## What You Will Learn

1. An improved method for synchronizing various components of your testbench.
2. Techniques to ensure your testbench does not terminate prematurely, before the DUT has been fully verified.
3. Ways to quickly identify where the testbench is stuck on timeout or other errors.
4. A method for generating a system “core dump” for easier debugging when errors are detected.

## Introduction

Testbench code often requires the use of multiple concurrent processes to achieve various objectives. These processes may be independent and run without regard to one another, or they may need to be synchronized to coordinate their activities. An example of the first is an independent process driving a clock and a common example of the latter is a bus functional model (BFM) that waits for another process to provide the necessary address and data for the next bus transaction.

VUnit provides an event mechanism to facilitate such synchronization between processes. An event in this context indicates that something has happened and can be used to notify other processes about that happening. The other processes can then wait for the event to become active before proceeding with their tasks.

Before proceeding we need to distinguish the VUnit event mechanism from similar concepts. For example, VHDL has its own event concept for signals, which allows processes to wait for a signal event using the wait statement.

```
wait on new_data_set;
```

A common misunderstanding is that the wait statement will be activated if the process responsible for driving the `new_data_set` signal makes an assignment to it. However, this is not the case. In VHDL, the wait statement will remain blocked unless there is a **change** in the value of the signal. Let’s say the core of the data producing process is as follows:

```
for data_set in 1 to 3 loop
  produce_data(data_set);
  new_data_set <= true;
end loop;
```

The process waiting for data looks like this:

```
listening_to_vhdl_events : process is
  constant logger : logger_t := get_logger("Process waiting on VHDL events");
begin
  wait on new_data_set;
  trace(logger, "Got new data set");
  handle(data);
end process;
```

If the intial value of `new_data_set` is `false`, the waiting process will react to the first assignment, but miss all subsequent events and data. To fix this issue, we can use the `transaction` attribute on the signal. Every VHDL assignment is considered a *transaction* and the process below should not miss any data.

```
listening_to_vhdl_transactions : process is
  constant logger : logger_t := get_logger("Process waiting on VHDL transactions");
begin
  wait on new_data_set'transaction;
  trace(logger, "Got new data set");
  handle(data);
end process;
```

This is confirmed by the simulation trace log, which shows that the latter receiving process works as intended, while the first process misses all but the first event.

```
1000000 fs - Process waiting on VHDL events       -   TRACE - Got new data set (tb_traditional.vhd:81)
1000000 fs - Process waiting on VHDL transactions -   TRACE - Got new data set (tb_traditional.vhd:93)
2000000 fs - Process waiting on VHDL transactions -   TRACE - Got new data set (tb_traditional.vhd:93)
3000000 fs - Process waiting on VHDL transactions -   TRACE - Got new data set (tb_traditional.vhd:93)
```

Another solution to this problem is to toggle the event signal rather than setting it to `true`:

```
for data_set in 1 to 3 loop
  produce_data(data_set);
  new_data_set <= not new_data_set;
end loop;
```

Although the signal name may now seem a bit misleading, we no longer need the `transaction` attribute for the wait statement to function properly, as shown in this log:

```
1000000 fs - Process waiting on VHDL events       -   TRACE - Got new data set (tb_traditional.vhd:81)
1000000 fs - Process waiting on VHDL transactions -   TRACE - Got new data set (tb_traditional.vhd:93)
2000000 fs - Process waiting on VHDL events       -   TRACE - Got new data set (tb_traditional.vhd:81)
2000000 fs - Process waiting on VHDL transactions -   TRACE - Got new data set (tb_traditional.vhd:93)
3000000 fs - Process waiting on VHDL events       -   TRACE - Got new data set (tb_traditional.vhd:81)
3000000 fs - Process waiting on VHDL transactions -   TRACE - Got new data set (tb_traditional.vhd:93)
```

There are also other approaches that can be used, but the main point is that creating events that behave as intended requires an understanding of the inner workings of VHDL, and such details are better hidden behind an abstraction to prevent confusion and mistakes. By establishing an abstraction, it is also possible to further extend the event functionality without adding unnecessary complexity to the user's code. These are the main motivations for the VUnit event types.

This is a similar step to how the design of synchronous processes has evolved over time. Previously, they were based on the `clk'event and clk = '1'` condition, but now the more common approach is to use `rising_edge(clk)`. The latter hides the details of VHDL events and is a more powerful solution. While the previous solution would react to a `std_logic` `clk` changing from `X` to `1`, or even worse from `H` to `1`, the latter correctly does not.

Another synchronization mechanism that is often confused with the VUnit style of events is the binary semaphore. The main difference is that an event occurs at a **point in time**, and a process starting to wait for an event will block until the **next** event occurs. Binary semaphores, on the other hand, have a binary state. One of the states, often denoted with the integer 1 and representing the availablity of some resource, allow a process to proceed immediately regardless of whether the state was active when the process started waiting. There is no need for something to happen, an event, in order for the process to proceed. A semaphore checks its state first and only if the value is 0 there is a need to wait for the event indicating that the state has been set to 1. After a process has been allowed to proceed, it will set the semaphore value to 0 to prevent other processes from claiming the resource.

Events can be used not only to build semaphores, but also to create other, more complex synchronization mechanisms. VUnit message passing is an example of such a mechanism that is based on events. If you have used it, you may have noticed the `net` signal appearing in many subprogram calls and wondered what it is used for. The answer is that `net` is an event that indicates that something has happened in the message passing system. In addition to the `net` event, VUnit provides other events that are useful to users. We will discuss some of these events in this blog.

## Two Types of VUnit Events

VUnit events come in two types: `basic_event_t` and `event_t`. `basic_event_t` events, such as `net`, are provided by the VUnit framework and behave similarly to user-defined events of type `event_t`. The main difference is that basic events are predefined rather than dynamically created. Predefined events enable us to create a cleaner architecture within VUnit where low-level functionality can use events without creating many dependencies on other parts of VUnit. However, from a user-point of view the basic events behave very similar to user-defined events and provide the same user interface. For that reason we'll start explaining `event_t` before presenting the basic events.

## User-Defined Events

User-defined events can be created from identities (see identity package) or directly from a name string. In the latter case an identity is created automatically for that name unless it already exists.

```
signal new_data_set : event_t := new_event("new_data_set");
-- The above is equivalent to
-- signal new_data_set : event_t := new_event(get_id("new_data_set"));
```

> **Important**
>
> An event is always declared as a signal.

In the following example, we have two processes: the `test_runner` process and the `dut_checker` process. `test_runner` generates stimuli input for the device under test (DUT) and `dut_checker` verifies the DUT's response to that stimuli. The stimuli is created from a number of data set files and `test_runner` starts by pushing the total number of samples in each set to a VUnit queue. Next, it notifies `dut_checker` via the `new_data_set` event. `test_runner` then pushes the individual sample values to the same queue.

The code below shows the body of the `test_runner` process but also a `test_runner_watchdog`. We'll get back to that later.

```
begin
  test_runner_setup(runner, runner_cfg);

  for data_set_idx in 0 to n_data_sets - 1 loop
    data_set := load_data_from_file(data_set_idx);
    n_samples := length(data_set);
    push(queue, n_samples);
    notify(new_data_set);

    for sample_idx in 0 to n_samples - 1 loop
      sample := get(data_set, sample_idx);
      drive_dut(sample);
      push(queue, sample);
    end loop;
  end loop;

  test_runner_cleanup(runner);
end process;

test_runner_watchdog(runner, 500 ns);
```

If the queue is empty, the `dut_checker` process waits for the `new_data_set` event to be activated using the `is_active` function. When the event arrives, `dut_checker` pops the number of samples to expect from the queue. It then waits for that number of outputs from the DUT, and for each output it pops the corresponding input sample from the queue in order to calculate the expected output. The expected value is then compared to the actual value. The `dut_checker` process operates in parallel with the `test_runner` process but is slightly separated in time due to the latency of the DUT.

```
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';
    check_equal(output_tdata, calculate_expected_output(pop(queue)));
  end loop;
end process;
```

After `dut_checker` has received and checked the expected number of values it will look for a new data set in the queue and wait if not already present. In this case there is no need to wait because `test_runner` is applying the data sets back-to-back and operates ahead of `dut_checker` due to the DUT's latency.

The design described in the previous sections has a major issue that needs to be addressed: when `test_runner` has driven the last sample of the last data set, it directly calls the `test_runner_cleanup` function before the DUT has processed and output the result of all stimuli. As a result, there are DUT outputs that are never verified. One common fix for this problem is to insert a wait statement before the `test_runner_cleanup` function that adds a few clock cycles of delay, allowing the DUT pipeline to drain of all remaining data. However, this is a very dangerous solution as it relies on the latency of the DUT remaining constant. If the latency of the DUT increases due to updates to the design, the delay may not be sufficient and the issue will reappear. Additionally, if there is a bug that causes the DUT not to produce all outputs, or perhaps no outputs at all, the delay will expire regardless of how much safety margin is added, resulting in a potentially faulty test being marked as passing. A more robust solution is needed to ensure that all results are properly verified.

The key here is that we've assigned the task of verifying the output to the `dut_checker` process. Only this process can determine when the task is fully completed. Therefore, we will create a second event called `dut_checker_done`. `dut_checker` will signal this event when it has verified a data set and can't find any new input in the queue. It will still go and wait for more input in case it comes later.

```
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';
    check_equal(output_tdata, calculate_expected_output(pop(queue)));
  end loop;

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

In `test_runner` we add a safety barrier by waiting for this event before calling `test_runner_cleanup`.

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

test_runner_watchdog(runner, 500 ns);
```

In this case we're using `is_active_msg` instead of `is_active`. It has the same function but also produces a log message when the input event is active.

```
        0 fs - default                    -    INFO - Identity hierarchy: (tb_event.vhd:134)
                                                  dut_checker
                                                  \---done
326000000 fs - vunit_lib:event_pkg        -    INFO - Event dut_checker:done activated (tb_event.vhd:151)
```

Also note that we gave the event a hierarchical name `dut_checker:done`, i.e. the event `done` is owned by the `dut_checker` (pretty-printing of the identity hierarchy is done by the `get_tree` function in the identity package).

With the new event we have a more solid strategy for terminating the simulation, so let's experiment to see what happens if a new bug in the DUT causes it to stop producing data prematurely:

```
500000000 fs - runner                     -   ERROR - Test runner timeout after 500000000 fs. (tb_event.vhd:155)
```

The `test_runner_watchdog` mentioned previously helps us identify this issue, and in this simple example, it's relatively easy to find the root cause. However, in more complex situations, it can be challenging to know where to begin debugging because there are many potential points where the testbench may become stuck. VUnit provides a number of basic events that are useful in situations like these.

## VUnit-Provided Events

A method to identify blocking wait statements that cause the simulation to timeout is to use the VUnit-provided `runner_timeout` event in combination with the `log_active` function. The `log_active` function produces a log message just like `is_active_msg` but it always returns `false`. This means it can be used to identify blocking wait statements without unblocking them. We can use this in the `dut_checker` wait statement.

```
wait until (rising_edge(clk) and output_tvalid = '1') or log_active(runner_timeout);
check_equal(output_tdata, calculate_expected_output(pop(queue)));
```

This addition will create an extra log entry and the location information pinpoints the exact location of the wait statement.

```
500000000 fs - vunit_lib:event_pkg          -    INFO - Event runner:timeout activated (tb_event.vhd:368)
500000000 fs - runner                       -   ERROR - Test runner timeout after 500000000 fs. (tb_event.vhd:378)
```

If you are using Active-HDL or Riviera-PRO and compiling your code with VHDL-2019, log location is automatically supported. If that is no option, you can use VUnit's location preprocessor to achieve the same result (see log location). If you are not using either of these options, the extra entry (or entries if you have multiple wait statements) will not be helpful. However, there are other ways to identify the problematic wait statement(s):

1. Use a Custom logger

   We can create a logger representing `dut_checker` using either identities or a name string like this:

   ```
   constant dut_checker_logger : logger_t := get_logger("dut_checker");
   ```

   Next, we hand `dut_checker_logger` to the `log_active` function.

   ```
   wait until (rising_edge(clk) and output_tvalid = '1') or log_active(runner_timeout, logger => dut_checker_logger);
   ```

   This will bring our log message closer to the source ( `dut_checker` ) should we not have the exact location.

   ```
   500000000 fs - dut_checker                  -    INFO - Event runner:timeout activated (tb_event.vhd:392)
   500000000 fs - runner                       -   ERROR - Test runner timeout after 500000000 fs. (tb_event.vhd:411)
   ```

2. Use a Custom Message

In case `dut_checker` has several wait statements and we want to know which one is stuck, we can add a message to `log_active`.

```
wait until (rising_edge(clk) and output_tvalid = '1') or log_active(runner_timeout, "Waiting on output data", logger =>
dut_checker_logger);
```

The source of the problem is easier to identify but at the expense of losing information about why the log entry was produced (the runner:timeout event was activated).

```
500000000 fs - dut_checker                 -    INFO - Waiting on output data (tb_event.vhd:396)
500000000 fs - runner                      -   ERROR - Test runner timeout after 500000000 fs. (tb_event.vhd:411)
```

3. Use a Decorated Message

We can restore the lost event information by using a decorated message. Decoration is done with the `decorate` function which combines the automatically generated information with a message provided by the user. This technique may be familiar to those who have used the `result` function in check subprograms. `result` implements the same idea and is actually an alias for `decorate`.

```
wait until (rising_edge(clk) and output_tvalid = '1') or log_active(runner_timeout, decorate("while waiting on output data"), logger
=> dut_checker_logger);
```

The resulting log is as follows:

```
500000000 fs - dut_checker                 -    INFO - Event runner:timeout activated while waiting on output data
(tb_event.vhd:400)
500000000 fs - runner                      -   ERROR - Test runner timeout after 500000000 fs. (tb_event.vhd:411)
```

Using events to identify blocking wait statements is useful not only for timeout errors, but for any type of error. VUnit offers a more generic `vunit_error` event that is activated in addition to the `runner_timeout` event and can be activated from other error sources as well, including errors found by the testbench itself.

```
wait until (rising_edge(clk) and output_tvalid = '1') or log_active(vunit_error, decorate("while waiting on output data"), logger =>
dut_checker_logger);
```

Let's say we have a requirement on the maximum latency for our DUT and we want to check that. To do that we decide to apply the first data set on the DUT input, wait for the maximum latency, and then read a status register in the DUT containing a field with the number of processed samples. We expect that field to be the number of samples applied if the latency is within the requirement. This piece of code is placed after the loop applying all samples in a data set.

```
if data_set_idx = 0 then
  wait for max_latency;
  read_register(status_reg_addr, status);
  check_equal(status(n_samples_field), n_samples, result("for #processed samples"));
end if;
```

Tip

Register fields in VHDL can be defined by creating an integer subtype with the range set to the range of bits occupied by the field. In this case `n_samples_field` is defined as:

```
subtype n_samples_field is integer range 8 downto 1;
```

Running this code reveals that only half of the applied samples have been processed after the maximum latency.

```
174000000 fs - check                          -    ERROR - Equality check failed for #processed samples - Got 0000_0101 (5). Expected
10 (0000_1010). (tb_event.vhd:206)
```

If we could notify `vunit_error` when we encounter this error, we would expect `dut_checker`'s wait statement for output data to create a log entry since not all data have been produced. That would help confirming the latency issue. To do that we need to divide the `check_equal` procedure into two parts: the analysis and the action. The analysis compares the expected and actual number of samples, while the action logs an error in response to a failing equality as determined by the analysis.

The `check_equal` procedure has an equivalent function that only performs the analysis and returns the result. There is also a `log` procedure that takes the result as input and performs the action part. To address the issue we are facing, we want to have an alternative action procedure that notifies `vunit_error` if the result indicates a failing equality and *then* calls the `log` procedure. This alternative action procedure, called `notify_if_fail`, is already provided and takes the analysis result and an event to notify on failure as input.

```
if data_set_idx = 0 then
  wait for max_latency;
  read_register(status_reg_addr, status);
  wait until rising_edge(clk);
  notify_if_fail(check_equal(status(n_samples_field), n_samples, result("for #processed samples")), vunit_error);
end if;
```

Running the simulation again reveals the following log:

```
178000000 fs - vunit_lib:event_pkg          -    INFO - Event vunit_lib:vunit_error activated while waiting on "wait until
is_active(new_data_set);"
178000000 fs - check                        -   ERROR - Equality check failed for #processed samples - Got 0000_0101 (5). Expected
10 (0000_1010). (tb_event.vhd:265)
```

The `dut_checker` isn't stuck where we expected it to be. Instead it is waiting for the next data set which suggests that all of the expected output data have been received before the max latency, and we should look for the bug elsewhere. It turns out that the root cause of the issue is an error in the range definition of the `n_samples_field`. The range is shifted one bit which causes the read value to be half of the actual value. The additional information provided by the wait statements triggered by `vunit_error` can be very helpful in situations like these, as it can confirm our initial suspicions or direct us towards another possible explanation.

You may have noticed that the wait statement triggered in this case wasn't prepared with a call to the `log_active` function. So, how was the log entry generated? What we did was to create a `VUnit preprocessor` to identify wait statements and then modified them to include a call to `log_active`. This allowed us to automatically generate log entries for wait statements not prepared to generate extra debug information. The design of such a preprocessor somewhat depends on the project setup but you can use this example as a template for your own project.

```python
from vunit.ui.preprocessor import Preprocessor


class WaitStatementPreprocessor(Preprocessor):
    """A preprocessor is a class with a run method that transforms code. It is based on the Preprocessor class."""

    def __init__(self, order):
        """The order argument to the constructor controls the order in which preprocessors are applied.
        Lowest number first."""

        # Call constructor of base class
        super().__init__(order)

        # Regular expression finding wait statements on the form
        # wait [on sensitivity_list] [until condition] [for timeout];
        self._wait_re = re.compile(
            r"wait(\s+on\s+(?P<sensitivity_list>.*?))?(\s+until\s+(?P<condition>.*?))?(\s+for\s+(?P<timeout>.*?))?;",
            re.MULTILINE | re.DOTALL | re.IGNORECASE,
        )

    def run(self, code, file_name):
        """The run method must take the code string and the file_name as arguments."""

        # Only process testbenches
        if "runner_cfg" not in code:
            return code

        # Find all wait statements and sort them in reverse order of appearance to simplify processing
        wait_statements = list(self._wait_re.finditer(code))
        wait_statements.sort(key=lambda wait_statement: wait_statement.start(), reverse=True)

        for wait_statement in wait_statements:
            modified_wait_statement = "wait"

            # If the wait statement has an explicit sensitivity list (on ...), then vunit_error must be added to that
            sensitivity_list = wait_statement.group("sensitivity_list")
            if sensitivity_list is not None:
                new_sensitivity_list = f"{sensitivity_list}, vunit_error"
                modified_wait_statement += f" on {new_sensitivity_list}"

            # Add log_active to an existing condition clause (until ...) or create one if not present
            original_wait_statement = wait_statement.group(0)
            log_message = f'decorate("while waiting on ""{original_wait_statement}""")'
            condition = wait_statement.group("condition")
```

```
            if condition is None:
                new_condition = f"log_active(vunit_error, {log_message})"
            elif "vunit_error" in condition:
                continue  # Don't touch a wait statement already triggering on vunit_error
            else:
                new_condition = f"({condition}) or log_active(vunit_error, {log_message})"

            modified_wait_statement += f" until {new_condition}"

            # The time clause (for ...) is not modified
            timeout = wait_statement.group("timeout")
            if timeout is not None:
                modified_wait_statement += f" for {timeout}"

            modified_wait_statement += ";"

            # Replace original wait statement
            code = code[: wait_statement.start()] + modified_wait_statement + code[wait_statement.end() :]

        return code
```

The preprocessor is added to the project using the `add_preprocessor()` method. The order number must be higher than that of the location preprocessor which is 1000 by default. This is to avoid unintended interference between the two.

```
vu = VUnit.from_argv()
vu.enable_location_preprocessing()  # order = 1000 if no other value is provided
vu.add_preprocessor(WaitStatementPreprocessor(order=1001))
```

#### Exercise

Create a preprocessor that wraps check procedures in a `notify_if_fail` call such that all detected errors triggers the `vunit_error` event.

In this example, we were fortunate in that the error occurred during a wait statement that was directly related to the issue at hand, making it easy to locate the bug. However, this is not always the case. To fully understand the issue, it is often necessary to examine the internal signal state. One way to address this is to continuously log a large number of signals at every clock edge. However, this approach quickly leads to unwieldy and difficult-to-manage logs. A more effective solution is to

1. Continuously log a smaller, targeted set of interesting signals and events that can provide insights into what led up to the error.
2. At the time of an error, log a much larger set of signals to obtain detailed information about the state of the system.

The larger set of signals is provided by one or several “core dump” processes that are triggered by `vunit_error`. For example, in our code example, the DUT has a data processing pipeline and a control block that manage register reads and writes. Both of these blocks have states that are interesting for debugging. A core dump of these states might look like this:

```
-- pragma translate_off
core_dump : process
  constant logger : logger_t := get_logger("incrementer:core_dump");
begin
  wait until is_active(vunit_error);

  info(logger, "Control state is " & control_state_t'image(control_state));
  info(logger, "Data processing state is " & data_processing_state_t'image(data_processing_state));
end process;
-- pragma translate_on
```

In this case the core dump process was encapsulated in the RTL code and pragmas were used to exclude it from synthesis. An alternative, if supported by your simulator, is to add the process to the testbench and then use external names to access the DUT-internal state signals.

The updated log now shows that both blocks are idle when the error occurs. This also confirms that there is no latency issue, as there are no pending data being processed.

```
178000000 fs - vunit_lib:event_pkg          -    INFO - Event vunit_lib:vunit_error activated while waiting on "wait until
is_active(new_data_set);"
178000000 fs - incrementer:core_dump        -    INFO - Control state is idle (incrementer.vhd:180)
178000000 fs - incrementer:core_dump        -    INFO - Data processing state is idle (incrementer.vhd:181)
178000000 fs - check                        -   ERROR - Equality check failed for #processed samples - Got 0000_0101 (5). Expected 10
(0000_1010). (tb_event.vhd:265)
```

## Close, but No Cigar

So far VUnit events helped us synchronize processes, perform core dumps and reveal blocking wait statements to aid debugging, and prevent premature termination of a simulation. However, using events to create a barrier for premature terminations is a solution with several problems:

1. It doesn’t scale well. For every process that has to complete we need a new event.

2. There is a race condition. If a process completes before the test runner process starts waiting for the completion event, the test runner process will block and never call `test_runner_cleanup` to end the simulation.
3. If we fail to recognize that there is a completion event for a process, or simply forget to add it, we still face the risk of a premature simulation termination.

A better solution would be one that allows any process to prevent `test_runner_cleanup` from ending the simulation before that process has completed. There is only one `test_runner_cleanup` call so it scales well and forgetting to add it will cause the testbench to fail. The event race condition is also removed since a process completing before the test runner process reaches `test_runner_cleanup` will have stopped preventing simulation termination.

Fortunately VUnit provides such a solution. It's called VUnit phases and it will be the topic for the next blog.