Basic CocoTB Tutorial For CARP

Introduction

I figured I'd go ahead and make a quick introduction/tutorial for CocoTB since the quickstart guide/installation guide on the CocoTB website is incomplete/outdated. This guide will go over the three basic steps of getting a CocoTB testbench set up and running: Installation, use, and running. If you have any questions beyond this guide, feel free to reach out on the Discord, preferably in the Backend channel.

Installation

For the first step, we will be following the provided CocoTB installation guide from their website at: docs.cocotb.org/en/stable/install.html
Follow this guide (using the installation for Linux) until the following step:

```
pip install "cocotb~=2.0"
```

We will come back to this later, as Linux doesn't like to play nice with the pip package installer. If you were to run it now, you would run into an error like:

```
error: externally managed environment
```

For the next step, we will overcome this problem by installing a virtual environment. The reason that Linux requires us to do this (or other, more difficult means) is to prevent things like mismatched dependency versions and help with reproducibility. To install a virtual environment, run the following commands in your terminal:

To ensure that a virtual environment can be installed:

```
sudo apt install python3-venv
```

Then, set up the virtual environment (a common name is *.venv*). Be sure to do this in your home directory (which can be accessed by typing *cd* into your terminal):

```
python3 -m venv venvname
```

To activate the virtual environment:

```
source venvname/bin/activate
```

Keep in mind that the *venvname* directory is the same as what you put in the previous command. Additionally, this is the command that you will enter to activate the virtual environment whenever you wish to install something using pip (which is only once for this tutorial). You should then see a prefix before your user in the command line, like so (keep in mind *venv* is simply the name I chose for my virtual environment):

Now, we can actually install CocoTB. Run the following command:

```
pip install "cocotb~=2.0"
```

This may take several moments, as CocoTB is a fairly large package, but this is the last step for installation.

Usage

For this next section, I will go over the two required parts of a CocoTB testbench, the testbench itself and the Makefile. Aside from being in Python and not (System) Verilog, CocoTB provides many nice features that developers can use to make writing testbenches easier AND more effective. If you wish to go into a deep(er) dive on those, feel free to check out the CocoTB documentation. However, this guide will give you the bare necessities for getting a working testbench.

For the testbench itself, there are several things that are required to be put into your program: Imports and interfaces (not the OOP kind).

For imports, this is fairly simple. Most people who have programmed Python before know how to do it, but for those who don't, imports are a way of including functions/features from beyond the scope of the program you're writing. For CocoTB, it's as simple as writing the following lines of code at the top of your file:

```
import cocotb
from cocotb.triggers import RisingEdge, Timer
from cocotb.clock import Clock
```

This will make sure that you can actually use the built-in CocoTB features. Now, for the other necessary step in writing a CocoTB testbench, interfacing your program with CocoTB. There are a couple of lines that you MUST include in your program to get CocoTB to actually run it:

Firstly, there should be one main function that has one argument: the DUT (design under test). You can think of the DUT as the instance of your RTL module you wrote. Above the function definition, you should tell CocoTB that the function is a test to be run. In all, the function declaration should look something like this:

```
@cocotb.test()
async def test_my_module(dut):
```

Now that we've instantiated the function, we can write a test. This is what will vary the most depending on what you're trying to verify, but there are a couple of things that they will have in common. Most modules more complicated than an adder will have a clock signal, so let's start there. You initialize a clock by calling the Clock function like so:

```
clock = Clock(dut.clk, 10, units="ns")
```

What this line does is tell CocoTB that the clock signal in your module (dut.clk) is used as a clock and has a period of 10 nanoseconds. Depending on the name of your clock, you will change the first argument, but it will always have *dut.* in front of its name.

Next, we need to tell CocoTB to start the clock signal. We accomplish this with the following:

```
cocotb.start_soon(clock.start())
```

This simply tells the testbench to start the clock.

Next, we will discuss how to access variables in your module. Everything exists within the DUT, so if we wanted to access a variable called *my_var*, here is how:

```
dut.my_var.value
```

These can both be read from and written to, just like any other standard variable in python. They also have the same properties as the variable in your module, so if your module says *my_var* is 6 bits, so is the above value.

Next, let's actually write the test part of the testbench. We want to ensure that the value our design is producing is the same as the value that we expect, so we need a way to compare them. This is where assert statements come in:

```
assert actual == expected, f"Mismatch: got {actual:#x}, expected
{expected:#x}"
```

This statement does several things: it compares between the values of actual and expected (which are instantiated earlier in the program), and if they aren't equal, then it prints an error message and stops the program. This is the workhorse in your testbench, and what enables you to stop looking at waveforms for correct values.

However, what if we need to wait for a signal? We don't want to test these values if they aren't ready yet. This is where the following line comes in:

```
await RisingEdge(dut.clk)
```

The await keyword stops your program until the condition is met. In this case, the program will stop until it reaches the rising edge of the next clock cycle. If your module takes 3 cycles to produce an output from an input, chain three of these together.

A full example of a CocoTB testbench can be found on my GitHub at:
https://github.com/rbloomfi/carp-core/blob/main/tests/tb_MUL/mul_test.py

Next, we will discuss the second file needed to perform a CocoTB test, the Makefile. A Makefile is a special kind of file that contains scripts that the shell can run. So, instead of the user having to input 20 different terminal commands, they can simply type one command, and the Makefile handles the rest. It is important to note that for CocoTB, **the Makefile and Python test file must be in the same directory, and the Makefile must be named *Makefile*** with that exact capitalization and no file extension.

A Makefile has some pretty wacky syntax, including tab characters being required in

certain spots, similar to Python. However, here is a simple Makefile for CocoTB use:

```
TOPLEVEL_LANG = verilog

RTL_PATH = ~/path/to/your/rtl/code/directory

VERILOG_SOURCES = $(RTL_PATH)/top_level.sv \
                  $(RTL_PATH)/rtl1.sv \
                  $(RTL_PATH)/rtl2.sv \
                  $(RTL_PATH)/rtl3.sv
TOPLEVEL = top_level.sv
MODULE = cocotb_file

SIM = icarus

include $(shell cocotb-config --makefiles)/Makefile.sim
```

Let's break this down:

The first line simply tells CocoTB what language you're working with, whether it be Verilog, VHDL, or something else. It doesn't distinguish between Verilog and System Verilog.

The RTL_PATH is simply the path to where your System Verilog files reside. If you don't know the path, but can open its location in the terminal, simply enter the command:

```
pwd
```

And copy-paste the result!

Further, the VERILOG_SOURCES section is a list of all your rtl files (most projects will have one). Note that the names of the files are appended to the path above.

Next, TOPLEVEL is the top-level module (woah) for your RTL code. Think of this as your main module, the one at the top of the hierarchy pyramid.

MODULE is the name of your CocoTB Python file (with the .py subtracted).

SIM is the simulator we are going to use, which, for now, is icarus.

The last line is some configuration information, and can just be copied and pasted.

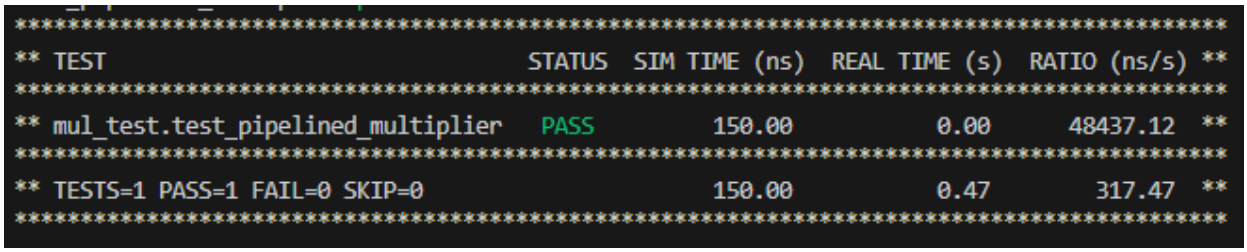
A working example of this Makefile (which I built the template from) can be found here: https://github.com/rbloomfi/carp-core/blob/main/tests/tb_MUL/Makefile

Running

Running a CocoTB test is fairly simple. All you have to do is make sure that your terminal is in the right directory (where the Makefile and testbench are) and type:

```
make
```

And your CocoTB test should run! Here is what the output should look like for a passed test:



If any errors occur or if you have any questions, feel free to reach out on Discord!