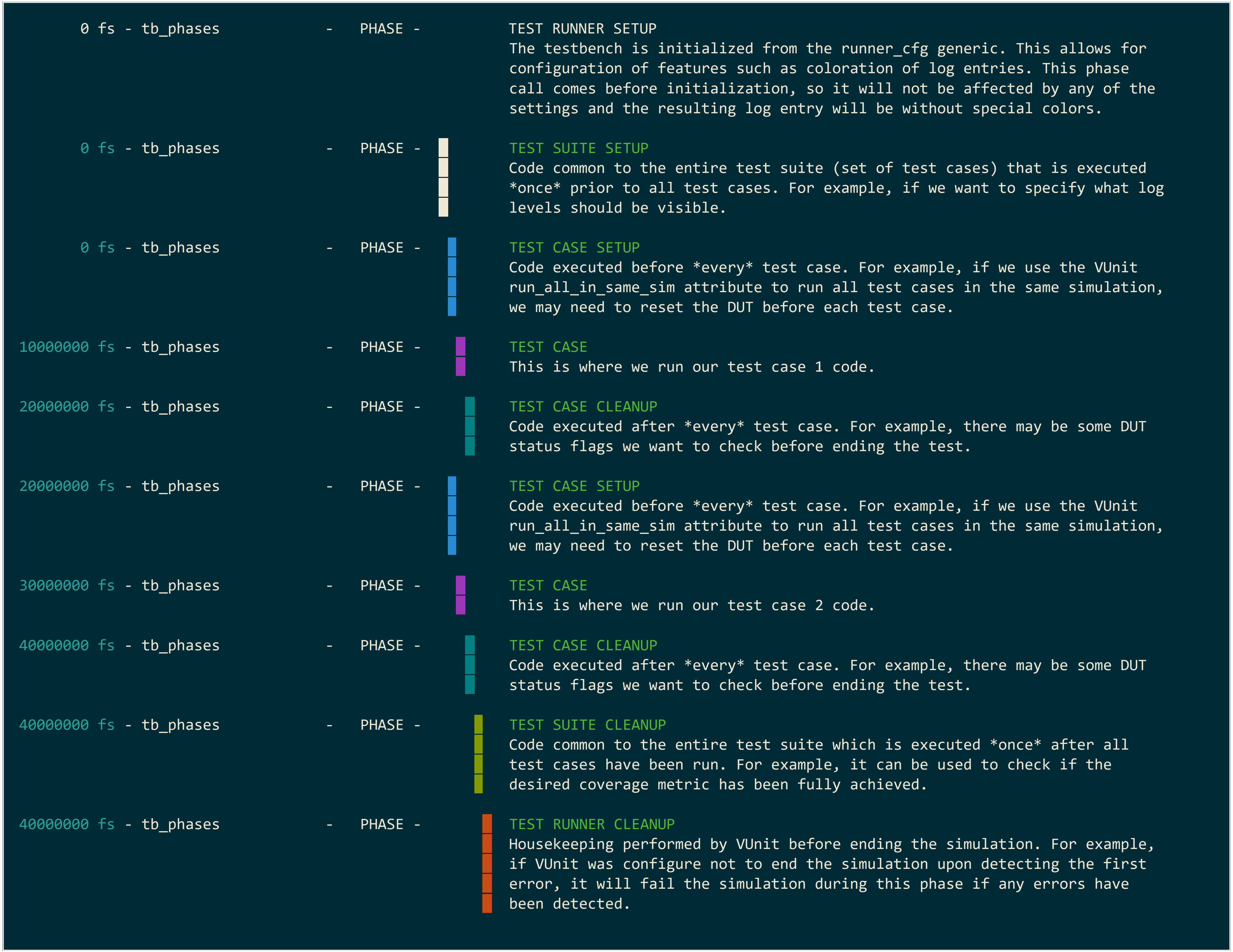
What You Will Learn

1. The different phases a VUnit testbench traverses during its execution.
2. How phases can be used by a process, such as a third part verification component (VC), to prevent a premature simulation exit without requiring non-intuitive coding constructs on the user side.
3. How testbench processes can be made aware of an imminent simulation exit, allowing them to carry out exit tasks such as verifying the correctness of the final DUT state or logging interesting summary information.

Introduction

During the course of a simulation, VUnit guides the testbench through a number of distinct *phases*. These phases are determined by the structure of the test runner process - the main process controlling the execution of the testbench. Some phases are completely encapsulated within a VUnit procedure, such as `test_runner_setup` and `test_runner_cleanup`, while others are defined as a code region within the test runner process. The code example below outlines a test runner process consisting of two test cases. The phases involved are named and described by the `phase` procedure calls. This procedure, created specifically for this example, provides a visual representation of the different phases and their transitions as a VUnit log.

```vhdl
test_runner : process
begin
  phase("TEST RUNNER SETUP",
    "The testbench is initialized from the runner_cfg generic. This allows for " &
    "configuration of features such as coloration of log entries. This phase " &
    "call comes before initialization, so it will not be affected by any of the " &
    "settings and the resulting log entry will be without special colors."
  );
  test_runner_setup(runner, runner_cfg);

  phase("TEST SUITE SETUP",
    "Code common to the entire test suite (set of test cases) that is executed *once* " &
    "prior to all test cases. For example, if we want to specify what log levels should " &
    "be visible."
  );
  show(display_handler, debug);

  while test_suite loop
    phase("TEST CASE SETUP",
      "Code executed before *every* test case. For example, if we use the VUnit " &
      "run_all_in_same_sim attribute to run all test cases in the same simulation, we " &
      "may need to reset the DUT before each test case."
    );
    -- vunit: run_all_in_same_sim
    reset <= '1';
    wait for 10 ns;
    reset <= '0';

    if run("Test case 1") then
      phase("TEST CASE",
        "This is where we run our test case 1 code."
      );
      wait for 10 ns; -- The test code is just a wait statement in this dummy example

    elsif run("Test case 2") then
      phase("TEST CASE",
        "This is where we run our test case 2 code."
      );
      wait for 10 ns; -- The test code is just a wait statement in this dummy example

    end if;

    phase("TEST CASE CLEANUP",
      "Code executed after *every* test case. For example, there may be some DUT status " &
      "flags we want to check before ending the test."
    );
    check_equal(error_flag, '0');

  end loop;

  phase("TEST SUITE CLEANUP",
    "Code common to the entire test suite which is executed *once* after all test " &
    "cases have been run. For example, it can be used to check if the desired coverage " &
    "metric has been fully achieved."
  );
  check_true(full_coverage);

  phase("TEST RUNNER CLEANUP",
    "Housekeeping performed by VUnit before ending the simulation. For example, " &
    "if VUnit was configure not to end the simulation upon detecting the first error, " &
    "it will fail the simulation during this phase if any errors have been detected."
  );
  test_runner_cleanup(runner);
end process;
```



```
       0 fs - tb_phases           -  PHASE -        TEST RUNNER SETUP
                                                    The testbench is initialized from the runner_cfg generic. This allows for
                                                    configuration of features such as coloration of log entries. This phase
                                                    call comes before initialization, so it will not be affected by any of the
                                                    settings and the resulting log entry will be without special colors.

       0 fs - tb_phases           -  PHASE -        TEST SUITE SETUP
                                                    Code common to the entire test suite (set of test cases) that is executed
                                                    *once* prior to all test cases. For example, if we want to specify what log
                                                    levels should be visible.

       0 fs - tb_phases           -  PHASE -        TEST CASE SETUP
                                                    Code executed before *every* test case. For example, if we use the VUnit
                                                    run_all_in_same_sim attribute to run all test cases in the same simulation,
                                                    we may need to reset the DUT before each test case.

10000000 fs - tb_phases           -  PHASE -        TEST CASE
                                                    This is where we run our test case 1 code.

20000000 fs - tb_phases           -  PHASE -        TEST CASE CLEANUP
                                                    Code executed after *every* test case. For example, there may be some DUT
                                                    status flags we want to check before ending the test.

20000000 fs - tb_phases           -  PHASE -        TEST CASE SETUP
                                                    Code executed before *every* test case. For example, if we use the VUnit
                                                    run_all_in_same_sim attribute to run all test cases in the same simulation,
                                                    we may need to reset the DUT before each test case.

30000000 fs - tb_phases           -  PHASE -        TEST CASE
                                                    This is where we run our test case 2 code.

40000000 fs - tb_phases           -  PHASE -        TEST CASE CLEANUP
                                                    Code executed after *every* test case. For example, there may be some DUT
                                                    status flags we want to check before ending the test.

40000000 fs - tb_phases           -  PHASE -        TEST SUITE CLEANUP
                                                    Code common to the entire test suite which is executed *once* after all
                                                    test cases have been run. For example, it can be used to check if the
                                                    desired coverage metric has been fully achieved.

40000000 fs - tb_phases           -  PHASE -        TEST RUNNER CLEANUP
                                                    Housekeeping performed by VUnit before ending the simulation. For example,
                                                    if VUnit was configure not to end the simulation upon detecting the first
                                                    error, it will fail the simulation during this phase if any errors have
                                                    been detected.
```

To infer as little overhead as possible (but not less than that), VUnit permits testbenches without named test cases, resulting in the elimination of certain phases. However, `test_runner_setup` and `test_runner_cleanup` remain present at all times.

```
test_runner : process
begin
  phase("TEST RUNNER SETUP",
    "The testbench is initialized from the runner_cfg generic. This allows for " &
    "configuration of features such as coloration of log entries. This phase " &
    "call comes before initialization, so it will not be affected by any of the " &
    "settings and the resulting log entry will be without special colors."
  );
  test_runner_setup(runner, runner_cfg);

  phase("TEST CASE",
    "This is where we run all the test code."
  );
  reset <= '1';
  wait for 10 ns;
  reset <= '0';
  wait for 10 ns; -- The test code is just a wait statement in this dummy example
  check_equal(error_flag, '0');
  check_true(full_coverage);

  phase("TEST RUNNER CLEANUP",
    "Housekeeping performed by VUnit before ending the simulation. For example, " &
    "if VUnit was configure not to end the simulation upon detecting the first error, " &
    "it will fail the simulation during this phase if any errors have been detected."
  );
  test_runner_cleanup(runner);
end process;
```

Of these phases, the test runner cleanup phase is the most useful and the focal point of this blog. To understand how we can use this phase, we can start by showing the usually hidden trace messages from the `runner` logger when the `test_runner_cleanup` function is invoked:

```
        0 fs - tb_phases             -   PHASE -          TEST RUNNER SETUP
                                                          The testbench is initialized from the runner_cfg generic. This allows for
                                                          configuration of features such as coloration of log entries. This phase
                                                          call comes before initialization, so it will not be affected by any of the
                                                          settings and the resulting log entry will be without special colors.

        0 fs - tb_phases             -   PHASE -          TEST CASE
                                                          This is where we run all the test code.

 20000000 fs - tb_phases             -   PHASE -          TEST RUNNER CLEANUP
                                                          Housekeeping performed by VUnit before ending the simulation. For example,
                                                          if VUnit was configure not to end the simulation upon detecting the first
                                                          error, it will fail the simulation during this phase if any errors have
                                                          been detected.

 20000000 fs - runner                -   TRACE - Entering test runner cleanup phase.
 20000000 fs - runner                -   TRACE - Passed test runner cleanup phase entry gate.
 20000000 fs - runner                -   TRACE - Passed test runner cleanup phase exit gate.
 20000000 fs - runner                -   TRACE - Entering test runner exit phase.
```

As we can see VUnit keeps track of the phases internally and there is a concept of a *gate* when entering and exiting a phase. In this example, VUnit passes through the gates but it's possible for a user to prevent that by locking a gate. This capability can be used to prevent VUnit from cleaning up and exiting the simulation until all processes are done, resolving the issues that arose with using an event as a barrier in our previous blog:

1. The phase lock solution scales well. `test_runner_cleanup` is already present in every VUnit testbench and it eliminates the need for creating a new event for every process to wait on.
2. There is no race condition as it doesn't matter in which order processes complete. `test_runner_cleanup` will wait until every process has removed its lock.
3. With the absence of any additional constructs in the test runner, there is no risk of forgetting to wait for another process, and any process can introduce locks without requiring external changes. This is especially important for verification components, as they can start using locks without the necessity of their users having to update their code.

Phase Gate Locks

To see the phase gate locks in action we're going to revisit the example provided in the event blog. In that example our `dut_checker` process notified its completion with the `dut_checker_done` event.

```
dut_checker : process
begin
  if is_empty(queue) then
    wait until is_active(new_data_set);
  end if;

  for i in 1 to pop(queue) loop
    wait until (rising_edge(clk) and output_tvalid = '1') or log_active(vunit_error, decorate("while waiting on output data"), logger =>
dut_checker_logger);
    check_equal(output_tdata, calculate_expected_output(pop(queue)));
  end loop;

  if is_empty(queue) then
    notify(dut_checker_done);
  end if;
end process;
```

The test runner process waited for that event before calling `test_runner_cleanup`.

```
wait until is_active_msg(dut_checker_done);
test_runner_cleanup(runner);
```

Before adding the locks, we need to remove the `dut_checker_done` event, the notification of that event in the `dut_checker`, and the wait statement waiting for that event in `test_runner`. Afterwards, we can update the `dut_checker` according to the code listing provided below.

```
dut_checker : process
  constant key : key_t := get_entry_key(test_runner_cleanup);
begin
  if is_empty(queue) then
    wait until is_active(new_data_set);
  end if;
  lock(runner, key, dut_checker_logger);

  for i in 1 to pop(queue) loop
    wait until (rising_edge(clk) and output_tvalid = '1') or log_active(vunit_error, decorate("while waiting on output data"), logger =>
dut_checker_logger);
    check_equal(output_tdata, calculate_expected_output(pop(queue)));
  end loop;

  if is_empty(queue) then
    unlock(runner, key, dut_checker_logger);
  end if;
end process;
```

The first step is to acquire a unique key for the gate we want to control, in this case the entry gate for the `test_runner_cleanup` phase. This is done by calling the `get_entry_key` function with `test_runner_cleanup` as the parameter. Each gate has many locks and the returned key fits one of those locks. An alternative design for locking a gate would be to use a keyless system, wherein locks can be added and removed to/from gates. However, such a design is prone to a class of bugs where a process unlocks a gate more than it locks it. This will lead to locks previously added by other processes being removed and the protection against premature termination of the simulation is lost.

The second step is to determine when to lock the gate. Generally, this is done when the process has a task that requires completion before the end of the simulation. In this case, this is when the queue is not empty. Locking is done by passing the `runner` signal, the key, and optionally a logger, to the `lock` procedure. The `runner` signal in VUnit contains several events, of which `runner_phase` is used to indicate that something occurred related to the VUnit phases.

The third step is to decide when to unlock. Unlocking should be done when the process has completed a task, provided there are no more tasks left to complete. In this case we unlock if the queue is empty. If we were to unlock before checking the queue, and the test runner process has already pushed all remaining data sets to the queue, we allow `test_runner_cleanup` to end the simulation and data is lost. When unlocking eventually does take place, `runner_phase` is activated and triggers `test_runner_cleanup` to verify that all locks are unlocked such that it can proceed.

Let’s take a look at what the log looks like after implementing these updates.

```
        0 fs - dut_checker                    -    TRACE - Locked test runner cleanup phase entry gate.
114000000 fs - dut_checker                    -    TRACE - Locked test runner cleanup phase entry gate.
222000000 fs - dut_checker                    -    TRACE - Locked test runner cleanup phase entry gate.
290000000 fs - runner                         -    TRACE - Entering test runner cleanup phase.
290000000 fs - runner                         -    TRACE - Halting on test runner cleanup phase entry gate.
326000000 fs - dut_checker                    -    TRACE - Unlocked test runner cleanup phase entry gate.
326000000 fs - runner                         -    TRACE - Passed test runner cleanup phase entry gate.
326000000 fs - runner                         -    TRACE - Passed test runner cleanup phase exit gate.
326000000 fs - runner                         -    TRACE - Entering test runner exit phase.
```

First, we can observe that `dut_checker` is locking the gate three times in succession without unlocking it in between. This is perfectly fine as it keeps a locked gate locked and allows for simpler code. The same goes for an unlocked gate; if it is unlocked multiple times, it will remain unlocked. This is another advantage of having unique keys, as a keyless design would not accommodate this behaviour.

We can also see how `test_runner_cleanup` halts on the entry gate and proceeds immediately after the locked gate lock is unlocked.

An alternative design solution for the `dut_checker` is to move the last if statement to the top:

```
dut_checker : process
  constant key : key_t := get_entry_key(test_runner_cleanup);
begin
  if is_empty(queue) then
    unlock(runner, key, dut_checker_logger);
  end if;
  if is_empty(queue) then
    wait until is_active(new_data_set);
  end if;
  lock(runner, key, dut_checker_logger);

  for i in 1 to pop(queue) loop
    wait until (rising_edge(clk) and output_tvalid = '1') or log_active(vunit_error, decorate("while waiting on output data"), logger =>
dut_checker_logger);
    check_equal(output_tdata, calculate_expected_output(pop(queue)));
  end loop;
end process;
```

The only difference is that there will be an initial unlock because of the initially empty queue but that is, as mentioned before, allowed.

```
        0 fs - dut_checker                   -   TRACE - Unlocked test runner cleanup phase entry gate.
        0 fs - dut_checker                   -   TRACE - Locked test runner cleanup phase entry gate.
114000000 fs - dut_checker                   -   TRACE - Locked test runner cleanup phase entry gate.
222000000 fs - dut_checker                   -   TRACE - Locked test runner cleanup phase entry gate.
290000000 fs - runner                        -   TRACE - Entering test runner cleanup phase.
290000000 fs - runner                        -   TRACE - Halting on test runner cleanup phase entry gate.
326000000 fs - dut_checker                   -   TRACE - Unlocked test runner cleanup phase entry gate.
326000000 fs - runner                        -   TRACE - Passed test runner cleanup phase entry gate.
326000000 fs - runner                        -   TRACE - Passed test runner cleanup phase exit gate.
326000000 fs - runner                        -   TRACE - Entering test runner exit phase.
```

It may be tempting to combine the two initial if statements into one where we first call `unlock` and then wait for the `new_data_set` event. However, this is a bad idea since it exposes us to a potential Time-Of-Check To Time-Of-Use bug. When we call `unlock`, it triggers `runner_phase`, which is an operation consuming delta cycles. During this time, a new data set may have been pushed to the queue and the `new_data_set` event is activated before we return from the `unlock` procedure. Consequently, `dut_checker` will miss the first event and block on the wait statement until the second event arrives some time after the DUT responded to the first data set. However, the first data set is still at the front of the queue and it will cause a failure when it is used to verify the DUT response to the second data set. This is a bug that only occurs under unfortunate timing circumstances and in this example we've added a dummy procedure with a finely tuned delay to showcase the scenario. Even if the risk of encountering this bug is low, or not even possible with the timing at hand, it would be unwise to make any assumptions about the test runner timing as that can change in the future.

```vhdl
dut_checker : process
  constant key : key_t := get_entry_key(test_runner_cleanup);
begin
  a_procedure_adding_some_delay;
  if is_empty(queue) then
    unlock(runner, key, dut_checker_logger);
    wait until is_active(new_data_set);
  end if;
  lock(runner, key, dut_checker_logger);

  for i in 1 to pop(queue) loop
    wait until (rising_edge(clk) and output_tvalid = '1') or log_active(vunit_error, decorate("while waiting on output data"), logger =>
dut_checker_logger);
    check_equal(output_tdata, calculate_expected_output(pop(queue)));
  end loop;
end process;
```

```
       0 fs - dut_checker                    -    TRACE - Unlocked test runner cleanup phase entry gate.
78000000 fs - dut_checker                    -    TRACE - Locked test runner cleanup phase entry gate.
82000000 fs - check                          -    ERROR - Equality check failed - Got 0000_0000_0110_1011 (107). Expected
0000_0001_0000_0000 (256).
```

With the two separate if statements, there is still a potential issue to consider. What if the test runner process pushes the last data set to the queue, notifies the `new_data_set` event, and then immediately calls `test_runner_cleanup`? If the `dut_checker` just missed that last data set and unlocks the gate, will it then have enough time to detect the new data set after the unlocking and lock the gate again, before `test_runner_cleanup` terminates the simulation? Fortunately, there is time for that, as long as the presented design is used.

Keeping both of the if statements at the beginning serves no purpose other than explaining the dangers of misplaced code optimizations. We recommend keeping the if statement that unlocks the gate at the bottom, thereby keeping the optimization temptation "out of sight".

At this point, we have designed a testbench where the `dut_checker` is solely responsible for carrying out its intended task and ensuring it is fully completed before the simulation ends (high cohesion). There are no responsibilities for the `test_runner` process. We have also ensured that the `dut_checker` does not make any assumptions about the timing of the `test_runner` (low coupling).

Phase Transition Events

In the previous chapters, we described how a process can prevent phase transitions by locking gates. However, there are also use cases that require processes to simply be aware of such transitions without needing to prevent them. To illustrate, let us take a look at the AXI Stream standard. This standard provides a set of protocol assertions that can be employed to verify if a stream conforms to the protocol. One such assertion, `AXI4STREAM_ERRM_STREAM_ALL_DONE_EOS`, states that

"At the end of simulation, all streams have had their corresponding TLAST transfer"

An AXI Stream protocol checker will not prevent a simulation from exiting but it must be given the opportunity to check that all streams have ended when that is about to happen. Let's see how this is solved in the ARM-provided protocol checker IP:

"The testbench that you are using has a signal called EOS_signal. You must drive EOS_signal HIGH at the end of the simulation for at least one clock cycle."

This is an example of a scenario in which a user must take a non-obvious step to ensure that a verification component works correctly. This is due to the fact that the ARM IP lacks a testbench structure to rely upon. A VUnit verification component, on the other hand, has a known structure in the mandatory presence of the `test_runner_cleanup` procedure. This procedure will trigger the `runner_phase` event when the phase changes to `test_runner_cleanup`, giving the AXI Stream protocol checker the option to act. However, remember that `runner_phase` is activated on all phase changes and also when there is activity related to phase gate locks. Therefore, before checking stream status, the protocol checker must verify that the event was caused by an imminent simulation exit and not some other change. This is done by confirming that the active phase is `test_runner_cleanup` and that the testbench is *within* the gates of that phase, i.e. it has passed the entry gate but not yet the exit gate. The principle for the `AXI4STREAM_ERRM_STREAM_ALL_DONE_EOS` assertion is outlined below. After ensuring that the simulation is about to terminate, the process is able to make its final check, provided it is done within a single delta cycle.

```
end_of_simulation_process : process
begin
  wait until is_active(runner_phase) and (get_phase = test_runner_cleanup) and is_within_gates;
  check_stream_activity;
  wait;
end process;
```

Incorporating this protocol checker into our example testbench generates the following log:

```
        0 fs - dut_checker                     -   TRACE - Unlocked test runner cleanup phase entry gate.
        0 fs - dut_checker                     -   TRACE - Locked test runner cleanup phase entry gate.
114000000 fs - dut_checker                     -   TRACE - Locked test runner cleanup phase entry gate.
222000000 fs - dut_checker                     -   TRACE - Locked test runner cleanup phase entry gate.
290000000 fs - runner                          -   TRACE - Entering test runner cleanup phase.
290000000 fs - runner                          -   TRACE - Halting on test runner cleanup phase entry gate.
326000000 fs - dut_checker                     -   TRACE - Unlocked test runner cleanup phase entry gate.
326000000 fs - runner                          -   TRACE - Passed test runner cleanup phase entry gate.
326000000 fs - axis_checker:STREAM_ALL_DONE_EOS -    PASS - Equality check passed for number of active streams - Got 0.
326000000 fs - runner                          -   TRACE - Passed test runner cleanup phase exit gate.
326000000 fs - runner                          -   TRACE - Entering test runner exit phase.
```

As you can see from the pass message, the check is perform just before the simulation comes to an end.

Final Words

In this blog we've shown how phases can be used to handle the final ticks of your simulation in a robust and reliable way while adhering to two key concepts of good code design: high cohesion and low coupling.

Do you have your own personal use cases for phases, or perhaps use cases that you think are not supported? Feel free to reach out and share them with us!