable sequences in functions
- Use a cocotb extension to control a UART standard interface
- Get familiar with the data format used by the UART extension, and how to make conversions.

Lab 4 - Design



Lab 4 - Work

Update the runner to include the multi-file design

- add all VHDL files

Encapsulate init sequence and end-of-sim time

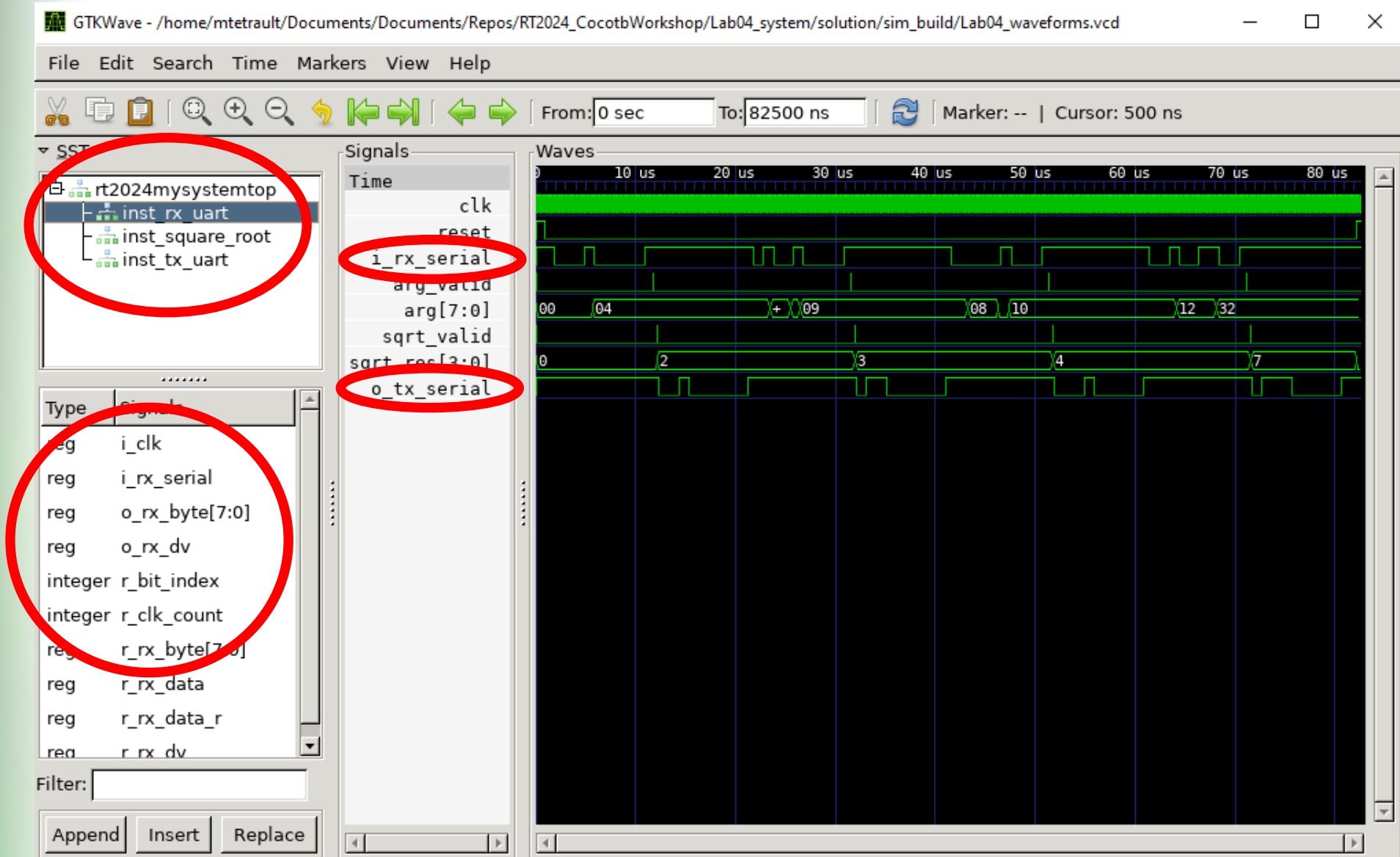
- use a function, not forgetting the special keywords

Use cocotbext-UART

- Add and use a driver and a sink object

Use native Python functions for conversion from/to bytearray

Lab 4 – Expected outcome



3!T

Lab 5

Code reuse 2 – object oriented programming

Lab 5

DAQ designs are complex : the verification code is not simpler

Universal Verification Methodology (UVM)

- Leverages Object Oriented Programming (OOP)
- Not supported by VHDL/Verilog; typically SystemVerilog
- Standardize structure and methods; excellent when buying a verification code

Steep ramp-up, requires simulation-specific SystemVerilog training, few students/scientists have basics on this topic.



Lab 5 – Cocotb with OOP

Python supports OOP, so cocotb does as well

UVM not required for small/medium sized designs

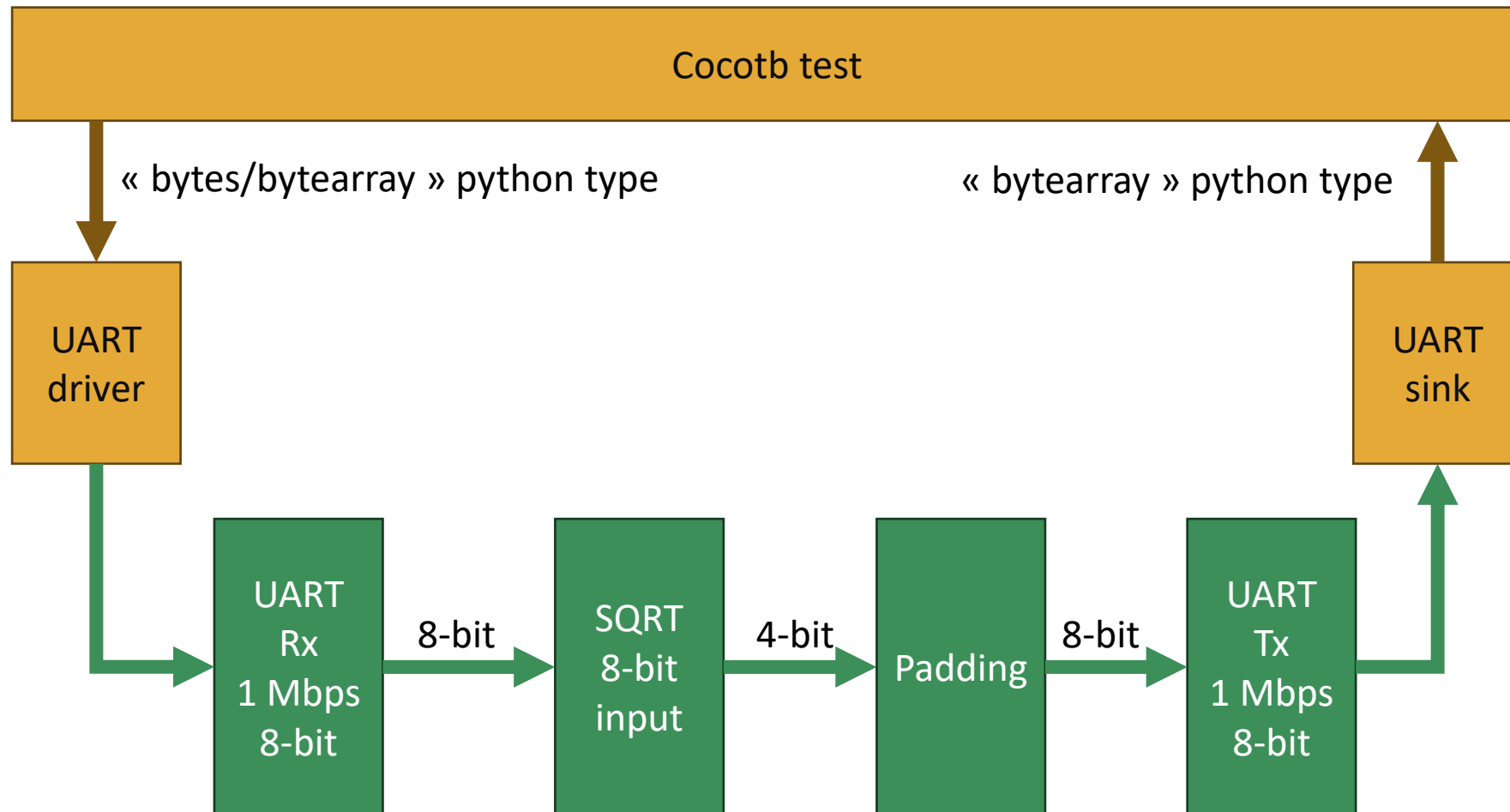
Planning a base class ahead will save a lot of time

- This is software programming, not firmware/HDL programming
- Larger pool of trained students and scientists can contribute

Lab 5 - Objectives

- Get familiar with a simple object oriented testbench structure.
- Populate the provided template with code prepared in previous labs.
- Write two different cocotb tests sharing the same base class.

Lab 5 – Design (same as lab 4)



Lab 5 – Base Environment

constructor (`__init__`)

- save dut pointer
- initialize logging utility

build environment

Init I/Os, clock and reset

configure dut

start environment

target test (pure virtual function)

post-test sequences

- wait for ongoing transactions

run function

- executes these steps one after the other

Lab 5 – Base Environment

constructor (`__init__`)
- save dut pointer
- initialize logging utility

build environment
Init I/Os, clock and reset

configure dut
start environment
target test (pure virtual function)
post-test sequences
- wait for ongoing transactions

run function
- executes these steps one after the other

Constructor – dut and logs pointers

Build the environment

- connect drivers, sinks, checkers
(not previously covered)

Start clock and reset dut (same as
lab 4)

Lab 5 – Base Environment

```
constructor (__init__)  
- save dut pointer  
- initialize logging utility  
  
build environment  
Init I/Os, clock and reset  
configure dut  
start environment  
target test (pure virtual function)  
post-test sequences  
- wait for ongoing transactions  
  
run function  
- executes these steps one after the other
```

Configure DUT – enable channels, set thresholds, set bias, etc.

Start the environment

- enable drivers, waveform generators, enable checkers...

Post test – wait for unfinished transactions or packets

Lab 5 – Base Environment

constructor (`__init__`)
- save dut pointer
- initialize logging utility

build environment
Init I/Os, clock and reset
configure dut
start environment

target test (pure virtual function)
post-test sequences
- wait for ongoing transactions

run function
- executes these steps one after the other

Test: undefined in base class, forces designers to derive the class and override the test.

Run: executes steps in order for all tests.

Lab 5 – Child Classes

constructor (`__init__`)
- save dut pointer
- initialize logging utility

build environment
Init I/Os, clock and reset
configure dut
start environment
target test (pure virtual function)
post-test sequences
- wait for ongoing transactions

run function
- executes these steps one after the other

Inheritance

Actual test 1

Actual test 2

Actual test 3

Contributors can focus their efforts on the test, relying on the environment

Lab 5 – Template and work

constructor (`__init__`)
- save dut pointer (done)
- initialize logging utility (done)

build environment
Init I/Os, clock and reset
~~configure dut~~
start environment
target test (pure virtual function)
post-test sequences
- wait for ongoing transactions

run function
- executes these steps one after the other

test 1: hard-coded values for sqrt

test 2: random values for sqrt

Lab 5 – Expected outcome

- Same waveform patterns as in lab 4.

Note: the two simulations will be appended in the same VCD file. Raising the reset at the end of a test (i.e. in the post-test sequence) helps to see this

3!T

Lab 6

Code reuse 3 – Monitors, Models and Checkers

Lab 6

Labs 4 and 5 see the DUT as a black box

When an error occurs, it is not always clear where the problem originates from. The designer needs to read the waveforms to find the issue.

Localized tests accelerate bug localization

System Verilog Assertions:

- industry standard, but...
- not supported by free/open source simulators
- Except perhaps Modelsim for Intel? To be confirmed.

Lab 6 – MMC construct

Monitor(s), Model and Checker

Monitors are probes, only recovering useful data from signals

- Conditions to record data depends on the interface
- Example: AXI bus needs to consider address, valid, ready and data signals
- Most simple interface : enable + data (sqrt core)

Lab 6 – MMC construct

Monitor(s), Model and Checker

Monitors are threads, basically while(true) loop.

In some cases, should not run while design is not in a known state

- For example, during the reset sequence

Monitor threads thus often have start/stop methods.

Lab 6 – MMC construct

Monitor(s), Model and Checker

Models generate the expected result from the HDL module. For example, CRC core, square root, FIFO, Data packet, compressed packet, etc.

Models have no notion of clock or signals. Only data

- Ensures model is different from HDL, improving error detection
- Should make model easier to code compared to HDL

Lab 6 – MMC construct

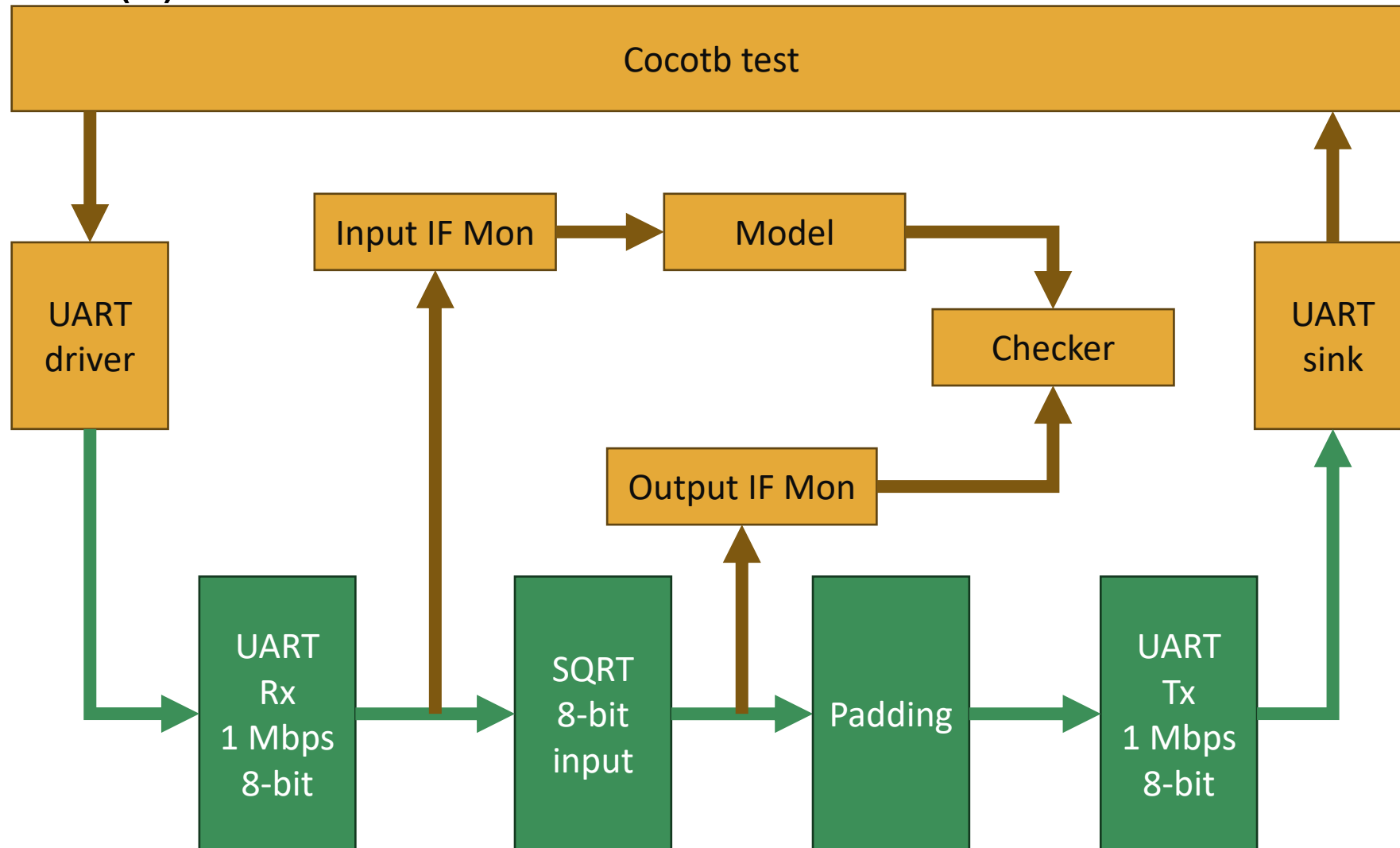
Monitor(s), Model and Checker

Checkers compare the result from the model and the HDL module

- Declares an error when differences are found
- Log utility provides instance location within the dut

Lab 6 – MMC construct

Monitor(s), Model and Checker



Lab 6 – MMC construction overview

How to write a checker (unit test) class

- 1- Create a monitor on the HDL input interface, connecting with its signals, in the constructor.
- 2- Create a monitor on the HDL output interface, connecting with its signals, in the constructor
- 3- Write a model method
- 4- Write a checker/test method
- 5- Write a “start” and a “stop method, launching and stopping the threads for the two monitors and the checker

Lab 6 – MMC insertion in base class

- 1- Add an MMC instance in the “BuildEnvironment” method
- 2- Add a “StartEnvironment” method, where the MMC.start() is called
- 3- In the post-simulation method, call the MMC.stop() method
- 4- Run the existing test(s)

Lab 6 - Objectives

- Reuse a monitor class from the cocotb main repository
- Adapt MMC class to the sqrt core.
- Attach MMC object to the base environment from lab 5

Warning: the template class from the cocotb repo uses efficient but less easy to understand native python constructs. Read the added comments for an initial explanations on these if they are not familiar to you.

Lab 6 – Bench split in 2 files

- Monitor and MMC class in separate file for clarity
- File with base environment class must import MMC class

Lab 6 – Expected outcome

- Same waveform patterns as in lab 4.
- Error messages will pinpoint the error location
Introduce an error in the model to generate a failure
- The assertion in the test is still relevant:
would indicate that something is wrong after the sqrt core

Lab 6 – Interesting « bug »

If test fails, it stops at the assertion

Does not start a new simulation, but continues where the previous one stopped.

Notice here the UART modules have no reset signal.

If the simulation failed while the UART is transmitting, it will do so regardless that the first test stopped, making the next test fail.

Fix 1 – add reset to uart in HDL

Fix 2 – make reset time much longer, and clear the `uart_sink` after the long reset.