Hardware Modeling [VU] (191.011)
– WS24 –
Simulation and Testbenches

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

This lecture gives an introduction on how the functionality of VHDL modules is verified using appropriate testbenches and a simulator.



# Hardware Modeling [VU] (191.011)
# – WS24 –
## Simulation and Testbenches

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

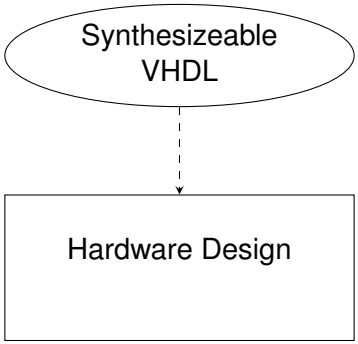
WS 2024/25

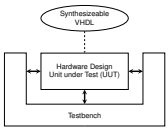
Modified: 2025-03-12, 16:24 (b25118c)


In previous lectures we talked about how VHDL can be used to design hardware. We also learned that in order to make it possible for a synthesis tool to take our code and convert it into a gate-level circuit we have to stick to the synthesizable subset of the language.

# Introduction



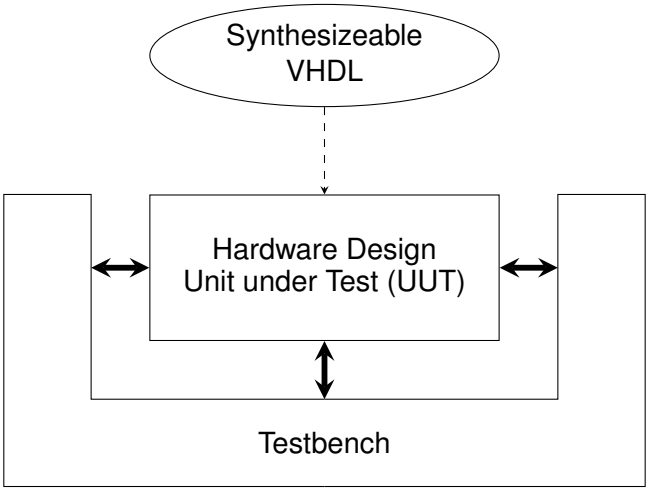


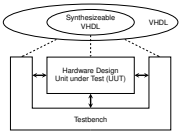



In this lecture we will now learn how to implement testbenches for our designs. Testbenches provide a controlled simulation environment for testing a hardware module without being part of the actual design itself. A testbench is used to apply stimulus – that is, generate input signals – to the unit under test and observe the resulting outputs.

# Introduction



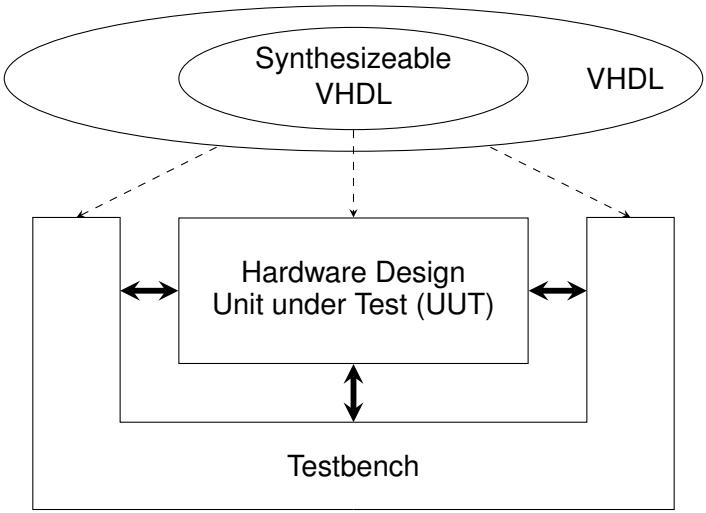





Since testbenches are not synthesized into hardware, but only ran in a simulator to validate the functional correctness of a VHDL design, we can use the complete VHDL language to implement them.

# Introduction





Testbenches are regular entities/architectures

Conceptionally, testbenches in VHDL don't use any special language constructs. A testbench is a regular entity with an architecture that contains the actual test code.

## Introduction (cont'd)



- Testbenches are regular entities/architectures

- Testbenches are regular entities/architectures
- Testbench architectures
  - create the instance of the unit under test (UUT)
  - produce input signals for the UUT
  - check the outputs of the UUT for correctness

The architecture of a testbench creates an instance of the module it tests and perform the necessary operations on it to ensure the hardware design behaves as expected. Common names for this instance are UUT or D-U-T. The latter acronym standing for device under test.

# Introduction (cont'd)



- Testbenches are regular entities/architectures
- Testbench architectures
  - create the instance of the unit under test (UUT)
  - produce input signals for the UUT
  - check the outputs of the UUT for correctness

- Testbenches are regular entities/architectures
- Testbench architectures
  - create the instance of the unit under test (UUT)
  - produce input signals for the UUT
  - check the outputs of the UUT for correctness
- Testbench entities
  - have no ports
  - may have generics

One specialty of testbench entities is that they don't have any port signals. This is not something the language itself forbids, it is rather a consequence of the role testbenches play in simulation. A testbench is the top-level module in the simulator and is responsible for generating stimulus and checking responses. It mimics a self-contained real-world environment for the circuit and, hence, can not depend on external communication through ports. However, there is nothing wrong with generics on a testbench entity. Those can, for example, be used to parametrize the simulation and select between different scenarios or environmental conditions. They are also utilized heavily in some more advanced VHDL testing frameworks.

## Introduction (cont'd)



- Testbenches are regular entities/architectures
- Testbench architectures
  - create the instance of the unit under test (UUT)
  - produce input signals for the UUT
  - check the outputs of the UUT for correctness
- Testbench entities
  - have no ports
  - may have generics

- Testbenches are regular entities/architectures
- Testbench architectures
  - create the instance of the unit under test (UUT)
  - produce input signals for the UUT
  - check the outputs of the UUT for correctness
- Testbench entities
  - have no ports
  - may have generics
- Unit test for a hardware module

Finally, to again draw a comparison to the software world: You can think of a testbench as a unit test for a hardware module.

# Introduction (cont'd)



- Testbenches are regular entities/architectures
- Testbench architectures
  - create the instance of the unit under test (UUT)
  - produce input signals for the UUT
  - check the outputs of the UUT for correctness
- Testbench entities
  - have no ports
  - may have generics
- Unit test for a hardware module

Wait statements used so far
Unconditional "wait;" (at the end of a process)
Sensitivity lists (equivalent to a "wait on [...];" at the end of a process)

Before we look at some example testbenches, let us first discuss the non-synthesizable features of VHDL that are vital for testbenches and simulation. The most important non-synthesizeable VHDL constructs for testbenches are arguably the different flavors of wait statements and they way how they can be used. We have already seen a few wait statements in previous lectures and discussed their semantics. So far all processes containing non-synthesizable code ended in an unconditional wait statement, which basically stops the process. In the previous lecture, we have also learned about Sensitivity lists and the equivalent wait-on statements at the end of a synthesizable process.

## Wait Statements



- Wait statements used so far
  - Unconditional "`wait;`" (at the end of a process)
  - Sensitivity lists (equivalent to a "`wait on [...];`" at the end of a process)

- Wait statements used so far
  - Unconditional "wait;" (at the end of a process)
  - Sensitivity lists (equivalent to a "wait on [...];" at the end of a process)
- Wait until a condition **becomes** true:
  `wait until condition;`
- Wait for a specific amount of time:
  `wait for expression;`

Besides those variants there also exist the ”wait-until” and ”wait-for” statements. Wait-until statements waits until the specified condition becomes true. The reason why we emphasize the word ”becomes” is that, if the condition is already true when the simulator comes to this statement, it will not simply go past it and continue. It will first wait until the condition is false and then wait until it becomes true. This is a common pitfall for beginners in VHDL. Wait-for statements are a little bit easier to understand. They simply wait for the delay period specified by the expression after the ”for” keyword.

# Wait Statements



- Wait statements used so far
  - Unconditional “`wait;`” (at the end of a process)
  - Sensitivity lists (equivalent to a “`wait on [...];`” at the end of a process)
- Wait until a condition **becomes** true:

```
wait until condition;
```

- Wait for a specific amount of time:

```
wait for expression;
```

- Wait statements used so far
  - Unconditional "wait;" (at the end of a process)
  - Sensitivity lists (equivalent to a "wait on [...];" at the end of a process)
- Wait until a condition **becomes** true:
  `wait until condition;`
- Wait for a specific amount of time:
  `wait for expression;`
- Control the flow of time in the simulator

Generally, ”wait-until” statements can only be synthesized in very special cases, which we won’t discuss any further in this lecture. Wait-for statements can not be synthesized but allow us to control the flow of time within simulations.

# Wait Statements



- Wait statements used so far
  - Unconditional “`wait;`” (at the end of a process)
  - Sensitivity lists (equivalent to a “`wait on [...];`” at the end of a process)
- Wait until a condition **becomes** true:

```
wait until condition;
```

- Wait for a specific amount of time:

```
wait for expression;
```

- Control the flow of time in the simulator

This slide shows a simple example that demonstrates how the ”wait-for” and ”wait-until” statements work.

# Wait Statements - Example



```
1 entity wait_example is
2 end entity;
3
```

The architecture to the ”wait-example” entity only contains a single signal ”X”.

## Wait Statements - Example



```
1 entity wait_example is
2 end entity;
3
4 architecture arch of wait_example is
5  signal x : std_ulogic;
6 begin
```

In the statement part we define the process ”proc-A”, which initially sets ”X” to zero, then waits for 2.5 nanoseconds before setting it to one. Finally, an unconditional wait statement is used to stop the process.

## Wait Statements - Example



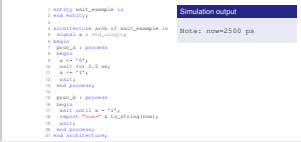
```
1  entity wait_example is
2  end entity;
3
4  architecture arch of wait_example is
5   signal x : std_ulogic;
6  begin
7   proc_a : process
8   begin
9    x <= '0';
10   wait for 2.5 ns;
11   x <= '1';
12   wait;
13  end process;
```

The process ”proc-B” uses a wait-until statement to wait until ”X” is set to one and then reports the current simulation time, which can conveniently be accessed by the global ”now”.

# Wait Statements - Example



```
1  entity wait_example is
2  end entity;
3
4  architecture arch of wait_example is
5   signal x : std_ulogic;
6  begin
7   proc_a : process
8   begin
9    x <= '0';
10   wait for 2.5 ns;
11   x <= '1';
12   wait;
13  end process;
14
15  proc_b : process
16  begin
17   wait until x = '1';
18   report "now=" & to_string(now);
19   wait;
20  end process;
21 end architecture;
```



Hence, the output of this example code is 2500 picoseconds. Finally, the simulation ends, since all processes stopped because of the unconditional wait statements.

## Wait Statements - Example



```
1  entity wait_example is
2  end entity;
3
4  architecture arch of wait_example is
5   signal x : std_ulogic;
6  begin
7   proc_a : process
8   begin
9    x <= '0';
10   wait for 2.5 ns;
11   x <= '1';
12   wait;
13  end process;
14
15  proc_b : process
16  begin
17   wait until x = '1';
18   report "now=" & to_string(now);
19   wait;
20  end process;
21 end architecture;
```

**Simulation output**

```
Note: now=2500 ps
```

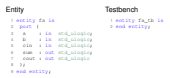

Let's now look at our first testbench. We start with a simple testbench for the full adder circuit that we have already encountered in previous lectures. During the course of this lecture we will then further refine this testbench. To refresh your memory, here is the entity declaration of the full adder. Recall that it has three single-bit inputs, for which it calculates the sum and carry.

# Example: Full Adder Testbench



## Entity

```
1 entity fa is
2  port (
3   a    : in  std_ulogic;
4   b    : in  std_ulogic;
5   cin  : in  std_ulogic;
6   sum  : out std_ulogic;
7   cout : out std_ulogic
8  );
9 end entity;
```




To implement the testbench, we begin with its entity declaration. As previously mentioned, testbench entities do not include any port signals. Additionally, since we are working with a very basic example, no generic parameters are needed. As a result, the entity we use here is entirely empty. Notice that the name of the testbench ends with the suffix ”underscore-t-b”. We will use this naming convention throughout this course. A testbench is always named after the module it tests and ends in underscore-t-b.

# Example: Full Adder Testbench



## Entity

```
1 entity fa is
2  port (
3   a    : in  std_ulogic;
4   b    : in  std_ulogic;
5   cin  : in  std_ulogic;
6   sum  : out std_ulogic;
7   cout : out std_ulogic
8  );
9 end entity;
```

## Testbench

```
1 entity fa_tb is
2 end entity;
```


After the entity declaration follows the declaration of the accompanying architecture. Its declarative part contains at least the declarations for the signals that are then connected to the port signals of the unit under test. The names of these local signals usually exactly match the names of the port signals of the UUT.

# Example: Full Adder Testbench



## Entity

```
1 entity fa is
2  port (
3   a    : in  std_ulogic;
4   b    : in  std_ulogic;
5   cin  : in  std_ulogic;
6   sum  : out std_ulogic;
7   cout : out std_ulogic
8  );
9 end entity;
```

## Testbench

```
1 entity fa_tb is
2 end entity;
3
4 architecture tb of fa_tb is
5  signal a, b, cin, sum, cout : std_ulogic;
6 begin
```


In the statement part the actual instance of the UUT is created. The port map clause connects its port signals to the local signals declared above.

# Example: Full Adder Testbench



## Entity

```
1 entity fa is
2  port (
3   a     : in  std_ulogic;
4   b     : in  std_ulogic;
5   cin   : in  std_ulogic;
6   sum   : out std_ulogic;
7   cout  : out std_ulogic
8  );
9 end entity;
```

## Testbench

```
1 entity fa_tb is
2 end entity;
3
4 architecture tb of fa_tb is
5  signal a, b, cin, sum, cout : std_ulogic;
6 begin
7  uut : entity work.fa
8  port map (
9   a     => a,
10  b     => b,
11  cin   => cin,
12  sum   => sum,
13  cout  => cout
14 );
15
```


Finally, and most importantly we have to implement the actual test code that interacts with the UUT. For simple testbenches this can usually be done in a single process – here named stimulus. However, more complex modules might require several processes that serve different, independent interfaces of the UUT. A simple example – that we will encounter later in this course – would be a clock generation process for synchronous modules. However, as for now, we are only dealing with purely combinational modules such as the full-adder, we don't yet have to worry about that.

# Example: Full Adder Testbench



## Entity

```
1 entity fa is
2  port (
3   a     : in  std_ulogic;
4   b     : in  std_ulogic;
5   cin   : in  std_ulogic;
6   sum   : out std_ulogic;
7   cout  : out std_ulogic
8  );
9 end entity;
```

## Testbench

```
1 entity fa_tb is
2 end entity;
3
4 architecture tb of fa_tb is
5  signal a, b, cin, sum, cout : std_ulogic;
6 begin
7  uut : entity work.fa
8  port map (
9   a    => a,
10  b    => b,
11  cin  => cin,
12  sum  => sum,
13  cout => cout
14 );
15
16 stimulus : process [...]
17
18 end architecture;
```

OK, let's see how such a stimulus process for our full adder circuit can look like. We start with a regular process that initially sets all the inputs of the UUT to zero.

# Example: Full Adder Testbench (cont'd)



```
1 stimulus : process
2 begin
3  a <= '0';
4  b <= '0';
5  cin <= '0';
```

Then we use some a "wait-for" statement to advance the simulation time. Recall from the previous lecture, that according to the semantics of processes, the signal assignments only take effect when the wait statement is executed. Since the full adder circuit we are simulating does not contain any timing information the actual amount of time we wait here is unimportant. Every change at the inputs of the full adder will immediately be reflected by its outputs – "immediately" meaning that no simulation time elapses. However, it is still common practice to include some delay in the testbench to allow for better visualization of the input and output changes during simulation. This makes the waveforms generated by the simulator easier to read and interpret. This instantaneous behavior is, of course, not very realistic. However, it allows us to verify if the circuit works on a logical level. In an upcoming lecture, we will learn about different types of simulations and how a more realistic timing behavior can be incorporated.

## Example: Full Adder Testbench (cont'd)



```
1 stimulus : process
2 begin
3  a <= '0';
4  b <= '0';
5  cin <= '0';
6  wait for 1 ns;
```

After the first set of input values, we continue with the rest of the 8 possible input combinations, until we finally reach the state where all inputs are one.

# Example: Full Adder Testbench (cont'd)



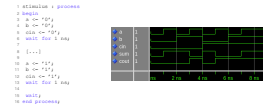
```
1  stimulus : process
2  begin
3    a <= '0';
4    b <= '0';
5    cin <= '0';
6    wait for 1 ns;
7
8    [...]
9
10   a <= '1';
11   b <= '1';
12   cin <= '1';
13   wait for 1 ns;
```

Finally, we need an unconditional wait statement to stop the process at the end of the simulation. This will signal the simulator that no more signal changes are possible, and the simulation can be stopped. If this wait statement would be omitted, the execution of the stimulus process would restart at the beginning.

# Example: Full Adder Testbench (cont'd)



```
1  stimulus : process
2  begin
3   a <= '0';
4   b <= '0';
5   cin <= '0';
6   wait for 1 ns;
7
8   [...]
9
10  a <= '1';
11  b <= '1';
12  cin <= '1';
13  wait for 1 ns;
14
15  wait;
16 end process;
```

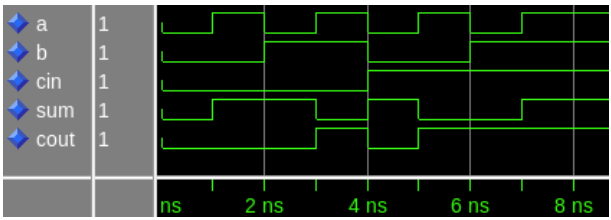


The right side of the slide shows the waveform output of the simulation in "Questa-Sim" – the simulation tool that we will use in the exercise part of this course.

# Example: Full Adder Testbench (cont'd)



```
1  stimulus : process
2  begin
3    a <= '0';
4    b <= '0';
5    cin <= '0';
6    wait for 1 ns;
7
8    [...]
9
10   a <= '1';
11   b <= '1';
12   cin <= '1';
13   wait for 1 ns;
14
15   wait;
16 end process;
```

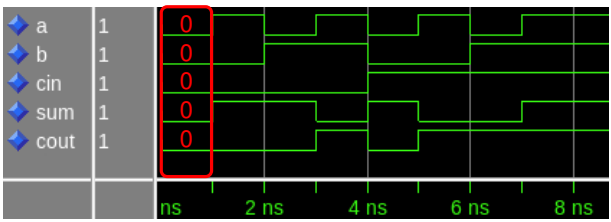


If we look at the first nanosecond of execution time, we see that, indeed, the inputs are set to zero and that the outputs c-out and sum are both zero, as well. We can hence conclude that the circuit works as intended for this set of input values.

# Example: Full Adder Testbench (cont'd)



```
1  stimulus : process
2  begin
3    a <= '0';
4    b <= '0';
5    cin <= '0';
6    wait for 1 ns;
7
8    [...]
9
10   a <= '1';
11   b <= '1';
12   cin <= '1';
13   wait for 1 ns;
14
15   wait;
16 end process;
```

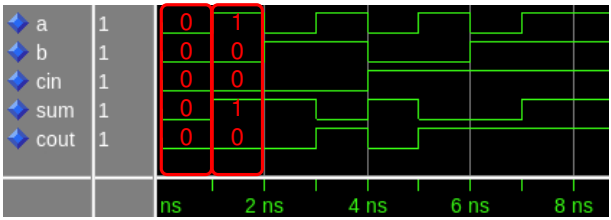


For the next set of input values, just the input A is set to one, which results in the output "sum" to be set to one.

# Example: Full Adder Testbench (cont'd)



```
1  stimulus : process
2  begin
3    a <= '0';
4    b <= '0';
5    cin <= '0';
6    wait for 1 ns;
7
8    [...]
9
10   a <= '1';
11   b <= '1';
12   cin <= '1';
13   wait for 1 ns;
14
15   wait;
16 end process;
```

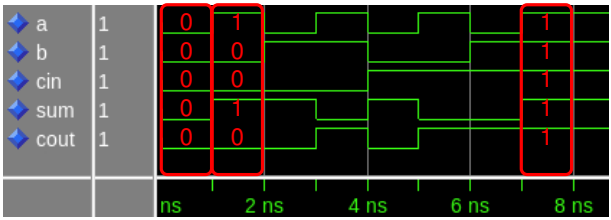


Finally, we reach the state where all inputs are one. According to the specification of the full adder, we know that in this case both outputs must also be one – which is clearly the case in the simulation. Notice that this testbench is exhaustive, which means that it tests every possible input combination. Hence, when checking the waveform we can be absolutely sure that our full adder works as intended. However, unfortunately in practice such exhaustive tests are generally not possible. The number of input combinations rises exponentially with the number of inputs. Making matters worse, usually modules also have internal state that must be taken into account. Thus, when devising testbenches, we have to be careful with the actual number of tests we can perform such that the simulation is feasible and still provides good test coverage. However, we will hear more about that in an upcoming lecture.

# Example: Full Adder Testbench (cont'd)



```
1  stimulus : process
2  begin
3    a <= '0';
4    b <= '0';
5    cin <= '0';
6    wait for 1 ns;
7
8    [...]
9
10   a <= '1';
11   b <= '1';
12   cin <= '1';
13   wait for 1 ns;
14
15   wait;
16 end process;
```



Note
Checking waveforms is **hard** and **time-consuming**! It is completely infeasible to verify large designs this way.

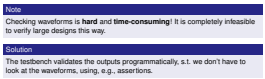
The full adder testbench on the previous slide only applied input stimulus. In order to assess the correctness of the implementation, we had to check the output waveform of the simulator. Even for this simple example it takes quite some time to go through all the output values and see if they match the expected values. To make matters worse, if we change something about the implementation of the full adder we would have to redo all this work. This is a painstaking process, and – as you might suspect – it does not scale with larger and more complex designs. Hence, we need a better way of verifying the outputs of the UUT.

# Assertions



## Note

Checking waveforms is **hard** and **time-consuming**! It is completely infeasible to verify large designs this way.




This is where assertions come into play. You might already be familiar with this concept, as almost all programming languages come with a comparable feature. Using assertions, the testbench can directly check the signals produced by the UUT against some specification. A testbench that is constructed in this way, can easily be run whenever something is changed in a design without the need to look at any waveforms.

# Assertions

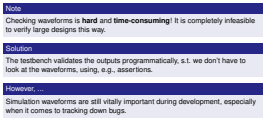


## Note

Checking waveforms is **hard** and **time-consuming**! It is completely infeasible to verify large designs this way.

## Solution

The testbench validates the outputs programmatically, s.t. we don't have to look at the waveforms, using, e.g., assertions.




Nevertheless, the graphical output of the simulation is still vitally important during the hardware development process. Especially when you have to dig down into the inner workings of a module when you are looking for bugs.

# Assertions



## Note

Checking waveforms is **hard** and **time-consuming**! It is completely infeasible to verify large designs this way.

## Solution

The testbench validates the outputs programmatically, s.t. we don't have to look at the waveforms, using, e.g., assertions.

## However, ...

Simulation waveforms are still vitally important during development, especially when it comes to tracking down bugs.


Assertion statements in VHDL can be viewed as conditional report statements. They use the ”assert” keyword, followed by an expression that must evaluate to a value of the built-in type boolean. If this value is false, the assertion is violated and the simulator executes the optional report statement. As with regular report statements the expression after the report keyword must evaluate to a string. Finally, the severity level of the assertion is selected by the last expression.

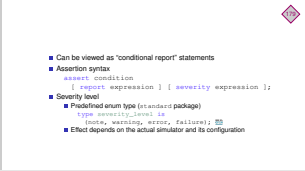
## Assertions (cont'd)





- Can be viewed as “conditional report” statements
- Assertion syntax

```
assert condition
  [ report expression ] [ severity expression ];
```




This expression must evaluate to a value of the built-in enum type "severity-level", defined in the standard package. The severity level defines how the simulator reacts to assertion violations. However, the actual effect of the severity level on the simulation is dependent on the simulator and how it is configured. We will now briefly discuss how they are handled by "Questa-Sim". The default level is "error" in which case an error message is printed, but the simulation is continued. If you want a behavior similar to a conditional report statements, you use the severity level "note". In fact, report statements can also be optionally equipped with a severity level. The levels "warning" and "error" increase the warning and error counter in the simulator. You can use these levels for minor and major issues detected during simulation. The failure level immediately leads to the termination of the simulation. It should be used when an error encountered that is so severe that continuing the simulation is futile – for example when the internal state of some module is corrupted.

# Assertions (cont'd)





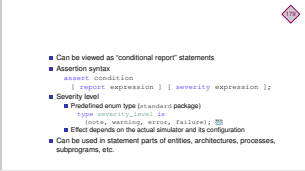
- Can be viewed as "conditional report" statements
- Assertion syntax

```
assert condition
  [ report expression ] [ severity expression ];
```

- Severity level
  - Predefined enum type (`standard` package)

```
type severity_level is
  (note, warning, error, failure);
```

  - Effect depends on the actual simulator and its configuration




Assertions can be used in the statement parts of various VHDL constructs, such as entities, architectures, processes and subprograms. An assertion in an entity declaration can, for example, be used to perform a sanity check on generic values. Such an assertion can also make sense in synthesizable code, as it can be used to raise an error during synthesis for illegal configuration parameters.

## Assertions (cont'd)





- Can be viewed as "conditional report" statements
- Assertion syntax

```
assert condition
  [ report expression ] [ severity expression ];
```

- Severity level
  - Predefined enum type (`standard` package)

```
type severity_level is
  (note, warning, error, failure);
```

  - Effect depends on the actual simulator and its configuration
- Can be used in statement parts of entities, architectures, processes, subprograms, etc.


Ok, let us now apply this knowledge to the full adder testbench example. Here, we see a modified version of the stimulus process. First we print a message that indicates which input combination is currently being tested. Next, we apply the input signals and wait for some time, exactly as in the previous version. After the wait statement, we have added two assertions that check the sum and carry-out signals. This sequence of operations is then repeated for all remaining input combinations.

# Assertions - Example



## Stimulus process with assertions

```
1  stimulus : process
2  begin
3   report "testing input 000";
4   a <= '0';
5   b <= '0';
6   cin <= '0';
7   wait for 1 ns;
8   assert cout = '0'
9    report "wrong carry" severity error;
10  assert sum = '0'
11   report "wrong sum" severity error;
12
13  [...]
14
15  wait;
16 end process;
```


On the right side of the slide, we now see the output when this testbench is run in Questa-Sim To make things a little bit more interesting we introduced a bug in the full adder implementation. As you can see, this modification leads to four assertion violations for the carry signal. Also notice that, for each executed report statement the simulation also outputs the current simulation time. At the end of the simulation, the total number of errors and warnings is reported. You might want to pause the video at this point to really study the simulation output. As a little exercise you can also try to figure out what we changed in the full adder circuit to produce the shown results.

# Assertions - Example



## Stimulus process with assertions

```
1  stimulus : process
2  begin
3   report "testing input 000";
4   a <= '0';
5   b <= '0';
6   cin <= '0';
7   wait for 1 ns;
8   assert cout = '0'
9    report "wrong carry" severity error;
10  assert sum = '0'
11   report "wrong sum" severity error;
12
13  [...]
14
15  wait;
16 end process;
```

## Simulator Output (QuestaSim)

```
# ** Note: testing input 000
#    Time: 0 ps  Iteration: 0  Instance: /fa_tb
# ** Note: testing input 001
#    Time: 1 ns  Iteration: 0  Instance: /fa_tb
# ** Note: testing input 010
#    Time: 2 ns  Iteration: 0  Instance: /fa_tb
# ** Note: testing input 011
#    Time: 3 ns  Iteration: 0  Instance: /fa_tb
# ** Error: wrong carry
#    Time: 4 ns  Iteration: 0  Instance: /fa_tb
# ** Note: testing input 100
#    Time: 4 ns  Iteration: 0  Instance: /fa_tb
# ** Note: testing input 101
#    Time: 5 ns  Iteration: 0  Instance: /fa_tb
# ** Error: wrong carry
#    Time: 6 ns  Iteration: 0  Instance: /fa_tb
# ** Note: testing input 110
#    Time: 6 ns  Iteration: 0  Instance: /fa_tb
# ** Error: wrong carry
#    Time: 7 ns  Iteration: 0  Instance: /fa_tb
# ** Note: testing input 111
#    Time: 7 ns  Iteration: 0  Instance: /fa_tb
# ** Error: wrong carry
#    Time: 8 ns  Iteration: 0  Instance: /fa_tb
#  quit
# End time: xx:xx:xx on xx xx,xx, Elapsed time: xx
# Errors: 4, Warnings: 0
```

You have probably noticed that the stimulus process of the testbenches shown so far are quite long and contain a lot of repetitive code. This makes them hard to maintain and error-prone during development. Hence, the question arises: Can we do better by applying some of the language constructs we have already learned? This is what we are going to do on this slide, by implementing an improved stimulus process.

## Further Automation



```
1 stimulus : process
```

We first declare two variables that we are going to use for temporary values in the process.

# Further Automation



```
1 stimulus : process
2  variable v : std_ulogic_vector(2 downto 0);
3  variable h : natural;
4 begin
```

Then, instead of going through all input combination manually, we use a for-loop that goes through the values 0 to 7.

## Further Automation



```
1 stimulus : process
2  variable v : std_ulogic_vector(2 downto 0);
3  variable h : natural;
4 begin
5  for i in 0 to 7 loop
```

In this for-loop, we first perform a type conversion to convert the loop counter integer variable ”i” to a three-bit `std_ulogic_vector` and assign it to the variable ”v”. The individual elements of this vector are then assigned to the input signals ”A”, ”B” and ”c-in” of the UUT.

## Further Automation



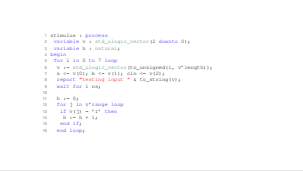
```
1 stimulus : process
2  variable v : std_ulogic_vector(2 downto 0);
3  variable h : natural;
4 begin
5  for i in 0 to 7 loop
6   v := std_ulogic_vector(to_unsigned(i, v'length));
7   a <= v(0); b <= v(1); cin <= v(2);
```

The report statement prints the current input combination in the same way the previous version of the stimulus process did.

# Further Automation



```
1 stimulus : process
2  variable v : std_ulogic_vector(2 downto 0);
3  variable h : natural;
4 begin
5  for i in 0 to 7 loop
6   v := std_ulogic_vector(to_unsigned(i, v'length));
7   a <= v(0); b <= v(1); cin <= v(2);
8   report "testing input " & to_string(v);
9   wait for 1 ns;
```

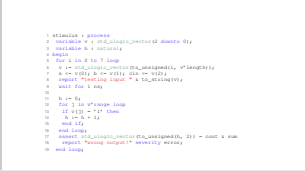


The part after the wait statement is a little tricky. Here, we use another for-loop to determine the number of bits in the variable ”V” that are set to one. Since we are actually calculating the Hamming weight of ”V” we named the variable ”H”. Because we have three inputs this number must be within the range 0 to 3.

## Further Automation



```
1  stimulus : process
2   variable v : std_ulogic_vector(2 downto 0);
3   variable h : natural;
4  begin
5   for i in 0 to 7 loop
6    v := std_ulogic_vector(to_unsigned(i, v'length));
7    a <= v(0); b <= v(1); cin <= v(2);
8    report "testing input " & to_string(v);
9    wait for 1 ns;
10
11   h := 0;
12   for j in v'range loop
13    if v(j) = '1' then
14     h := h + 1;
15    end if;
16   end loop;
```

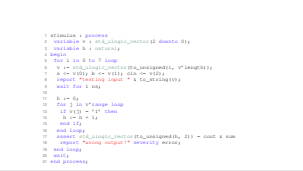


After the for-loop we finally come to the assertion. The assertion converts the variable ”H” into a `std_ulogic_vector` and compares it to the vector formed by the concatenation of sum and c-out. You can pause to video to really understand why this assertion works.

## Further Automation



```
1  stimulus : process
2   variable v : std_ulogic_vector(2 downto 0);
3   variable h : natural;
4  begin
5   for i in 0 to 7 loop
6    v := std_ulogic_vector(to_unsigned(i, v'length));
7    a <= v(0); b <= v(1); cin <= v(2);
8    report "testing input " & to_string(v);
9    wait for 1 ns;
10
11   h := 0;
12   for j in v'range loop
13    if v(j) = '1' then
14     h := h + 1;
15    end if;
16   end loop;
17   assert std_ulogic_vector(to_unsigned(h, 2)) = cout & sum
18    report "wrong output!" severity error;
19  end loop;
```



Finally, we need the unconditional wait to terminate the simulation.

# Further Automation



```
1  stimulus : process
2   variable v : std_ulogic_vector(2 downto 0);
3   variable h : natural;
4  begin
5   for i in 0 to 7 loop
6    v := std_ulogic_vector(to_unsigned(i, v'length));
7    a <= v(0); b <= v(1); cin <= v(2);
8    report "testing input " & to_string(v);
9    wait for 1 ns;
10
11   h := 0;
12   for j in v'range loop
13    if v(j) = '1' then
14     h := h + 1;
15    end if;
16   end loop;
17   assert std_ulogic_vector(to_unsigned(h, 2)) = cout & sum
18    report "wrong output!" severity error;
19  end loop;
20  wait;
21 end process;
```

Thank you for listening! We recommend you to immediately take the self-check test in TUWEL, to see if you understood the material presented in this lecture.



# Lecture Complete!