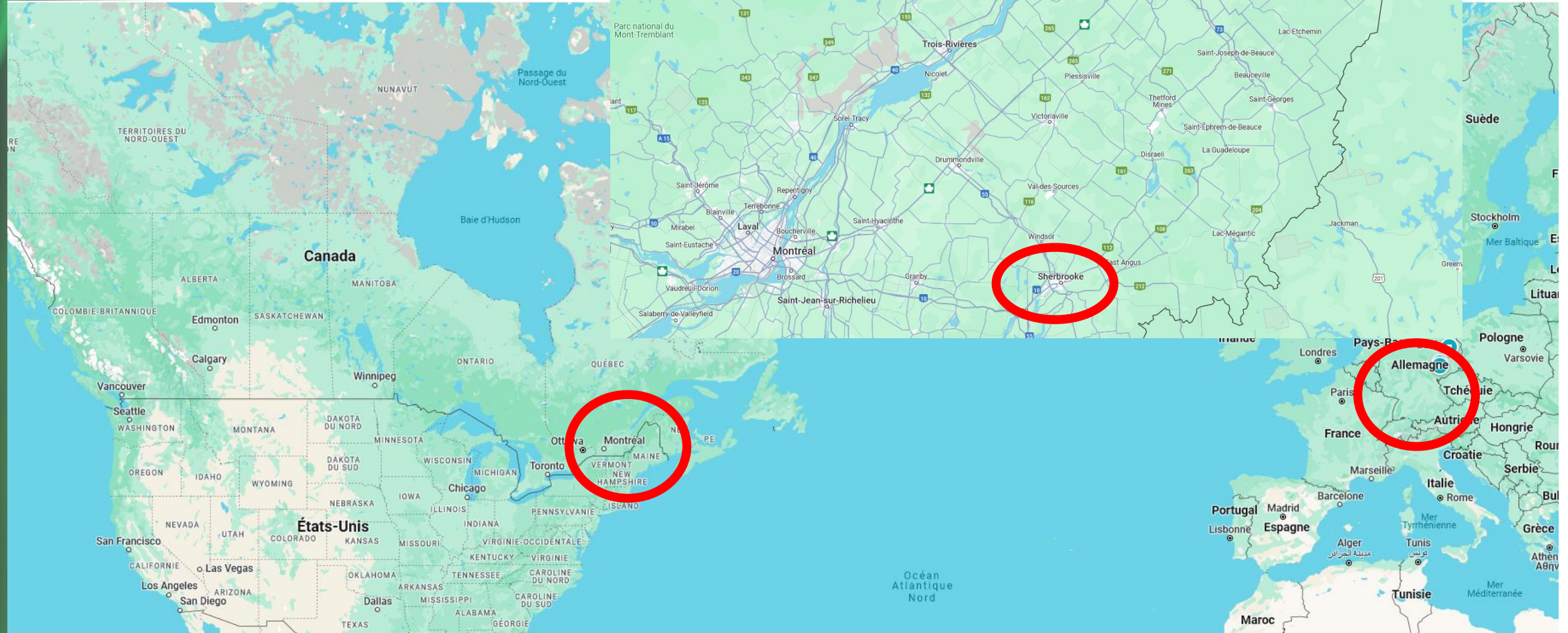


FPGA Ignite 2024

Cocotb Primer

Marc-André Tétrault, Eng., PhD.

6/08/2024





Université de
Sherbrooke



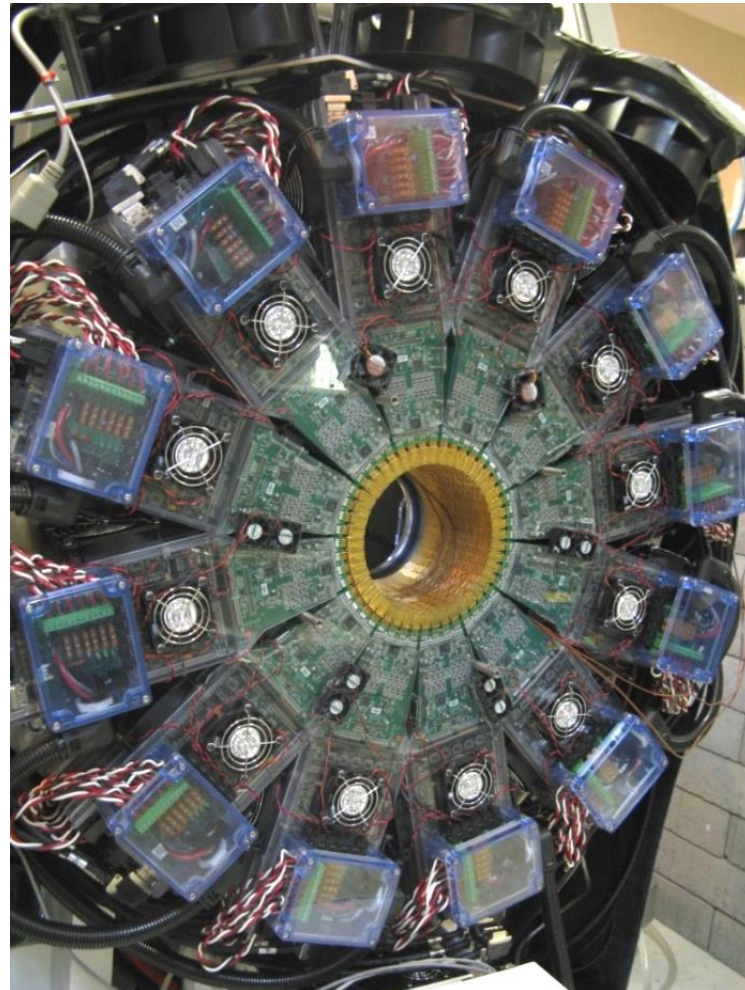
Distributed digital design for medical imaging

Lecomte et al

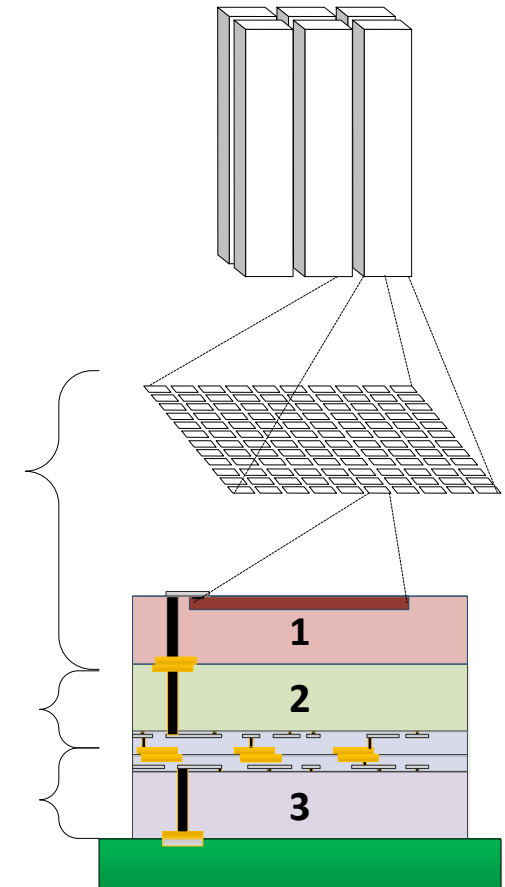
Sherbrooke Prototype



Fontaine, Lecomte et al
LabPET



Pratte, Charlebois et al
3D dSiPM



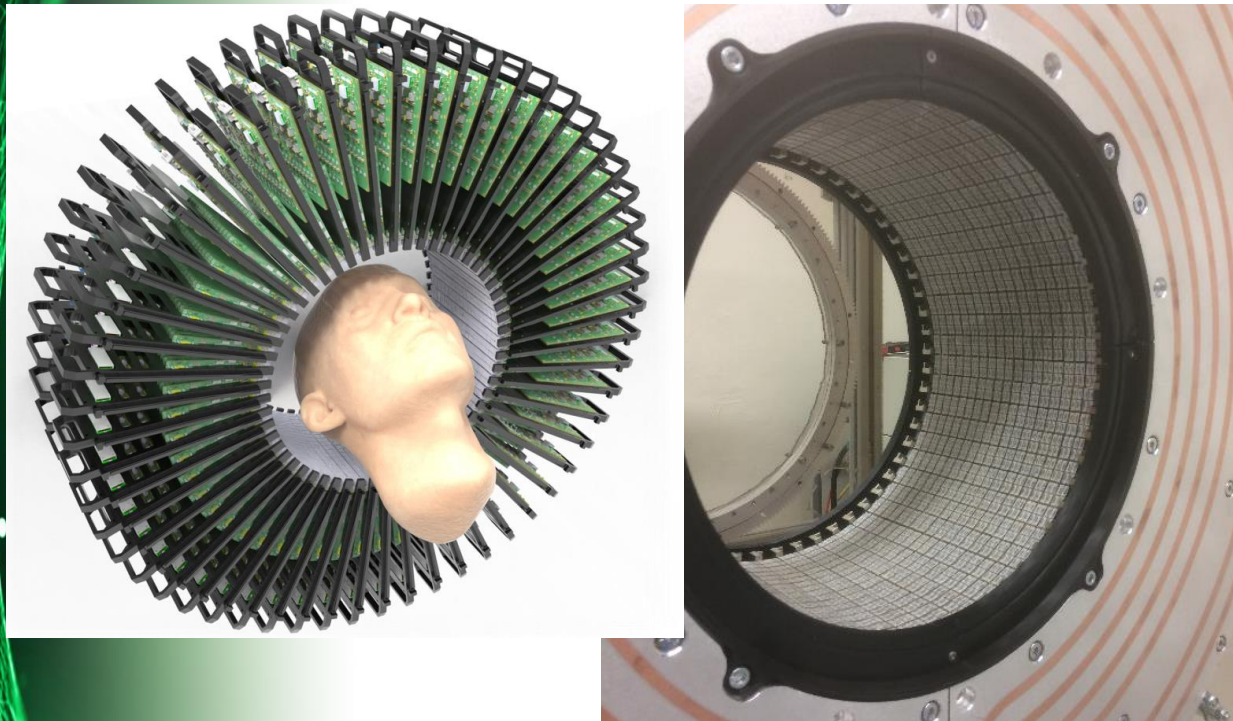
3!T

Recent project/collaborations

Postdoc

UHR/SAVANT Brain scanners

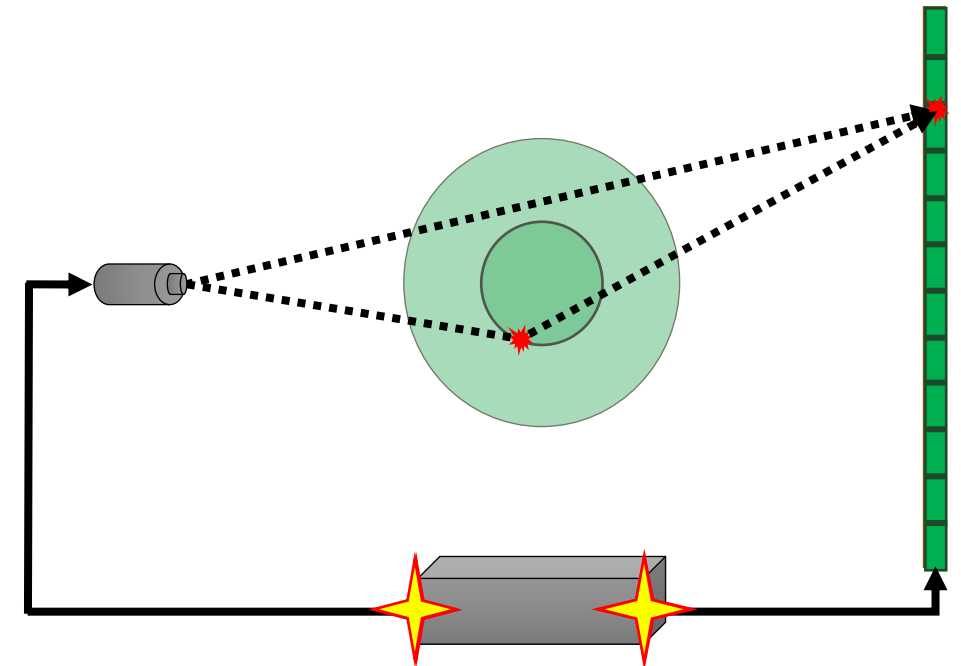
Lecomte, El Fakhri, Fontaine et al



Current

ToF CT

Bérubé-Lauzière, Corbeil Therrien, Tétrault, Fontaine et al



J.Rossignol et al. Time-of-Flight Computed Tomography: Proof of Principle and Challenges



Cocotb

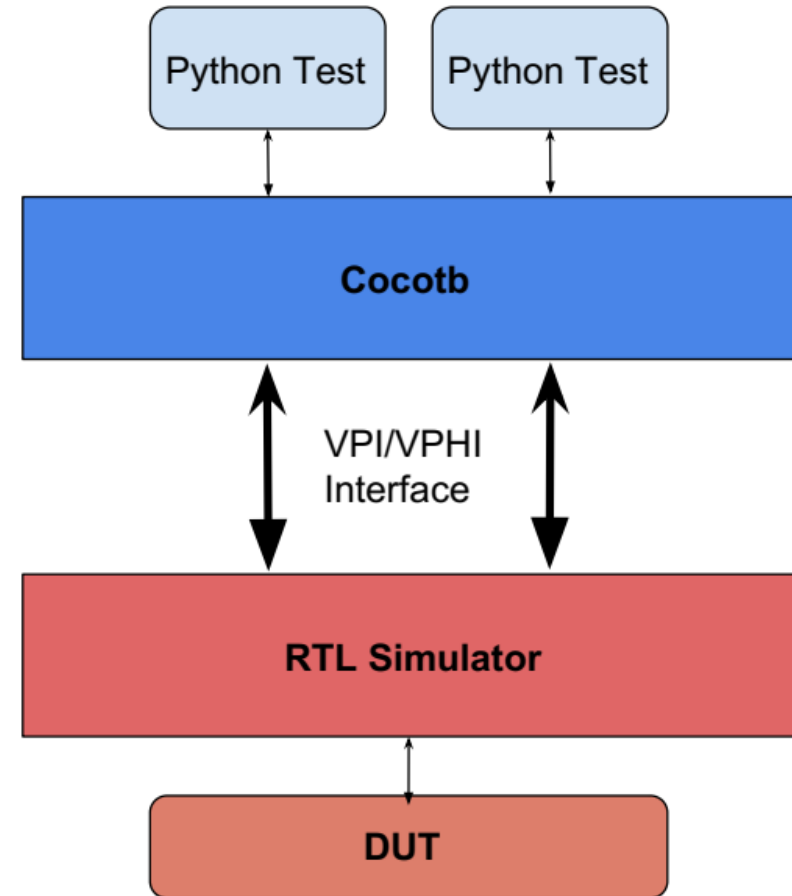
Coroutine cosimulation testbench

Uses Python instead of HDL/tailored language

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

Cocotb – Link with simulator

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



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

Supported simulators

Commercial

Incisive/Xcellium

Questa/Modelsim

VCS

Open-source

Verilator

GHDL

Icarus Verilog

...

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

Opensource vs commercial simulators

Commercial

Mixed language

Advanced features (SVA)

Waveform viewer

Open-source

Single language

Community support

External waveform viewer

Why change simulation language?

Verification compares models

- The HDL model (Behavioural, RTL, gate level)

vs

- The testbench model (expected outcomes)

Same language → risk of very similar models (and errors)

Cocotb – Hands-on Primer

What I wished I had seen or known when I started out

- First look at major components (launcher and sample test)
- How to debug this test bench?
- How to make it easy to reuse and modify my testbench code
- Verification automation

Cocotb – Hands-on Primer

Primer expects basic HDL/Linux experience

Aims to rely on Open Source tools

- Uses VHDL, with the GHDL simulator
- Uses GtkWave to view waveforms
- Basic text editors (nano, vim, gedit) for file edition
- VSCodium (not quite VSCode) 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 only
- 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 reads "GTKWave -". The menu bar includes "File", "Edit", "Search", "Time", "Markers", "View", and "Help". The toolbar contains various icons, with a magnifying glass icon circled in red. Below the toolbar, the "From:" field is set to "0 sec" and the "To:" field is set to "20 ns". The "Marker:" field is empty, and the "Cursor:" is at "12190 ps".

The main window is divided into several panes. On the left, the "Project" pane shows a tree view with "adder" selected, also circled in red. Below it, the "Signals" pane lists the following signals:

Type	Signal
reg	a[3:0]
reg	b[3:0]
reg	x[4:0]

The "Waves" pane on the right displays a timing diagram with a 10 ns scale. The diagram shows three signals: "E", "B", and "x". The "E" signal has values 1, F, C, E, 4, 7, C, A, 2. The "B" signal has values 4, A, 5, 6, 5, C, 6, 5. The "x" signal has values 19, 05, 19, 11, 14, 09, 0C, 18, 10, 07.

At the bottom of the window, there is a "Filter:" field and three buttons: "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

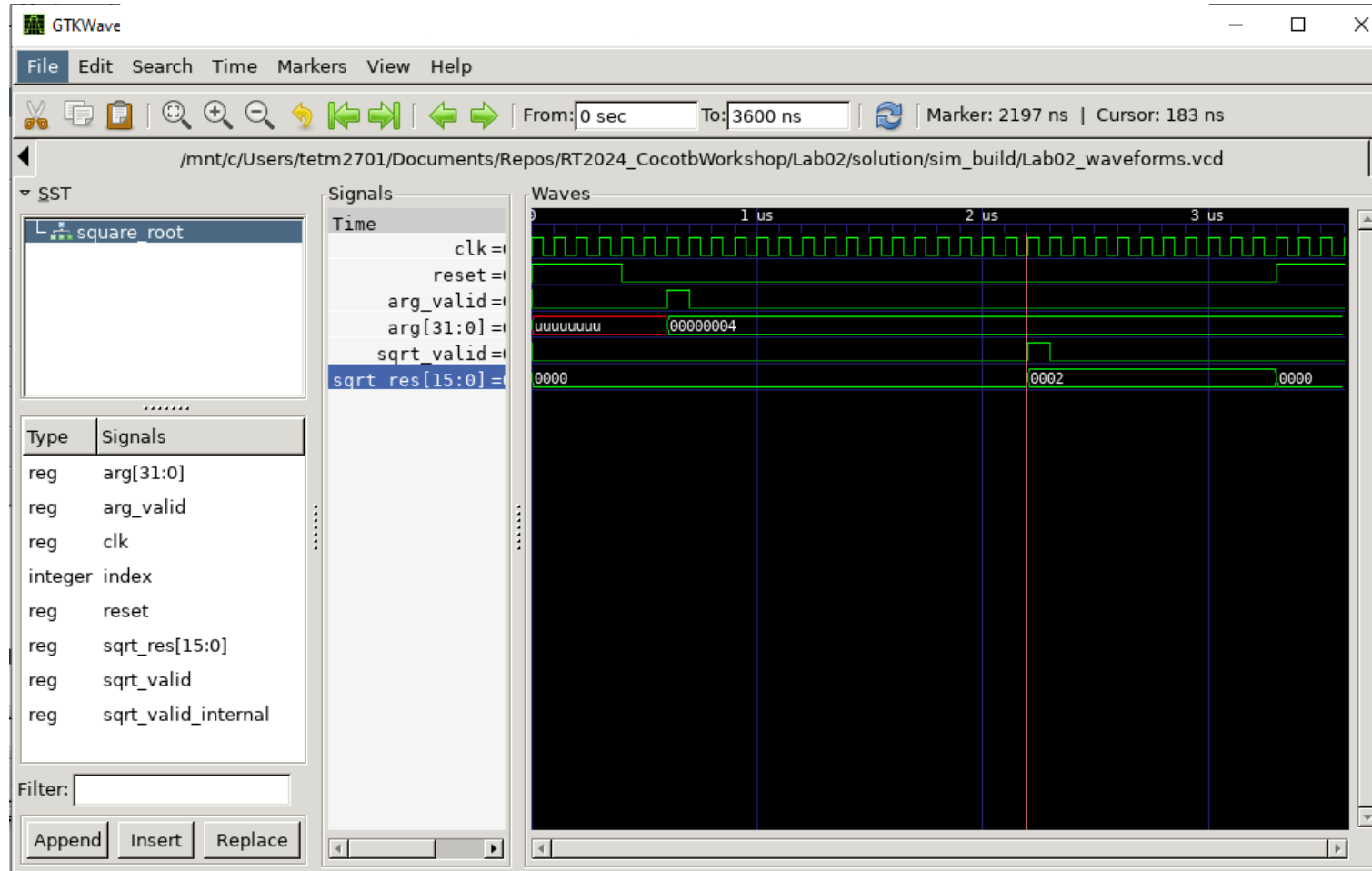
- Modify the runner to match the provided design
- Learn Cocotb relevant Python keywords
- Add very simple test for arithmetic core

Note: Python “async” and “await” keywords are not in typical Python scripts and programs (yield in older cocotb versions)

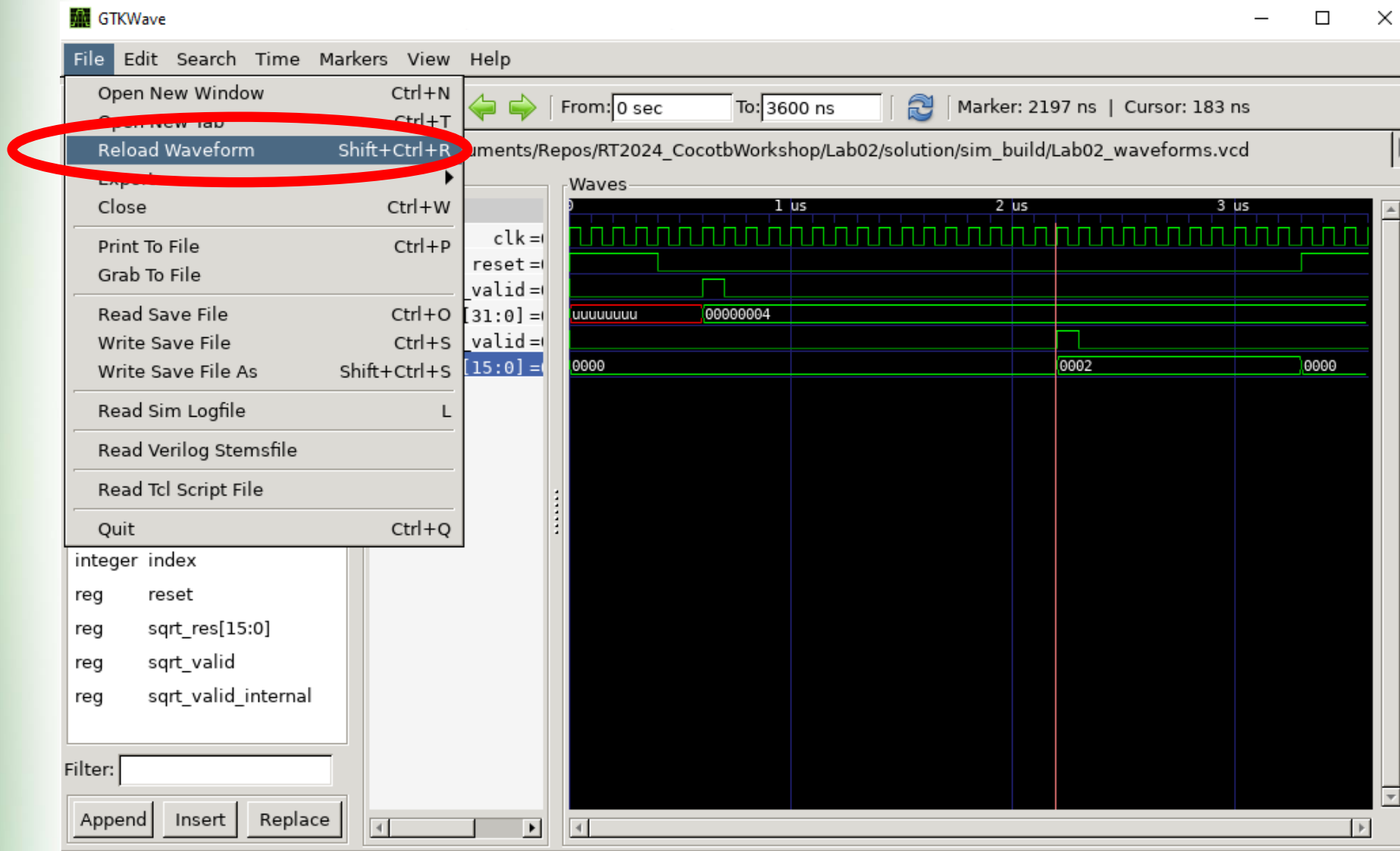
Use the Snippets File!

- Faster than google 😊

Lab 2 – Expected outcome



Lab 2 – GtkWave Tip



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

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
```

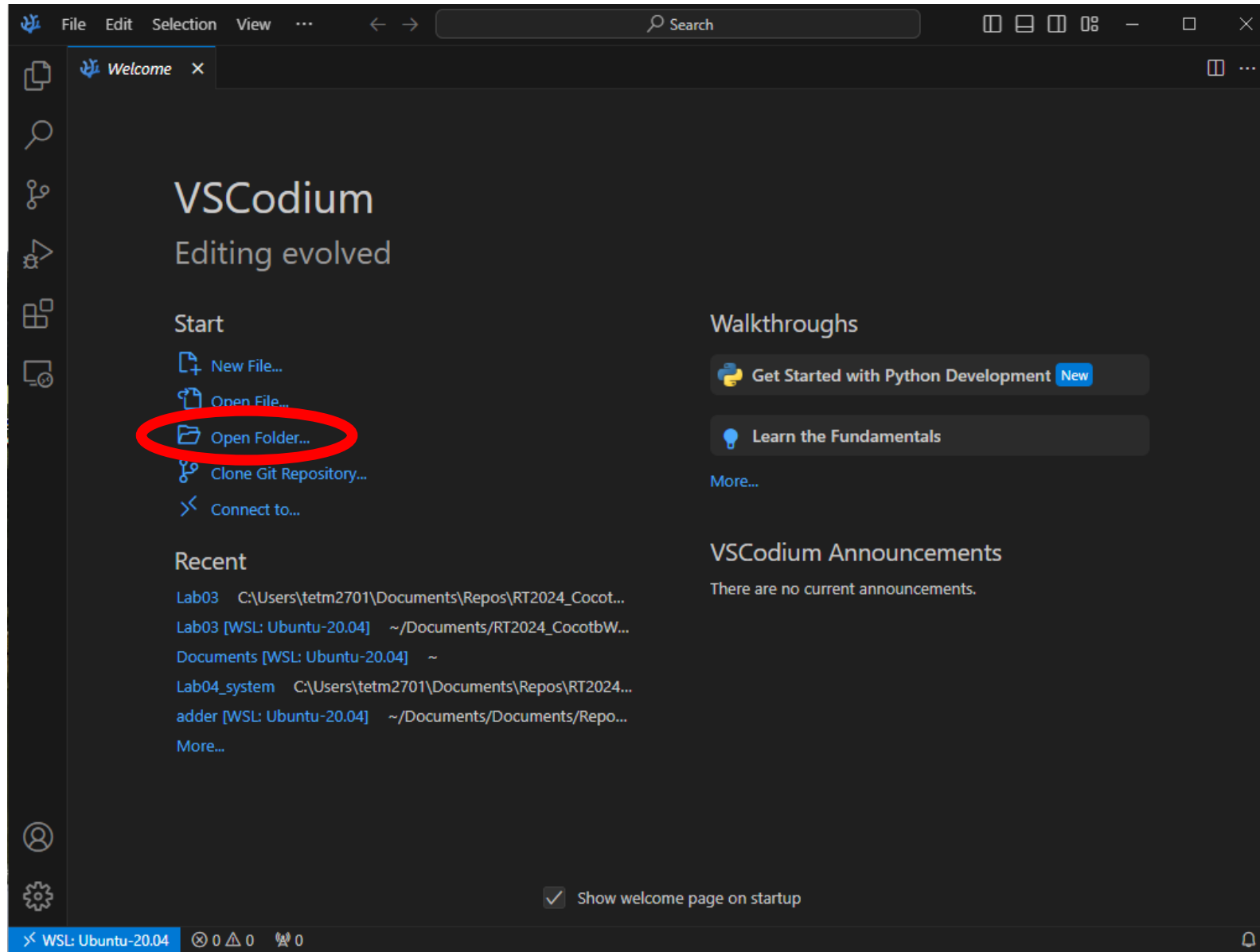
```
arg_valid
```

```
sqrt_valid
```

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

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

Lab 3 – Visual steps



Lab 3 – Visual steps

1

2

3

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

Show all automatic debug configurations.

Select debugger

Python Debugger Suggested

```
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
```

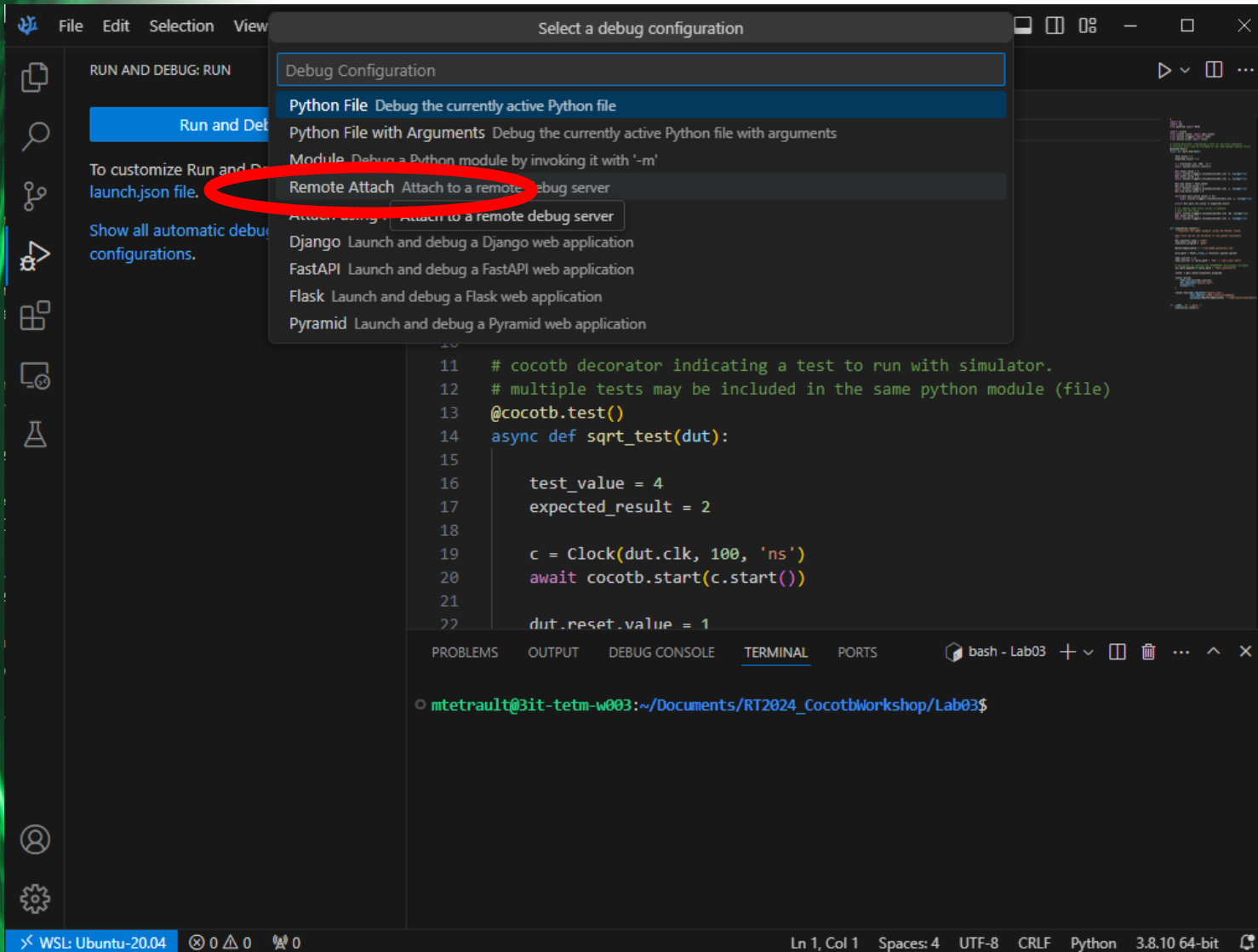
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

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

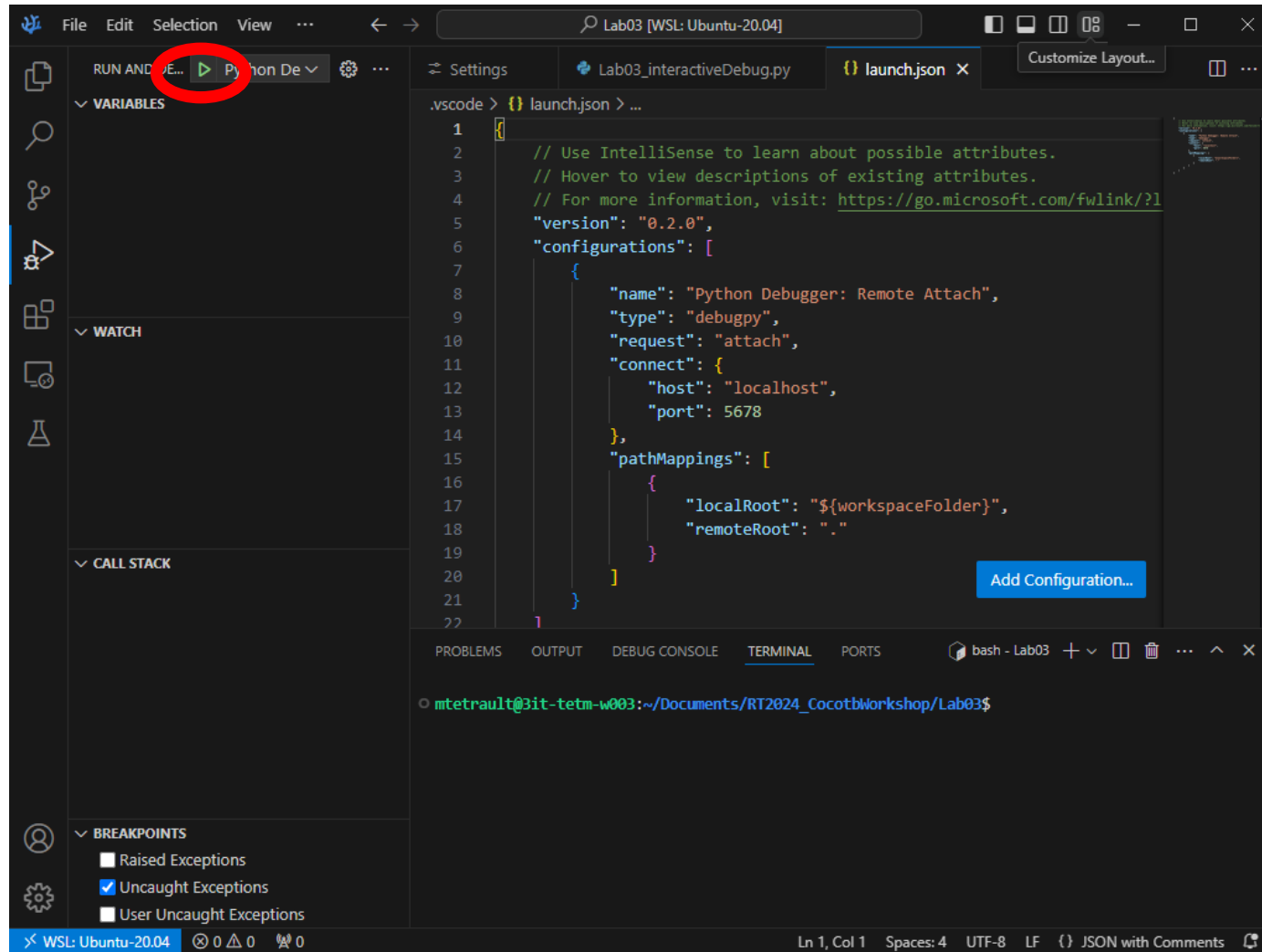
Lab 3 – Visual steps



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. The top menu bar includes File, Edit, Selection, View, and a search bar. The left sidebar contains icons for Explorer, Search, Run and Debug, and Extensions. The main editor area shows a file named `launch.json` with the following content:

```
1  {}
2  // Use IntelliSense to learn about possible attributes.
3  // Hover to view descriptions of existing attributes.
4  // For more information, visit: https://go.microsoft.com/fwlink/?l
5  "version": "0.2.0",
6  "configurations": [
7    {
8      "name": "Python Debugger: Remote Attach",
9      "type": "debugpy",
10     "request": "attach",
11     "connect": {
12       "host": "localhost",
13       "port": 5678
14     },
15     "pathMappings": [
16       {
17         "localRoot": "${workspaceFolder}",
18         "remoteRoot": "."
19       }
20     ]
21   }
22 ]
```

A red circle highlights the play button icon in the top-left corner of the Run and Debug toolbar. The bottom panel shows the TERMINAL view with a bash shell prompt: `mtetrault@3it-tetm-w003:~/Documents/RT2024_Cocotbworkshop/Lab03$`. The status bar at the bottom indicates the current file is `Ln 1, Col 1` with `Spaces: 4`, `UTF-8` encoding, and `LF` line endings. The bottom-left corner shows the active workspace as `WSL: Ubuntu-20.04`.

Lab 3 – Visual steps

The screenshot displays a Python IDE interface with a dark theme. The main window shows a code editor with the following Python 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:

- Left Panel:** Contains 'VARIABLES', 'WATCH', 'CALL STACK', and 'BREAKPOINTS' sections. The 'CALL STACK' shows 'sqrt_test Lab03_interactiveDebug...'. The 'BREAKPOINTS' section has 'Uncaught Exceptions' checked and two breakpoints set at lines 24 and 27.
- Top Panel:** Shows 'RUN AND DE...' and 'Python De' buttons. A red circle highlights the 'Debug' icon (a play button with a bug) in the top toolbar.
- Right Panel:** Contains 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL', and 'PORTS' sections. The 'DEBUG CONSOLE' is currently selected.
- Bottom Panel:** Displays the status bar with 'WSL: Ubuntu-20.04', '0 0 0 0', 'Python Debugger: Remote Attach (Lab03)', 'Ln 21, Col 1', 'Spaces: 4', 'Python', and '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, AXI, spi, uart...

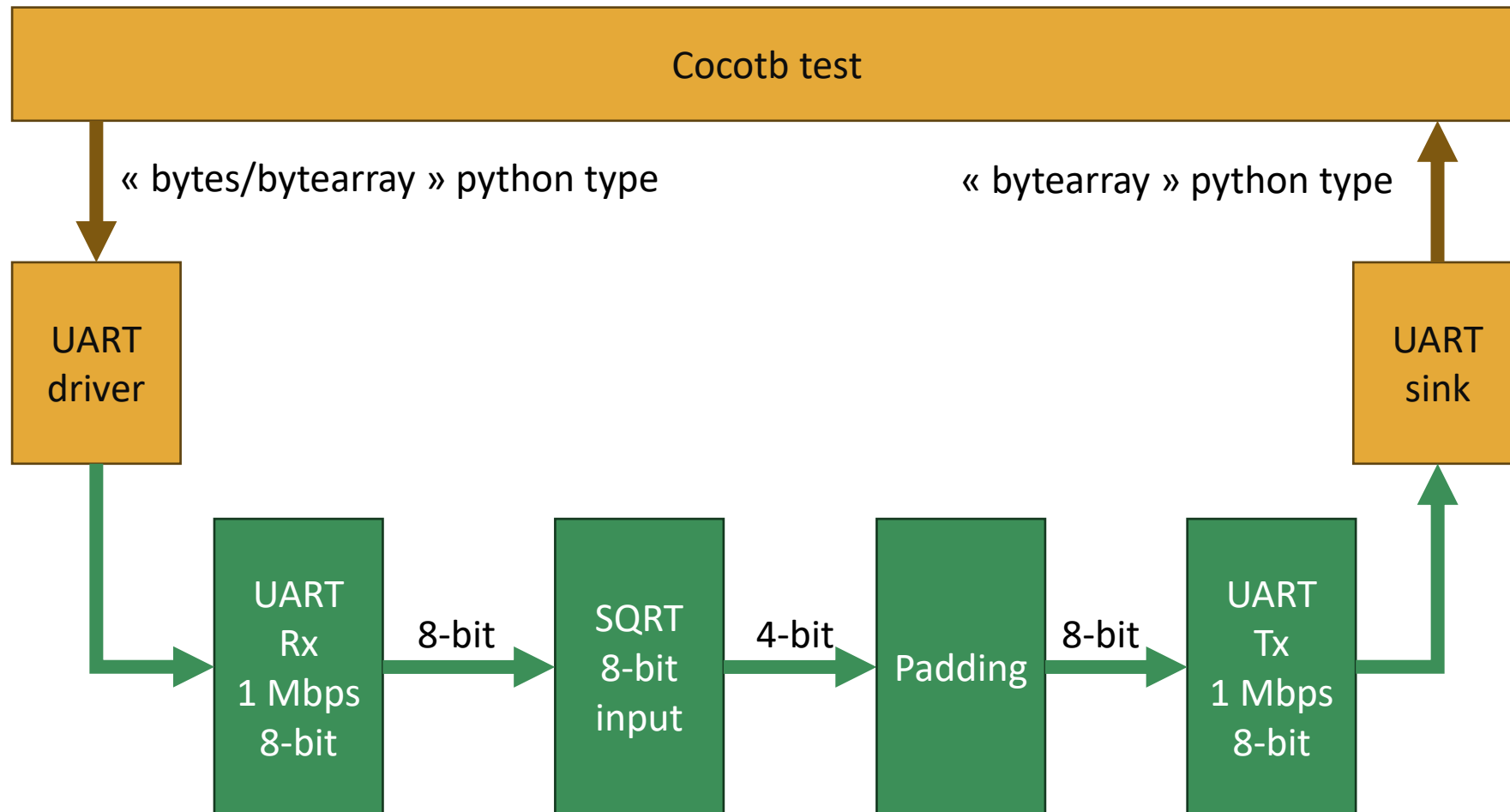
<https://pypi.org/search/?q=cocotbext>

Others available from private repositories: ahb, ...

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

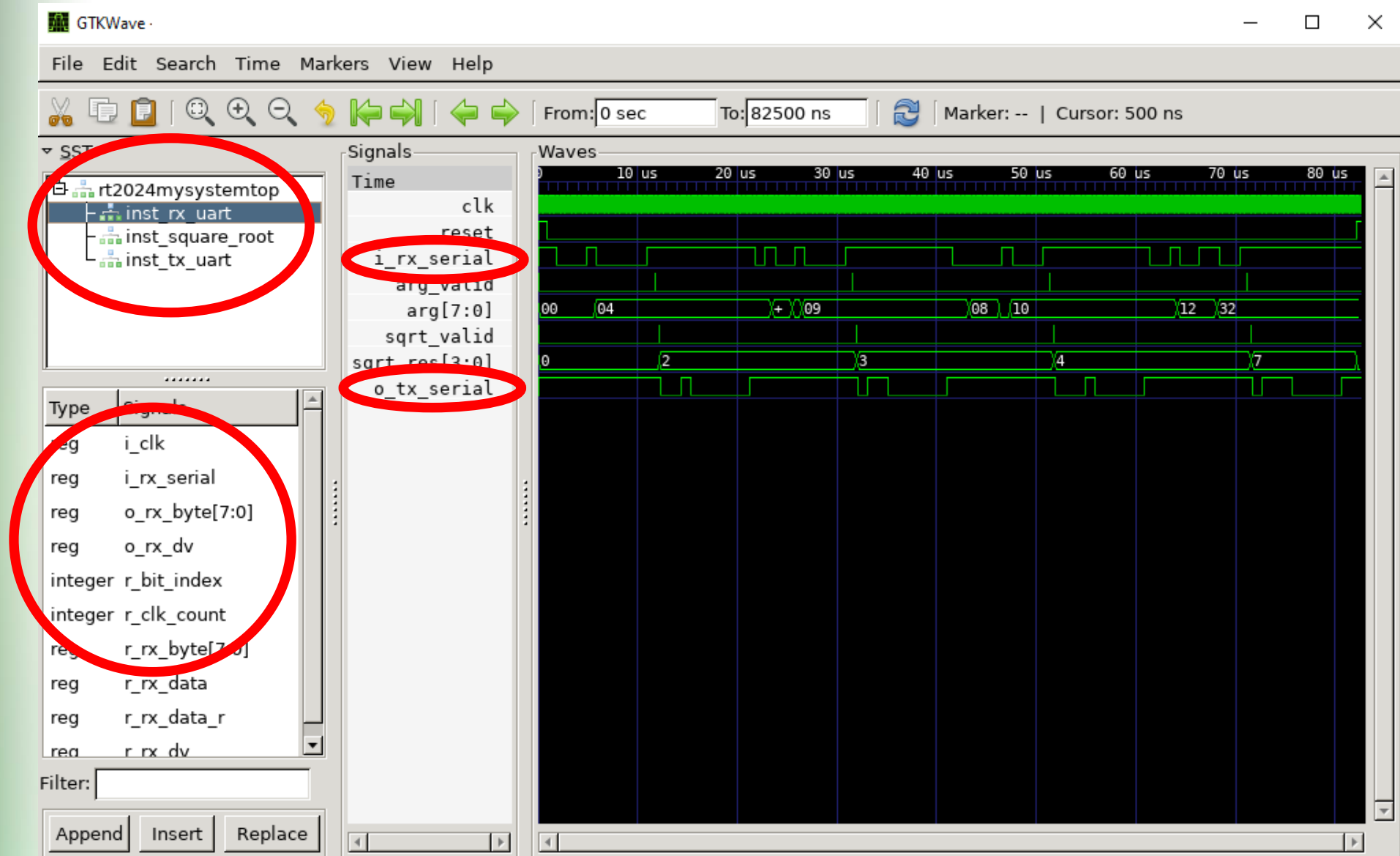
Use cocotbext-UART

- Add and use a driver and a sink object

Use native Python functions for conversion from/to bytearray

Note: Don't forget the snippets file!

Lab 4 – Expected outcome



3!T

Lab 5

Code reuse 2 – object oriented programming

Lab 5

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

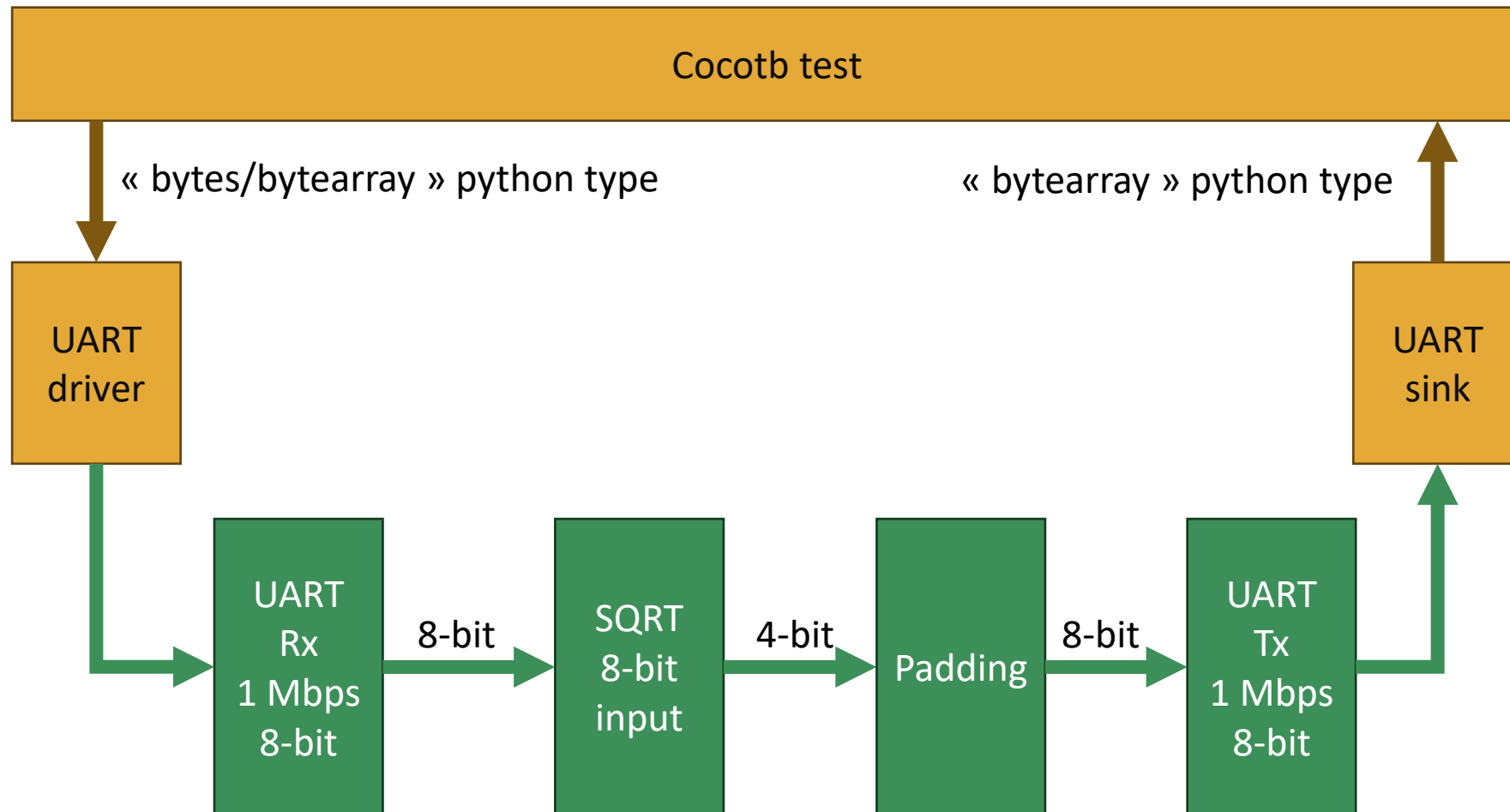
Planning a base class ahead will save a lot of time

- 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 to finish

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 to finish

run function
- executes these steps one after the other

Constructor – dut and logs pointers

Build the environment

- Connect drivers, sinks, etc.
- Connect checkers (lab 6)

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 to finish  
  
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 to finish

run function
- executes these steps one after the other

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

Run: executes steps in the same 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 to finish

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 to finish

run function
- executes these steps one after the other

test 1: hard-coded values for sqrt

test 2: random values for sqrt

In this lab, simple core, so no need for configuration phase.

« Start environment » will be used in lab 6.

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
- Exceptions? If you know, I want to know about them!

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 in an unstable state

- For example, not 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 should not have notion of clock or signals. Only data.

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

Lab 6 – MMC construct

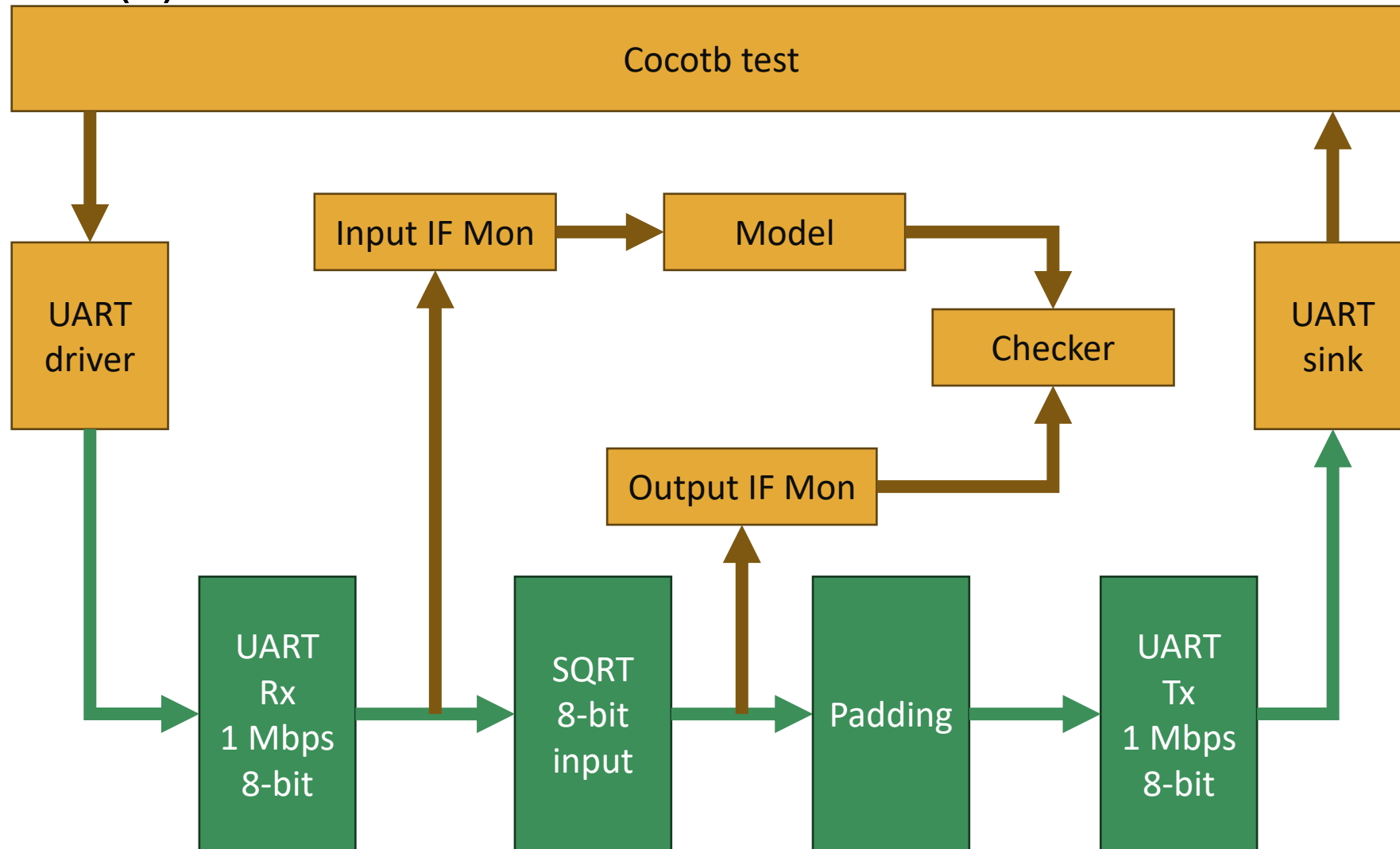
Monitor(s), Model and Checker

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 - 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 6.1 – 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 6.2 – MMC insertion in base class

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

Lab 6 – Solution split in 2 files

- You could put everything in the same file, but...
- Monitor and MMC classes together in a separate file for clarity and portability
- File with base environment class must import MMC class (basic Python import keyword)

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 a test fails, it stops at the (Python) assertion

It 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.

Fix 3 – What could be your solution?

3!T

Last slide