cocotb Documentation

Release 1.1

PotentialVentures

Jan 09, 2020

CONTENTS

Contents:

CHAPTER

ONE

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
- GHDL
- Aldec Riviera-PRO
- Synopsys VCS
- Cadence Incisive
- Mentor Modelsim (DE and SE)

Windows Platform

- Icarus Verilog
- Aldec Riviera-PRO
- Mentor Modelsim (DE and SE)

A (possibly older) version of 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.

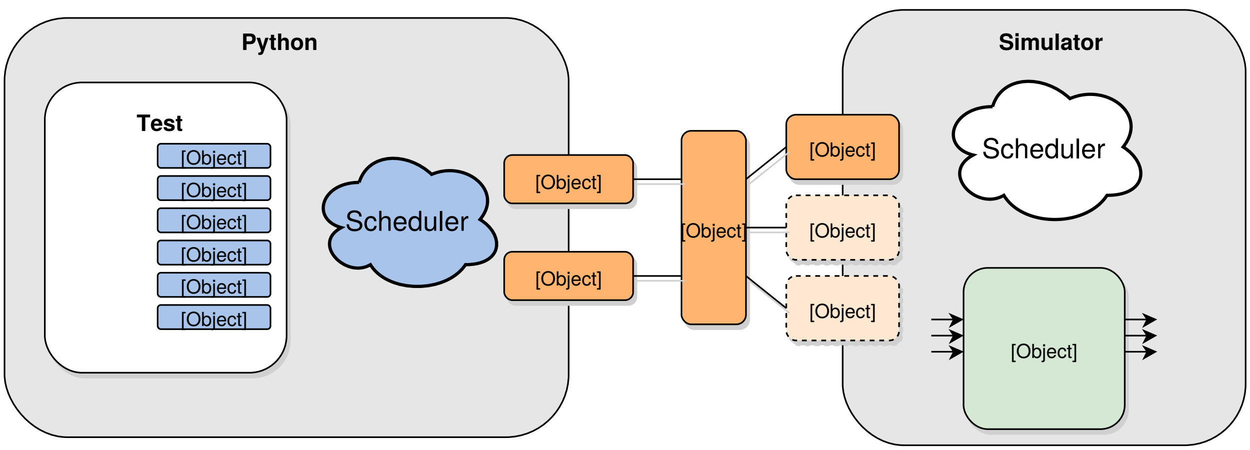
All verification is done using Python which has various advantages over using SystemVerilog or VHDL for verification:

- 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 the full list of contributions).

We also have a list of talks and papers, libraries and examples at our wiki page Further Resources. Feel free to add links to cocotb-related content that we are still missing!

CHAPTER

TWO

QUICKSTART GUIDE

2.1 Installing cocotb

2.1.1 Pre-requisites

Cocotb has the following requirements:

- Python 2.7+
- Python-dev packages
- GCC and associated development packages
- GNU Make
- A Verilog or VHDL simulator, depending on your source RTL code

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 32-bit 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 32-bit development libraries and Python are needed.

2.1.3 32-bit Python

Additional development libraries are needed for building 32-bit 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 32-bit Python by setting the architecture and placing the 32-bit 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

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.

https://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 macOS Packages

You need a few packages installed to get cocotb running on macOS. 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 install python icarus-verilog gtkwave
```

2.2 Installing cocotb

Cocotb can be installed by running (recommended Python3)

```
$> pip3 install cocotb
```

or

```
$> pip install cocotb
```

**For user local install follow* pip User Guide.

For development version:

```
$> git clone https://github.com/cocotb/cocotb
$> pip install -e ./cocotb
```

2.2.1 Running an example

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

To run a test using a different simulator:

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

2.2.2 Running a VHDL example

The `endian_swapper` example includes both a VHDL and a 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=ghdl TOPLEVEL_LANG=vhdl
```

2.3 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.

2.3.1 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  # the module name in your Verilog or VHDL file
MODULE=test_my_design  # the name of the Python test file
include $(shell cocotb-config --makefiles)/Makefile.inc
include $(shell cocotb-config --makefiles)/Makefile.sim
```

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

2.3.2 Creating a test

The test is written in Python. Cocotb wraps your top level with the handle you pass it. In this documentation, and most of the examples in the project, that handle is `dut`, but you can pass your own preferred name in instead. The handle is used in all Python files referencing your RTL project. 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.

2.3.3 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
```

2.3.4 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
```

The syntax `sig <= new_value` is a short form of `sig.value = new_value`. It not only resembles HDL-syntax, but also has the same semantics: writes are not applied immediately, but delayed until the next write cycle. Use `sig.setimmediatevalue(new_val)` to set a new value immediately (see *setimmediatevalue()*).

2.3.5 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 *integer* 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
>>> # Show number of bits in a value
>>> print(count.n_bits)
6
```

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

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

2.3.6 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")
```

CHAPTER

THREE

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.

VHDL_SOURCES_lib A list of the VHDL source files to include in the VHDL library *lib* (currently GHDL only).

COMPILE_ARGS Any arguments or flags to pass to the compile stage of the simulation.

SIM_ARGS Any arguments or flags to pass to the execution of the compiled simulation.

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 of VHPI traces when using the nvc VHDL simulator.

SIM_BUILD Use to define a scratch directory for use by the simulator. The path is relative to the Makefile location. If not provided, the default scratch directory is `sim_build`.

3.2 Environment Variables

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.

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 stimuli use the following:

```
make RANDOM_SEED=1377424946
```

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

COCOTB_REDUCED_LOG_FMT
If defined, log lines displayed in terminal will be shorter. It will print only time, message type (`INFO`, `WARNING`, `ERROR`) and log message.

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

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

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

3.2.1 Additional Environment Variables

COCOTB_ATTACH
In order to give yourself time to attach a debugger to the simulator process before it starts to run, you can set the environment variable *COCOTB_ATTACH* to a pause time value in seconds. If set, Cocotb will print the process ID (PID) to attach to and wait the specified time before actually letting the simulator run.

COCOTB_ENABLE_PROFILING
Enable performance analysis of the Python portion of Cocotb. When set, a file `test_profile.pstat` will be written which contains statistics about the cumulative time spent in the functions.

From this, a callgraph diagram can be generated with gprof2dot and `graphviz`. See the `profile` Make target in the `endian_swapper` example on how to set this up.

COCOTB_HOOKS
A comma-separated list of modules that should be executed before the first test. You can also use the *cocotb.hook* decorator to mark a function to be run before test code.

COCOTB_LOG_LEVEL
Default logging level to use. This is set to `INFO` unless overridden.

COCOTB_RESOLVE_X
Defines how to resolve bits with a value of `X`, `Z`, `U` or `W` when being converted to integer. Valid settings are:

VALUE_ERROR raise a `ValueError` exception

ZEROS resolve to `0`

ONES resolve to `1`

RANDOM randomly resolve to a `0` or a `1`

Set to `VALUE_ERROR` by default.

COCOTB_SCHEDULER_DEBUG
Enable additional log output of the coroutine scheduler.

COVERAGE
Enable to report python coverage data. For some simulators, this will also report HDL coverage.

This needs the `coverage` python module

MEMCHECK
HTTP port to use for debugging Python's memory usage. When set to e.g. `8088`, data will be presented at http://localhost:8088.

This needs the `cherrypy` and `dowser` Python modules installed.

COCOTB_PY_DIR
Path to the directory containing the cocotb Python package in the `cocotb` subdirectory.

COCOTB_SHARE_DIR
Path to the directory containing the cocotb Makefiles and simulator libraries in the subdirectories `lib`, `include`, and `makefiles`.

VERSION
The version of the Cocotb installation. You probably don't want to modify this.

CHAPTER FOUR

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 can return a value, so that they can be used by other coroutines. Before Python 3.3, this requires a *`ReturnValue`* to be raised.

```
@cocotb.coroutine
def get_signal(clk, signal):
    yield RisingEdge(clk)
    raise ReturnValue(signal.value)

@cocotb.coroutine
def get_signal_python_33(clk, signal):
    # newer versions of Python can use return normally
    yield RisingEdge(clk)
    return signal.value

@cocotb.coroutine
def check_signal_changes(dut):
    first = yield get_signal(dut.clk, dut.signal)
    second = yield get_signal(dut.clk, dut.signal)
    if first == second:
        raise TestFailure("Signal did not change")
```

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

```
@cocotb.coroutine
def packet_with_timeout(monitor, timeout):
```

(continues on next page)

(continued from previous page)

```
    """Wait for a packet but time out if nothing arrives"""
    yield [Timer(timeout), RisingEdge(dut.ready)]
```

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 time out if nothing arrives"""
    tout_trigger = Timer(timeout)
    result = yield [tout_trigger, RisingEdge(dut.ready)]
    if result is tout_trigger:
        raise TestFailure("Timed out waiting for packet")
```

Coroutines can be forked for parallel operation within a function of that code and the forked code.

```
@cocotb.test()
def test_act_during_reset(dut):
    """While reset is active, toggle signals"""
    tb = uart_tb(dut)
    # "Clock" is a built in class for toggling a clock signal
    cocotb.fork(Clock(dut.clk, 1000).start())
    # reset_dut is a function -
    # part of the user-generated "uart_tb" class
    cocotb.fork(tb.reset_dut(dut.rstn, 20000))

    yield Timer(10000)
    print("Reset is still active: %d" % dut.rstn)
    yield Timer(15000)
    print("Reset has gone inactive: %d" % dut.rstn)
```

Coroutines can be joined to end parallel operation within a function.

```
@cocotb.test()
def test_count_edge_cycles(dut, period=1000, clocks=6):
    cocotb.fork(Clock(dut.clk, period).start())
    yield RisingEdge(dut.clk)

    timer = Timer(period + 10)
    task = cocotb.fork(count_edges_cycles(dut.clk, clocks))
    count = 0
    expect = clocks - 1

    while True:
        result = yield [timer, task.join()]
        if count > expect:
            raise TestFailure("Task didn't complete in expected time")
        if result is timer:
            dut._log.info("Count %d: Task still running" % count)
            count += 1
        else:
            break
```

Coroutines can be killed before they complete, forcing their completion before they'd naturally end.

```
@cocotb.test()
def test_different_clocks(dut):
    clk_1mhz   = Clock(dut.clk, 1.0, units='us')
    clk_250mhz = Clock(dut.clk, 4.0, units='ns')

    clk_gen = cocotb.fork(clk_1mhz.start())
    start_time_ns = get_sim_time(units='ns')
    yield Timer(1)
    yield RisingEdge(dut.clk)
    edge_time_ns = get_sim_time(units='ns')
    # NOTE: isclose is a python 3.5+ feature
    if not isclose(edge_time_ns, start_time_ns + 1000.0):
        raise TestFailure("Expected a period of 1 us")

    clk_gen.kill()

    clk_gen = cocotb.fork(clk_250mhz.start())
    start_time_ns = get_sim_time(units='ns')
    yield Timer(1)
    yield RisingEdge(dut.clk)
    edge_time_ns = get_sim_time(units='ns')
    # NOTE: isclose is a python 3.5+ feature
    if not isclose(edge_time_ns, start_time_ns + 4.0):
        raise TestFailure("Expected a period of 4 ns")
```

4.1 Async functions

Python 3.5 introduces `async` functions, which provide an alternative syntax. For example:

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

To wait on a trigger or a nested coroutine, these use `await` instead of `yield`. Provided they are decorated with `@cocotb.coroutine`, `async def` functions using `await` and regular functions using `yield` can be used interchangeable - the appropriate keyword to use is determined by which type of function it appears in, not by the sub-coroutinue being called.

Note: It is not legal to `await` a list of triggers as can be done in `yield`-based coroutine with `yield [trig1, trig2]`. Use `await First(trig1, trig2)` instead.

4.1.1 Async generators

In Python 3.6, a `yield` statement within an `async` function has a new meaning (rather than being a `SyntaxError`) which matches the typical meaning of `yield` within regular python code. It can be used to create a special type of generator function that can be iterated with `async for`:

```
async def ten_samples_of(clk, signal):
    for i in range(10):
        await RisingEdge(clk)
        yield signal.value  # this means "send back to the for loop"

@cocotb.test()
async def test_samples_are_even(dut):
    async for sample in ten_samples_of(dut.clk, dut.signal):
        assert sample % 2 == 0
```

More details on this type of generator can be found in **PEP 525**.

CHAPTER

FIVE

TRIGGERS

Triggers are used to indicate when the cocotb scheduler should resume coroutine execution. Typically a coroutine will `yield` a trigger or a list of triggers, while it is waiting for them to complete.

5.1 Simulation Timing

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

`ReadOnly()` Registers a callback which will continue execution of the coroutine when the current simulation timestep moves to the `ReadOnly` phase of the RTL simulator. The `ReadOnly` phase is entered when the current timestep no longer has any further delta steps. This should be a point where all the signal values are stable as there are no more RTL events scheduled for the timestep. The simulator should not allow scheduling of more events in this timestep. 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

`Edge(signal)` Registers a callback that will continue execution of the coroutine on any value change of *signal*.

`RisingEdge(signal)` Registers a callback that will continue execution of the coroutine on a transition from `0` to `1` of *signal*.

`FallingEdge(signal)` Registers a callback that will continue execution of the coroutine on a transition from `1` to `0` of *signal*.

`ClockCycles(signal, num_cycles)` Registers a callback that will continue execution of the coroutine when *num_cycles* transitions from `0` to `1` have occured on *signal*.

5.3 Python Triggers

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

`Join(coroutine_2)` Will block the coroutine until *coroutine_2* has completed.

CHAPTER

SIX

TESTBENCH TOOLS

6.1 Logging

Cocotb extends the Python logging library. Each DUT, monitor, driver, and scoreboard (as well as any other function using the coroutine decorator) implements its own `logging` object, and each can be set to its own logging level. Within a DUT, each hierarchical object can also have individual logging levels set.

When logging HDL objects, beware that `_log` is the preferred way to use logging. This helps minimize the change of name collisions with an HDL log component with the Python logging functionality.

Log printing levels can also be set on a per-object basis.

```
class EndianSwapperTB(object):

    def __init__(self, dut, debug=False):
        self.dut = dut
        self.stream_in = AvalonSTDriver(dut, "stream_in", dut.clk)
        self.stream_in_recovered = AvalonSTMonitor(dut, "stream_in", dut.clk,
                                                   callback=self.model)

        # Set verbosity on our various interfaces
        level = logging.DEBUG if debug else logging.WARNING
        self.stream_in.log.setLevel(level)
        self.stream_in_recovered.log.setLevel(level)
        self.dut.reset_n._log.setLevel(logging.DEBUG)
```

And when the logging is actually called

```
class AvalonSTPkts(BusMonitor):
...
    @coroutine
    def _monitor_recv(self):
        ...
        self.log.info("Received a packet of %d bytes" % len(pkt))

class Scoreboard(object):
    ...
    def add_interface(self):
        ...
        self.log.info("Created with reorder_depth %d" % reorder_depth)

class EndianSwapTB(object):
    ...
    @cocotb.coroutine
```

(continues on next page)

(continued from previous page)

```
    def reset():
        self.dut._log.debug("Resetting DUT")
```

will display as something like

```
0.00ns INFO                     cocotb.scoreboard.endian_swapper_sv       scoreboard.
↪py:177  in add_interface                          Created with reorder_depth 0
0.00ns DEBUG                    cocotb.endian_swapper_sv           .._endian_swapper.
↪py:106  in reset                                  Resetting DUT
160000000000000.00ns INFO       cocotb.endian_swapper_sv.stream_out           avalon.
↪py:151  in _monitor_recv                          Received a packet of 125 bytes
```

6.2 Busses

Busses are simply defined as collection of signals. The *Bus* class will automatically bundle any group of signals together that are named similar to `dut.<bus_name><separator><signal_name>`. For instance,

```
dut.stream_in_valid
dut.stream_in_data
```

have a bus name of `stream_in`, a separator of `_`, and signal names of `valid` and `data`. A list of signal names, or a dictionary mapping attribute names to signal names is also passed into the *Bus* class. Busses can have values driven onto them, be captured (returning a dictionary), or sampled and stored into a similar object.

```
stream_in_bus = Bus(dut, "stream_in", ["valid", "data"]) # '_' is the default␣
↪separator
```

6.3 Driving Busses

Examples and specific bus implementation bus drivers (AMBA, Avalon, XGMII, and others) exist in the *Driver* class enabling a test to append transactions to perform the serialization of transactions onto a physical interface. Here is an example using the Avalon bus driver in the `endian_swapper` example:

```
class EndianSwapperTB(object):

    def __init__(self, dut, debug=False):
        self.dut = dut
        self.stream_in = AvalonSTDriver(dut, "stream_in", dut.clk)

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

    if idle_inserter is not None:
        tb.stream_in.set_valid_generator(idle_inserter())
```

(continues on next page)

(continued from previous page)

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

6.4 Monitoring Busses

For our testbenches to actually be useful, we have to monitor some of these busses, and not just drive them. That's where the `Monitor` class comes in, with prebuilt monitors for Avalon and XGMII busses. The Monitor class is a base class which you are expected to derive for your particular purpose. You must create a `_monitor_recv()` function which is responsible for determining 1) at what points in simulation to call the `_recv()` function, and 2) what transaction values to pass to be stored in the monitors receiving queue. Monitors are good for both outputs of the DUT for verification, and for the inputs of the DUT, to drive a test model of the DUT to be compared to the actual DUT. For this purpose, input monitors will often have a callback function passed that is a model. This model will often generate expected transactions, which are then compared using the `Scoreboard` class.

```
# ==============================================================================
class BitMonitor(Monitor):
    """Observes single input or output of DUT."""
    def __init__(self, name, signal, clock, callback=None, event=None):
        self.name = name
        self.signal = signal
        self.clock = clock
        Monitor.__init__(self, callback, event)

    @coroutine
    def _monitor_recv(self):
        clkedge = RisingEdge(self.clock)

        while True:
            # Capture signal at rising edge of clock
            yield clkedge
            vec = self.signal.value
            self._recv(vec)

# ==============================================================================
def input_gen():
    """Generator for input data applied by BitDriver"""
    while True:
        yield random.randint(1,5), random.randint(1,5)

# ==============================================================================
class DFF_TB(object):
    def __init__(self, dut, init_val):

        self.dut = dut

        # Create input driver and output monitor
        self.input_drv = BitDriver(dut.d, dut.c, input_gen())
        self.output_mon = BitMonitor("output", dut.q, dut.c)

        # Create a scoreboard on the outputs
        self.expected_output = [ init_val ]
```

(continues on next page)

(continued from previous page)

```
        # Reconstruct the input transactions from the pins
        # and send them to our 'model'
        self.input_mon = BitMonitor("input", dut.d, dut.c, callback=self.model)

    def model(self, transaction):
        """Model the DUT based on the input transaction."""
        # Do not append an output transaction for the last clock cycle of the
        # simulation, that is, after stop() has been called.
        if not self.stopped:
            self.expected_output.append(transaction)
```

6.5 Tracking testbench errors

The `Scoreboard` class is used to compare the actual outputs to expected outputs. Monitors are added to the scoreboard for the actual outputs, and the expected outputs can be either a simple list, or a function that provides a transaction. Here is some code from the `dff` example, similar to above with the scoreboard added.

```
class DFF_TB(object):
    def __init__(self, dut, init_val):
        self.dut = dut

        # Create input driver and output monitor
        self.input_drv = BitDriver(dut.d, dut.c, input_gen())
        self.output_mon = BitMonitor("output", dut.q, dut.c)

        # Create a scoreboard on the outputs
        self.expected_output = [ init_val ]
        self.scoreboard = Scoreboard(dut)
        self.scoreboard.add_interface(self.output_mon, self.expected_output)

        # Reconstruct the input transactions from the pins
        # and send them to our 'model'
        self.input_mon = BitMonitor("input", dut.d, dut.c,callback=self.model)
```

CHAPTER

SEVEN

LIBRARY REFERENCE

7.1 Test Results

The exceptions in this module can be raised at any point by any code and will terminate the test.

`cocotb.result.raise_error(obj, msg)`
Creates a `TestError` exception and raises it after printing a traceback.

Parameters

- **obj** – Object with a log method.
- **msg** (*str*) – The log message.

`cocotb.result.create_error(obj, msg)`
Like `raise_error()`, but return the exception rather than raise it, simply to avoid too many levels of nested *try/except* blocks.

Parameters

- **obj** – Object with a log method.
- **msg** (*str*) – The log message.

exception `cocotb.result.ReturnValue(retval)`
Helper exception needed for Python versions prior to 3.3.

exception `cocotb.result.TestComplete(*args, **kwargs)`
Exception showing that test was completed. Sub-exceptions detail the exit status.

exception `cocotb.result.ExternalException(exception)`
Exception thrown by external functions.

exception `cocotb.result.TestError(*args, **kwargs)`
Exception showing that test was completed with severity Error.

exception `cocotb.result.TestFailure(*args, **kwargs)`
Exception showing that test was completed with severity Failure.

exception `cocotb.result.TestSuccess(*args, **kwargs)`
Exception showing that test was completed successfully.

exception `cocotb.result.SimFailure(*args, **kwargs)`
Exception showing that simulator exited unsuccessfully.

7.2 Writing and Generating tests

class cocotb.**test**(*f*, *timeout=None*, *expect_fail=False*, *expect_error=False*, *skip=False*, *stage=None*)
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.

Used as `@cocotb.test(...)`.

Parameters

- **timeout** (*int, optional*) – value representing simulation timeout (not implemented).
- **expect_fail** (*bool, optional*) – Don't mark the result as a failure if the test fails.
- **expect_error** (*bool, optional*) – Don't mark the result as an error if an error is raised. This is for cocotb internal regression use when a simulator error is expected.
- **skip** (*bool, optional*) – Don't execute this test as part of the regression.
- **stage** (*int, optional*) – Order tests logically into stages, where multiple tests can share a stage.

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

`log` methods will log to `cocotb.coroutine.name`.

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

Used as `@cocotb.coroutine`.

class cocotb.**external**(*func*)
Decorator to apply to an external function to enable calling from cocotb. This currently creates a new execution context for each function that is called. Scope for this to be streamlined to a queue in future.

class cocotb.**function**(*func*)
Decorator class that allows a function to block.

This allows a function to internally block while externally appear to yield.

class cocotb.**hook**(*f*)
Decorator to mark a function as a hook for cocotb.

Used as `@cocotb.hook()`.

All hooks are run at the beginning of a cocotb test suite, prior to any test code being run.

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 generation 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])
```

(continues on next page)

(continued from previous page)

```
>>> 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.

Parameters

- **name** (*str*) – 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 exhaustive set of tests using the cartesian product of the possible keyword arguments.

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

Parameters

- **prefix** (*str*) – 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** (*str*) – 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.

7.3 Interacting with the Simulator

class cocotb.binary.**BinaryRepresentation**

UNSIGNED = 0
Unsigned format

SIGNED_MAGNITUDE = 1
Sign and magnitude format

TWOS_COMPLEMENT = 2
Two's complement format

class cocotb.binary.**BinaryValue**(*value=None*, *n_bits=None*, *bigEndian=True*, *binaryRepresentation=0*, *bits=None*)
Representation of values in binary format.

The underlying value can be set or accessed using these 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.

Parameters **value** (*str or int or long*) – The value to assign.

get_value()
Return the integer representation of the underlying vector.

get_value_signed()
Return the signed integer representation of the underlying vector.

is_resolvable
Does the value contain any `X`'s? Inquiring minds want to know.

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

integer
The integer representation of the underlying vector.

signed_integer
The signed integer representation of the underlying vector.

get_buff()
Attribute *buff* represents the value as a binary string buffer.

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

buff
Access to the value as a buffer.

get_binstr()
Attribute *binstr* is the binary representation stored as a string of `1` and `0`.

binstr
Access to the binary string.

n_bits
Access to the number of bits of the binary value.

class cocotb.bus.**Bus**(*entity*, *name*, *signals*, *optional_signals=[]*, *bus_separator='_'*, *array_idx=None*)
Wraps up a collection of signals.

Assumes we have a set of signals/nets named `entity.<bus_name><separator><signal>`.

For example a bus `stream_in` with signals `valid` and `data` is assumed to be named `dut.stream_in_valid` and `dut.stream_in_data` (with the default separator '_').

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

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

Parameters

- **obj** – Object with attribute names that match the bus signals.
- **strict** (*bool, optional*) – Check that all signals are being assigned.

Raises **AttributeError** – If not all signals have been assigned when `strict=True`.

capture()
Capture the values from the bus, returning an object representing the capture.

Returns A dictionary that supports access by attribute, where each attribute corresponds to each signal's value.

Return type dict

Raises **RuntimeError** – If signal not present in bus, or attempt to modify a bus capture.

sample(*obj*, *strict=False*)
Sample the values from the bus, assigning them to *obj*.

Parameters

- **obj** – Object with attribute names that match the bus signals.
- **strict** (*bool, optional*) – Check that all signals being sampled are present in *obj*.

Raises **AttributeError** – If attribute is missing in *obj* when `strict=True`.

class cocotb.clock.**Clock**(*signal*, *period*, *units=None*)
Simple 50:50 duty cycle clock driver.

Instances of this class should call its *start()* method and fork the result. This will create a clocking thread that drives the signal at the desired period/frequency.

Example:

```
c = Clock(dut.clk, 10, 'ns')
cocotb.fork(c.start())
```

Parameters

- **signal** – The clock pin/signal to be driven.
- **period** (*int*) – The clock period. Must convert to an even number of timesteps.
- **units** (*str, optional*) – One of None, 'fs', 'ps', 'ns', 'us', 'ms', 'sec'. When no *units* is given (None) the timestep is determined by the simulator.

7.3.1 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.

class cocotb.triggers.**Trigger**

Base class to derive from.

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**

Execution will resume when the readonly portion of the sim cycles is reached.

class cocotb.triggers.**NextTimeStep**

Execution will resume when the next time step is started.

class cocotb.triggers.**ClockCycles**(*signal*, *num_cycles*, *rising=True*)

Execution will resume after *num_cycles* rising edges or *num_cycles* falling edges.

Signal related

class cocotb.triggers.**Edge**(*signal*)

Triggers on either edge of the provided signal.

class cocotb.triggers.**RisingEdge**(*signal*)

Triggers on the rising edge of the provided signal.

class cocotb.triggers.**FallingEdge**(*signal*)

Triggers on the falling edge of the provided signal.

Python Triggers

class cocotb.triggers.**Combine**(**args*)

Waits until all the passed triggers have fired.

Like most triggers, this simply returns itself.

class cocotb.triggers.**Event**(*name=''*)

Event to permit synchronisation between two coroutines.

set(*data=None*)

Wake up any coroutines blocked on this event.

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

If the event has already been fired, this returns `NullTrigger`. To reset the event (and enable the use of `wait` again), `clear()` should be called.

clear()
Clear this event that has fired.

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

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

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

release()
Release the lock.

class cocotb.triggers.**Join**(*coroutine*)
Join a coroutine, firing when it exits.

retval
The return value of the joined coroutine.

If the coroutine threw an exception, this attribute will re-raise it.

prime(*callback*)
Set a callback to be invoked when the trigger fires.

The callback will be invoked with a single argumement, *self*.

Subclasses must override this, but should end by calling the base class method.

Do not call this directly within coroutines, it is intended to be used only by the scheduler.

7.4 Testbench Structure

7.4.1 Driver

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.

kill()
Kill the coroutine sending stuff.

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

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

Parameters

- **transaction** (*any*) – The transaction to be sent.
- **callback** (`callable, optional`) – Optional function to be called when the transaction has been sent.

- **event** (*optional*) – *Event* to be set when the transaction has been sent.
- ****kwargs** – Any additional arguments used in child class' *_driver_send* method.

clear()

Clear any queued transactions without sending them onto the bus.

send

Blocking send call (hence must be "yielded" rather than called).

Sends the transaction over the bus.

Parameters

- **transaction** (*any*) – The transaction to be sent.
- **sync** (*bool, optional*) – Synchronise the transfer by waiting for a rising edge.
- ****kwargs** (*dict*) – Additional arguments used in child class' *_driver_send* method.

_driver_send(*transaction*, *sync=True*, ***kwargs*)

Actual implementation of the send.

Subclasses should override this method to implement the actual *send()* routine.

Parameters

- **transaction** (*any*) – The transaction to be sent.
- **sync** (*boolean, optional*) – Synchronise the transfer by waiting for a rising edge.
- ****kwargs** – Additional arguments if required for protocol implemented in subclass.

_send

Send coroutine.

Parameters

- **transaction** (*any*) – The transaction to be sent.
- **callback** (*callable, optional*) – Optional function to be called when the transaction has been sent.
- **event** (*optional*) – event to be set when the transaction has been sent.
- **sync** (*boolean, optional*) – Synchronise the transfer by waiting for a rising edge.
- ****kwargs** – Any additional arguments used in child class' *_driver_send* method.

class cocotb.drivers.**BitDriver**(*signal*, *clk*, *generator=None*)

Bases: object

Drives a signal onto a single bit.

Useful for exercising ready / valid.

start(*generator=None*)

Start generating data.

Parameters generator (*optional*) – Generator yielding data.

stop()

Stop generating data.

class cocotb.drivers.**BusDriver**(*entity*, *name*, *clock*, ***kwargs*)

Bases: *cocotb.drivers.Driver*

Wrapper around common functionality for busses which have:

- a list of `_signals` (class attribute)
- a list of `_optional_signals` (class attribute)
- a clock
- a name
- an entity

Parameters

- **entity** (*SimHandle*) – A handle to the simulator entity.
- **name** (*str or None*) – Name of this bus. `None` for nameless bus, e.g. bus-signals in an interface or a modport. (untested on struct/record, but could work here as well).
- **clock** (*SimHandle*) – A handle to the clock associated with this bus.
- **array_idx** (*int or None, optional*) – Optional index when signal is an array.

_driver_send

Implementation for BusDriver.

Parameters

- **transaction** – The transaction to send.
- **sync** (*bool, optional*) – Synchronise the transfer by waiting for a rising edge.

_wait_for_signal

This method will return when the specified signal has hit logic `1`. The state will be in the *ReadOnly* phase so sim will need to move to *NextTimeStep* before registering more callbacks can occur.

_wait_for_nsignal

This method will return when the specified signal has hit logic `0`. The state will be in the *ReadOnly* phase so sim will need to move to *NextTimeStep* before registering more callbacks can occur.

7.4.2 Monitor

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

Base class for Monitor objects.

Monitors are passive 'listening' objects that monitor pins going in or out of a DUT. This class should not be used directly, but should be subclassed and the internal *_monitor_recv* method should be overridden and decorated as a *coroutine*. This *_monitor_recv* method should capture some behavior of the pins, form a transaction, and pass this transaction to the internal *_recv* method. The *_monitor_recv* method is added to the cocotb scheduler during the `__init__` phase, so it should not be yielded anywhere.

The primary use of a Monitor is as an interface for a *Scoreboard*.

Parameters

- **callback** (*callable*) – Callback to be called with each recovered transaction as the argument. If the callback isn't used, received transactions will be placed on a queue and the event used to notify any consumers.
- **event** (*event*) – Object that supports a `set` method that will be called when a transaction is received through the internal *_recv* method.

for ... in wait_for_recv(*timeout=None*)
With *timeout*, `wait()` for transaction to arrive on monitor and return its data.

Parameters **timeout** (*optional*) – The timeout value for `Timer`. Defaults to `None`.

Returns: Data of received transaction.

_monitor_recv
Actual implementation of the receiver.

Subclasses should override this method to implement the actual receive routine and call `_recv` with the recovered transaction.

_recv(*transaction*)
Common handling of a received transaction.

class cocotb.monitors.**BusMonitor**(*entity*, *name*, *clock*, *reset=None*, *reset_n=None*, *callback=None*, *event=None*, *bus_separator='_'*, *array_idx=None*)
Bases: `cocotb.monitors.Monitor`

Wrapper providing common functionality for monitoring busses.

in_reset
Boolean flag showing whether the bus is in reset state or not.

7.4.3 Scoreboard

Common scoreboarding capability.

class cocotb.scoreboard.**Scoreboard**(*dut*, *reorder_depth=0*, *fail_immediately=True*)
Bases: `object`

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.

Parameters

- **dut** (*SimHandle*) – Handle to the DUT.
- **reorder_depth** (*int, optional*) – Consider up to *reorder_depth* elements of the expected result list as passing matches. Default is 0, meaning only the first element in the expected result list is considered for a passing match.
- **fail_immediately** (*bool, optional*) – Raise `TestFailure` immediately when something is wrong instead of just recording an error. Default is `True`.

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

Returns If not all expected output was received or error were recorded during the test.

Return type `TestFailure`

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

Can be re-implemented by a subclass.

Parameters

- **got** – The received transaction.
- **exp** – The expected transaction.
- **log** – The logger for reporting messages.
- **strict_type** (*bool, optional*) – Require transaction type to match exactly if `True`, otherwise compare its string representation.

Raises **TestFailure** – If received transaction differed from expected transaction when `fail_immediately` is `True`. If *strict_type* is `True`, also the transaction type must match.

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.

Parameters

- **monitor** – The monitor object.
- **expected_output** – Queue of expected outputs.
- **compare_fn** (*callable, optional*) – Function doing the actual comparison.
- **reorder_depth** (*int, optional*) – Consider up to *reorder_depth* elements of the expected result list as passing matches. Default is 0, meaning only the first element in the expected result list is considered for a passing match.
- **strict_type** (*bool, optional*) – Require transaction type to match exactly if `True`, otherwise compare its string representation.

Raises **TypeError** – If no monitor is on the interface or *compare_fn* is not a callable function.

7.4.4 Clock

class cocotb.clock.**Clock** (*signal*, *period*, *units=None*)
Simple 50:50 duty cycle clock driver.

Instances of this class should call its *start()* method and fork the result. This will create a clocking thread that drives the signal at the desired period/frequency.

Example:

```
c = Clock(dut.clk, 10, 'ns')
cocotb.fork(c.start())
```

Parameters

- **signal** – The clock pin/signal to be driven.
- **period** (*int*) – The clock period. Must convert to an even number of timesteps.

- **units** (*str, optional*) – One of None, 'fs', 'ps', 'ns', 'us', 'ms', 'sec'. When no *units* is given (None) the timestep is determined by the simulator.

start

Clocking coroutine. Start driving your clock by forking a call to this.

Parameters **cycles** (*int, optional*) – Cycle the clock *cycles* number of times, or if None then cycle the clock forever. Note: 0 is not the same as None, as 0 will cycle no times.

7.5 Utilities

cocotb.utils.**get_sim_time**(*units=None*)

Retrieves the simulation time from the simulator.

Parameters **units** (*str or None, optional*) – String specifying the units of the result (one of None, 'fs', 'ps', 'ns', 'us', 'ms', 'sec'). None will return the raw simulation time.

Returns The simulation time in the specified units.

cocotb.utils.**get_time_from_sim_steps**(*steps, units*)

Calculates simulation time in the specified *units* from the *steps* based on the simulator precision.

Parameters

- **steps** (*int*) – Number of simulation steps.
- **units** (*str*) – String specifying the units of the result (one of 'fs', 'ps', 'ns', 'us', 'ms', 'sec').

Returns The simulation time in the specified units.

cocotb.utils.**get_sim_steps**(*time, units=None*)

Calculates the number of simulation time steps for a given amount of *time*.

Parameters

- **time** (*int or float*) – The value to convert to simulation time steps.
- **units** (*str or None, optional*) – String specifying the units of the result (one of None, 'fs', 'ps', 'ns', 'us', 'ms', 'sec'). None means time is already in simulation time steps.

Returns The number of simulation time steps.

Return type int

Raises **ValueError** – If given *time* cannot be represented by simulator precision.

cocotb.utils.**pack**(*ctypes_obj*)

Convert a ctypes structure into a Python string.

Parameters **ctypes_obj** (*ctypes.Structure*) – The ctypes structure to convert to a string.

Returns New Python string containing the bytes from memory holding *ctypes_obj*.

cocotb.utils.**unpack**(*ctypes_obj, string, bytes=None*)

Unpack a Python string into a ctypes structure.

If the length of *string* is not the correct size for the memory footprint of the ctypes structure then the *bytes* keyword argument must be used.

Parameters

- **ctypes_obj** (*ctypes.Structure*) – The ctypes structure to pack into.
- **string** (*str*) – String to copy over the ctypes_obj memory space.
- **bytes** (*int, optional*) – Number of bytes to copy. Defaults to `None`, meaning the length of *string* is used.

Raises

- **ValueError** – If length of *string* and size of *ctypes_obj* are not equal.
- **MemoryError** – If *bytes* is longer than size of *ctypes_obj*.

`cocotb.utils.`**`hexdump`**(*x*)

Hexdump a buffer.

Parameters **x** – Object that supports conversion via the `str` built-in.

Returns A string containing the hexdump.

Example:

```
print(hexdump('this somewhat long string'))
```

```
0000   74 68 69 73 20 73 6F 6D 65 77 68 61 74 20 6C 6F   this somewhat lo
0010   6E 67 20 73 74 72 69 6E 67                        ng string
```

`cocotb.utils.`**`hexdiffs`**(*x*, *y*)

Return a diff string showing differences between two binary strings.

Parameters

- **x** – Object that supports conversion via the `str` built-in.
- **y** – Object that supports conversion via the `str` built-in.

Example:

```
print(hexdiffs('this short thing', 'this also short'))
```

```
0000      746869732073686F 7274207468696E67 this short thing
     0000 7468697320616C73 6F  2073686F7274 this also  short
```

`cocotb.utils.`**`with_metaclass`**(*meta*, **bases*)

This provides:

```
class Foo(with_metaclass(Meta, Base1, Base2)): pass
```

which is a unifying syntax for:

```
# python 3
class Foo(Base1, Base2, metaclass=Meta): pass

# python 2
class Foo(Base1, Base2):
    __metaclass__ = Meta
```

class `cocotb.utils.`**`ParametrizedSingleton`**(**args*, ***kwargs*)

A metaclass that allows class construction to reuse an existing instance.

We use this so that `RisingEdge(sig)` and `Join(coroutine)` always return the same instance, rather than creating new copies.

class `cocotb.utils.`**`nullcontext`**(*enter_result=None*)

Context manager that does no additional processing. Used as a stand-in for a normal context manager, when a particular block of code is only sometimes used with a normal context manager:

```
>>> cm = optional_cm if condition else nullcontext()
>>> with cm:
>>>     # Perform operation, using optional_cm if condition is True
```

`cocotb.utils.`**`reject_remaining_kwargs`**(*name, kwargs*)

Helper function to emulate python 3 keyword-only arguments.

Use as:

```
def func(x1, **kwargs):
    a = kwargs.pop('a', 1)
    b = kwargs.pop('b', 2)
    reject_remaining_kwargs('func', kwargs)
    ...
```

To emulate the Python 3 syntax:

```
def func(x1, *, a=1, b=2):
    ...
```

class `cocotb.utils.`**`lazy_property`**(*fget*)

A property that is executed the first time, then cached forever.

It does this by replacing itself on the instance, which works because unlike *@property* it does not define __set__.

This should be used for expensive members of objects that are not always used.

7.6 Simulation Object Handles

class `cocotb.handle.`**`SimHandleBase`**(*handle, path*)

Bases: `object`

Base class for all simulation objects.

We maintain a handle which we can use for GPI calls.

class `cocotb.handle.`**`RegionObject`**(*handle, path*)

Bases: `cocotb.handle.SimHandleBase`

Region objects don’t have values, they are effectively scopes or namespaces.

class `cocotb.handle.`**`HierarchyObject`**(*handle, path*)

Bases: `cocotb.handle.RegionObject`

Hierarchy objects are namespace/scope objects.

class `cocotb.handle.`**`HierarchyArrayObject`**(*handle, path*)

Bases: `cocotb.handle.RegionObject`

Hierarchy Arrays are containers of Hierarchy Objects.

class `cocotb.handle.`**`AssignmentResult`**(*signal, value*)

Bases: `object`

Object that exists solely to provide an error message if the caller is not aware of cocotb's meaning of <=.

class cocotb.handle.**NonHierarchyObject**(*handle*, *path*)

Bases: *cocotb.handle.SimHandleBase*

Common base class for all non-hierarchy objects.

value

A reference to the value

class cocotb.handle.**ConstantObject**(*handle*, *path*, *handle_type*)

Bases: *cocotb.handle.NonHierarchyObject*

Constant objects have a value that can be read, but not set.

We can also cache the value since it is elaboration time fixed and won't change within a simulation.

class cocotb.handle.**NonHierarchyIndexableObject**(*handle*, *path*)

Bases: *cocotb.handle.NonHierarchyObject*

class cocotb.handle.**NonConstantObject**(*handle*, *path*)

Bases: *cocotb.handle.NonHierarchyIndexableObject*

for ... in drivers()

An iterator for gathering all drivers for a signal.

for ... in loads()

An iterator for gathering all loads on a signal.

class cocotb.handle.**ModifiableObject**(*handle*, *path*)

Bases: *cocotb.handle.NonConstantObject*

Base class for simulator objects whose values can be modified.

setimmediatevalue(*value*)

Set the value of the underlying simulation object to value.

This operation will fail unless the handle refers to a modifiable object, e.g. net, signal or variable.

We determine the library call to make based on the type of the value because assigning integers less than 32 bits is faster.

Parameters **value** (*ctypes.Structure*, *cocotb.binary.BinaryValue*, *int*, *double*) – The value to drive onto the simulator object.

Raises **TypeError** – If target is not wide enough or has an unsupported type for value assignment.

class cocotb.handle.**RealObject**(*handle*, *path*)

Bases: *cocotb.handle.ModifiableObject*

Specific object handle for Real signals and variables.

setimmediatevalue(*value*)

Set the value of the underlying simulation object to value.

This operation will fail unless the handle refers to a modifiable object, e.g. net, signal or variable.

Parameters **value** (*float*) – The value to drive onto the simulator object.

Raises **TypeError** – If target has an unsupported type for real value assignment.

class cocotb.handle.**EnumObject**(*handle*, *path*)

Bases: *cocotb.handle.ModifiableObject*

Specific object handle for enumeration signals and variables.

setimmediatevalue(*value*)
Set the value of the underlying simulation object to value.

This operation will fail unless the handle refers to a modifiable object, e.g. net, signal or variable.

Parameters **value** (*int*) – The value to drive onto the simulator object.

Raises **TypeError** – If target has an unsupported type for integer value assignment.

class cocotb.handle.**IntegerObject**(*handle*, *path*)
Bases: *cocotb.handle.ModifiableObject*

Specific object handle for Integer and Enum signals and variables.

setimmediatevalue(*value*)
Set the value of the underlying simulation object to value.

This operation will fail unless the handle refers to a modifiable object, e.g. net, signal or variable.

Parameters **value** (*int*) – The value to drive onto the simulator object.

Raises **TypeError** – If target has an unsupported type for integer value assignment.

class cocotb.handle.**StringObject**(*handle*, *path*)
Bases: *cocotb.handle.ModifiableObject*

Specific object handle for String variables.

setimmediatevalue(*value*)
Set the value of the underlying simulation object to value.

This operation will fail unless the handle refers to a modifiable object, e.g. net, signal or variable.

Parameters **value** (*str*) – The value to drive onto the simulator object.

Raises **TypeError** – If target has an unsupported type for string value assignment.

cocotb.handle.**SimHandle**(*handle*, *path=None*)
Factory function to create the correct type of *SimHandle* object.

7.7 Implemented Testbench Structures

7.7.1 Drivers

AD9361

Analog Devices AD9361 RF Transceiver.

class cocotb.drivers.ad9361.**AD9361**(*dut*, *rx_channels=1*, *tx_channels=1*, *tx_clock_half_period=16276*, *rx_clock_half_period=16276*, *loopback_queue_maxlen=16*)
Driver for the AD9361 RF Transceiver.

send_data(*i_data*, *q_data*, *i_data2=None*, *q_data2=None*, *binaryRepresentation=BinaryRepresentation.TWOS_COMPLEMENT*)
Forks the `rx_data_to_ad9361` coroutine to send data.

Parameters

- **i_data** (*int*) – Data of the I0 channel.
- **q_data** (*int*) – Data of the Q0 channel.

- **i_data2** (*int, optional*) – Data of the I1 channel.
- **q_data2** (*int, optional*) – Data of the Q1 channel.
- **binaryRepresentation** (BinaryRepresentation) – The representation of the binary value. Default is *TWOS_COMPLEMENT*.

for ... in rx_data_to_ad9361 (*i_data, q_data, i_data2=None, q_data2=None, binaryRepresentation=BinaryRepresentation.TWOS_COMPLEMENT*)

Receive data to AD9361.

This is a coroutine.

Parameters

- **i_data** (*int*) – Data of the I0 channel.
- **q_data** (*int*) – Data of the Q0 channel.
- **i_data2** (*int, optional*) – Data of the I1 channel.
- **q_data2** (*int, optional*) – Data of the Q1 channel.
- **binaryRepresentation** (BinaryRepresentation) – The representation of the binary value. Default is *TWOS_COMPLEMENT*.

ad9361_tx_to_rx_loopback ()

Create loopback from tx to rx.

Forks a coroutine doing the actual task.

tx_data_from_ad9361 ()

Transmit data from AD9361.

Forks a coroutine doing the actual task.

AMBA

Advanced Microcontroller Bus Architecture.

class cocotb.drivers.amba.**AXI4LiteMaster** (*entity, name, clock*)

AXI4-Lite Master.

TODO: Kill all pending transactions if reset is asserted.

for ... in write (*address, value, byte_enable=0xf, address_latency=0, data_latency=0*)

Write a value to an address.

Parameters

- **address** (*int*) – The address to write to.
- **value** (*int*) – The data value to write.
- **byte_enable** (*int, optional*) – Which bytes in value to actually write. Default is to write all bytes.
- **address_latency** (*int, optional*) – Delay before setting the address (in clock cycles). Default is no delay.
- **data_latency** (*int, optional*) – Delay before setting the data value (in clock cycles). Default is no delay.
- **sync** (*bool, optional*) – Wait for rising edge on clock initially. Defaults to True.

Returns The write response value.

Return type *BinaryValue*

Raises **AXIProtocolError** – If write response from AXI is not `OKAY`.

for ... in read(*address*, *sync=True*)

Read from an address.

Parameters

- **address** (`int`) – The address to read from.
- **sync** (`bool, optional`) – Wait for rising edge on clock initially. Defaults to True.

Returns The read data value.

Return type *BinaryValue*

Raises **AXIProtocolError** – If read response from AXI is not `OKAY`.

class cocotb.drivers.amba.**AXI4Slave**(*entity*, *name*, *clock*, *memory*, *callback=None*, *event=None*, *big_endian=False*)

AXI4 Slave

Monitors an internal memory and handles read and write requests.

Avalon

class cocotb.drivers.avalon.**AvalonMM**(*entity*, *name*, *clock*, ***kwargs*)

Bases: `cocotb.drivers.BusDriver`

Avalon Memory Mapped Interface (Avalon-MM) Driver.

Currently we only support the mode required to communicate with SF avalon_mapper which is a limited subset of all the signals.

Blocking operation is all that is supported at the moment, and for the near future as well. Posted responses from a slave are not supported.

class cocotb.drivers.avalon.**AvalonMaster**(*entity*, *name*, *clock*, ***kwargs*)

Avalon Memory Mapped Interface (Avalon-MM) Master

for ... in write(*address*, *value*)

Issue a write to the given address with the specified value.

Parameters

- **address** (`int`) – The address to write to.
- **value** (`int`) – The data value to write.

Raises **TestError** – If master is read-only.

for ... in read(*address*, *sync=True*)

Issue a request to the bus and block until this comes back. Simulation time still progresses but syntactically it blocks.

Parameters

- **address** (`int`) – The address to read from.
- **sync** (`bool, optional`) – Wait for rising edge on clock initially. Defaults to True.

Returns The read data value.

Return type *BinaryValue*

Raises `TestError` – If master is write-only.

class `cocotb.drivers.avalon.`**`AvalonMemory`**(*entity*, *name*, *clock*, *readlatency_min=1*, *readlatency_max=1*, *memory=None*, *avl_properties={}*)

Bases: `cocotb.drivers.BusDriver`

Emulate a memory, with back-door access.

class `cocotb.drivers.avalon.`**`AvalonST`**(**args*, ***kwargs*)

Bases: `cocotb.drivers.ValidatedBusDriver`

Avalon Streaming Interface (Avalon-ST) Driver

class `cocotb.drivers.avalon.`**`AvalonSTPkts`**(**args*, ***kwargs*)

Bases: `cocotb.drivers.ValidatedBusDriver`

Avalon Streaming Interface (Avalon-ST) Driver, packetised.

OPB

class `cocotb.drivers.opb.`**`OPBMaster`**(*entity*, *name*, *clock*)

On-chip peripheral bus master.

for ... in write(*address*, *value*, *sync=True*)

Issue a write to the given address with the specified value.

Parameters

- **address** (`int`) – The address to read from.
- **value** (`int`) – The data value to write.
- **sync** (`bool, optional`) – Wait for rising edge on clock initially. Defaults to True.

Raises `OPBException` – If write took longer than 16 cycles.

for ... in read(*address*, *sync=True*)

Issue a request to the bus and block until this comes back.

Simulation time still progresses but syntactically it blocks.

Parameters

- **address** (`int`) – The address to read from.
- **sync** (`bool, optional`) – Wait for rising edge on clock initially. Defaults to True.

Returns The read data value.

Return type *BinaryValue*

Raises `OPBException` – If read took longer than 16 cycles.

XGMII

class cocotb.drivers.xgmii.**XGMII**(*signal*, *clock*, *interleaved=True*)
Bases: *cocotb.drivers.Driver*

XGMII (10 Gigabit Media Independent Interface) driver.

staticmethod layer1(*packet*)
Take an Ethernet packet (as a string) and format as a layer 1 packet.

Pad to 64 bytes, prepend preamble and append 4-byte CRC on the end.

Parameters **packet** (*str*) – The Ethernet packet to format.

Returns The formatted layer 1 packet.

Return type str

idle()
Helper function to set bus to IDLE state.

terminate(*index*)
Helper function to terminate from a provided lane index.

Parameters **index** (*int*) – The index to terminate.

7.7.2 Monitors

Avalon

class cocotb.monitors.avalon.**AvalonST**(**args*, ***kwargs*)
Bases: *cocotb.monitors.BusMonitor*

Avalon-ST bus.

Non-packetised so each valid word is a separate transaction.

class cocotb.monitors.avalon.**AvalonSTPkts**(**args*, ***kwargs*)
Bases: *cocotb.monitors.BusMonitor*

Packetised Avalon-ST bus.

XGMII

class cocotb.monitors.xgmii.**XGMII**(*signal*, *clock*, *interleaved=True*, *callback=None*, *event=None*)
Bases: *cocotb.monitors.Monitor*

XGMII (10 Gigabit Media Independent Interface) Monitor.

Assumes a single vector, either 4 or 8 bytes plus control bit for each byte.

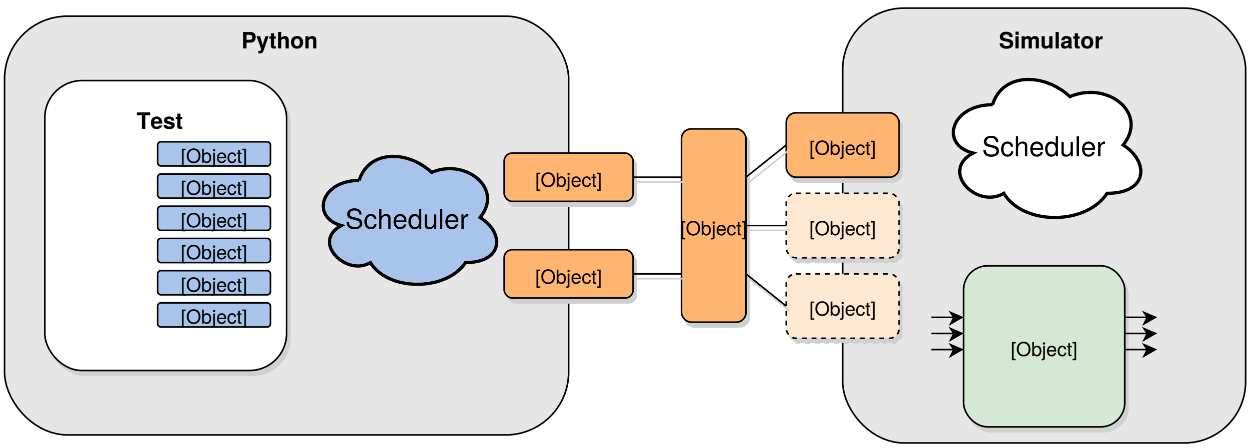
If interleaved is `True` then the control bits are adjacent to the bytes.

CHAPTER

EIGHT

C CODE LIBRARY REFERENCE

Cocotb contains a library called `GPI` (in directory `cocotb/share/lib/gpi/`) written in C++ that is an abstraction layer for the VPI, VHPI, and FLI simulator interfaces.



The interaction between Python and GPI is via a Python extension module called `simulator` (in directory `cocotb/share/lib/simulator/`) which provides routines for traversing the hierarchy, getting/setting an object's value, registering callbacks etc.

8.1 API Documentation

8.1.1 Class list

Class FliEnumObjHdl

`class FliEnumObjHdl` : **`public`** *FliValueObjHdl*

Class FliImpl

class FliImpl : **public** *GpiImplInterface*

Native Check Create

Determine whether a simulation object is native to FLI and create a handle if it is

Get current simulation time

Get current simulation time

NB units depend on the simulation configuration

Find the root handle

Find the root handle using an optional name

Get a handle to the root simulator object. This is usually the toplevel.

If no name is provided, we return the first root instance.

If name is provided, we check the name against the available objects until we find a match. If no match is found we return NULL

Class FliIntObjHdl

class FliIntObjHdl : **public** *FliValueObjHdl*

Class FliIterator

class FliIterator : **public** *GpiIterator*

Find the root handle

Find the root handle using an optional name

Get a handle to the root simulator object. This is usually the toplevel.

If no name is provided, we return the first root instance.

If name is provided, we check the name against the available objects until we find a match. If no match is found we return NULL

Class FliLogicObjHdl

class FliLogicObjHdl : **public** *FliValueObjHdl*

Class FliNextPhaseCbHdl

class FliNextPhaseCbHdl : **public** *FliSimPhaseCbHdl*

Class FliObj

class FliObj

Subclassed by *FliObjHdl*, *FliSignalObjHdl*

Class FliObjHdl

class FliObjHdl : **public** *GpiObjHdl*, **public** *FliObj*

Class FliProcessCbHdl

class FliProcessCbHdl : **public virtual** *GpiCbHdl*

Subclassed by *FliShutdownCbHdl*, *FliSignalCbHdl*, *FliSimPhaseCbHdl*, *FliStartupCbHdl*, *FliTimedCbHdl*

cleanup callback

Called while unwinding after a GPI callback

We keep the process but de-sensitise it

NB need a way to determine if should leave it sensitised, hmmm…

Class FliReadOnlyCbHdl

class FliReadOnlyCbHdl : **public** *FliSimPhaseCbHdl*

Class FliReadWriteCbHdl

class FliReadWriteCbHdl : **public** *FliSimPhaseCbHdl*

Class FliRealObjHdl

class FliRealObjHdl : **public** *FliValueObjHdl*

Class FliShutdownCbHdl

class FliShutdownCbHdl : **public** *FliProcessCbHdl*

cleanup callback

Called while unwinding after a GPI callback

We keep the process but de-sensitise it

NB need a way to determine if should leave it sensitised, hmmm…

Class FliSignalCbHdl

class FliSignalCbHdl : **public** *FliProcessCbHdl*, **public** *GpiValueCbHdl*

cleanup callback

Called while unwinding after a GPI callback

We keep the process but de-sensitise it

NB need a way to determine if should leave it sensitised, hmmm…

Class FliSignalObjHdl

class FliSignalObjHdl : **public** *GpiSignalObjHdl*, **public** *FliObj*
Subclassed by *FliValueObjHdl*

Class FliSimPhaseCbHdl

class FliSimPhaseCbHdl : **public** *FliProcessCbHdl*
Subclassed by *FliNextPhaseCbHdl*, *FliReadOnlyCbHdl*, *FliReadWriteCbHdl*

cleanup callback

Called while unwinding after a GPI callback

We keep the process but de-sensitise it

NB need a way to determine if should leave it sensitised, hmmm…

Class FliStartupCbHdl

class FliStartupCbHdl : **public** *FliProcessCbHdl*

cleanup callback

Called while unwinding after a GPI callback

We keep the process but de-sensitise it

NB need a way to determine if should leave it sensitised, hmmm…

Class FliStringObjHdl

class FliStringObjHdl : **public** *FliValueObjHdl*

Class FliTimedCbHdl

class FliTimedCbHdl : **public** *FliProcessCbHdl*

cleanup callback

Called while unwinding after a GPI callback

We keep the process but de-sensitise it

NB need a way to determine if should leave it sensitised, hmmm…

Class FliTimerCache

class FliTimerCache

Find the root handle

Find the root handle using an optional name

Get a handle to the root simulator object. This is usually the toplevel.

If no name is provided, we return the first root instance.

If name is provided, we check the name against the available objects until we find a match. If no match is found we return NULL

Class FliValueObjHdl

class FliValueObjHdl : **public** *FliSignalObjHdl*

Subclassed by *FliEnumObjHdl*, *FliIntObjHdl*, *FliLogicObjHdl*, *FliRealObjHdl*, *FliStringObjHdl*

Class GpiCbHdl

class GpiCbHdl : **public** *GpiHdl*

Subclassed by *FliProcessCbHdl*, *GpiValueCbHdl*, *VhpiCbHdl*, *VpiCbHdl*

Class GpiClockHdl

class GpiClockHdl

Class GpiHdl

class GpiHdl

Subclassed by *GpiCbHdl*, *GpiIterator*, *GpiObjHdl*

Class GpiImplInterface

class GpiImplInterface

Subclassed by *FliImpl*, *VhpiImpl*, *VpiImpl*

Class GpiIterator

class GpiIterator : **public** *GpiHdl*

Subclassed by *FliIterator*, *VhpiIterator*, *VpiIterator*, *VpiSingleIterator*

Class GpiIteratorMapping

template<class **Ti**, class **Tm**>
class GpiIteratorMapping

Class GpiObjHdl

class GpiObjHdl : **public** *GpiHdl*

Subclassed by *FliObjHdl*, *GpiSignalObjHdl*, *VhpiArrayObjHdl*, *VhpiObjHdl*, *VpiArrayObjHdl*, *VpiObjHdl*

Class GpiSignalObjHdl

class GpiSignalObjHdl : **public** *GpiObjHdl*

Subclassed by *FliSignalObjHdl*, *VhpiSignalObjHdl*, *VpiSignalObjHdl*

Class GpiValueCbHdl

class GpiValueCbHdl : **public virtual** *GpiCbHdl*
Subclassed by *FliSignalCbHdl*, *VhpiValueCbHdl*, *VpiValueCbHdl*

Class VhpiArrayObjHdl

class VhpiArrayObjHdl : **public** *GpiObjHdl*

Class VhpiCbHdl

class VhpiCbHdl : **public virtual** *GpiCbHdl*
Subclassed by *VhpiNextPhaseCbHdl*, *VhpiReadOnlyCbHdl*, *VhpiReadwriteCbHdl*, *VhpiShutdownCbHdl*, *VhpiStartupCbHdl*, *VhpiTimedCbHdl*, *VhpiValueCbHdl*

Class VhpiImpl

class VhpiImpl : **public** *GpiImplInterface*

Class VhpiIterator

class VhpiIterator : **public** *GpiIterator*

Class VhpiLogicSignalObjHdl

class VhpiLogicSignalObjHdl : **public** *VhpiSignalObjHdl*

Class VhpiNextPhaseCbHdl

class VhpiNextPhaseCbHdl : **public** *VhpiCbHdl*

Class VhpiObjHdl

class VhpiObjHdl : **public** *GpiObjHdl*

Class VhpiReadOnlyCbHdl

class VhpiReadOnlyCbHdl : **public** *VhpiCbHdl*

Class VhpiReadwriteCbHdl

class VhpiReadwriteCbHdl : **public** *VhpiCbHdl*

Class VhpiShutdownCbHdl

class VhpiShutdownCbHdl : **public** *VhpiCbHdl*

Class VhpiSignalObjHdl

class VhpiSignalObjHdl : **public** *GpiSignalObjHdl*
Subclassed by *VhpiLogicSignalObjHdl*

Class VhpiStartupCbHdl

class VhpiStartupCbHdl : **public** *VhpiCbHdl*

Class VhpiTimedCbHdl

class VhpiTimedCbHdl : **public** *VhpiCbHdl*

Class VhpiValueCbHdl

class VhpiValueCbHdl : **public** *VhpiCbHdl*, **public** *GpiValueCbHdl*

Class VpiArrayObjHdl

class VpiArrayObjHdl : **public** *GpiObjHdl*

Class VpiCbHdl

class VpiCbHdl : **public virtual** *GpiCbHdl*
Subclassed by *VpiNextPhaseCbHdl*, *VpiReadOnlyCbHdl*, *VpiReadwriteCbHdl*, *VpiShutdownCbHdl*, *VpiStartupCbHdl*, *VpiTimedCbHdl*, *VpiValueCbHdl*

Class VpiImpl

class VpiImpl : **public** *GpiImplInterface*

Class VpiIterator

class VpiIterator : **public** *GpiIterator*

Class VpiNextPhaseCbHdl

class VpiNextPhaseCbHdl : **public** *VpiCbHdl*

Class VpiObjHdl

class VpiObjHdl : **public** *GpiObjHdl*

Class VpiReadOnlyCbHdl

class VpiReadOnlyCbHdl : **public** *VpiCbHdl*

Class VpiReadwriteCbHdl

class VpiReadwriteCbHdl : **public** *VpiCbHdl*

Class VpiShutdownCbHdl

class VpiShutdownCbHdl : **public** *VpiCbHdl*

Class VpiSignalObjHdl

class VpiSignalObjHdl : **public** *GpiSignalObjHdl*

Class VpiSingleIterator

class VpiSingleIterator : **public** *GpiIterator*

Class VpiStartupCbHdl

class VpiStartupCbHdl : **public** *VpiCbHdl*

Class VpiTimedCbHdl

class VpiTimedCbHdl : **public** *VpiCbHdl*

Class VpiValueCbHdl

class VpiValueCbHdl : **public** *VpiCbHdl*, **public** *GpiValueCbHdl*

Class cocotb_entrypoint

class cocotb_entrypoint

Class cocotb_entrypoint::cocotb_arch

class cocotb_arch

8.1.2 File list

File FliCbHdl.cpp

File FliImpl.cpp

Find the root handle

Find the root handle using an optional name

Get a handle to the root simulator object. This is usually the toplevel.

If no name is provided, we return the first root instance.

If name is provided, we check the name against the available objects until we find a match. If no match is found we return NULL

void **fli_mappings** (*GpiIteratorMapping*<int, *FliIterator*::OneToMany> &*map*)

void **handle_fli_callback** (void **data*)

static void **register_initial_callback** (void)

static void **register_final_callback** (void)

static void **register_embed** (void)

void **cocotb_init** (void)

GPI_ENTRY_POINT (fli, *register_embed*)

Variables

FliProcessCbHdl ***sim_init_cb**

FliProcessCbHdl ***sim_finish_cb**

FliImpl ***fli_table**

File FliImpl.h

Functions

void **cocotb_init** (void)

void **handle_fli_callback** (void **data*)

class FliProcessCbHdl : **public virtual** *GpiCbHdl*

Subclassed by *FliShutdownCbHdl*, *FliSignalCbHdl*, *FliSimPhaseCbHdl*, *FliStartupCbHdl*, *FliTimedCbHdl*

cleanup callback

Called while unwinding after a GPI callback

We keep the process but de-sensitise it

NB need a way to determine if should leave it sensitised, hmmm...

int **cleanup_callback** (void)

Public Functions

FliProcessCbHdl (*GpiImplInterface* **impl*)

virtual ~FliProcessCbHdl ()

virtual int **arm_callback** (void) = 0

Protected Attributes

mtiProcessIdT **m_proc_hdl**

bool **m_sensitised**

class FliSignalCbHdl : **public** *FliProcessCbHdl*, **public** *GpiValueCbHdl*

cleanup callback

Called while unwinding after a GPI callback

We keep the process but de-sensitise it

NB need a way to determine if should leave it sensitised, hmmm...

FliSignalCbHdl (*GpiImplInterface* **impl*, *FliSignalObjHdl* **sig_hdl*, unsigned int *edge*)

int **arm_callback** (void)

Public Functions

virtual ~FliSignalCbHdl()

int **cleanup_callback**(void)

Private Members

mtiSignalIdT **m_sig_hdl**

class FliSimPhaseCbHdl : **public** *FliProcessCbHdl*
Subclassed by *FliNextPhaseCbHdl*, *FliReadOnlyCbHdl*, *FliReadWriteCbHdl*

cleanup callback

Called while unwinding after a GPI callback

We keep the process but de-sensitise it

NB need a way to determine if should leave it sensitised, hmmm…

int **arm_callback**(void)

Public Functions

FliSimPhaseCbHdl(*GpiImplInterface* **impl*, mtiProcessPriorityT *priority*)

virtual ~FliSimPhaseCbHdl()

Protected Attributes

mtiProcessPriorityT **m_priority**

class FliReadWriteCbHdl : **public** *FliSimPhaseCbHdl*

Public Functions

FliReadWriteCbHdl(*GpiImplInterface* **impl*)

virtual ~FliReadWriteCbHdl()

class FliNextPhaseCbHdl : **public** *FliSimPhaseCbHdl*

Public Functions

FliNextPhaseCbHdl(*GpiImplInterface* **impl*)

virtual ~FliNextPhaseCbHdl()

class FliReadOnlyCbHdl : **public** *FliSimPhaseCbHdl*

Public Functions

FliReadOnlyCbHdl (*GpiImplInterface *impl*)

virtual ~FliReadOnlyCbHdl ()

class FliStartupCbHdl : **public** *FliProcessCbHdl*

cleanup callback

Called while unwinding after a GPI callback

We keep the process but de-sensitise it

NB need a way to determine if should leave it sensitised, hmmm…

int **arm_callback** (void)

int **run_callback** (void)

Public Functions

FliStartupCbHdl (*GpiImplInterface *impl*)

virtual ~FliStartupCbHdl ()

class FliShutdownCbHdl : **public** *FliProcessCbHdl*

cleanup callback

Called while unwinding after a GPI callback

We keep the process but de-sensitise it

NB need a way to determine if should leave it sensitised, hmmm…

int **arm_callback** (void)

int **run_callback** (void)

Public Functions

FliShutdownCbHdl (*GpiImplInterface *impl*)

virtual ~FliShutdownCbHdl ()

class FliTimedCbHdl : **public** *FliProcessCbHdl*

cleanup callback

Called while unwinding after a GPI callback

We keep the process but de-sensitise it

NB need a way to determine if should leave it sensitised, hmmm. . .

FliTimedCbHdl (*GpiImplInterface* **impl*, uint64_t *time_ps*)

int **arm_callback** (void)

int **cleanup_callback** (void)

Public Functions

virtual ~FliTimedCbHdl ()

void **reset_time** (uint64_t *new_time*)

Private Members

uint64_t **m_time_ps**

class FliObj
Subclassed by *FliObjHdl*, *FliSignalObjHdl*

Public Functions

FliObj (int *acc_type*, int *acc_full_type*)

virtual ~FliObj ()

int **get_acc_type** (void)

int **get_acc_full_type** (void)

Protected Attributes

int **m_acc_type**

int **m_acc_full_type**

class FliObjHdl : **public** *GpiObjHdl*, **public** *FliObj*

Public Functions

FliObjHdl (*GpiImplInterface* **impl*, void **hdl*, gpi_objtype_t *objtype*, int *acc_type*, int *acc_full_type*)

FliObjHdl (*GpiImplInterface* **impl*, void **hdl*, gpi_objtype_t *objtype*, int *acc_type*, int *acc_full_type*, bool *is_const*)

virtual ~FliObjHdl ()

int **initialise** (std::string &*name*, std::string &*fq_name*)

class FliSignalObjHdl : **public** *GpiSignalObjHdl*, **public** *FliObj*
Subclassed by *FliValueObjHdl*

Public Functions

FliSignalObjHdl (*GpiImplInterface* **impl*, void **hdl*, gpi_objtype_t *objtype*, bool *is_const*, int *acc_type*, int *acc_full_type*, bool *is_var*)

virtual ~FliSignalObjHdl ()

GpiCbHdl ***value_change_cb** (unsigned int *edge*)

int **initialise** (std::string &*name*, std::string &*fq_name*)

bool **is_var** (void)

Protected Attributes

bool **m_is_var**

FliSignalCbHdl **m_rising_cb**

FliSignalCbHdl **m_falling_cb**

FliSignalCbHdl **m_either_cb**

class FliValueObjHdl : **public** *FliSignalObjHdl*
Subclassed by *FliEnumObjHdl*, *FliIntObjHdl*, *FliLogicObjHdl*, *FliRealObjHdl*, *FliStringObjHdl*

Public Functions

FliValueObjHdl (*GpiImplInterface* **impl*, void **hdl*, gpi_objtype_t *objtype*, bool *is_const*, int *acc_type*, int *acc_full_type*, bool *is_var*, mtiTypeIdT *valType*, mtiTypeKindT *typeKind*)

virtual ~FliValueObjHdl ()

const char ***get_signal_value_binstr** (void)

const char ***get_signal_value_str** (void)

double **get_signal_value_real** (void)

long **get_signal_value_long** (void)

int **set_signal_value** (**const** long *value*)

int **set_signal_value** (**const** double *value*)

int **set_signal_value** (std::string &*value*)

void ***get_sub_hdl** (int *index*)

int **initialise** (std::string &*name*, std::string &*fq_name*)

mtiTypeKindT **get_fli_typekind** (void)

mtiTypeIdT **get_fli_typeid** (void)

Protected Attributes

mtiTypeKindT **m_fli_type**

mtiTypeIdT **m_val_type**

char ***m_val_buff**

void ****m_sub_hdls**

class FliEnumObjHdl : **public** *FliValueObjHdl*

Public Functions

FliEnumObjHdl(*GpiImplInterface* **impl*, void **hdl*, gpi_objtype_t *objtype*, bool *is_const*, int *acc_type*, int *acc_full_type*, bool *is_var*, mtiTypeIdT *valType*, mtiTypeKindT *typeKind*)

virtual ~FliEnumObjHdl()

const char ***get_signal_value_str**(void)

long **get_signal_value_long**(void)

int **set_signal_value**(**const** long *value*)

int **initialise**(std::string &*name*, std::string &*fq_name*)

Private Members

char ****m_value_enum**

mtiInt32T **m_num_enum**

class FliLogicObjHdl : **public** *FliValueObjHdl*

Public Functions

FliLogicObjHdl(*GpiImplInterface* **impl*, void **hdl*, gpi_objtype_t *objtype*, bool *is_const*, int *acc_type*, int *acc_full_type*, bool *is_var*, mtiTypeIdT *valType*, mtiTypeKindT *typeKind*)

virtual ~FliLogicObjHdl()

const char ***get_signal_value_binstr**(void)

int **set_signal_value**(**const** long *value*)

int **set_signal_value**(std::string &*value*)

int **initialise**(std::string &*name*, std::string &*fq_name*)

Private Members

char ***m_mti_buff**

char ****m_value_enum**

mtiInt32T **m_num_enum**

std::map<char, mtiInt32T> **m_enum_map**

class FliIntObjHdl : **public** *FliValueObjHdl*

Public Functions

FliIntObjHdl (*GpiImplInterface* **impl*, void **hdl*, gpi_objtype_t *objtype*, bool *is_const*, int *acc_type*, int *acc_full_type*, bool *is_var*, mtiTypeIdT *valType*, mtiTypeKindT *typeKind*)

virtual ~FliIntObjHdl ()

const char ***get_signal_value_binstr** (void)

long **get_signal_value_long** (void)

int **set_signal_value** (**const** long *value*)

int **initialise** (std::string &*name*, std::string &*fq_name*)

class FliRealObjHdl : **public** *FliValueObjHdl*

Public Functions

FliRealObjHdl (*GpiImplInterface* **impl*, void **hdl*, gpi_objtype_t *objtype*, bool *is_const*, int *acc_type*, int *acc_full_type*, bool *is_var*, mtiTypeIdT *valType*, mtiTypeKindT *typeKind*)

virtual ~FliRealObjHdl ()

double **get_signal_value_real** (void)

int **set_signal_value** (**const** double *value*)

int **initialise** (std::string &*name*, std::string &*fq_name*)

Private Members

double ***m_mti_buff**

class FliStringObjHdl : **public** *FliValueObjHdl*

Public Functions

FliStringObjHdl(*GpiImplInterface* **impl*, void **hdl*, gpi_objtype_t *objtype*, bool *is_const*, int *acc_type*, int *acc_full_type*, bool *is_var*, mtiTypeIdT *valType*, mtiTypeKindT *typeKind*)

virtual ~FliStringObjHdl()

const char ***get_signal_value_str**(void)

int **set_signal_value**(std::string &*value*)

int **initialise**(std::string &*name*, std::string &*fq_name*)

Private Members

char ***m_mti_buff**

class FliTimerCache

Find the root handle

Find the root handle using an optional name

Get a handle to the root simulator object. This is usually the toplevel.

If no name is provided, we return the first root instance.

If name is provided, we check the name against the available objects until we find a match. If no match is found we return NULL

FliTimedCbHdl ***get_timer**(uint64_t *time_ps*)

void **put_timer**(*FliTimedCbHdl* **hdl*)

Public Functions

FliTimerCache(*FliImpl* **impl*)

~FliTimerCache()

Private Members

std::queue<*FliTimedCbHdl* *> **free_list**

FliImpl ***impl**

class FliIterator : **public** *GpiIterator*

Find the root handle

Find the root handle using an optional name

Get a handle to the root simulator object. This is usually the toplevel.

If no name is provided, we return the first root instance.

If name is provided, we check the name against the available objects until we find a match. If no match is found we return NULL

GpiIteratorMapping<int, *FliIterator*::OneToMany> **iterate_over**

FliIterator (*GpiImplInterface* **impl*, *GpiObjHdl* **hdl*)

GpiIterator::Status **next_handle** (std::string &*name*, *GpiObjHdl* ***hdl*, void ***raw_hdl*)

void **populate_handle_list** (*OneToMany childType*)

Public Types

enum OneToMany

Values:

OTM_END = 0

OTM_CONSTANTS

OTM_SIGNALS

OTM_REGIONS

OTM_SIGNAL_SUB_ELEMENTS

OTM_VARIABLE_SUB_ELEMENTS

Public Functions

virtual ~FliIterator ()

Private Members

std::vector<*OneToMany*> ***selected**

std::vector<*OneToMany*>::iterator **one2many**

std::vector<void *> **m_vars**

std::vector<void *> **m_sigs**

std::vector<void *> **m_regs**

std::vector<void *> ***m_currentHandles**

std::vector<void *>::iterator **m_iterator**

class FliImpl : **public** *GpiImplInterface*

Native Check Create

Determine whether a simulation object is native to FLI and create a handle if it is

GpiObjHdl ***native_check_create**(std::string &*name*, *GpiObjHdl* **parent*)

GpiObjHdl ***native_check_create**(int32_t *index*, *GpiObjHdl* **parent*)

const char ***reason_to_string**(int *reason*)

Get current simulation time

Get current simulation time

NB units depend on the simulation configuration

void **get_sim_time**(uint32_t **high*, uint32_t **low*)

void **get_sim_precision**(int32_t **precision*)

Find the root handle

Find the root handle using an optional name

Get a handle to the root simulator object. This is usually the toplevel.

If no name is provided, we return the first root instance.

If name is provided, we check the name against the available objects until we find a match. If no match is found we return NULL

GpiObjHdl ***get_root_handle**(**const** char **name*)

GpiIterator ***iterate_handle**(*GpiObjHdl* **obj_hdl*, gpi_iterator_sel_t *type*)

GpiCbHdl ***register_timed_callback**(uint64_t *time_ps*)

GpiCbHdl ***register_readonly_callback**(void)

GpiCbHdl ***register_nexttime_callback**(void)

GpiCbHdl ***register_readwrite_callback**(void)

int **deregister_callback**(*GpiCbHdl* **obj_hdl*)

Public Functions

FliImpl(**const** std::string &*name*)

void **sim_end**(void)

GpiObjHdl ***native_check_create**(void **raw_hdl*, *GpiObjHdl* **paret*)

GpiObjHdl ***create_gpi_obj_from_handle**(void **hdl*, std::string &*name*, std::string &*fq_name*, int *accType*, int *accFullType*)

Public Members

FliTimerCache **cache**

Private Functions

bool **isValueConst** (int *kind*)

bool **isValueLogic** (mtiTypeIdT *type*)

bool **isValueChar** (mtiTypeIdT *type*)

bool **isValueBoolean** (mtiTypeIdT *type*)

bool **isTypeValue** (int *type*)

bool **isTypeSignal** (int *type*, int *full_type*)

Private Members

FliReadOnlyCbHdl **m_readonly_cbhdl**

FliNextPhaseCbHdl **m_nexttime_cbhdl**

FliReadWriteCbHdl **m_readwrite_cbhdl**

File FliObjHdl.cpp

File GpiCbHdl.cpp

Defines

ret = #_X; \ break]

File GpiCommon.cpp

Defines

CHECK_AND_STORE (_x) _x

DOT_LIB_EXT "." xstr(LIB_EXT)

Functions

int **gpi_print_registered_impl** (void)

int **gpi_register_impl** (*GpiImplInterface* **func_tbl*)

void **gpi_embed_init** (gpi_sim_info_t **info*)

void **gpi_embed_end** (void)

void **gpi_sim_end** (void)

void **gpi_embed_event** (gpi_event_t *level*, **const** char **msg*) static void **gpi_load_libs** (std::vector<std::string> *to_load*)

void **gpi_load_extra_libs** (void)

void **gpi_get_sim_time** (uint32_t **high*, uint32_t **low*)

void **gpi_get_sim_precision** (int32_t **precision*)

gpi_sim_hdl **gpi_get_root_handle** (const char **name*)

static GpiObjHdl ***__gpi_get_handle_by_name** (GpiObjHdl **parent*, std::string *name*, GpiImplInterface **skip_impl*)

static GpiObjHdl ***__gpi_get_handle_by_raw** (GpiObjHdl **parent*, void **raw_hdl*, GpiImplInterface **skip_impl*)

gpi_sim_hdl **gpi_get_handle_by_name** (gpi_sim_hdl *parent*, const char **name*)

gpi_sim_hdl **gpi_get_handle_by_index** (gpi_sim_hdl *parent*, int32_t *index*)

gpi_iterator_hdl **gpi_iterate** (gpi_sim_hdl *base*, gpi_iterator_sel_t *type*)

gpi_sim_hdl **gpi_next** (gpi_iterator_hdl *iterator*)

const char ***gpi_get_definition_name** (gpi_sim_hdl *sig_hdl*)

const char ***gpi_get_definition_file** (gpi_sim_hdl *sig_hdl*)

const char ***gpi_get_signal_value_binstr** (gpi_sim_hdl *sig_hdl*)

const char ***gpi_get_signal_value_str** (gpi_sim_hdl *sig_hdl*)

double **gpi_get_signal_value_real** (gpi_sim_hdl *sig_hdl*)

long **gpi_get_signal_value_long** (gpi_sim_hdl *sig_hdl*)

const char ***gpi_get_signal_name_str** (gpi_sim_hdl *sig_hdl*)

const char ***gpi_get_signal_type_str** (gpi_sim_hdl *sig_hdl*)

gpi_objtype_t **gpi_get_object_type** (gpi_sim_hdl *sig_hdl*)

int **gpi_is_constant** (gpi_sim_hdl *sig_hdl*)

int **gpi_is_indexable** (gpi_sim_hdl *sig_hdl*)

void **gpi_set_signal_value_long** (gpi_sim_hdl *sig_hdl*, long *value*)

void **gpi_set_signal_value_str** (gpi_sim_hdl *sig_hdl*, const char **str*)

void **gpi_set_signal_value_real** (gpi_sim_hdl *sig_hdl*, double *value*)

int **gpi_get_num_elems** (gpi_sim_hdl *sig_hdl*)

int **gpi_get_range_left** (gpi_sim_hdl *sig_hdl*)

int **gpi_get_range_right** (gpi_sim_hdl *sig_hdl*)

gpi_sim_hdl **gpi_register_value_change_callback** (int (**gpi_function*)) const void *
, void **gpi_cb_data*, gpi_sim_hdl *sig_hdl*, unsigned int *edge*

gpi_sim_hdl **gpi_register_timed_callback** (int (**gpi_function*)) const void *
, void **gpi_cb_data*, uint64_t *time_ps*

gpi_sim_hdl **gpi_register_readonly_callback** (int (**gpi_function*)) const void *
, void **gpi_cb_data*

gpi_sim_hdl **gpi_register_nexttime_callback** (int (**gpi_function*)) const void *
, void **gpi_cb_data*

gpi_sim_hdl **gpi_register_readwrite_callback** (int (**gpi_function*)) **const** void *
, void **gpi_cb_data*

gpi_sim_hdl **gpi_create_clock** (gpi_sim_hdl *clk_signal*, **const** int *period*)

void **gpi_stop_clock** (gpi_sim_hdl *clk_object*)

void **gpi_deregister_callback** (gpi_sim_hdl *hdl*)

Variables

vector<*GpiImplInterface* *> **registered_impls**

File VhpiCbHdl.cpp

Defines

VHPI_TYPE_MIN (1000)

Functions

void **handle_vhpi_callback** (**const** vhpiCbDataT **cb_data*)

bool **get_range** (vhpiHandleT *hdl*, vhpiIntT *dim*, int **left*, int **right*)

void **vhpi_mappings** (*GpiIteratorMapping*<vhpiClassKindT, vhpiOneToManyT> &*map*)

File VhpiImpl.cpp

Defines

CASE_STR (_X) case _X: return #_X

Functions

bool **is_const** (vhpiHandleT *hdl*)

bool **is_enum_logic** (vhpiHandleT *hdl*)

bool **is_enum_char** (vhpiHandleT *hdl*)

bool **is_enum_boolean** (vhpiHandleT *hdl*)

void **handle_vhpi_callback** (**const** vhpiCbDataT **cb_data*)

static void **register_initial_callback** (void)

static void **register_final_callback** (void)

static void **register_embed** (void)

void **vhpi_startup_routines_bootstrap** (void)

Variables

VhpiCbHdl ***sim_init_cb**

VhpiCbHdl ***sim_finish_cb**

VhpiImpl ***vhpi_table**

void(* vhpi_startup_routines[])(void)= { register_embed, register_initial_callback,

File VhpiImpl.h

Defines

GEN_IDX_SEP_LHS "__"

GEN_IDX_SEP_RHS ""

__check_vhpi_error(__FILE__, __func__, __LINE__); \ } while (0)]

Functions

static int **__check_vhpi_error** (**const** char **file*, **const** char **func*, long *line*)

class VhpiCbHdl : **public virtual** *GpiCbHdl*

Subclassed by *VhpiNextPhaseCbHdl*, *VhpiReadOnlyCbHdl*, *VhpiReadwriteCbHdl*, *VhpiShutdownCbHdl*, *VhpiStartupCbHdl*, *VhpiTimedCbHdl*, *VhpiValueCbHdl*

Public Functions

VhpiCbHdl (*GpiImplInterface* **impl*)

virtual ~VhpiCbHdl ()

int **arm_callback** (void)

int **cleanup_callback** (void)

Protected Attributes

vhpiCbDataT **cb_data**

vhpiTimeT **vhpi_time**

class VhpiValueCbHdl : **public** *VhpiCbHdl*, **public** *GpiValueCbHdl*

Public Functions

VhpiValueCbHdl (*GpiImplInterface* **impl*, *VhpiSignalObjHdl* **sig*, int *edge*)

virtual ~VhpiValueCbHdl ()

int **cleanup_callback** (void)

Private Members

std::string **initial_value**

bool **rising**

bool **falling**

VhpiSignalObjHdl ***signal**

class VhpiTimedCbHdl : **public** *VhpiCbHdl*

Public Functions

VhpiTimedCbHdl (*GpiImplInterface* **impl*, uint64_t *time_ps*)

virtual ~VhpiTimedCbHdl ()

int **cleanup_callback** ()

class VhpiReadOnlyCbHdl : **public** *VhpiCbHdl*

Public Functions

VhpiReadOnlyCbHdl (*GpiImplInterface* **impl*)

virtual ~VhpiReadOnlyCbHdl ()

class VhpiNextPhaseCbHdl : **public** *VhpiCbHdl*

Public Functions

VhpiNextPhaseCbHdl (*GpiImplInterface* **impl*)

virtual ~VhpiNextPhaseCbHdl ()

class VhpiStartupCbHdl : **public** *VhpiCbHdl*

Public Functions

VhpiStartupCbHdl(*GpiImplInterface* **impl*)

int **run_callback**(void)

int **cleanup_callback**(void)

virtual ~VhpiStartupCbHdl()

class VhpiShutdownCbHdl : **public** *VhpiCbHdl*

Public Functions

VhpiShutdownCbHdl(*GpiImplInterface* **impl*)

int **run_callback**(void)

int **cleanup_callback**(void)

virtual ~VhpiShutdownCbHdl()

class VhpiReadwriteCbHdl : **public** *VhpiCbHdl*

Public Functions

VhpiReadwriteCbHdl(*GpiImplInterface* **impl*)

virtual ~VhpiReadwriteCbHdl()

class VhpiArrayObjHdl : **public** *GpiObjHdl*

Public Functions

VhpiArrayObjHdl(*GpiImplInterface* **impl*, vhpiHandleT *hdl*, gpi_objtype_t *objtype*)

virtual ~VhpiArrayObjHdl()

int **initialise**(std::string &*name*, std::string &*fq_name*)

class VhpiObjHdl : **public** *GpiObjHdl*

Public Functions

VhpiObjHdl(*GpiImplInterface* **impl*, vhpiHandleT *hdl*, gpi_objtype_t *objtype*)

virtual ~VhpiObjHdl()

int **initialise**(std::string &*name*, std::string &*fq_name*)

class VhpiSignalObjHdl : **public** *GpiSignalObjHdl*

Subclassed by *VhpiLogicSignalObjHdl*

Public Functions

VhpiSignalObjHdl (*GpiImplInterface* **impl*, vhpiHandleT *hdl*, gpi_objtype_t *objtype*, bool *is_const*)

~VhpiSignalObjHdl ()

const char ***get_signal_value_binstr** (void)

const char ***get_signal_value_str** (void)

double **get_signal_value_real** (void)

long **get_signal_value_long** (void)

int **set_signal_value** (**const** long *value*)

int **set_signal_value** (**const** double *value*)

int **set_signal_value** (std::string &*value*)

GpiCbHdl ***value_change_cb** (unsigned int *edge*)

int **initialise** (std::string &*name*, std::string &*fq_name*)

Protected Functions

const vhpiEnumT **chr2vhpi** (**const** char *value*)

Protected Attributes

vhpiValueT **m_value**

vhpiValueT **m_binvalue**

VhpiValueCbHdl **m_rising_cb**

VhpiValueCbHdl **m_falling_cb**

VhpiValueCbHdl **m_either_cb**

class VhpiLogicSignalObjHdl : **public** *VhpiSignalObjHdl*

Public Functions

VhpiLogicSignalObjHdl (*GpiImplInterface* **impl*, vhpiHandleT *hdl*, gpi_objtype_t *objtype*, bool *is_const*)

virtual ~VhpiLogicSignalObjHdl ()

int **set_signal_value** (**const** long *value*)

int **set_signal_value** (std::string &*value*)

int **initialise** (std::string &*name*, std::string &*fq_name*)

class VhpiIterator : **public** *GpiIterator*

Public Functions

VhpiIterator (*GpiImplInterface* **impl*, *GpiObjHdl* **hdl*)

~VhpiIterator ()

GpiIterator::Status **next_handle** (std::string &*name*, *GpiObjHdl* ***hdl*, void ***raw_hdl*)

Private Members

vhpiHandleT **m_iterator**

vhpiHandleT **m_iter_obj**

std::vector<vhpiOneToManyT> ***selected**

std::vector<vhpiOneToManyT>::iterator **one2many**

Private Static Attributes

GpiIteratorMapping<vhpiClassKindT, vhpiOneToManyT> **iterate_over**

class VhpiImpl : **public** *GpiImplInterface*

Public Functions

VhpiImpl (**const** std::string &*name*)

void **sim_end** (void)

void **get_sim_time** (uint32_t **high*, uint32_t **low*)

void **get_sim_precision** (int32_t **precision*)

GpiObjHdl ***get_root_handle** (**const** char **name*)

GpiIterator ***iterate_handle** (*GpiObjHdl* **obj_hdl*, gpi_iterator_sel_t *type*)

GpiCbHdl ***register_timed_callback** (uint64_t *time_ps*)

GpiCbHdl ***register_readonly_callback** (void)

GpiCbHdl ***register_nexttime_callback** (void)

GpiCbHdl ***register_readwrite_callback** (void)

int **deregister_callback** (*GpiCbHdl* **obj_hdl*)

GpiObjHdl ***native_check_create** (std::string &*name*, *GpiObjHdl* **parent*)

GpiObjHdl ***native_check_create** (int32_t *index*, *GpiObjHdl* **parent*)

GpiObjHdl ***native_check_create** (void **raw_hdl*, *GpiObjHdl* **parent*)

const char ***reason_to_string** (int *reason*)

const char ***format_to_string** (int *format*)

GpiObjHdl ***create_gpi_obj_from_handle**(vhpiHandleT *new_hdl*, std::string &*name*, std::string &*fq_name*)

Private Members

VhpiReadwriteCbHdl **m_read_write**

VhpiNextPhaseCbHdl **m_next_phase**

VhpiReadOnlyCbHdl **m_read_only**

File VpiCbHdl.cpp

Defines

VPI_TYPE_MAX (1000)

Functions

int32_t **handle_vpi_callback**(p_cb_data *cb_data*)

void **vpi_mappings**(*GpiIteratorMapping*<int32_t, int32_t> &*map*)

File VpiImpl.cpp

Defines

CASE_STR(_X) case _X: return #_X

Functions

gpi_objtype_t **to_gpi_objtype**(int32_t *vpitype*)

int32_t **handle_vpi_callback**(p_cb_data *cb_data*)

static void **register_embed**(void)

static void **register_initial_callback**(void)

static void **register_final_callback**(void)

static int **system_function_compiletf**(char **userdata*)

static int **system_function_overload**(char **userdata*)

static void **register_system_functions**(void)

void **vlog_startup_routines_bootstrap**(void)

Variables

VpiCbHdl ***sim_init_cb**

VpiCbHdl ***sim_finish_cb**

VpiImpl ***vpi_table**

int **systf_info_level** = GPIInfo

int **systf_warning_level** = GPIWarning

int **systf_error_level** = GPIError

int **systf_fatal_level** = GPICritical

void(* vlog_startup_routines[])(void)= { register_embed, register_system_functions,

File VpiImpl.h

Defines

__check_vpi_error(__FILE__, __func__, __LINE__); \ } while (0)]

Functions

static int **__check_vpi_error** (**const** char **file*, **const** char **func*, long *line*)

class VpiCbHdl : **public virtual** *GpiCbHdl*
Subclassed by *VpiNextPhaseCbHdl*, *VpiReadOnlyCbHdl*, *VpiReadwriteCbHdl*, *VpiShutdownCbHdl*, *VpiStartupCbHdl*, *VpiTimedCbHdl*, *VpiValueCbHdl*

Public Functions

VpiCbHdl (*GpiImplInterface* **impl*)

virtual ~VpiCbHdl ()

int **arm_callback** (void)

int **cleanup_callback** (void)

Protected Attributes

s_cb_data **cb_data**

s_vpi_time **vpi_time**

class VpiValueCbHdl : **public** *VpiCbHdl*, **public** *GpiValueCbHdl*

Public Functions

VpiValueCbHdl (*GpiImplInterface *impl*, *VpiSignalObjHdl *sig*, int *edge*)

virtual ~VpiValueCbHdl ()

int **cleanup_callback** (void)

Private Members

s_vpi_value **m_vpi_value**

class VpiTimedCbHdl : **public** *VpiCbHdl*

Public Functions

VpiTimedCbHdl (*GpiImplInterface *impl*, uint64_t *time_ps*)

virtual ~VpiTimedCbHdl ()

int **cleanup_callback** ()

class VpiReadOnlyCbHdl : **public** *VpiCbHdl*

Public Functions

VpiReadOnlyCbHdl (*GpiImplInterface *impl*)

virtual ~VpiReadOnlyCbHdl ()

class VpiNextPhaseCbHdl : **public** *VpiCbHdl*

Public Functions

VpiNextPhaseCbHdl (*GpiImplInterface *impl*)

virtual ~VpiNextPhaseCbHdl ()

class VpiReadwriteCbHdl : **public** *VpiCbHdl*

Public Functions

VpiReadwriteCbHdl (*GpiImplInterface *impl*)

virtual ~VpiReadwriteCbHdl ()

class VpiStartupCbHdl : **public** *VpiCbHdl*

Public Functions

VpiStartupCbHdl (*GpiImplInterface* **impl*)

int **run_callback** (void)

int **cleanup_callback** (void)

virtual ~VpiStartupCbHdl ()

class VpiShutdownCbHdl : **public** *VpiCbHdl*

Public Functions

VpiShutdownCbHdl (*GpiImplInterface* **impl*)

int **run_callback** (void)

int **cleanup_callback** (void)

virtual ~VpiShutdownCbHdl ()

class VpiArrayObjHdl : **public** *GpiObjHdl*

Public Functions

VpiArrayObjHdl (*GpiImplInterface* **impl*, vpiHandle *hdl*, gpi_objtype_t *objtype*)

virtual ~VpiArrayObjHdl ()

int **initialise** (std::string &*name*, std::string &*fq_name*)

class VpiObjHdl : **public** *GpiObjHdl*

Public Functions

VpiObjHdl (*GpiImplInterface* **impl*, vpiHandle *hdl*, gpi_objtype_t *objtype*)

virtual ~VpiObjHdl ()

int **initialise** (std::string &*name*, std::string &*fq_name*)

class VpiSignalObjHdl : **public** *GpiSignalObjHdl*

Public Functions

VpiSignalObjHdl (*GpiImplInterface* **impl*, vpiHandle *hdl*, gpi_objtype_t *objtype*, bool *is_const*)

virtual ~VpiSignalObjHdl ()

const char ***get_signal_value_binstr** (void)

const char ***get_signal_value_str** (void)

double **get_signal_value_real** (void)

long **get_signal_value_long** (void)

int **set_signal_value** (**const** long *value*)

int **set_signal_value** (**const** double *value*)

int **set_signal_value** (std::string &*value*)

GpiCbHdl ***value_change_cb** (unsigned int *edge*)

int **initialise** (std::string &*name*, std::string &*fq_name*)

Private Functions

int **set_signal_value** (s_vpi_value *value*)

Private Members

VpiValueCbHdl **m_rising_cb**

VpiValueCbHdl **m_falling_cb**

VpiValueCbHdl **m_either_cb**

class VpiIterator : **public** *GpiIterator*

Public Functions

VpiIterator (*GpiImplInterface* **impl*, *GpiObjHdl* **hdl*)

~VpiIterator ()

GpiIterator::Status **next_handle** (std::string &*name*, *GpiObjHdl* ***hdl*, void ***raw_hdl*)

Private Members

vpiHandle **m_iterator**

std::vector<int32_t> ***selected**

std::vector<int32_t>::iterator **one2many**

Private Static Attributes

GpiIteratorMapping<int32_t, int32_t> **iterate_over**

class VpiSingleIterator : **public** *GpiIterator*

Public Functions

VpiSingleIterator (*GpiImplInterface* *impl, *GpiObjHdl* *hdl, int32_t *vpitype*)

virtual ~VpiSingleIterator ()

GpiIterator::Status **next_handle** (std::string &*name*, *GpiObjHdl* ***hdl*, void ***raw_hdl*)

Protected Attributes

vpiHandle **m_iterator**

class VpiImpl : **public** *GpiImplInterface*

Public Functions

VpiImpl (**const** std::string &*name*)

void **sim_end** (void)

void **get_sim_time** (uint32_t **high*, uint32_t **low*)

void **get_sim_precision** (int32_t **precision*)

GpiObjHdl ***get_root_handle** (**const** char **name*)

GpiIterator ***iterate_handle** (*GpiObjHdl* **obj_hdl*, gpi_iterator_sel_t *type*)

GpiObjHdl ***next_handle** (*GpiIterator* **iter*)

GpiCbHdl ***register_timed_callback** (uint64_t *time_ps*)

GpiCbHdl ***register_readonly_callback** (void)

GpiCbHdl ***register_nexttime_callback** (void)

GpiCbHdl ***register_readwrite_callback** (void)

int **deregister_callback** (*GpiCbHdl* **obj_hdl*)

GpiObjHdl ***native_check_create** (std::string &*name*, *GpiObjHdl* **parent*)

GpiObjHdl ***native_check_create** (int32_t *index*, *GpiObjHdl* **parent*)

GpiObjHdl ***native_check_create** (void **raw_hdl*, *GpiObjHdl* **parent*)

const char ***reason_to_string** (int *reason*)

GpiObjHdl ***create_gpi_obj_from_handle** (vpiHandle *new_hdl*, std::string &*name*, std::string &*fq_name*)

Private Members

VpiReadwriteCbHdl **m_read_write**

VpiNextPhaseCbHdl **m_next_phase**

VpiReadOnlyCbHdl **m_read_only**

File cocotb_utils.c

Functions

void **to_python** (void)

void **to_simulator** (void)

void ***utils_dyn_open** (**const** char **lib_name*)

void ***utils_dyn_sym** (void **handle*, **const** char **sym_name*)

Variables

int **is_python_context** = 0

File entrypoint.vhdl

class cocotb_entrypoint

class cocotb_arch

Public Members

cocotb_arch:architecture is "cocotb_fli_init fli.so" cocotb_entrypoint.cocotb_ar

File gpi_embed.c

Initialisation

Called by the simulator on initialisation.

Load cocotb python module

GILState before calling: Not held

GILState after calling: Not held

Makes one call to PyGILState_Ensure and one call to PyGILState_Release

Loads the Python module called cocotb and calls the _initialise_testbench function

COCOTB_MODULE "cocotb"

int **get_module_ref** (**const** char **modname*, PyObject ***mod*)

int **embed_sim_init** (gpi_sim_info_t **info*)

void **embed_sim_event** (gpi_event_t *level*, **const** char **msg*)

Initialise the python interpreter

Create and initialise the python interpreter

GILState before calling: N/A

GILState after calling: released

Stores the thread state for cocotb in static variable gtstate

void **embed_init_python** (void)

Variables

PyThreadState ***gtstate** = NULL

char **progname**[] = "cocotb"

char ***argv**[] = {*progname*}

PyObject ***pEventFn** = NULL

File gpi_logging.c

GPI logging

Write a log message using cocotb SimLog class

GILState before calling: Unknown

GILState after calling: Unknown

Makes one call to PyGILState_Ensure and one call to PyGILState_Release

If the Python logging mechanism is not initialised, dumps to stderr.

void **gpi_log** (**const** char **name*, long *level*, **const** char **pathname*, **const** char **funcname*, long *lineno*, **const** char **msg*, ...)

Defines

LOG_SIZE 512

Functions

void **set_log_handler** (void **handler*)

void **set_log_filter** (void **filter*)

void **set_log_level** (**enum** gpi_log_levels *new_level*)

const char ***log_level** (long *level*)

Variables

PyObject ***pLogHandler**

PyObject ***pLogFilter**

gpi_log_levels **local_level** = GPIInfo

struct _log_level_table log_level_table[]= { { 10, "DEBUG" }, { 20,

char **log_buff**[**LOG_SIZE**]

struct _log_level_table

Public Members

long **level**

const char ***levelname**

File gpi_priv.h

Defines

const void NAME##_entry_point(void) \ { \ func(); \ } \ }]

Typedefs

typedef enum *gpi_cb_state* **gpi_cb_state_e**

typedef const void (***layer_entry_func**) (void)

Enums

enum gpi_cb_state

Values:

GPI_FREE = 0

GPI_PRIMED = 1

GPI_CALL = 2

GPI_REPRIME = 3

GPI_DELETE = 4

Functions

template<class **To**>
To **sim_to_hdl** (gpi_sim_hdl *input*)

int **gpi_register_impl** (*GpiImplInterface* **func_tbl*)

void **gpi_embed_init** (gpi_sim_info_t **info*)

void **gpi_embed_end** (void)

void **gpi_embed_event** (gpi_event_t *level*, **const** char **msg*)

void **gpi_load_extra_libs** (void)

class GpiHdl
Subclassed by *GpiCbHdl*, *GpiIterator*, *GpiObjHdl*

Public Functions

GpiHdl (*GpiImplInterface* **impl*)

GpiHdl (*GpiImplInterface* **impl*, void **hdl*)

virtual ~GpiHdl ()

int **initialise** (std::string &*name*)

template<typename **T**>
T **get_handle** (void) **const**

char ***gpi_copy_name** (**const** char **name*)

bool **is_this_impl** (*GpiImplInterface* **impl*)

Public Members

GpiImplInterface ***m_impl**

Protected Attributes

void ***m_obj_hdl**

Private Functions

GpiHdl ()

class GpiObjHdl : **public** *GpiHdl*
Subclassed by *FliObjHdl*, *GpiSignalObjHdl*, *VhpiArrayObjHdl*, *VhpiObjHdl*, *VpiArrayObjHdl*, *VpiObjHdl*

Public Functions

GpiObjHdl (*GpiImplInterface* **impl*)

GpiObjHdl (*GpiImplInterface* **impl*, void **hdl*, gpi_objtype_t *objtype*)

GpiObjHdl (*GpiImplInterface* **impl*, void **hdl*, gpi_objtype_t *objtype*, bool *is_const*)

virtual ~GpiObjHdl ()

const char ***get_name_str** (void)

const char ***get_fullname_str** (void)

const char ***get_type_str** (void)

gpi_objtype_t **get_type** (void)

bool **get_const** (void)

int **get_num_elems** (void)

int **get_range_left** (void)

int **get_range_right** (void)

int **get_indexable** (void)

const std::string &**get_name** (void)

const std::string &**get_fullname** (void)

virtual const char ***get_definition_name** ()

virtual const char ***get_definition_file** ()

bool **is_native_impl** (*GpiImplInterface* **impl*)

int **initialise** (std::string &*name*, std::string &*full_name*)

Protected Attributes

int **m_num_elems**

bool **m_indexable**

int **m_range_left**

int **m_range_right**

std::string **m_name**

std::string **m_fullname**

std::string **m_definition_name**

std::string **m_definition_file**

gpi_objtype_t **m_type**

bool **m_const**

class GpiSignalObjHdl : **public** *GpiObjHdl*

Subclassed by *FliSignalObjHdl*, *VhpiSignalObjHdl*, *VpiSignalObjHdl*

Public Functions

GpiSignalObjHdl (*GpiImplInterface* **impl*, void **hdl*, gpi_objtype_t *objtype*, bool *is_const*)

virtual ~GpiSignalObjHdl ()

virtual const char ***get_signal_value_binstr** (void) = 0

virtual const char ***get_signal_value_str** (void) = 0

virtual double **get_signal_value_real** (void) = 0

virtual long **get_signal_value_long** (void) = 0

virtual int **set_signal_value** (**const** long *value*) = 0

virtual int **set_signal_value** (**const** double *value*) = 0

virtual int **set_signal_value** (std::string &*value*) = 0

virtual *GpiCbHdl* ***value_change_cb** (unsigned int *edge*) = 0

Public Members

int **m_length**

class GpiCbHdl : **public** *GpiHdl*

Subclassed by *FliProcessCbHdl*, *GpiValueCbHdl*, *VhpiCbHdl*, *VpiCbHdl*

Public Functions

GpiCbHdl (*GpiImplInterface* **impl*)

int **arm_callback** (void) = 0

int **run_callback** (void)

int **cleanup_callback** (void) = 0

int **set_user_data** (int (**gpi_function*)) **const** void *
, **const** void **data*

const void ***get_user_data** (void)

void **set_call_state** (*gpi_cb_state_e* *new_state*)

gpi_cb_state_e **get_call_state** (void)

~GpiCbHdl ()

Protected Attributes

int (***gpi_function**) (**const** void *)

const void ***m_cb_data**

gpi_cb_state_e **m_state**

class GpiValueCbHdl : **public virtual** *GpiCbHdl*
Subclassed by *FliSignalCbHdl*, *VhpiValueCbHdl*, *VpiValueCbHdl*

Public Functions

GpiValueCbHdl (*GpiImplInterface* **impl*, *GpiSignalObjHdl* **signal*, int *edge*)

virtual ~GpiValueCbHdl ()

int **run_callback** (void)

virtual int **cleanup_callback** (void) = 0

Protected Attributes

std::string **required_value**

GpiSignalObjHdl ***m_signal**

class GpiClockHdl

Public Functions

GpiClockHdl (*GpiObjHdl* **clk*)

GpiClockHdl (**const** char **clk*)

~GpiClockHdl ()

int **start_clock** (**const** int *period_ps*)

int **stop_clock** (void)

class GpiIterator : **public** *GpiHdl*
Subclassed by *FliIterator*, *VhpiIterator*, *VpiIterator*, *VpiSingleIterator*

Public Types

enum Status
Values:

NATIVE

NATIVE_NO_NAME

NOT_NATIVE

NOT_NATIVE_NO_NAME

END

Public Functions

GpiIterator(*GpiImplInterface* **impl*, *GpiObjHdl* **hdl*)

virtual **~GpiIterator**()

virtual *Status* **next_handle**(std::string &*name*, *GpiObjHdl* ***hdl*, void ***raw_hdl*)

GpiObjHdl ***get_parent**(void)

Protected Attributes

GpiObjHdl ***m_parent**

template<class **Ti**, class **Tm**>
class GpiIteratorMapping

Public Functions

GpiIteratorMapping(void (**populate*)) *GpiIteratorMapping*<Ti, Tm>&

std::vector<Tm> ***get_options**(Ti *type*)

void **add_to_options**(Ti *type*, Tm **options*)

Private Members

std::map<Ti, std::vector<Tm>> **options_map**

class GpiImplInterface
Subclassed by *FliImpl*, *VhpiImpl*, *VpiImpl*

Public Functions

GpiImplInterface(**const** std::string &*name*)

const char ***get_name_c**(void)

const string &**get_name_s**(void)

virtual ~GpiImplInterface()

virtual void **sim_end**(void) = 0

virtual void **get_sim_time**(uint32_t **high*, uint32_t **low*) = 0

virtual void **get_sim_precision**(int32_t **precision*) = 0

virtual *GpiObjHdl* ***native_check_create**(std::string &*name*, *GpiObjHdl* **parent*) = 0

virtual *GpiObjHdl* ***native_check_create**(int32_t *index*, *GpiObjHdl* **parent*) = 0

virtual *GpiObjHdl* ***native_check_create**(void **raw_hdl*, *GpiObjHdl* **parent*) = 0

virtual *GpiObjHdl* ***get_root_handle**(**const** char **name*) = 0 virtual *GpiIterator* ***iterate_handle**(*GpiObjHdl* **obj_hdl*, gpi_iterator_sel_t *type*) = 0

virtual *GpiCbHdl* ***register_timed_callback**(uint64_t *time_ps*) = 0

virtual *GpiCbHdl* ***register_readonly_callback**(void) = 0

virtual *GpiCbHdl* ***register_nexttime_callback**(void) = 0

virtual *GpiCbHdl* ***register_readwrite_callback**(void) = 0

virtual int **deregister_callback**(*GpiCbHdl* **obj_hdl*) = 0

virtual const char ***reason_to_string**(int *reason*) = 0

Private Members

std::string **m_name**

File python3_compat.h

Defines

GETSTATE(m) (&_state)

MODULE_ENTRY_POINT initsimulator

INITERROR return

struct module_state

Public Members

PyObject ***error**

File simulatormodule.c

Python extension to provide access to the simulator.

Uses GPI calls to interface to the simulator.

Callback Handling

Handle a callback coming from GPI

GILState before calling: Unknown

GILState after calling: Unknown

Makes one call to TAKE_GIL and one call to DROP_GIL

Returns 0 on success or 1 on a failure.

Handles a callback from the simulator, all of which call this function.

We extract the associated context and find the Python function (usually cocotb.scheduler.react) calling it with a reference to the trigger that fired. The scheduler can then call next() on all the coroutines that are waiting on that particular trigger.

TODO:

- Tidy up return values
- Ensure cleanup correctly in exception cases

int **handle_gpi_callback** (void **user_data*)

static PyObject ***log_msg** (PyObject **self*, PyObject **args*)

static PyObject ***register_readonly_callback** (PyObject **self*, PyObject **args*)

static PyObject ***register_rwsynch_callback** (PyObject **self*, PyObject **args*)

static PyObject ***register_nextstep_callback** (PyObject **self*, PyObject **args*)

static PyObject ***register_timed_callback** (PyObject **self*, PyObject **args*)

static PyObject ***register_value_change_callback** (PyObject **self*, PyObject **args*)

static PyObject ***iterate** (PyObject **self*, PyObject **args*)

static PyObject ***next** (PyObject **self*, PyObject **args*)

static PyObject ***get_signal_val_binstr** (PyObject **self*, PyObject **args*)

static PyObject ***get_signal_val_str** (PyObject **self*, PyObject **args*)

static PyObject ***get_signal_val_real** (PyObject **self*, PyObject **args*)

static PyObject ***get_signal_val_long** (PyObject **self*, PyObject **args*)

static PyObject ***set_signal_val_str** (PyObject **self*, PyObject **args*)

static PyObject ***set_signal_val_real** (PyObject **self*, PyObject **args*)

static PyObject ***set_signal_val_long** (PyObject **self*, PyObject **args*)

static PyObject ***get_definition_name** (PyObject **self*, PyObject **args*)

static PyObject ***get_definition_file** (PyObject **self*, PyObject **args*)

static PyObject ***get_handle_by_name** (PyObject **self*, PyObject **args*)

static PyObject ***get_handle_by_index** (PyObject **self*, PyObject **args*)

static PyObject ***get_root_handle** (PyObject **self*, PyObject **args*)

static PyObject ***get_name_string** (PyObject **self*, PyObject **args*)

static PyObject ***get_type** (PyObject **self*, PyObject **args*)

static PyObject ***get_const** (PyObject **self*, PyObject **args*)

static PyObject ***get_type_string** (PyObject **self*, PyObject **args*)

static PyObject ***get_sim_time** (PyObject **self*, PyObject **args*)

static PyObject ***get_precision** (PyObject **self*, PyObject **args*)

static PyObject ***get_num_elems** (PyObject **self*, PyObject **args*)

static PyObject ***get_range** (PyObject **self*, PyObject **args*)

static PyObject ***stop_simulator** (PyObject **self*, PyObject **args*)

static PyObject ***deregister_callback** (PyObject **self*, PyObject **args*)

static PyObject ***log_level** (PyObject **self*, PyObject **args*)

static void **add_module_constants** (PyObject **simulator*)

Typedefs

typedef int (***gpi_function_t**) (**const** void *)

Functions

PyGILState_STATE **TAKE_GIL** (void)

void **DROP_GIL** (PyGILState_STATE *state*)

static int **gpi_sim_hdl_converter** (PyObject **o*, gpi_sim_hdl **data*)

static int **gpi_iterator_hdl_converter** (PyObject **o*, gpi_iterator_hdl **data*)

Variables

int **takes** = 0

int **releases** = 0

struct *sim_time* **cache_time**

struct sim_time

Public Members

uint32_t **high**

uint32_t **low**

File simulatormodule.h

Defines

COCOTB_ACTIVE_ID 0xC0C07B

COCOTB_INACTIVE_ID 0xDEADB175

MODULE_NAME "simulator"

Typedefs

typedef struct *t_callback_data* **s_callback_data**

typedef struct *t_callback_data* ***p_callback_data**

Functions

static PyObject ***error_out** (PyObject **m*)

static PyObject ***log_msg** (PyObject **self*, PyObject **args*)

static PyObject ***get_signal_val_long** (PyObject **self*, PyObject **args*)

static PyObject ***get_signal_val_real** (PyObject **self*, PyObject **args*)

static PyObject ***get_signal_val_str** (PyObject **self*, PyObject **args*)

static PyObject ***get_signal_val_binstr** (PyObject **self*, PyObject **args*)

static PyObject ***set_signal_val_long** (PyObject **self*, PyObject **args*)

static PyObject ***set_signal_val_real** (PyObject **self*, PyObject **args*)

static PyObject ***set_signal_val_str** (PyObject **self*, PyObject **args*)

static PyObject ***get_definition_name** (PyObject **self*, PyObject **args*)

static PyObject ***get_definition_file** (PyObject **self*, PyObject **args*)

static PyObject ***get_handle_by_name** (PyObject **self*, PyObject **args*)

static PyObject ***get_handle_by_index** (PyObject **self*, PyObject **args*)

static PyObject ***get_root_handle** (PyObject **self*, PyObject **args*)

static PyObject ***get_name_string** (PyObject **self*, PyObject **args*)

static PyObject ***get_type** (PyObject **self*, PyObject **args*)

static PyObject ***get_const** (PyObject **self*, PyObject **args*)

static PyObject ***get_type_string** (PyObject **self*, PyObject **args*)

static PyObject ***get_num_elems** (PyObject **self*, PyObject **args*)

static PyObject ***get_range** (PyObject **self*, PyObject **args*)

static PyObject ***register_timed_callback** (PyObject **self*, PyObject **args*)

static PyObject ***register_value_change_callback** (PyObject **self*, PyObject **args*)

static PyObject ***register_readonly_callback** (PyObject **self*, PyObject **args*)

static PyObject ***register_nextstep_callback** (PyObject **self*, PyObject **args*)

static PyObject ***register_rwsynch_callback** (PyObject **self*, PyObject **args*)

static PyObject ***stop_simulator** (PyObject **self*, PyObject **args*)

static PyObject ***iterate** (PyObject **self*, PyObject **args*)

static PyObject ***next** (PyObject **self*, PyObject **args*)

static PyObject ***get_sim_time** (PyObject **self*, PyObject **args*)

static PyObject ***get_precision** (PyObject **self*, PyObject **args*)

static PyObject ***deregister_callback** (PyObject **self*, PyObject **args*)

static PyObject ***log_level** (PyObject **self*, PyObject **args*)

Variables

PyMethodDef **SimulatorMethods**[]

struct t_callback_data

Public Members

PyThreadState ***_saved_thread_state**

uint32_t **id_value**

PyObject ***function**

PyObject ***args**

PyObject ***kwargs**

gpi_sim_hdl **cb_hdl**

File simulatormodule_python2.c

Functions

static PyObject ***error_out** (PyObject **m*)

PyMODINIT_FUNC **MODULE_ENTRY_POINT** (void)

Variables

char error_module[] = MODULE_NAME ".Error"

struct *module_state* **_state**

File simulatormodule_python3.c

Functions

static PyObject ***error_out** (PyObject **m*)

static int **simulator_traverse** (PyObject **m*, visitproc *visit*, void **arg*)

static int **simulator_clear** (PyObject **m*)

PyMODINIT_FUNC **MODULE_ENTRY_POINT** (void)

Variables

struct PyModuleDef **moduledef** = {PyModuleDef_HEAD_INIT, , , , , , , , }

8.1.3 Struct list

Struct _log_level_table

`struct _log_level_table`

Struct module_state

`struct module_state`

Struct sim_time

`struct sim_time`

Struct t_callback_data

`struct t_callback_data`

CHAPTER

NINE

TUTORIAL: ENDIAN SWAPPER

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

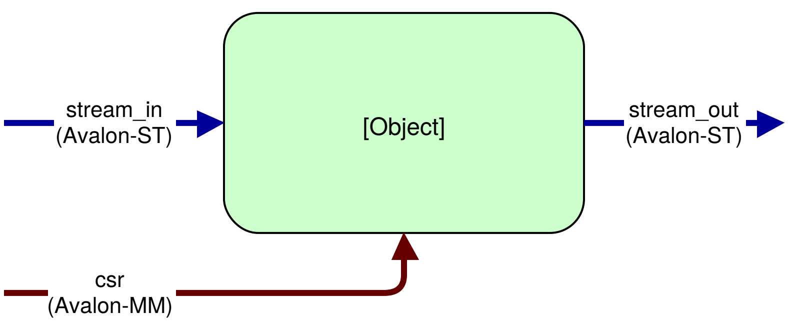
Note: All the code and sample output from this example are available on EDA Playground

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
```

9.1 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.

9.2 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.

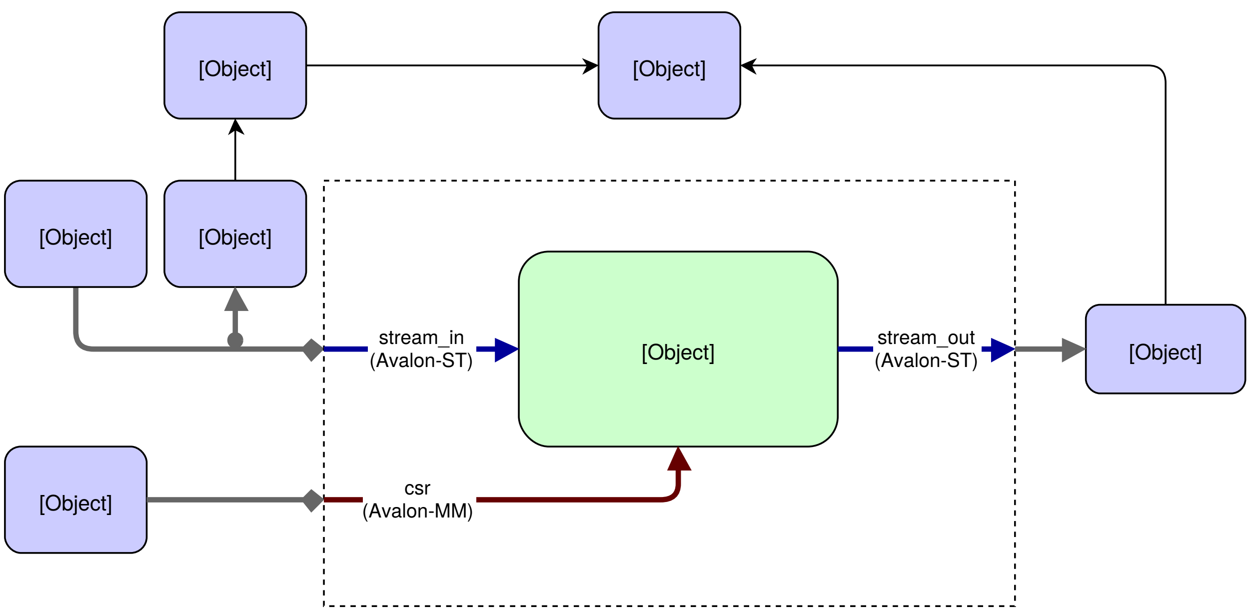
```
class EndianSwapperTB(object):

    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 are 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	startofpacket	Asserted by the source to mark the beginning of a packet
stream_in_endofpacket	endofpacket	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
```

9.2.1 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 range(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.

9.2.2 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 the options provided for each argument. Note that we utilise some of the built-in generators to toggle backpressure and insert idle cycles.

CHAPTER

TEN

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
```

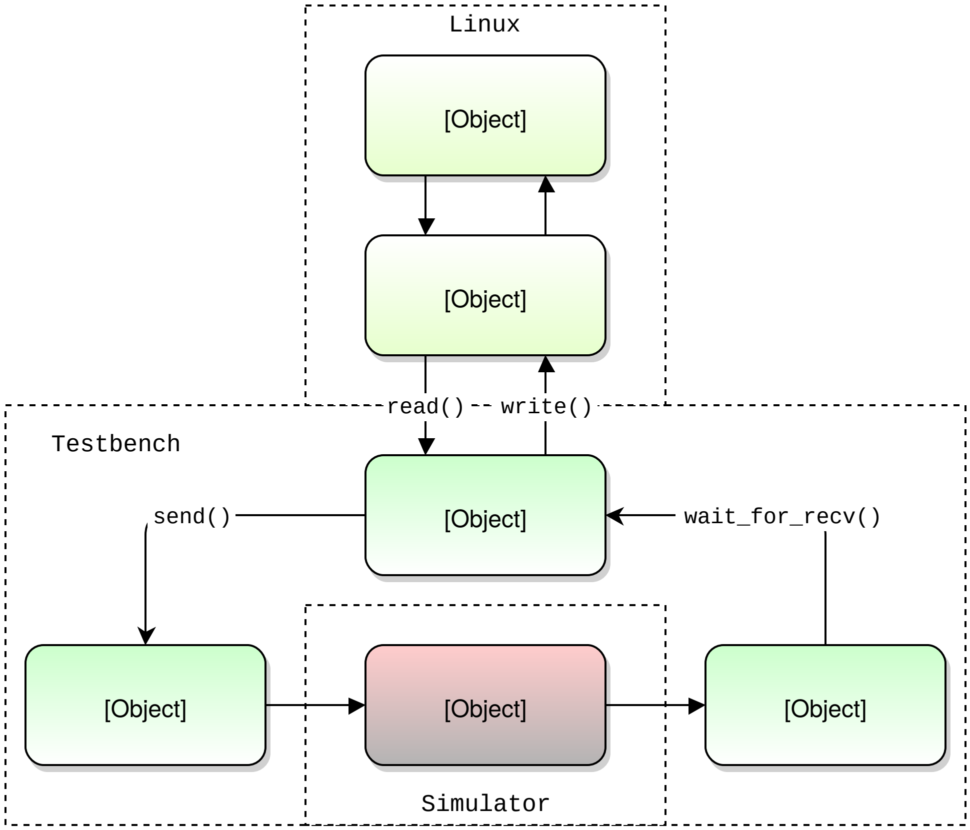
Note: To create a virtual interface the test either needs root permissions or have `CAP_NET_ADMIN` capability.

10.1 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.



10.2 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 range(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!

10.3 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

ELEVEN

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

11.1 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.

11.1.1 Calling the HAL from a test

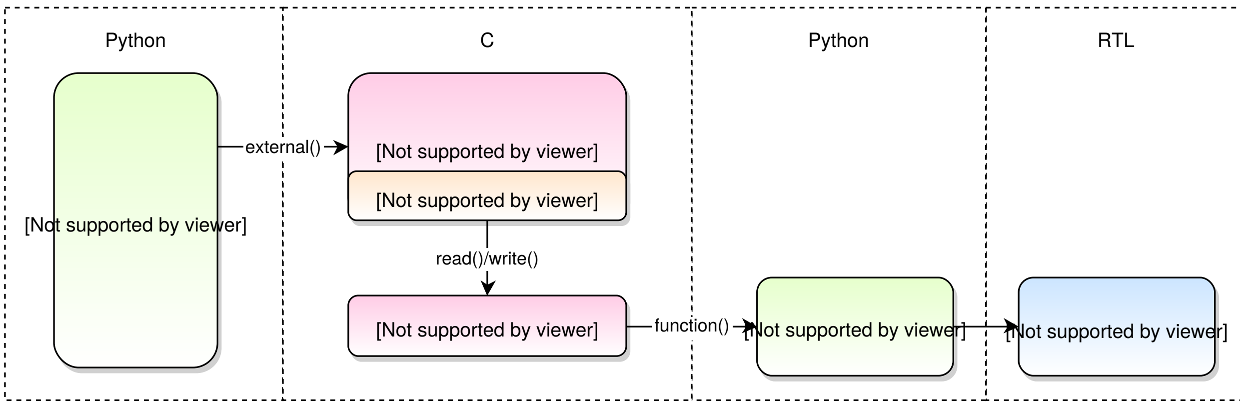
Typically the software component (often referred to as a Hardware Abstraction Layer or HAL) is written in C. We need to call this software from our test written in Python. There are multiple ways to call C code from Python, in this tutorial we'll use SWIG to generate Python bindings for our HAL.

11.1.2 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.

11.2 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:



11.3 Implementation

11.3.1 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

11.3.2 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.

11.3.3 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.

11.3.4 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 initialise 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.

11.4 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

TWELVE

MORE EXAMPLES

Apart from the examples covered with full tutorials in the previous sections, the directory `cocotb/examples/` contains some more smaller modules you may want to take a look at.

12.1 Adder

The directory `cocotb/examples/adder/` contains an `adder` RTL in both Verilog and VHDL, an `adder_model` implemented in Python, and the cocotb testbench with two defined tests a simple `adder_basic_test()` and a slightly more advanced `adder_randomised_test()`.

This example does not use any *`Driver`*, *`Monitor`*, or *`Scoreboard`*; not even a clock.

12.2 D Flip-Flop

The directory `cocotb/examples/dff/` contains a simple D flip-flop, implemented in both VDHL and Verilog.

The HDL has the data input port `d`, the clock port `c`, and the data output `q` with an initial state of `0`. No reset port exists.

The cocotb testbench checks the initial state first, then applies random data to the data input. The flip-flop output is captured at each rising edge of the clock and compared to the applied input data using a *`Scoreboard`*.

The testbench defines a `BitMonitor` (a subclass of *`Monitor`*) as a pendant to the cocotb-provided *`BitDriver`*. The *`BitDriver`*'s *`start()`* and *`stop()`* methods are used to start and stop generation of input data.

A *`TestFactory`* is used to generate the random tests.

12.3 Mean

The directory `cocotb/examples/mean/` contains a module that calculates the mean value of a data input bus `i` (with signals `i_data` and `i_valid`) and outputs it on `o` (with `i_data` and `o_valid`).

It has implementations in both VHDL and SystemVerilog.

The testbench defines a `StreamBusMonitor` (a subclass of *`BusMonitor`*), a clock generator, a `value_test` helper coroutine and a few tests. Test `mean_randomised_test` uses the `StreamBusMonitor` to feed a *`Scoreboard`* with the collected transactions on input bus `i`.

12.4 Mixed Language

The directory `cocotb/examples/mixed_language/` contains two toplevel HDL files, one in VHDL, one in SystemVerilog, that each instantiate the `endian_swapper` in SystemVerilog and VHDL in parallel and chains them together so that the endianness is swapped twice.

Thus, we end up with SystemVerilog+VHDL instantiated in VHDL and SystemVerilog+VHDL instantiated in SystemVerilog.

The cocotb testbench pulls the reset on both instances and checks that they behave the same.

Todo: This example is not complete.

12.5 AXI Lite Slave

The directory `cocotb/examples/axi_lite_slave/` contains . . .

Todo: Write documentation, see README.md

12.6 Sorter

Example testbench for snippet of code from comp.lang.verilog:

```
@cocotb.coroutine
def run_test(dut, data_generator=random_data, delay_cycles=2):
    """Send data through the DUT and check it is sorted output."""
    cocotb.fork(Clock(dut.clk, 100).start())

    # Don't check until valid output
    expected = [None] * delay_cycles

    for index, values in enumerate(data_generator(bits=len(dut.in1))):
        expected.append(sorted(values))

        yield RisingEdge(dut.clk)
        dut.in1 = values[0]
        dut.in2 = values[1]
        dut.in3 = values[2]
        dut.in4 = values[3]
        dut.in5 = values[4]

        yield ReadOnly()
        expect = expected.pop(0)

        if expect is None:
            continue

        got = [int(dut.out5), int(dut.out4), int(dut.out3),
               int(dut.out2), int(dut.out1)]
```

(continues on next page)

(continued from previous page)

```
        if got != expect:
            dut._log.error('Expected %s' % expect)
            dut._log.error('Got %s' % got)
            raise TestFailure("Output didn't match")

    dut._log.info('Sucessfully sent %d cycles of data' % (index + 1))
```

CHAPTER

THIRTEEN

TROUBLESHOOTING

13.1 Simulation Hangs

Did you call a function that is marked as a coroutine directly, i.e. without using `yield`?

13.2 Increasing Verbosity

If things fail in the VPI/VHPI/FLI area, check your simulator's documentation to see if it has options to increase its verbosity about what may be wrong. You can then set these options on the `make` command line as `COMPILE_ARGS`, `SIM_ARGS` or `EXTRA_OPTS` (see *Build options and Environment Variables* for details).

13.3 Attaching a Debugger

In order to give yourself time to attach a debugger to the simulator process before it starts to run, you can set the environment variable `COCOTB_ATTACH` to a pause time value in seconds. If set, cocotb will print the process ID (PID) to attach to and wait the specified time before actually letting the simulator run.

For the GNU debugger GDB, the command is `attach <process-id>`.

CHAPTER

FOURTEEN

SIMULATOR SUPPORT

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

14.1 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`.

14.2 Synopsys VCS

14.3 Aldec Riviera-PRO

The `$LICENSE_QUEUE` environment variable can be used for this simulator – this setting will be mirrored in the TCL `license_queue` variable to control runtime license checkouts.

14.4 Mentor Questa

14.5 Mentor Modelsim

Any ModelSim PE or ModelSim PE derivative (like ModelSim Microsemi, Intel, Lattice Edition) does not support the VHDL FLI feature. If you try to run with FLI enabled, you will see a `vsim-FLI-3155` error:

```
** Error (suppressible): (vsim-FLI-3155) The FLI is not enabled in this version of␣
↪ModelSim.
```

ModelSim DE and SE (and Questa, of course) supports the FLI.

14.6 Cadence Incisive, Cadence Xcelium

14.7 GHDL

Support is preliminary. Noteworthy is that despite GHDL being a VHDL simulator, it implements the VPI interface.

CHAPTER

FIFTEEN

ROADMAP

cocotb is in active development.

We use GitHub issues to track our pending tasks. Take a look at the open Feature List 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

SIXTEEN

RELEASE NOTES

All releases are available from the GitHub Releases Page.

16.1 cocotb 1.2

Released on 24 July 2019

16.1.1 New features

- cocotb is now built as Python package and installable through pip. (#517, #799, #800, #803, #805)
- Support for async functions and generators was added (Python 3 only). Please have a look at *Async functions* for an example how to use this new feature.
- VHDL block statements can be traversed. (#850)
- Support for Python 3.7 was added.

16.1.2 Notable changes and bug fixes

- The heart of cocotb, its scheduler, is now even more robust. Many small bugs, inconsistencies and unreliable behavior have been ironed out.
- Exceptions are now correctly propagated between coroutines, giving users the "natural" behavior they'd expect with exceptions. (#633)
- The `setimmediatevalue()` function now works for values larger than 32 bit. (#768)
- The documentation was cleaned up, improved and extended in various places, making it more consistent and complete.
- Tab completion in newer versions of IPython is fixed. (#825)
- Python 2.6 is officially not supported any more. cocotb supports Python 2.7 and Python 3.5+.
- The cocotb GitHub project moved from `potentialventures/cocotb` to `cocotb/cocotb`. Redirects for old URLs are in place.

16.1.3 Known issues

- Depending on your simulation, cocotb 1.2 might be roughly 20 percent slower than cocotb 1.1. Much of the work in this release cycle went into fixing correctness bugs in the scheduler, sometimes at the cost of performance. We are continuing to investigate this in issue #961. Independent of the cocotb version, we recommend using the latest Python 3 version, which is shown to be significantly faster than previous Python 3 versions, and slightly faster than Python 2.7.

Please have a look at the issue tracker for more outstanding issues and contribution opportunities.

16.2 cocotb 1.1

Released on 24 Jan 2019.

This release is the result of four years of work with too many bug fixes, improvements and refactorings to name them all. Please have a look at the release announcement on the mailing list for further information.

16.3 cocotb 1.0

Released on 15 Feb 2015.

16.3.1 New features

- FLI support for Modelsim
- Mixed Language, Verilog and VHDL
- Windows
- 300% performance improvement with VHPI interface
- Wavedrom support for wave diagrams.

16.4 cocotb 0.4

Released on 25 Feb 2014.

16.4.1 New features

- Issue #101: Implement Lock primitive to support mutex
- Issue #105: Compatibility with Aldec Riviera-Pro
- Issue #109: Combine multiple results.xml into a single results file
- Issue #111: XGMII drivers and monitors added
- Issue #113: Add operators to BinaryValue class
- Issue #116: Native VHDL support by implementing VHPI layer
- Issue #117: Added AXI4-Lite Master BFM

16.4.2 Bugs fixed

- Issue #100: Functional bug in endian_swapper example RTL
- Issue #102: Only 1 coroutine wakes up of multiple coroutines wait() on an Event
- Issue #114: Fix build issues with Cadence IUS simulator

16.4.3 New examples

- Issue #106: TUN/TAP example using ping

16.5 cocotb 0.3

Released on 27 Sep 2013.

This contains a raft of fixes and feature enhancements.

16.6 cocotb 0.2

Released on 19 Jul 2013.

16.6.1 New features

- Release 0.2 supports more simulators and increases robustness over 0.1.
- A centralised installation is now supported (see documentation) with supporting libraries build when the simulation is run for the first time.

16.7 cocotb 0.1

Released on 9 Jul 2013.

- The first release of cocotb.
- Allows installation and running against Icarus, VCS, Aldec simulators.

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

G

R

S

T

U

V

W

X