cocotb Documentation

Release 1.0

PotentialVentures

Nov 11, 2017

Contents

1 Introduction **3**
- 1.1 What is cocotb? . . . 3
- 1.2 How is Cocotb different? . . . 3
- 1.3 How does Cocotb work? . . . 4
- 1.4 Contributors . . . 4

2 Quickstart Guide **5**
- 2.1 Installing cocotb . . . 5
- 2.2 Using Cocotb . . . 7

3 Build options and Environment Variables **11**
- 3.1 Make System . . . 11
- 3.2 Environment Variables . . . 12

4 Coroutines **15**

5 Triggers **17**
- 5.1 Simulation Timing . . . 17
- 5.2 Signal related . . . 17
- 5.3 Python Triggers . . . 18

6 Library Reference **19**
- 6.1 Test Results . . . 19
- 6.2 Writing and Generating tests . . . 19
- 6.3 Interacting with the Simulator . . . 21
- 6.4 Testbench Structure . . . 23

7 Tutorial: Endian Swapper **25**
- 7.1 Design . . . 25
- 7.2 Testbench . . . 25

8 Tutorial: Ping **29**
- 8.1 Architecture . . . 29
- 8.2 Implementation . . . 29
- 8.3 Further work . . . 31

9 Tutorial: Driver Cosimulation **33**
- 9.1 Difficulties with Driver Co-simulation . . . 33

9.2 Cocotb infrastructure . . . 34
9.3 Implementation . . . 34
9.4 Further Work . . . 35

10 Roadmap **37**

11 Simulator Support **39**
11.1 Icarus . . . 39
11.2 Synopsys VCS . . . 39
11.3 Aldec Riviera-PRO . . . 39
11.4 Mentor Questa . . . 39
11.5 Cadence Incisive . . . 39

12 Indices and tables **41**

Contents:

CHAPTER 1

Introduction

1.1 What is cocotb?

Cocotb is a *COroutine* based *COsimulation TestBench* environment for verifying VHDL/Verilog RTL using Python.

Cocotb is completely free, open source (under the BSD License) and hosted on GitHub.

Cocotb requires a simulator to simulate the RTL. Simulators that have been tested and known to work with Cocotb:

Linux Platforms

- Icarus Verilog
- Aldec Riviera-PRO
- Synopsys VCS
- Cadence Incisive
- Mentor Modelsim

Windows Platform

- Icarus Verilog
- Aldec Riviera-PRO
- Mentor Modelsim

Cocotb can be used live in a web-browser using EDA Playground.

1.2 How is Cocotb different?

Cocotb encourages the same philosophy of design re-use and randomised testing as UVM, however is implemented in Python rather than SystemVerilog.

In Cocotb VHDL/Verilog/SystemVerilog are only used for the synthesisable design.

Cocotb has built-in support for integrating with the Jenkins continuous integration system.

Cocotb was specifically designed to lower the overhead of creating a test.

Cocotb automatically discovers tests so that no additional step is required to add a test to a regression.

All verification is done using Python which has various advantages over using SystemVerilog or VHDL for verification:

- Writing Python is **fast** - it's a very productive language
- It's **easy** to interface to other languages from Python
- Python has a huge library of existing code to **re-use** like packet generation libraries.
- Python is **interpreted**. Tests can be edited and re-run them without having to recompile the design or exit the simulator GUI.
- Python is **popular** - far more engineers know Python than SystemVerilog or VHDL

1.3 How does Cocotb work?

1.3.1 Overview

A typical cocotb testbench requires no additional RTL code. The Design Under Test (DUT) is instantiated as the toplevel in the simulator without any wrapper code. Cocotb drives stimulus onto the inputs to the DUT (or further down the hierarchy) and monitors the outputs directly from Python.

A test is simply a Python function. At any given time either the simulator is advancing time or the Python code is executing. The **yield** keyword is used to indicate when to pass control of execution back to the simulator. A test can spawn multiple coroutines, allowing for independent flows of execution.

1.4 Contributors

Cocotb was developed by Potential Ventures with the support of Solarflare Communications Ltd and contributions from Gordon McGregor and Finn Grimwood (see contributers for full list of contributions).

CHAPTER 2

Quickstart Guide

2.1 Installing cocotb

2.1.1 Pre-requisites

Cocotb has the following requirements:

- Python 2.6+
- Python-dev packages
- GCC and associated development packages
- GNU Make
- A Verilog simulator

Internal development is performed on Linux Mint 17 (x64). We also use Redhat 6.5(x64). Other Redhat and Ubuntu based distributions (x32 and x64) should work too but due fragmented nature of Linux we can not test everything. Instructions are provided for the main distributions we use.

2.1.2 Linux native arch installation

Ubuntu based installation

```
$> sudo apt-get install git make gcc g++ swig python-dev
```

This will allow building of the Cocotb libs for use with a 64 bit native simulator. If a 32 bit simulator is being used then additional steps to install 32bit development libraries and python are needed.

Redhat based installation

```
$> sudo yum install gcc gcc-c++ libstdc++-devel swig python-devel
```

This will allow building of the Cocotb libs for use with a 64 bit native simulator. If a 32 bit simulator is being used then additional steps to install 32bit development libraries and python are needed.

2.1.3 32 bit Python

Additional development libraries are needed for building 32bit python on 64 bit systems.

Ubuntu based installation

```
$> sudo apt-get install libx32gcc1 gcc-4.8-multilib lib32stdc++-4.8-dev
```

Replace 4.8 with the version of gcc that was installed on the system in the step above. Unlike on Redhat where 32 bit python can co-exist with native python ubuntu requires the source to be downloaded and built.

Redhat based installation

```
$> sudo yum install glibc.i686 glibc-devel.i386 libgcc.i686 libstdc++-devel.i686
```

Specific releases can be downloaded from https://www.python.org/downloads/ .

```
$> wget https://www.python.org/ftp/python/2.7.9/Python-2.7.9.tgz
$> tar xvf Python-2.7.9.tgz
$> cd Python-2.7.9
$> export PY32_DIR=/opt/pym32
$> ./configure CC="gcc -m32" LDFLAGS="-L/lib32 -L/usr/lib32 -Lpwd/lib32 -Wl,-rpath,/
↪lib32 -Wl,-rpath,$PY32_DIR/lib" --prefix=$PY32_DIR --enable-shared
$> make
$> sudo make install
```

Cocotb can now be built against 32bit python by setting the architecture and placing the 32bit python ahead of the native version in the path when running a test

```
$> export PATH=/opt/pym32/bin
$> cd <cocotb_dir>
$> ARCH=i686 make
```

2.1.4 Windows 7 installation

Recent work has been done with the support of the Cocotb community to enable Windows support using the MinGW/Msys environment. Download the MinGQ installer from.

http://sourceforge.net/projects/mingw/files/latest/download?source=files .

Run the GUI installer and specify a directory you would like the environment installed in. The installer will retrieve a list of possible packages, when this is done press continue. The MinGW Installation Manager is then launched.

The following packages need selecting by checking the tick box and selecting “Mark for installation”

```
Basic Installation
  -- mingw-developer-tools
  -- mingw32-base
  -- mingw32-gcc-g++
  -- msys-base
```

From the Installation menu then select “Apply Changes”, in the next dialog select “Apply”.

When installed a shell can be opened using the “msys.bat” file located under the <install_dir>/msys/1.0/

Python can be downloaded from https://www.python.org/ftp/python/2.7.9/python-2.7.9.msi, other versions of python can be used as well. Run the installer and download to your chosen location.

It is beneficial to add the path to Python to the windows system PATH variable so it can be used easily from inside Msys.

Once inside the Msys shell commands as given here will work as expected.

2.1.5 MAC Packages

You need a few packages installed to get cocotb running on mac. Installing a package manager really helps things out here.

Brew_ seems to be the most popular, so we'll assume you have that installed. .. _Brew: http://www.brew.sh

2.1.6 Running an example

```
$> git clone https://github.com/potentialventures/cocotb
$> cd cocotb/examples/endian_swapper/tests
$> make
```

To run a test using a different simulator:

```
$> make SIM=vcs
```

2.1.7 Running a VHDL example

The endian swapper example includes both a VHDL and Verilog RTL implementation. The Cocotb testbench can execute against either implementation using VPI for Verilog and VHPI/FLI for VHDL. To run the test suite against the VHDL implementation use the following command (a VHPI or FLI capable simulator must be used):

```
$> make SIM=aldec TOPLEVEL_LANG=vhdl
```

2.2 Using Cocotb

A typical Cocotb testbench requires no additional RTL code. The Design Under Test (DUT) is instantiated as the toplevel in the simulator without any wrapper code. Cocotb drives stimulus onto the inputs to the DUT and monitors the outputs directly from Python.

2.2.1 Creating a Makefile

To create a Cocotb test we typically have to create a Makefile. Cocotb provides rules which make it easy to get started. We simply inform Cocotb of the source files we need compiling, the toplevel entity to instantiate and the python test script to load.

```
VERILOG_SOURCES = $(PWD)/submodule.sv $(PWD)/my_design.sv
TOPLEVEL=my_design
MODULE=test_my_design
include $(COCOTB)/makefiles/Makefile.inc
include $(COCOTB)/makefiles/Makefile.sim
```

We would then create a file called `test_my_design.py` containing our tests.

2.2.2 Creating a test

The test is written in Python. Assuming we have a toplevel port called `clk` we could create a test file containing the following:

```
import cocotb
from cocotb.triggers import Timer

@cocotb.test()
def my_first_test(dut):
    """
    Try accessing the design
    """
    dut._log.info("Running test!")
    for cycle in range(10):
        dut.clk = 0
        yield Timer(1000)
        dut.clk = 1
        yield Timer(1000)
    dut._log.info("Running test!")
```

This will drive a square wave clock onto the `clk` port of the toplevel.

2.2.3 Accessing the design

When cocotb initialises it finds the top-level instantiation in the simulator and creates a handle called **dut**. Top-level signals can be accessed using the "dot" notation used for accessing object attributes in Python. The same mechanism can be used to access signals inside the design.

```
# Get a reference to the "clk" signal on the top-level
clk = dut.clk

# Get a reference to a register "count" in a sub-block "inst_sub_block"
count = dut.inst_sub_block.count
```

2.2.4 Assigning values to signals

Values can be assigned to signals using either the .value property of a handle object or using direct assignment while traversing the hierarchy.

```
# Get a reference to the "clk" signal and assign a value
clk = dut.clk
clk.value = 1

# Direct assignment through the hierarchy
dut.input_signal = 12

# Assign a value to a memory deep in the hierarchy
dut.sub_block.memory.array[4] = 2
```

2.2.5 Reading values from signals

Accessing the .value property of a handle object will return a `BinaryValue` object. Any unresolved bits are preserved and can be accessed using the binstr attribute, or a resolved integer value can be accessed using the value attribute.

```
>>> # Read a value back from the dut
>>> count = dut.counter.value
>>>
>>> print count.binstr
1X1010
>>> # Resolve the value to an integer (X or Z treated as 0)
>>> print count.integer
42
```

We can also cast the signal handle directly to an integer:

```
>>> print int(dut.counter)
42
```

2.2.6 Parallel and sequential execution of coroutines

```
@cocotb.coroutine
def reset_dut(reset_n, duration):
    reset_n <= 0
    yield Timer(duration)
    reset_n <= 1
    reset_n._log.debug("Reset complete")

@cocotb.test()
def parallel_example(dut):
    reset_n = dut.reset

    # This will call reset_dut sequentially
    # Execution will block until reset_dut has completed
    yield reset_dut(reset_n, 500)
    dut._log.debug("After reset")

    # Call reset_dut in parallel with this coroutine
    reset_thread = cocotb.fork(reset_dut(reset_n, 500)

    yield Timer(250)
    dut._log.debug("During reset (reset_n = %s)" % reset_n.value)

    # Wait for the other thread to complete
    yield reset_thread.join()
    dut._log.debug("After reset")
```

2.2.7 Creating a test

```
import cocotb
from cocotb.triggers import Timer

@cocotb.test(timeout=None)
```

```
def my_first_test(dut):

    # drive the reset signal on the dut
    dut.reset_n <= 0
    yield Timer(12345)
    dut.reset_n <= 1
```

CHAPTER 3

Build options and Environment Variables

3.1 Make System

Makefiles are provided for a variety of simulators in cocotb/makefiles/simulators. The common Makefile cocotb/makefiles/Makefile.sim includes the appropriate simulator makefile based on the contents of the SIM variable.

3.1.1 Make Targets

Makefiles define two targets, 'regression' and 'sim', the default target is sim.

Both rules create a results file in the calling directory called 'results.xml'. This file is a JUnit-compatible output file suitable for use with Jenkins. The 'sim' targets unconditionally re-runs the simulator whereas the regression target only re-builds if any dependencies have changed.

3.1.2 Make phases

Typically the makefiles provided with Cocotb for various simulators use a separate *compile* and *run* target. This allows for a rapid re-running of a simulator if none of the RTL source files have changed and therefore the simulator does not need to recompile the RTL.

3.1.3 Make Variables

GUI

Set this to 1 to enable the GUI mode in the simulator (if supported).

SIM

Selects which simulator Makefile to use. Attempts to include a simulator specific makefile from cocotb/makefiles/makefile.$(SIM)

VERILOG_SOURCES

A list of the Verilog source files to include.

VHDL_SOURCES

A list of the VHDL source files to include.

COMPILE_ARGS

Any arguments or flags to pass to the compile stage of the simulation. Only applies to simulators with a separate compilation stage (currently Icarus and VCS).

SIM_ARGS

Any arguments or flags to pass to the execution of the compiled simulation. Only applies to simulators with a separate compilation stage (currently Icarus, VCS and GHDL).

EXTRA_ARGS

Passed to both the compile and execute phases of simulators with two rules, or passed to the single compile and run command for simulators which don't have a distinct compilation stage.

CUSTOM_COMPILE_DEPS

Use to add additional dependencies to the compilation target; useful for defining additional rules to run pre-compilation or if the compilation phase depends on files other than the RTL sources listed in **VERILOG_SOURCES** or **VHDL_SOURCES**.

CUSTOM_SIM_DEPS

Use to add additional dependencies to the simulation target.

COCOTB_NVC_TRACE

Set this to 1 to enable display VHPI trace when using nvc VHDL simulator.

3.2 Environment Variables

3.2.1 TOPLEVEL

Used to indicate the instance in the hierarchy to use as the DUT. If this isn't defined then the first root instance is used.

3.2.2 RANDOM_SEED

Seed the Python random module to recreate a previous test stimulus. At the beginning of every test a message is displayed with the seed used for that execution:

```
INFO     cocotb.gpi                                  __init__.py:89   in _initialise_
↪testbench           Seeding Python random module with 1377424946
```

To recreate the same stimulis use the following:

```
make RANDOM_SEED=1377424946
```

3.2.3 COCOTB_ANSI_OUTPUT

Use this to override the default behaviour of annotating cocotb output with ANSI colour codes if the output is a terminal (isatty()).

COCOTB_ANSI_OUTPUT=1 forces output to be ANSI regardless of the type stdout

COCOTB_ANSI_OUTPUT=0 supresses the ANSI output in the log messages

3.2.4 MODULE

The name of the module(s) to search for test functions. Multiple modules can be specified using a comma-separated list.

3.2.5 TESTCASE

The name of the test function(s) to run. If this variable is not defined cocotb discovers and executes all functions decorated with @cocotb.test() decorator in the supplied modules.

Multiple functions can be specified in a comma-separated list.

CHAPTER 4

Coroutines

Testbenches built using Cocotb use coroutines. While the coroutine is executing the simulation is paused. The coroutine uses the `yield` keyword to pass control of execution back to the simulator and simulation time can advance again.

Typically coroutines `yield` a `Trigger` object which indicates to the simulator some event which will cause the coroutine to be woken when it occurs. For example:

```
@cocotb.coroutine
def wait_10ns():
    cocotb.log.info("About to wait for 10ns")
    yield Timer(10000)
    cocotb.log.info("Simulation time has advanced by 10 ns")
```

Coroutines may also yield other coroutines:

```
@cocotb.coroutine
def wait_100ns():
    for i in range(10):
        yield wait_10ns()
```

Coroutines may also yield a list of triggers to indicate that execution should resume if *any* of them fires:

```
@cocotb.coroutine
def packet_with_timeout(monitor, timeout):
    """Wait for a packet but timeout if nothing arrives"""
    yield [Timer(timeout), monitor.wait_for_recv()]
```

The trigger that caused execution to resume is passed back to the coroutine, allowing them to distinguish which trigger fired:

```
@cocotb.coroutine
def packet_with_timeout(monitor, timeout):
    """Wait for a packet but timeout if nothing arrives"""
    tout_trigger = Timer(timeout)
    result = yield [tout_trigger, monitor.wait_for_recv()]
```

```
    if result is tout_trigger:
        raise TestFailure("Timed out waiting for packet")
```

CHAPTER 5

Triggers

Triggers are used to indicate when the scheduler should resume coroutine execution. Typically a coroutine will **yield** a trigger or a list of triggers.

5.1 Simulation Timing

5.1.1 Timer(time)

Registers a timed callback with the simulator to continue execution of the coroutine after a specified simulation time period has elapsed.

Todo

What is the behaviour if time=0?

5.1.2 ReadOnly()

Registers a callback which will continue execution of the coroutine when the current simulation timestep moves to the ReadOnly phase. Useful for monitors which need to wait for all processes to execute (both RTL and cocotb) to ensure sampled signal values are final.

5.2 Signal related

5.2.1 Edge(signal)

Registers a callback that will continue execution of the coroutine on any value change of a signal.

Todo

Behaviour for vectors

5.2.2 RisingEdge(signal)

Registers a callback that will continue execution of the coroutine on a transition from 0 to 1 of signal.

5.2.3 FallingEdge(signal)

Registers a callback that will continue execution of the coroutine on a transition from 1 to 0 of signal.

5.2.4 ClockCycles(signal, num_cycles)

Registers a callback that will continue execution of the coroutine when num_cycles transistions from 0 to 1 have occured.

5.3 Python Triggers

5.3.1 Event()

Can be used to synchronise between coroutines. yielding Event.wait() will block the coroutine until Event.set() is called somewhere else.

5.3.2 Join(coroutine)

Will block the coroutine until another coroutine has completed.

CHAPTER 6

Library Reference

6.1 Test Results

The following exceptions can be raised at any point by any code and will terminate the test:

class cocotb.result.**TestComplete**(**args*, ***kwargs*)
Exceptions are used to pass test results around.

class cocotb.result.**TestError**(**args*, ***kwargs*)

class cocotb.result.**TestFailure**(**args*, ***kwargs*)

class cocotb.result.**TestSuccess**(**args*, ***kwargs*)

6.2 Writing and Generating tests

class cocotb.**test**(*timeout=None*, *expect_fail=False*, *expect_error=False*, *skip=False*)
Decorator to mark a function as a test

All tests are coroutines. The test decorator provides some common reporting etc, a test timeout and allows us to mark tests as expected failures.

KWargs:

timeout: (int) value representing simulation timeout (not implemented)

expect_fail: (bool): Don't mark the result as a failure if the test fails

expect_error: (bool): Don't make the result as an error if an error is raised This is for cocotb internal regression use

skip: (bool): Don't execute this test as part of the regression

class cocotb.**coroutine**(*func*)
Decorator class that allows us to provide common coroutine mechanisms:

log methods will will log to cocotb.coroutines.name

join() method returns an event which will fire when the coroutine exits

class cocotb.regression.**TestFactory**(*test_function*, **args*, ***kwargs*)

Used to automatically generate tests.

Assuming we have a common test function that will run a test. This test function will take keyword arguments (for example generators for each of the input interfaces) and generate tests that call the supplied function.

This Factory allows us to generate sets of tests based on the different permutations of the possible arguments to the test function.

For example if we have a module that takes backpressure and idles and have some packet generations routines gen_a and gen_b.

```
>>> tf = TestFactory(run_test)
>>> tf.add_option('data_in', [gen_a, gen_b])
>>> tf.add_option('backpressure', [None, random_backpressure])
>>> tf.add_option('idles', [None, random_idles])
>>> tf.generate_tests()
```

We would get the following tests:

- gen_a with no backpressure and no idles
- gen_a with no backpressure and random_idles
- gen_a with random_backpressure and no idles
- gen_a with random_backpressure and random_idles
- gen_b with no backpressure and no idles
- gen_b with no backpressure and random_idles
- gen_b with random_backpressure and no idles
- gen_b with random_backpressure and random_idles

The tests are appended to the calling module for auto-discovery.

Tests are simply named test_function_N. The docstring for the test (hence the test description) includes the name and description of each generator.

add_option(*name*, *optionlist*)

Add a named option to the test.

Args:

name (string): name of the option. passed to test as a keyword argument

optionlist (list): A list of possible options for this test knob

generate_tests(*prefix=''*, *postfix=''*)

Generates exhasutive set of tests using the cartesian product of the possible keyword arguments.

The generated tests are appended to the namespace of the calling module.

Args:

prefix: Text string to append to start of test_function name when naming generated test cases. This allows reuse of a single test_function with multiple TestFactories without name clashes.

postfix: Text string to append to end of test_function name when naming generated test cases. This allows reuse of a single test_function with multiple TestFactories without name clashes.

6.3 Interacting with the Simulator

class cocotb.binary.**BinaryValue**(*value=None*, *bits=None*, *bigEndian=True*, *binaryRepresentation=0*)

Represenatation of values in binary format.

The underlying value can be set or accessed using three aliasing attributes

- BinaryValue.integer is an integer
- BinaryValue.signed_integer is a signed integer
- BinaryValue.binstr is a string of "01xXzZ"
- BinaryValue.buff is a binary buffer of bytes
- BinaryValue.value is an integer *** deprecated ***

For example:

```
>>> vec = BinaryValue()
>>> vec.integer = 42
>>> print vec.binstr
101010
>>> print repr(vec.buff)
'*'
```

assign(*value*)

Decides how best to assign the value to the vector

We possibly try to be a bit too clever here by first of all trying to assign the raw string as a binstring, however if the string contains any characters that aren't 0, 1, X or Z then we interpret the string as a binary buffer...

binstr

Access to the binary string

buff

Access to the value as a buffer

get_binstr()

Attribute binstr is the binary representation stored as a string of 1s and 0s

get_buff()

Attribute self.buff represents the value as a binary string buffer

```
>>> "0100000100101111".buff == "A/"
True
```

get_value()

value is an integer representaion of the underlying vector

get_value_signed()

value is an signed integer representaion of the underlying vector

integer

Integer access to the value

signed_integer

Signed integer access to the value

value

Integer access to the value *** deprecated ***

class cocotb.bus.**Bus**(*entity*, *name*, *signals*, *optional_signals=[]*)
Wraps up a collection of signals

Assumes we have a set of signals/nets named:

entity.bus_name_signal

for example a bus named "stream_in" with signals ["valid", "data"] dut.stream_in_valid
dut.stream_in_data

TODO: Support for struct/record ports where signals are member names

drive(*obj*, *strict=False*)
Drives values onto the bus.

Args:

obj (any type) [object with attribute names that match the bus] signals

Kwargs: strict (bool) : Check that all signals are being assigned

Raises: AttributeError

6.3.1 Triggers

Triggers are used to indicate when the scheduler should resume coroutine execution. Typically a coroutine will **yield** a trigger or a list of triggers.

Simulation Timing

class cocotb.triggers.**Timer**(*time_ps*, *units=None*)
Execution will resume when the specified time period expires

Consumes simulation time

class cocotb.triggers.**ReadOnly**

Signal related

class cocotb.triggers.**Edge**

class cocotb.triggers.**RisingEdge**

Python Triggers

class cocotb.triggers.**Event**(*name=''*)
Event to permit synchronisation between two coroutines

clear()
Clear this event that's fired.

Subsequent calls to wait will block until set() is called again

set(*data=None*)
Wake up any coroutines blocked on this event

wait()
This can be yielded to block this coroutine until another wakes it

class cocotb.triggers.**Lock**(*name=''*)
Lock primitive (not re-entrant)

acquire()
This can be yielded to block until the lock is acquired

class cocotb.triggers.**Join**

6.4 Testbench Structure

class cocotb.drivers.**Driver**
Class defining the standard interface for a driver within a testbench

The driver is responsible for serialising transactions onto the physical pins of the interface. This may consume simulation time.

append(*transaction*, *callback=None*, *event=None*)
Queue up a transaction to be sent over the bus.

Mechanisms are provided to permit the caller to know when the transaction is processed

callback: optional function to be called when the transaction has been sent

event: event to be set when the tansaction has been sent

clear()
Clear any queued transactions without sending them onto the bus

class cocotb.monitors.**Monitor**(*callback=None*, *event=None*)

class cocotb.scoreboard.**Scoreboard**(*dut*, *reorder_depth=0*, *fail_immediately=True*)
Generic scoreboarding class

We can add interfaces by providing a monitor and an expected output queue

The expected output can either be a function which provides a transaction or a simple list containing the expected output.

TODO: Statistics for end-of-test summary etc.

add_interface(*monitor*, *expected_output*, *compare_fn=None*, *reorder_depth=0*, *strict_type=True*)
Add an interface to be scoreboarded.

Provides a function which the monitor will callback with received transactions

Simply check against the expected output.

compare(*got*, *exp*, *log*, *strict_type=True*)
Common function for comparing two transactions.

Can be re-implemented by a subclass.

result
Determine the test result, do we have any pending data remaining?

CHAPTER 7

Tutorial: Endian Swapper

In this tutorial we'll use some of the built-in features of Cocotb to quickly create a complex testbench.

Note: All the code and sample output from this example are available on EDA Playground

For the impatient this tutorial is provided as an example with Cocotb. You can run this example from a fresh checkout:

```
cd examples/endian_swapper/tests
make
```

7.1 Design

We have a relatively simplistic RTL block called the endian_swapper. The DUT has three interfaces, all conforming to the Avalon standard:

The DUT will swap the endianness of packets on the Avalon-ST bus if a configuration bit is set. For every packet arriving on the "stream_in" interface the entire packet will be endian swapped if the configuration bit is set, otherwise the entire packet will pass through unmodified.

7.2 Testbench

To begin with we create a class to encapsulate all the common code for the testbench. It is possible to write directed tests without using a testbench class however to encourage code re-use it is good practice to create a distinct class.

```
class EndianSwapperTB(object):

    def __init__(self, dut):
        self.dut = dut
```

```
        self.stream_in  = AvalonSTDriver(dut, "stream_in", dut.clk)
        self.stream_out = AvalonSTMonitor(dut, "stream_out", dut.clk)
        self.csr = AvalonMaster(dut, "csr", dut.clk)

        self.expected_output = []
        self.scoreboard = Scoreboard(dut)
        self.scoreboard.add_interface(self.stream_out, self.expected_output)

        # Reconstruct the input transactions from the pins and send them to our 'model
↪'
        self.stream_in_recovered = AvalonSTMonitor(dut, "stream_in", dut.clk,␣
↪callback=self.model)
```

With the above code we have created a testbench with the following structure:

If we inspect this line-by-line:

```
self.stream_in  = AvalonSTDriver(dut, "stream_in", dut.clk)
```

Here we're creating an AvalonSTDriver instance. The constructor requires 3 arguments - a handle to the entity containing the interface (**dut**), the name of the interface (**stream_in**) and the associated clock with which to drive the interface (**dut.clk**). The driver will auto-discover the signals for the interface, assuming that they follow the naming convention **interface_name** _ *signal*.

In this case we have the following signals defined for the **stream_in** interface:

Name	Type	Description (from Avalon Specification)
stream_in_data	data	The data signal from the source to the sink
stream_in_empty	empty	Indicates the number of symbols that are empty during cycles that contain the end of a packet
stream_in_valid	valid	Asserted by the source to qualify all other source to sink signals
stream_in_startofpacket	startof-packet	Asserted by the source to mark the beginning of a packet
stream_in_endofpacket	endof-packet	Asserted by the source to mark the end of a packet
stream_in_ready	ready	Asserted high to indicate that the sink can accept data

By following the signal naming convention the driver can find the signals associated with this interface automatically.

```
self.stream_out = AvalonSTMonitor(dut, "stream_out", dut.clk)
self.csr = AvalonMaster(dut, "csr", dut.clk)
```

We do the same to create the monitor on **stream_out** and the CSR interface.

```
self.expected_output = []
self.scoreboard = Scoreboard(dut)
self.scoreboard.add_interface(self.stream_out, self.expected_output)
```

The above lines create a Scoreboard instance and attach it to the **stream_out** monitor instance. The scoreboard is used to check that the DUT behaviour is correct. The call to **add_interface** takes a Monitor instance as the first argument and the second argument is a mechanism for describing the expected output for that interface. This could be a callable function but in this example a simple list of expected transactions is sufficient.

```
# Reconstruct the input transactions from the pins and send them to our 'model'
self.stream_in_recovered = AvalonSTMonitor(dut, "stream_in", dut.clk, callback=self.
↪model)
```

Finally we create another Monitor instance, this time connected to the **stream_in** interface. This is to reconstruct the transactions being driven into the DUT. It's good practice to use a monitor to reconstruct the transactions from the pin interactions rather than snooping them from a higher abstraction layer as we can gain confidence that our drivers and monitors are functioning correctly. We also pass the keyword argument **callback** to the monitor constructor which will result in the supplied function being called for each transaction seen on the bus with the transaction as the first argument. Our model function is quite straightforward in this case - we simply append the transaction to the expected output list and increment a counter:

```
def model(self, transaction):
    """Model the DUT based on the input transaction"""
    self.expected_output.append(transaction)
    self.pkts_sent += 1
```

7.2.1 Test Function

There are various 'knobs' we can tweak on this testbench to vary the behaviour:

- Packet size
- Backpressure on the **stream_out** interface
- Idle cycles on the **stream_in** interface
- Configuration switching of the endian swap register during the test.

We want to run different variations of tests but they will all have a very similar structure so we create a common `run_test` function. To generate backpressure on the **stream_out** interface we use the `BitDriver` class from `cocotb.drivers`.

```
@cocotb.coroutine
def run_test(dut, data_in=None, config_coroutine=None, idle_inserter=None,␣
↪backpressure_inserter=None):

    cocotb.fork(Clock(dut.clk, 5000).start())
    tb = EndianSwapperTB(dut)

    yield tb.reset()
    dut.stream_out_ready <= 1

    # Start off any optional coroutines
    if config_coroutine is not None:
        cocotb.fork(config_coroutine(tb.csr))
    if idle_inserter is not None:
        tb.stream_in.set_valid_generator(idle_inserter())
    if backpressure_inserter is not None:
        tb.backpressure.start(backpressure_inserter())

    # Send in the packets
    for transaction in data_in():
        yield tb.stream_in.send(transaction)

    # Wait at least 2 cycles where output ready is low before ending the test
    for i in xrange(2):
        yield RisingEdge(dut.clk)
        while not dut.stream_out_ready.value:
            yield RisingEdge(dut.clk)

    pkt_count = yield tb.csr.read(1)
```

```
    if pkt_count.integer != tb.pkts_sent:
        raise TestFailure("DUT recorded %d packets but tb counted %d" % (
                          pkt_count.integer, tb.pkts_sent))
    else:
        dut._log.info("DUT correctly counted %d packets" % pkt_count.integer)

    raise tb.scoreboard.result
```

We can see that this test function creates an instance of the testbench, resets the DUT by running the coroutine `tb.reset()` and then starts off any optional coroutines passed in using the keyword arguments. We then send in all the packets from `data_in`, ensure that all the packets have been received by waiting 2 cycles at the end. We read the packet count and compare this with the number of packets. Finally we use the `tb.scoreboard.result` to determine the status of the test. If any transactions didn't match the expected output then this member would be an instance of the `TestFailure` result.

7.2.2 Test permutations

Having defined a test function we can now auto-generate different permutations of tests using the `TestFactory` class:

```
factory = TestFactory(run_test)
factory.add_option("data_in",                   [random_packet_sizes])
factory.add_option("config_coroutine",          [None, randomly_switch_config])
factory.add_option("idle_inserter",             [None, wave, intermittent_single_cycles,
↪ random_50_percent])
factory.add_option("backpressure_inserter",     [None, wave, intermittent_single_cycles,
↪ random_50_percent])
factory.generate_tests()
```

This will generate 32 tests (named run_test_001 to run_test_032) with all possible permutations of options provided for each argument. Note that we utilise some of the built-in generators to toggle backpressure and insert idle cycles.

CHAPTER 8

Tutorial: Ping

One of the benefits of Python is the ease with which interfacing is possible. In this tutorial we'll look at interfacing the standard GNU ping command to the simulator. Using Python we can ping our DUT with fewer than 50 lines of code.

For the impatient this tutorial is provided as an example with Cocotb. You can run this example from a fresh checkout:

```
cd examples/ping_tun_tap/tests
sudo make
```

Note: To create a virtual interface the test either needs root permissions or have CAP_NET_ADMIN capability.

8.1 Architecture

We have a simple RTL block that takes ICMP echo requests and generates an ICMP echo response. To verify this behaviour we want to run the ping utility against our RTL running in the simulator.

In order to achieve this we need to capture the packets that are created by ping, drive them onto the pins of our DUT in simulation, monitor the output of the DUT and send any responses back to the ping process.

Linux has a TUN/TAP virtual network device which we can use for this purpose, allowing ping to run unmodified and unaware that it is communicating with our simulation rather than a remote network endpoint.

8.2 Implementation

First of all we need to work out how to create a virtual interface. Python has a huge developer base and a quick search of the web reveals a TUN example that looks like an ideal starting point for our testbench. Using this example we write a function that will create our virtual interface:

```
import subprocess, fcntl, struct

def create_tun(name="tun0", ip="192.168.255.1"):
    TUNSETIFF = 0x400454ca
    TUNSETOWNER = TUNSETIFF + 2
    IFF_TUN = 0x0001
    IFF_NO_PI = 0x1000
    tun = open('/dev/net/tun', 'r+b')
    ifr = struct.pack('16sH', name, IFF_TUN | IFF_NO_PI)
    fcntl.ioctl(tun, TUNSETIFF, ifr)
    fcntl.ioctl(tun, TUNSETOWNER, 1000)
    subprocess.check_call('ifconfig tun0 %s up pointopoint 192.168.255.2 up' % ip,␣
↪shell=True)
    return tun
```

Now we can get started on the actual test. First of all we'll create a clock signal and connect up the Avalon driver and monitor to the DUT. To help debug the testbench we'll enable verbose debug on the drivers and monitors by setting the log level to **logging.DEBUG**.

```
import cocotb
from cocotb.clock import Clock
from cocotb.drivers.avalon import AvalonSTPkts as AvalonSTDriver
from cocotb.monitors.avalon import AvalonSTPkts as AvalonSTMonitor

@cocotb.test()
def tun_tap_example_test(dut):
    cocotb.fork(Clock(dut.clk, 5000).start())

    stream_in  = AvalonSTDriver(dut, "stream_in", dut.clk)
    stream_out = AvalonSTMonitor(dut, "stream_out", dut.clk)

    # Enable verbose logging on the streaming interfaces
    stream_in.log.setLevel(logging.DEBUG)
    stream_out.log.setLevel(logging.DEBUG)
```

We also need to reset the DUT and drive some default values onto some of the bus signals. Note that we'll need to import the **Timer** and **RisingEdge** triggers.

```
# Reset the DUT
dut._log.debug("Resetting DUT")
dut.reset_n <= 0
stream_in.bus.valid <= 0
yield Timer(10000)
yield RisingEdge(dut.clk)
dut.reset_n <= 1
dut.stream_out_ready <= 1
```

The rest of the test becomes fairly straightforward. We create our TUN interface using our function defined previously. We'll also use the **subprocess** module to actually start the ping command.

We then wait for a packet by calling a blocking read call on the TUN file descriptor and simply append that to the queue on the driver. We wait for a packet to arrive on the monitor by yielding on wait_for_recv() and then write the received packet back to the TUN file descriptor.

```
# Create our interface (destroyed at the end of the test)
tun = create_tun()
fd = tun.fileno()
```

```
# Kick off a ping...
subprocess.check_call('ping -c 5 192.168.255.2 &', shell=True)

# Respond to 5 pings, then quit
for i in xrange(5):

    cocotb.log.info("Waiting for packets on tun interface")
    packet = os.read(fd, 2048)
    cocotb.log.info("Received a packet!")

    stream_in.append(packet)
    result = yield stream_out.wait_for_recv()

    os.write(fd, str(result))
```

That's it - simple!

8.3 Further work

This example is deliberately simplistic to focus on the fundamentals of interfacing to the simulator using TUN/TAP. As an exercise for the reader a useful addition would be to make the file descriptor non-blocking and spawn out separate coroutines for the monitor / driver, thus decoupling the sending and receiving of packets.

CHAPTER 9

Tutorial: Driver Cosimulation

Cocotb was designed to provide a common platform for hardware and software developers to interact. By integrating systems early, ideally at the block level, it's possible to find bugs earlier in the design process.

For any given component that has a software interface there is typically a software abstraction layer or driver which communicates with the hardware. In this tutorial we will call unmodified production software from our testbench and re-use the code written to configure the entity.

For the impatient this tutorial is provided as an example with Cocotb. You can run this example from a fresh checkout:

```
cd examples/endian_swapper/tests
make MODULE=test_endian_swapper_hal
```

Note: SWIG is required to compile the example

9.1 Difficulties with Driver Co-simulation

Co-simulating *un-modified* production software against a block-level testbench is not trivial - there are a couple of significant obstacles to overcome:

9.1.1 Calling the HAL from a test

Typically the software component (often referred to as a Hardware Abstraction Layer or HAL) is written in C. We need to call this software from our test written in Python. There are multiple ways to call C code from Python, in this tutorial we'll use SWIG to generate Python bindings for our HAL.

9.1.2 Blocking in the driver

Another difficulty to overcome is the fact that the HAL is expecting to call a low-level function to access the hardware, often something like `ioread32`. We need this call to block while simulation time advances and a value is either read or written on the bus. To achieve this we link the HAL against a C library that provides the low level read/write functions. These functions in turn call into Cocotb and perform the relevant access on the DUT.

9.2 Cocotb infrastructure

There are two decorators provided to enable this flow, which are typically used together to achieve the required functionality. The `cocotb.external` decorator turns a normal function that isn't a coroutine into a blocking coroutine (by running the function in a separate thread). The `cocotb.function` decorator allows a coroutine that consumes simulation time to be called by a normal thread. The call sequence looks like this:

9.3 Implementation

9.3.1 Register Map

The endian swapper has a very simple register map:

Byte Offset	Register	Bits	Access	Description
0	CONTROL	0	R/W	Enable
		31:1	N/A	Reserved
4	PACKET_COUNT	31:0	RO	Num Packets

9.3.2 HAL

To keep things simple we use the same RTL from the *Tutorial: Endian Swapper*. We write a simplistic HAL which provides the following functions:

```
endian_swapper_enable(endian_swapper_state_t *state);
endian_swapper_disable(endian_swapper_state_t *state);
endian_swapper_get_count(endian_swapper_state_t *state);
```

These functions call `IORD` and `IOWR` - usually provided by the Altera NIOS framework.

9.3.3 IO Module

This module acts as the bridge between the C HAL and the Python testbench. It exposes the `IORD` and `IOWR` calls to link the HAL against, but also provides a Python interface to allow the read/write bindings to be dynamically set (through `set_write_function` and `set_read_function` module functions).

In a more complicated scenario, this could act as an interconnect, dispatching the access to the appropriate driver depending on address decoding, for instance.

9.3.4 Testbench

First of all we set up a clock, create an Avalon Master interface and reset the DUT. Then we create two functions that are wrapped with the `cocotb.function` decorator to be called when the HAL attempts to perform a read or write. These are then passed to the *IO Module*:

```
@cocotb.function
def read(address):
    master.log.debug("External source: reading address 0x%08X" % address)
    value = yield master.read(address)
    master.log.debug("Reading complete: got value 0x%08x" % value)
    raise ReturnValue(value)

@cocotb.function
def write(address, value):
    master.log.debug("Write called for 0x%08X -> %d" % (address, value))
    yield master.write(address, value)
    master.log.debug("Write complete")

io_module.set_write_function(write)
io_module.set_read_function(read)
```

We can then intialise the HAL and call functions, using the `cocotb.external` decorator to turn the normal function into a blocking coroutine that we can `yield`:

```
state = hal.endian_swapper_init(0)
yield cocotb.external(hal.endian_swapper_enable)(state)
```

The HAL will perform whatever calls it needs, accessing the DUT through the Avalon-MM driver, and control will return to the testbench when the function returns.

Note: The decorator is applied to the function before it is called

---

9.4 Further Work

In future tutorials we'll consider co-simulating unmodified drivers written using `mmap` (for example built upon the UIO framework) and consider interfacing with emulators like QEMU to allow us to co-simulate when the software needs to execute on a different processor architecture.

CHAPTER 10

Roadmap

Cocotb is in active development.

We use GitHub issues to track our pending tasks. Take a look at the open Enhancements to see the work that's lined up.

If you have a GitHub account you can also raise an enhancement request to suggest new features.

CHAPTER 11

Simulator Support

This page documents any known quirks and gotchas in the various simulators.

11.1 Icarus

Accessing bits of a vector doesn't work:

```
dut.stream_in_data[2] <= 1
```

See "access_single_bit" test in examples/functionality/tests/test_discovery.py.

11.2 Synopsys VCS

11.3 Aldec Riviera-PRO

11.4 Mentor Questa

11.5 Cadence Incisive

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Index

A

acquire() (cocotb.triggers.Lock method), 23
add_interface() (cocotb.scoreboard.Scoreboard method), 23
add_option() (cocotb.regression.TestFactory method), 20
append() (cocotb.drivers.Driver method), 23
assign() (cocotb.binary.BinaryValue method), 21

B

BinaryValue (class in cocotb.binary), 21
binstr (cocotb.binary.BinaryValue attribute), 21
buff (cocotb.binary.BinaryValue attribute), 21
Bus (class in cocotb.bus), 22

C

clear() (cocotb.drivers.Driver method), 23
clear() (cocotb.triggers.Event method), 22
compare() (cocotb.scoreboard.Scoreboard method), 23
coroutine (class in cocotb), 19

D

drive() (cocotb.bus.Bus method), 22
Driver (class in cocotb.drivers), 23

E

Edge (class in cocotb.triggers), 22
Event (class in cocotb.triggers), 22

G

generate_tests() (cocotb.regression.TestFactory method), 20
get_binstr() (cocotb.binary.BinaryValue method), 21
get_buff() (cocotb.binary.BinaryValue method), 21
get_value() (cocotb.binary.BinaryValue method), 21
get_value_signed() (cocotb.binary.BinaryValue method), 21

I

integer (cocotb.binary.BinaryValue attribute), 21

J

Join (class in cocotb.triggers), 23

L

Lock (class in cocotb.triggers), 22

M

Monitor (class in cocotb.monitors), 23

R

ReadOnly (class in cocotb.triggers), 22
result (cocotb.scoreboard.Scoreboard attribute), 23
RisingEdge (class in cocotb.triggers), 22

S

Scoreboard (class in cocotb.scoreboard), 23
set() (cocotb.triggers.Event method), 22
signed_integer (cocotb.binary.BinaryValue attribute), 21

T

test (class in cocotb), 19
TestComplete (class in cocotb.result), 19
TestError (class in cocotb.result), 19
TestFactory (class in cocotb.regression), 20
TestFailure (class in cocotb.result), 19
TestSuccess (class in cocotb.result), 19
Timer (class in cocotb.triggers), 22

V

value (cocotb.binary.BinaryValue attribute), 21

W

wait() (cocotb.triggers.Event method), 22