2024 IEEE NPSS Real Time Conference Workshop

Introduction to Cocotb

Prepared by:

Marc-André Tétrault, Eng., Ph.D.

April 19, 2024

Contents

Expected experience

- Basic experience with Linux and command line terminal.
- Basic experience with Python.
- Basic experience with git (to download examples)
- Basic experience with HDL simulators (Vivado, Modelsim, etc)
- Basic experience with VHDL.

Setup

Getting the lab files

The virtual machine has almost everything pre-installed. You need to download the additional files from git:

1. Open terminal.
2. Move to " /Documents"
3. git clone https://github.com/mtetrault/RT2024_CocotbWorkshopFiles.git

1 Lab 1 - First contact

Learning objectives

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

Note: This first example is an adaptation of the adder example from the official cocotb repository (tag v1.8.1).

1.1 Launching a simulation

The more recent version of Cocotb supports launching simulations directly from Python.

1. Open a terminal.
2. Go to the Lab01/tests folder in the workshop files cloned in section .
3. type *python3 test_adder.py*.
4. Inspect the log text in the terminal.
5. The message should conclude that the test passes.

1.2 Adding arguments to the simulator

The Python file implements two parts. One is the actual test to stimulate the VHDL. The second is preparing and launching the simulation using a dedicated HDL simulator, in this case GHDL. Python, through the cocotb interface, calls the GHDL program the same as done with the command line interface. Cocotb has a few different methods for this, where this example splits it in three steps:

1. Creates a runner object based on the simulator.
2. Configures/builds the runner.
3. Launches the test(s).

In this lab, you will add two parameters to give to GHDL. The first is to generate the waveform associated with the test. The lab voluntarily does the first attempts with mistakes, to help understand how Cocotb manages the arguments, and how to debug sometimes hard to understand error messages.

1. Open the python file in *Lab01/tests.* Use a command line editor (nano, vi) or a simple graphic editor (Applications –> accessories –> text editor) for this lab.
2. Read the documentation for the *build* and *test* methods.

3. Create a python string variable containing the GHDL option to generate the waveforms file. For example, *VcdFile="–vcd=output.vcd"*.

4. The option is part of the "run" step for GHDL, so add the option with the *cocotb.runner.test* method using the *test_args* argument as indicated in the cocotb documentation (linked above).

5. Run the python file, as in section 1.1. There will be an error message.

6. Read the error message. GHDL (not Python/Cocotb) complains of an unknown option. Notice in the terminal that all the characters of the string appear seperated by spaces at the GHDL program call.

7. The *test_args* parameter expects Python lists, and take the provided string as a list of characters. This needs fixing.

8. Change the assignment to a Python list element using the "[]" characters. Example: *VcdFile=["–vcd=output.vcd"]*. Again, there will be an error message when running the test.

9. The error message is the same. Reading the GHDL documentation, this option is expected after the module name. Inspecting the command line to GHDL, this is inserted before the hdl module name ("adder") and causes the error.

10. Replace the *test_args* argument with the *plusargs* argument, also described in the cocotb documentation. Run the test again. This time, the argument is inserted after the module name. The test passes, and generates the .vcd file.

11. In the terminal, notice the line *../../src/openieee/v93/numeric_std-body.vhdl:398:9:@0ms:(assertion warning): NUMERIC_STD."+": non logical value detected.* This is a warning that will reappear often in the labs. Add the *"–ieee-asserts=disable"* argument to the argument list to remove these warnings. **Note**: In real designs, this might not be desirable.

12. Run the simulation again. The warning should be gone.

13. Confirm that the .vcd file was generated, in the *sim_build* subdirectory.

1.3 Inspecting waveforms

1. In the command terminal, type *gtkwave &* to launch the viewer. Or, from the applications menu, find it in the accessories category.

2. In the *file* menu, choose *open new tab*, then select the .vcd file generated above.

3. In the upper left pane, click on *adder*.

4. In the lower-left pane, double-click on the three signals.

5. In the top icon bar, click on the magnifying lense on the left (zoom fit).

6. You should now see the random test values and the results.

7. You can now inspect waveforms from your simulations.

2 Lab 2 - Customizing a Cocotb template

Learning objectives

- Create a cocotb python file customized for your own HDL design.
  - Builder function.
  - Simple test function.
- Use basic signal control commands (clock, reset, vectors).
- Get familiar with Python constructs leveraged by Cocotb.
  - *async* and *await* keywords.
  - Cocotb test function decorator.
- Control simulation execution time (clock cycles).
- Use Python assertions to validate conditions.

2.1 Introduction

Examples provide good starting points, but always need some modifications to be adapted to your own design. This lab is such an exercise to further your understanding of the mecanics behind the cocotb configuration.

The VHDL design under test (dut) for this lab is an iterative square-root arithmetic core [**?**]. It takes a N-bit integer argument, and generates a N/2-bit integer result after N/2 clock cycles. The core defaults to 32-bit values, which is used for this lab. The overall steps of the lab are to configure the cocotb builder function, add a clock, run a reset cycle and test one calculation throught the arithmetic core.

2.2 Builder configuration

1. Use a command line editor (nano, vi) or a simple graphic editor (Applications –> accessories –> text editor) for this lab.
2. Open the arithmetic VHDL file in the *Lab02/hdl* directory and inspect the entity definition.
3. Open the *Lab02/tests/Lab02_cocotb.py* file and use it as starting point.
4. Using Lab 1 as an example, update the *simulation_runner* function to simulate the square-root core. Note that the core filename and core name are not the same, and this will impact some parameters for the filename list, build and test methods.
5. Add the missing Python keywords to the *mytest* function, and the *dut* argument. See lab 1 or code snippets for reference.
6. Run the simulation to validate that the build and test functions are correctly configured.

2.3 Add clock, reset and stimulus signals

For this section, many of the commands have examples in the Code Snippets file. To save time, make sure to have first overview before starting. Then, use the snippets in your python file. However, make sure to always match the values for the signal names and parameters (example: clock period). Alternatively, also read the online cocotb documentation.

For the next steps, all of them have examples in the snippets file, or in Lab 1.

1. Remove the Timer delay in the provided *mytest* function.
2. Begin the test by creating and starting a 10 MHz clock (see snippets).
3. Set the dut's reset signal to 1 for 5 clock cycles, then lower the reset to 0 and wait for 2 clock cycles.
4. Set the dut's argument port to a known value (your choice, but preferrably not 0 or 1).
5. Set the dut's arg_valid port to 1 for a single clock cycle (impulse lasting one clock cycle).
6. The core takes 16 clocks to calculate the result. Use a while loop reading the *sqrt_valid*, waiting until it changes to 1.
7. Use the Python *assert* keyword to compare the core's output with the expected value and confirm if the core's result is the expected answer (see snippets).
8. Wait a few more clock cycles, then raise the reset signal to 1. Wait again a few cycles.
9. Configure GHDL to generate a vcd file from this simulation.
10. Run the simulation, the test should pass.
11. Inspect the waveform to double-check the result. When *sqrt_valid* rises to 1, the value should be correct.
12. To confirm the assertion works correctly, change the expected value to a wrong one. Launch the simulation and confirm that the "fail" message appears.

3 Interactive Debugging

Learning 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.

3.1 Introduction

When simulating a design, both the HDL design and the testbench need to be debugged and inspected. When using HDL for the testbench (VHDL, Verilog, SystemVerilog), commercial tools provides access to a GUI with breakpoints and stepping utilities. Python is not an HDL language and thus not supported. Furthermore, Cocotb is not the main process of a simulation, but rather functions called by the simulator itself. To have access to line-by-line debugging, one needs to use so-called "remote debugging".

There are several python remote debugging tools available. The current lab will use the *debugpy* module, and the VSCodium interactive development environment (IDE). Other IDE's supports this workflow, but not always in the basic version. For example, Pycharm, a very popular IDE, only offers remote debugging with the licenced project at the time of this writing.

3.2 Preparing VSCodium

1. On your workstation (virtual machine for the live workshop), find VSCodium in the application menu and start it.
2. VSCodium in the virtual machine has been preconfigured. The Python and Python debugger modules were added, and the Python interpreter set to the default Python3 installation. See VSCodium or VSCode documentation of details.
3. Use the "open folder" command, and choose the Lab_03 folder. The dut is the same as lab 2, with the python solution in the test folder.
4. On VSCodium's left pane, choose the file explorer and open the Lab03_cocotb.py file in the tests folder. Python highlighting and coloring should be enabled.
5. On the left pane, choose the "Run and debug" group.
6. With the Python test file displayed in the edit area, choose the "create a launch json file" under the "run and debug" button.
7. In the top center of the VSCode window, it will ask to choose a debugger.
8. Take the first choice (Python debugger), then from the list select "remote attach".

9. Leave the default "localhost" and press enter.
10. Leave the default "5678" and press enter.
11. Inspect the json file without changing it, then close it.

3.3 Preparing the cocotb test

1. Import the *debugpy* module in the cocotb python file.
2. In the test function (not main or simulation builder), add the .listen(port), .wait_for_client and .breakpoint() methods. See code snippets for example.

3.4 Launch simulation and attach debugger

1. In a terminal, launch the cocotb simulation as in previous labs.
2. Log messages will appear in the terminal, indicating the test is running, but will seem to freeze. This is expected.
3. Return to VSCode. While in the debug pane, look for the green "play" button. The text to its right should reflect the json file created earlier and indicate "remote attach".
4. Click on the "play" button.
5. The debugger should attach to the simulation, and stop just after the *debugpy.breakpoint()* line.
6. You can now add breakpoints in VSCodium by left-clicking to the left of the line numbers.
7. Above the code, typical debug buttons have appeared:
    - Continue
    - Step over
    - Step into
    - Step out
8. In the side pane, variables have appeared, including the dut object.
9. You can now debug the Python portion of the cocotb test, and access current design values.

3.5 Making things easy

Note : At the moment of this writing, the solution proposed below could not be validated, as edits to the VSCodium json file seems to cause errors. This will hopefully be fixed in the future.

The method presented above is executed in two steps. However, it should be possible to configure VSCodium to both launch the simulation and attach the debugger, but requires a bit of tweaking. If the json file problem can be resolved, a solution could be inspired by the combination of the following two posts, the first about compound launch and the second on how to delay a launch (or attach) command.

An approximate flow would be along the lines of:

1. Add a "python debugger" configuration to the json file using VSCodium.
2. Test it by running it with the "play" button. It should hang (because of the "listen" code line).
3. Add a *remote attach* configuration to the same file.
4. Add a prelaunch delay to the *remote attach* configuration, as in the second post.
5. Test it by running it with the "play" button. It should give an error because no debug server would be running.
6. Manually edit the json file to create the compound configuration.
7. Run the compound configuration with the "play" button. This time the test should start, followed by the debugger attachment, allowing the flow in a signal clic.

4 Code reuse 1 : functions and drivers

Learning objectives

- Replace manual signal toggling by functions or objects
- Use a driver to control a UART standard bus interface
- Get familiar with different data formats used by cocotb and its extensions.

4.1 Introduction

The simulation environment for any HDL project is generally of similar complexity to the design. As such, it becomes important to plan ahead and use layers of abstraction to focus on the features to test rather than how the testbench interacts with the design. Control busses are a very good example, like the AXI interface supported by the Vivado tool. The verification scientist may then focus on the dut rather than the protocol handshake with the HDL.

In this section, we will begin by encapsulate the clock instanciation and reset cycle in a function, isolating these standard steps to remove them from the main test function, improving readability. Next, the lab will introduce the cocotb extension library with the UART configurable driver (the AXI driver will be shown in the SURF workshop). The uart uses byte arrays, adding some data type conversions, but are worth the effort in terms of higher level programming.

The demonstration design resuses the square root arithmetic core, placing it between a UART receiver and transmitter. The design clock period is 10 MHz, and the UART speed is 1Mbps with 8-bits per transfer. The arithmetic core is configured as 32-bit, but only uses 8-bit values received from the UART. The uart also only returns the lower 8-bits from the result.

4.2 Build preperation

1. Open the cocotb/python file for lab 4. It is nearly identical to lab 2 and 3.
2. Add the uart and top-level HDL files to the VHDL list use by the simulation builder.
3. Update the hdl top-level name, as well as the test module name.
4. Confirm the configuration by running the test, for example with the *do_nothing_test* from the code snippets as the test.

4.3 Encapsulate clock, reset, and end of simulation

1. Create an empty function called *init* with *dut* as argument. Do not put the @cocotb.test decorator before its declaration. This is a subfunction.
2. Cut and paste the clock instanciation and reset sequence from lab 2 in the init function.
3. Call the init function in the cocotb test.

4. Run the simulation. If Python complains, the *async* and/or *await* keywords might be missing. See the code snippets for examples.

5. Create a second function, called PostTestDelay, which contains a delay of a few clock cycles and finishes by returning the reset to 1 and again a delay of a few clock cycles.

4.4 Write a test using UART driver and sink

1. Import in the python file the UARTSource and UARTSink classes from the cocotb extension library.

2. In the cocotb test, before calling the init function, create the uart driver and sink objects, attaching the uart signals with the DUT and specifying the proper configuration (1Mbps and 8-bits).

3. After calling the init function, and using the example in the code snippet, use the driver to send one byte with the UART driver (.write(...) method). **Note**: the driver expects a *bytes* object, so use the integrated Python *.to_bytes()* method for the conversion, shown in the snippets.

4. Add the .wait() method to let the test wait for the driver to complete the UART transaction.

5. Add a call to the *PostTestDelay* function written above.

6. Run the simulation, and inspect the waveforms on the UART and arithmetic core instances. This time the design has 2 levels of hierarchy, so you must use gtkwave to display the signals in the arithmetic core.

7. The arithmetic core should give a correct answer in the waveforms. Once validated, continue to the next step.

8. In the cocotb test, before the call to the *PostTestDelay* function and after the driver.wait() function, add a read request using the sink object, requesting 1 byte. It will return a ByteArray object.

9. Convert the *ByteArray* to *int* by casting it to *bytes* and then using the Python built-in conversion method (see snippets). Print the integer value.

10. Use an assert command to check if the returned value is correct.

11. Change the test to send multiple values from a list. Use a second list to provide the expected return values (Python 3 *for/zip* keywords, see snippets).

12. Run the simulation, let the assert command check if all values passe or fail, and inspect the waveforms for further confidence.

5 Code reuse 2 : Object Oriented Programming

Learning 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.

5.1 Introduction

Beyond using drivers, the verification community has developped different standards to promote code reuse. On example is the Universal Verification Methodology, or UVM. It leverages object oriented programming techniques not available in VHDL or Verilog, but enabled in SystemVerilog and the nowadays less used *e* verification language. While UVM promotes code reuse, maintaining a code library requires non negligible effort and training. The lab here introduces a simplified example, providing a basic but flexible starting point for small to medium-sized projects.

A use case in data acquisition systems would be where two modes must be supported by the same FPGA firmware, for example a triggered mode and a data streaming mode. The two modes share the same HDL modules, the same reset sequence, the same input signals, but their configuration, output format and data validation checks are different. At least two, but likely more than two tests are required to fully simulate the HDL design.

This lab provides a simple class definition with empty methods, to be filled with the code from the previous labs. The goal is to reorganize the code using a class, to have very small test functions leveraging class inherentence to reuse common code.

5.2 Move the testbench to class methods

1. Open the lab 5 python file in the tests folder.
2. Read the file and comments.
3. Note the ___init___ function, where it copies the pointer to the dut.
4. Populate the methods as suggested by the comments. *Reminder*: in Python, class members are referenced with the *self.* keyword.
5. To define the actual test, define a derived child class outside cocotb test function and override the *test* method. See *EmptyTestClass* in the lab 5 template as an example.
6. Similarly, add a cocotb test entry function with the appropriate decorator, which instanciates an object of the derived class.
7. Run the test.

8. Add a second derived class, this time wich contains a test randomly generating the values using the python package of your choice.
9. Replace the *print* functions/commands by *self.log.info()*.
10. Notice that the printed string is now preceeded by the Top-level HDL module name .
11. Compare your implementation with the solution. There are different good answers to the same problem!

6 Unit test methodology

Learning objectives

- Get familiar with unit tests using a Monitor-Model-Checker structure.

6.1 Introduction

6.2 Get familiar with a Monitor class

1. Open the Lab06_MMC_Sqrt.py file. It contains a copy of the *DataValidMonitor* class from the official cocotb git repository, from the matrix multiplier example, but with additional comments.
2. Carefully read the comments and class methods. There might be Python constructs you are unfamiliar with.
3. The function deciding when to sample signals is the *_run* method, which becomes a thread (coroutine).
4. This class does not require modifications.

This is not the only example of Monitor class. Others exist, but the principale is the same: capture waveforms under design-specific conditions, store them, and then provide a way to transfer the samples to a model or checker.

6.3 Adapt a module-level unit test class

1. Review the Monitor-Model-Checker figure from the presentation.
2. Get familiar with the *MMC_Sqrt* class in the Lab06_MMC_Sqrt.py file.
3. In this example, the constructor expects to receive a pointer to the instance in the top level design, not the dut (top level). Looking at the top VHDL module, we see it is named *dut.inst_square_root*, meaning the constructor will receive *dut.inst_square_root* as argument. This will be used later. No modifications to do here.
4. The constructor adds two monitor objects. There is a monitor for the read interface, and another one for the write interface.
5. Edit the lines below the comments indicating to do so, tying the monitor to the sqrt instance signals.
6. The start/stop methods manage the monitor and checker threads. Only read the two functions; no modifications are required.
7. The *model* method currently does nothing. Add code to predict the square root result using the method of your choice, returning an integer value.
8. The *_Check* method reads samples from the monitor class, calculates the predicted result with the model, and compares. Read the method.

9. You will notice that the *expected_inputs* and *actual* variables are python dictionaries, with elements named "Arguments" and "SqrtResult". These names are defined in the constructor, when creating the monitors. Can you find them?

6.4 Add a MMC object to the base environment

1. Open the Lab06_MainEnvironment.py file. This is mainly the solution from lab 5.
2. Notice the new *StartEnvironment* method, currently empty. It will run after the reset sequence is completed.
3. In *BuildEnvironment*, create a new class member, an instance of the *MMC_Sqrt* class, and assign the square root core instance to it (*dut.inst_square_root*).
4. In the *StartEnvironment* method, add a call to the MMC *start* method for the instance you just created.
5. Optionally, add a call to its *stop* method in the *postTest* section.
6. Launch the tests. If without errors, both tests will pass.

With this structure, if a design has multiple square root cores, separate MMC objects are each assigned one core to test, promoting code reuse. It also means different projects may reuse the same MMC class.

6.5 Surprising double bug?

1. In the MMC's model method, make any change to the expected result to force an assertion error.
2. Run the simulation.
3. Notice that both tests now fail, and not at the same place. The first is in the MMC class, the other is in the test assertion.
4. Open the waveform file. Remember, tests are run one after the other, not separately.
5. Find the reset signal toggle in the middle of the simulation. This is actually where the first test ends and the second begins for this case.
6. Notice that the UART transmitter is sending data even though the test changed. This is because the two simualtions are done one after the other, and because the UART core does not have a reset input signal!

Is this a real bug? Maybe, that depends. Of course, the problem is easily fixed by adding a reset to the UART core. On the other hand, if the intent is to have no reset signal on the UART, then the problem is with the verification environment.

To fix the environment, one way is to have the reset last longer, and also call the *uart_sink.clear()* method during the *StartEnvironment* stage. This will empty the sink of any data remaining from the previous test.