Coroutine Co-Simulation Test Bench (COCOTB)

Created By

NIGIL MOHRA

COCOTB

Introduction

1. COCOTB is an opensource framework hosted on GitHub.
2. COCOTB is a library for digital logic verification in Python.
3. Provides Python interface to control standard RTL Simulator.
4. Offers an alternative to using Verilog/System Verilog/VHDL framework for verification.

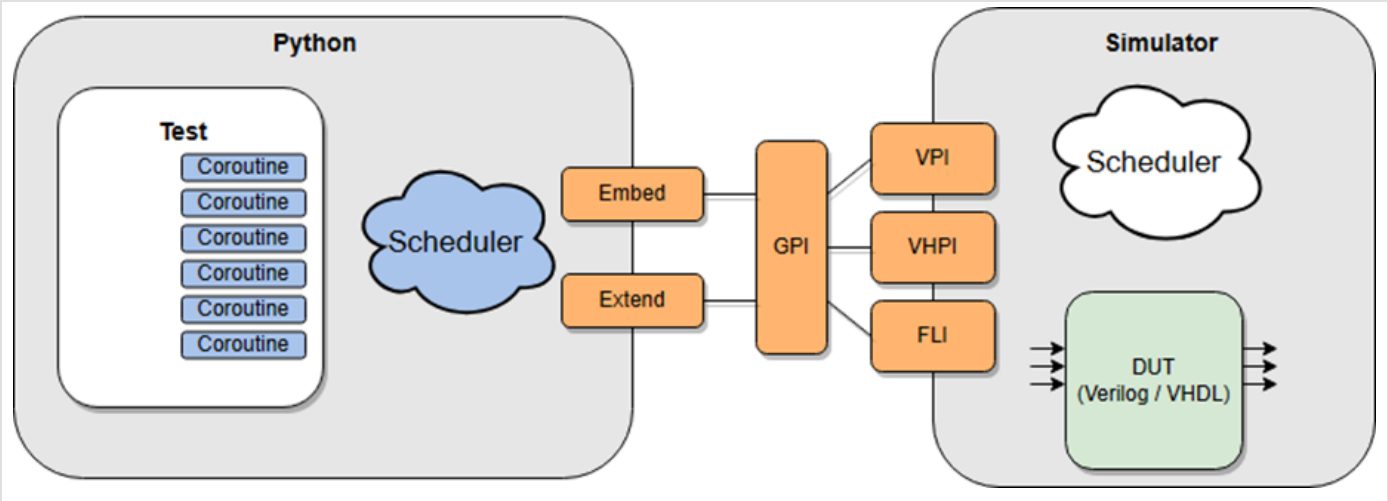

Figure 1. Architecture of Cocotb

Problems with Current Testbenches

1. Hardware design (Synthesizable) and verification (TB are Non Synthesizable) are different.
2. Using the same language for both might not be optimal. Higher level language concepts (like OOP) are useful when writing complex testbenches.
3. Add higher level programming features to the hardware description language. System Verilog is the first approach: simulation-only OOP language features.
4. UVM (Universal Verification Methodology) libraries written in System Verilog.
5. UVM is a well-defined set of coding guidelines with well-defined testbench structure, It's written in System Verilog and comes with System Verilog base class library for creating advanced reusable verification TB's.
6. Use an existing high level object oriented general purpose language for verification.
7. But, there are problems with this,
    1. The System Verilog is a fairly complex programming language.
    2. Specification is almost a thousand pages long.
    3. There are 221 keywords in the language, to 83 in C++.
8. It is a powerful, but it takes some time to master.
9. There is also SV-UVM but, it also has more than 300 keywords and though it is powerful, but it takes some time to master.

Verification using Python

1. COCOTB was developed by **Chris Higgs** and **Stuart Hodgson**, tried a different approach.
2. Use-a high level, general purpose, OOP language for developing test benches, they picked Python as their language of choice.
3. Python is simple (only 23 keywords) and easy to learn, but very powerful
4. Python has a large standard library and a huge ecosystem; lots of existing libraries.
5. Python is well documented and popular: lots of resources online.

Cosimulation (Triggers)

1. Design and TB simulated independently: this is called Cosimulation.
2. Communication through VPI/VHPI interfaces, generated by COCOTB "triggers".
3. When a trigger is yield/await, the testbench waits until the triggered condition is satisfied before resuming execution.

Coroutines

1. In COCOTB coroutines are just functions that obey two properties.
    1. Decorated using the @cocotb.coroutine decorator
    2. Contains at least one yield/await statement yielding another coroutine or trigger.
    3. Coroutine can be yielded, but they can also be forked to run in parallel. This allows the creation of something like a Verilog always block.

Available Triggers

Name	Type	Function
Timer	Time, Unit	Waits for a certain amount of simulation time to pass
Edge	Signal	Waits for a signal to change state (rising or falling edge)
Rising Edge	Signal	Waits for the rising edge of a signal
Falling Edge	Signal	Waits for the falling edge of a signal
Clock Cycles	Signal, Num	Waits for some number of clocks (transitions from 0 to 1)

Examples

Combinational Circuit

```
// MAIN MODULE | COMBINATIONAL CIRCUIT
module mux_2x1
{
        input a, b, sel,
        output y
};

assign y = sel ? a : b;

initial
begin
        $dumpfile('dump.vcd');
        $dumpvars(0, mux_2x1);
```

```
end
endmodule
```

```
# MAKEFILE
SIM ?= icarus
TOPLEVEL_LANG ?= verilog
PWD=$(shell pwd)
VERILOG_SOURCE = $(PWD)/mux.v
TOPLEVEL := mux_2x1
MODULE := test_mux
include $(shell cocotb-config --makerfiles)/Makefile.sim
```

```
# COCOTB TESTBENCH
import cocotb
from cocotb.triggers import Timer
from cocotb.result import TestFailure

@cocotb.test()
asyn def mux_test(dut):

        dut._log.info('start of test!')
        await Timer(1, 'ns')

        dut.a.value = 0
        dut.b.value = 0

        dut._log.info('Drive 0 to a & b inputs of 2x1 mux')
        await Timer(1, 'ns')

        dut.a.value = 1
        dut.sel.value = 1

        dut._log.info('Drive 1 to a & sel inputs of 2x1 mux')
        await Timer(1, 'ns')

        if dut.y.value != 1:
                raise TestFailure('Result is incorrect. %s !=1' %str(dut.y))
        else:
                dut._log.info('PASS !)

        dut.sel.value = 0
        dut.log.info('Drive 0 to sel inputs of 2x1 mux')

        if dut.y.value != 0:
                raise TestFailure('Result is incorrect. %s !=1' %str(dut.y))
        else:
                dut._log.info('PASS !)
```

Compiling and Running

1. The `make` command is typed in the root folder to run the simulation.
2. To view the waveform type `gtkwave` and go to the root folder and select the dump file.

Sequential Circuit

```
// MAIN MODULE | SEQUENTIAL CIRCUITS
module dff_rtl
```

```
{
        input clk, d,
        output reg q
};

always @(posedge clk)
begin
        q <= d;
end

initial
begin
        $dumpfile('dump.vcd');
        $dumpvars(l, dff_rtl);
end
endmodule
```

```
# MAKEFILE
SIM ?= icarus
TOPLEVEL_LANG ?= verilog
PWD=$(shell pwd)
VERILOG_SOURCE = $(PWD)/dff.v
TOPLEVEL := dff_rtl
MODULE := test_dff
COCOTB_HDL_TIMEUNIT = 1ns
COCOTB_HDL_TIMEPRECISION = 1ns
include $(shell cocotb-config --makefiles)/Makefile.sim
```

```
# COCOTB TESTBENCH
import cocotb
import random
from cocotb.clock import Clock
from cocotb.triggers import RisingEdge, FallingEdge, Timer
from cocotb.result import TestFailure

@cocotb.test()
async def test_dff(dut):

        dut._log.info('Start of the test here')

        cocotb.fork(Clock(dut.clk, 10, 'ns').start())
        dut._log.info('Generating a clk of 10ns')

        for i in range(10):
                dut.d.value = random.randint(0,1)
                await FallingEdge(dut.clk)

        if dut.q.value != dut.d.value:
                raise TestFailure('Incorect %s dut.q.value != dut.q.value')
                print('random value of d is', dut.d.value.binstr)
                print('value of output is q is' dut.q.value.binstr)
        else
                dut._log,info('Correct')
                print('random value of d is', dut.d.value.binstr)
                print('value of output is q is' dut.q.value.binstr)
        dut._log.info('End of test here')
```