VHDL Testbench Tutorial

Learning Goal: Learning how to write a testbench in VHDL.

Requirements: ModelSim.

Created by: Sahand Kashani-Akhavan.

1 Introduction

An engineer's job does not stop after having created a "solution" to a specific problem, but he/she must also be able to demonstrate, to various degrees of certitude, that the solution is *correct*. This statement is equally valid in both software & hardware engineering, but it is especially important in the hardware domain, as errors can generally not be fixed once a product has been shipped to customers!

When describing digital circuits in VHDL, one generally tests the correctness of their implementation with a VHDL *testbench*. A testbench is generally a *non-synthesizable* VHDL file which iteratively applies a sequence of controlled inputs to a circuit and compares its concrete output against the expected output. If a mismatch is detected, an error is displayed in the VHDL simulator's log which can then be consulted to help direct a designer search for the problem in the circuit's RTL description.

This tutorial introduces readers to the craft of writing simple VHDL testbenches. We start with an empty VHDL testbench file and iteratively explain our thought process and how it affects the way we construct the testbench. We start with a simple testbench for a combinatorial circuit, then move on towards a more complicated testbench for a sequential circuit.

For simplicity, we will introduce *black box* testing. This method tests a circuit by considering it is concealed in a black box, with only its interface visible to the person testing the system. It allows one to abstract away the implementation details of the circuit and only test its behaviour at its interface. By recursively performing black box testing on all subcircuits used in larger designs, we can, with high confidence, compose systems which are correct by construction.

Note that we will not be performing *exhaustive* testing, which means we will not test our circuits against all possible input combinations, but rather against a carefully-chosen set of *test vectors*. The number of input combinations grows exponentially with the size of a circuit's inputs, so it quickly becomes infeasible to try all combinations (unless you are willing to wait several hours for the result of a simulation).

2 Testbench architecture

There are multiple ways of developing a testbench, but the one we will develop throughout this tutorial is shown in Figure 1. It consists of three 3 parts:

1. The component we want to test, i.e. the *Design Under Test* (DUT).
2. A mechanism for supplying inputs to the DUT.
3. A mechanism for checking the outputs of the DUT against expected outputs.

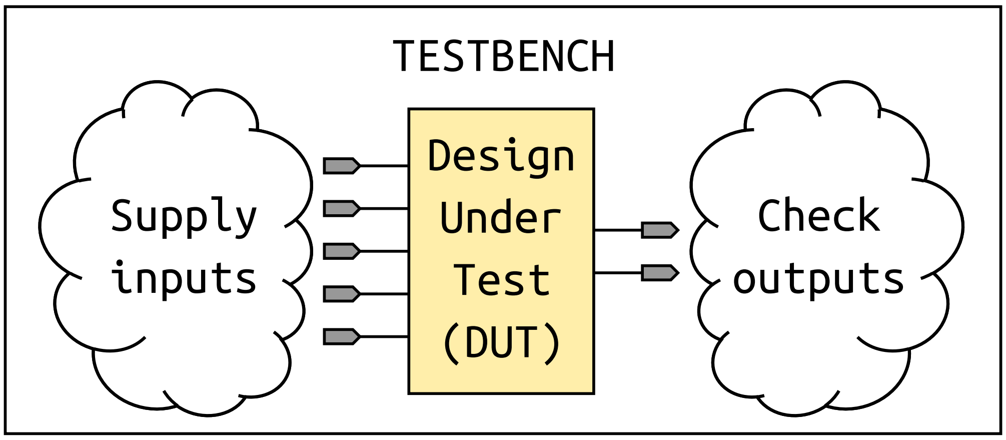


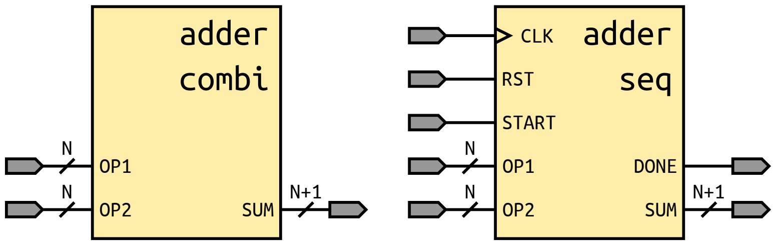
Figure 1: Testbench architecture

3 Victims / test subjects

We will be testing two RTL designs of a generic N-bit adder:

1. Combinatorial ripple-carry adder (Figure 2a.)
2. Sequential adder (Figure 2b.)

Each design is based upon the implementation of a 1-bit combinatorial full-adder. We assume the 1-bit full-adder is correct, so we will not be explicitly testing it.



(a) Combinatorial adder (b) Sequential adder

Figure 2: Adders

4 Setup

4.1 Project structure

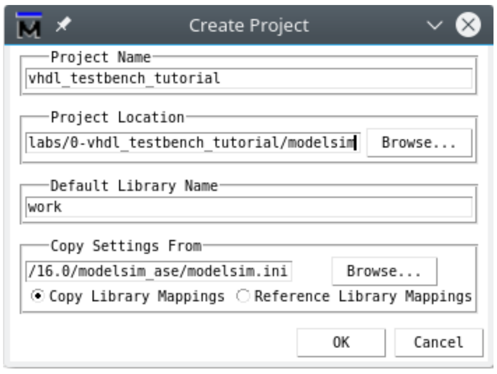
Download the provided template and extract it somewhere where the directory path does *not contain any spaces*. You should obtain the directory tree shown below:

```
vhdl_testbench_tutorial/
├─modelsim/
├─testbench/
│ ├─tb_adder_combinatorial.vhd
│ └─tb_adder_sequential.vhd
└─vhdl/
  ├─adder_combinatorial.vhd
  ├─adder_sequential.vhd
  └─full_adder.vhd
```

1. The **modelsim** folder will be used by ModelSim for its working directory.
2. The **testbench** folder contains 100% empty files in which we will write our testbenches for the combinatorial and sequential adders.
3. The **vhdl** folder contains the RTL designs of our 2 adders and of the 1-bit full-adder on which they are based.

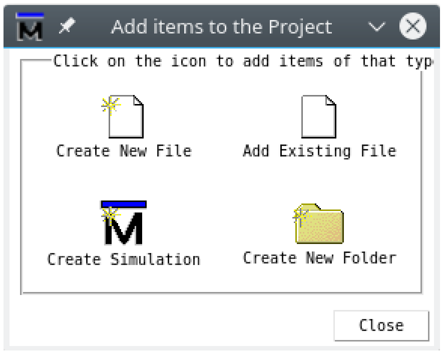
4.2 ModelSim setup

1. Launch ModelSim and create a new project with `File > New > Project...`
2. Name the project `vhdl_testbench_tutorial` and choose the **modelsim** folder from the extracted archive for the `Project Location` (Figure 3a)
3. Click on the `Add Existing File` button and add all files in the **vhdl** and **testbench** directories to the project (Figure 3b). You should obtain something similar to Figure 4.


Create Project

Project Name: vhdl_testbench_tutorial
Project Location: labs/0-vhdl_testbench_tutorial/modelsim
Default Library Name: work
Copy Settings From: /16.0/modelsim_ase/modelsim.ini
● Copy Library Mappings ○ Reference Library Mappings

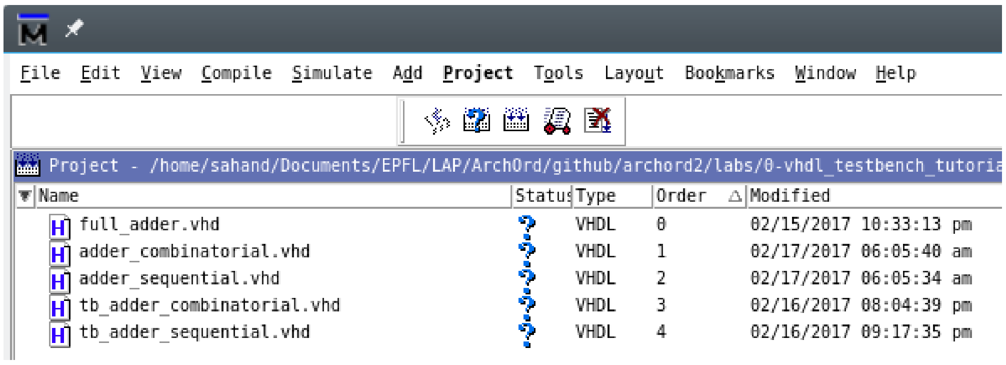
(a) Create project


Add items to the Project

Click on the icon to add items of that typ

(b) Add items files to the project

Figure 3: ModelSim project creation

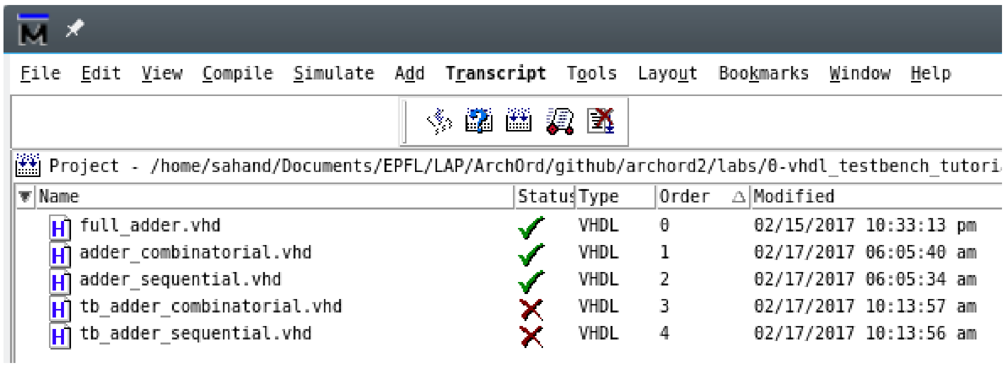

File Edit View Compile Simulate Add Project Tools Layout Bookmarks Window Help

Project - /home/sahand/Documents/EPFL/LAP/ArchOrd/github/archord2/labs/0-vhdl_testbench_tutoria

Name	Status	Type	Order	Modified
full_adder.vhd	?	VHDL	0	02/15/2017 10:33:13 pm
adder_combinatorial.vhd	?	VHDL	1	02/17/2017 06:05:40 am
adder_sequential.vhd	?	VHDL	2	02/17/2017 06:05:34 am
tb_adder_combinatorial.vhd	?	VHDL	3	02/16/2017 08:04:39 pm
tb_adder_sequential.vhd	?	VHDL	4	02/16/2017 09:17:35 pm

Figure 4: Files added to project

4. Click on the icon to compile all sources. The RTL files will compile successfully, but the testbenches will fail to compile (Figure 5). This is normal as the testbench files are empty, so ModelSim does not detect any entity to compile. If your output is not as in Figure 5, then it means you have added the various VHDL files in a different order from that shown in Figure 4 ("Order" column). You can solve this issue by clicking a few more times on the icon until ModelSim correctly determines the right compile order of the VHDL files based on their dependencies.


File Edit View Compile Simulate Add Transcript Tools Layout Bookmarks Window Help

Project - /home/sahand/Documents/EPFL/LAP/ArchOrd/github/archord2/labs/0-vhdl_testbench_tutori

Name	Status	Type	Order	Modified
full_adder.vhd	✓	VHDL	0	02/15/2017 10:33:13 pm
adder_combinatorial.vhd	✓	VHDL	1	02/17/2017 06:05:40 am
adder_sequential.vhd	✓	VHDL	2	02/17/2017 06:05:34 am
tb_adder_combinatorial.vhd	✗	VHDL	3	02/17/2017 10:13:57 am
tb_adder_sequential.vhd	✗	VHDL	4	02/17/2017 10:13:56 am

Figure 5: Compiling all files (empty testbenches)

The ModelSim project is now set up, so we can get to writing the testbenches.

5 Testing the combinatorial adder

We will now write, from scratch, the testbench for the combinatorial adder. All code presented in this section is to be written in `testbench/tb_adder_combinatorial.vhd`.

5.1 Minimum testbench

ModelSim complains about the fact the testbench is empty, so let's begin by filling it up with the minimum code needed to compile.

All VHDL files must have an *entity*, so we must write one for this testbench. An entity represents the input and output ports of a component. This makes sense if the VHDL file is describing some RTL design, but it doesn't make much sense in the case of a testbench. Indeed, a testbench is not a "component" which will be used in a design, but merely a simple VHDL file which is used to provide inputs to, and monitor the outputs from the DUT (the combinatorial adder in this case). So the testbench essentially has an *empty* entity.

```vhdl
entity tb_adder_combinatorial is
end tb_adder_combinatorial;
```

Listing 1: Minimum testbench (only empty entity)

ModelSim should now successfully compile this testbench.

We would like to actually do something useful with the testbench, so we need to add an *architecture*, as well as include the libraries containing various data types we are interested in manipulating (`std_logic`, `std_logic_vector`, `integer`, ...).

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_adder_combinatorial is
end tb_adder_combinatorial;

architecture test of tb_adder_combinatorial is
begin
end architecture test;
```

Listing 2: Add libraries & empty architecture

5.2 Instantiating the DUT

In order to test a component, we must have access to it. So the next step is to take the DUT, instantiate it, and wire it into the testbench. After this step, we would be able to interact with the DUT as if it were a component in the "design" (although we are not designing anything here, just testing).

Wiring the DUT into the testbench requires information about its entity. The combinatorial adder's entity is presented in Listing 3.

```vhdl
entity adder_combinatorial is
    generic(
        N_BITS : positive range 2 to positive'right  -- Operand size in bits
    );
    port(
        OP1 : in  std_logic_vector(N_BITS - 1 downto 0);   -- N-bit input.
        OP2 : in  std_logic_vector(N_BITS - 1 downto 0);   -- N-bit input.
        SUM : out std_logic_vector(N_BITS downto 0)   -- (N+1)-bit output.
    );
end entity adder_combinatorial;
```

Listing 3: `adder_combinatorial` entity

Note that the DUT is *generic*: at instantiation time, the component can be configured with a specific width depending on the value provided in its `N_BITS` parameter. Let's analyze this generic parameter, namely the line `N_BITS : positive range 2 to positive'right`.

- `N_BITS` is declared as being of type `positive`. In VHDL, `positive` is a constrained 32-bit integer type ranging from `1` to `2147483647`. However, the DUT is a *multi*-bit adder, so the author of the design provided a more constrained range than that of the standard `positive` type, namely a range with a lower bound of `2` instead of `1`.
- When constraining a type in VHDL, you must provide both a lower *and* an upper bound (the language does not allow you to just constrain one "side"). The `right` attribute of `positive` is used for this purpose here. When you write `positive'right`, you are telling the VHDL compiler to take the "rightmost" value of the `positive` type as the upper bound of `N_BITS`.

Now that we know what the DUT's entity looks like, we extract its details and update the testbench *architecture* as follows:

1. Create a *constant* for every *generic* parameter in the DUT's entity. You must also assign a value to the constant, as it will be used to instantiate the DUT with the configuration we want to test. We chose to implement a 4-bit adder.
2. Create a *signal* for every *port* in the DUT's entity.

Note that it is essential that you extract information from the DUT *as-is*, and that you do not modify any bounds in the corresponding constant and signal declarations. This is necessary to guarantee that the testbench is in sync with what the DUT expects as inputs (it doesn't make any sense to testbench a DUT when you are instantiating it with incorrect parameters).

Finally, we instantiate the DUT and wire it into the testbench with the signals created above.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_adder_combinatorial is
end tb_adder_combinatorial;

architecture test of tb_adder_combinatorial is

    -- adder_combinatorial GENERICS
    constant N_BITS : positive range 2 to positive'right := 4;

    -- adder_combinatorial PORTS
    signal OP1 : std_logic_vector(N_BITS - 1 downto 0);
    signal OP2 : std_logic_vector(N_BITS - 1 downto 0);
    signal SUM : std_logic_vector(N_BITS downto 0);

begin
```

```
    -- Instantiate DUT
    dut : entity work.adder_combinatorial
    generic map(N_BITS => N_BITS)
    port map(OP1 => OP1,
             OP2 => OP2,
             SUM => SUM);

end architecture test;
```

Listing 4: Instantiate DUT

5.3 Feeding inputs to the DUT

Now that the DUT is instantiated and wired into the testbench, we can start feeding it inputs to make the circuit actually do something.

5.3.1 Basic input feeding

Let's go over what we are trying to achieve to see how to write the part of the testbench responsible for feeding data to the DUT. We would like to feed test vectors in *sequence* to the unit, something like the execution below:

1. Supply inputs `OP1` and `OP2` from vector 1.
2. Check output `SUM` from vector 1.
3. Supply inputs `OP1` and `OP2` from vector 2.
4. Check output `SUM` from vector 2.
5. ...

In VHDL, recall that processes execute in *parallel* among each other, but the statements within each process execute in *sequence*. This observation is key to writing the input-feeding part of the testbench: we can provide all inputs in a single process, separated by time intervals, and each vector will be executed one after the other until there no longer are any statements left in the process. It may be complicated to see why this works in words, so let's look at a concrete example to see how this all works.

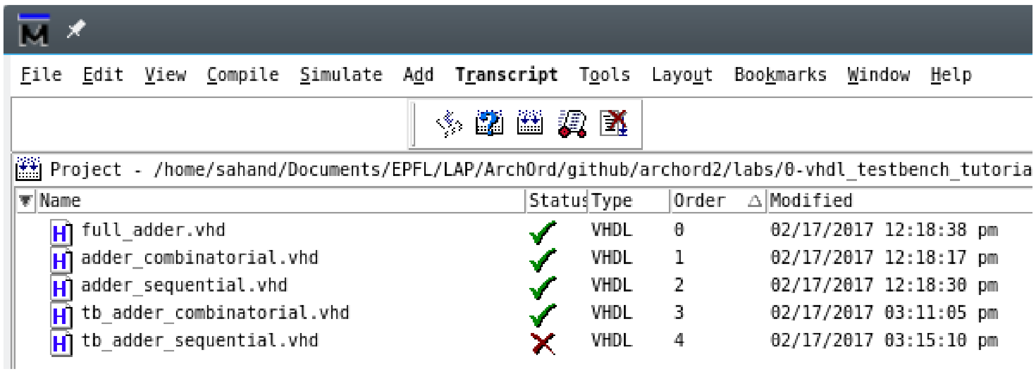
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_adder_combinatorial is
end tb_adder_combinatorial;

architecture test of tb_adder_combinatorial is

    -- "Time" that will elapse between test vectors we submit to the component.
    constant TIME_DELTA : time := 100 ns;

    -- adder_combinatorial GENERICS
    constant N_BITS : positive range 2 to positive'right := 4;

    -- adder_combinatorial PORTS
    signal OP1 : std_logic_vector(N_BITS - 1 downto 0);
    signal OP2 : std_logic_vector(N_BITS - 1 downto 0);
    signal SUM : std_logic_vector(N_BITS downto 0);

begin

    -- Instantiate DUT
    dut : entity work.adder_combinatorial
    generic map(N_BITS => N_BITS)
    port map(OP1 => OP1,
             OP2 => OP2,
             SUM => SUM);

    -- Test adder_combinatorial
    simulation : process
    begin
        -- Assign values to circuit inputs.
        OP1 <= "0001"; -- 1
        OP2 <= "0101"; -- 5

        -- OP1 and OP2 are NOT yet assigned. We have to wait for some time
        -- for the simulator to "propagate" their values. Any infinitesimal
        -- period would work here since we are testing a combinatorial
        -- circuit.
        wait for TIME_DELTA;

        -- Assign values to circuit inputs.
        OP1 <= "0011"; -- 3
        OP2 <= "0010"; -- 2

        -- OP1 and OP2 are NOT yet assigned. We have to wait for some time
        -- for the simulator to "propagate" their values. Any infinitesimal
        -- period would work here since we are testing a combinatorial
        -- circuit.
        wait for TIME_DELTA;
    end process simulation;

end architecture test;
```

Listing 5: Simulation process (basic)

Notice that the process does *not* have a sensitivity list. In VHDL, a process must either have a sensitivity list, or a `wait` statement. A testbench is not an RTL design, so it is not “sensitive” to any input signals. A testbench instead “controls” time to supply inputs at appropriate points, so the process contains 2 `wait` statements to propagate the 2 input vectors instead of a sensitivity list.

Let’s simulate our current testbench and see what we get.

1. Click on the icon to compile all sources. You should get something similar to Figure 6. Recall that the testbench for the sequential adder is not written yet, so it is normal it fails to compile.

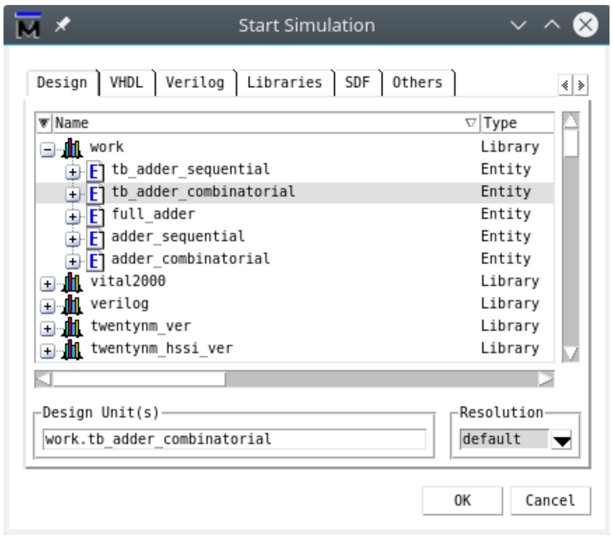

File Edit View Compile Simulate Add Transcript Tools Layout Bookmarks Window Help

Project - /home/sahand/Documents/EPFL/LAP/ArchOrd/github/archord2/labs/0-vhdl_testbench_tutoria

Name	Status	Type	Order	Modified
full_adder.vhd	✓	VHDL	0	02/17/2017 12:18:38 pm
adder_combinatorial.vhd	✓	VHDL	1	02/17/2017 12:18:17 pm
adder_sequential.vhd	✓	VHDL	2	02/17/2017 12:18:30 pm
tb_adder_combinatorial.vhd	✓	VHDL	3	02/17/2017 03:11:05 pm
tb_adder_sequential.vhd	✗	VHDL	4	02/17/2017 03:15:10 pm

Figure 6: Compiling all testbenches

2. Go to `Simulate > Start Simulation...`
3. Select `work > tb_adder_combinatorial`, then press `OK` (Figure 7).



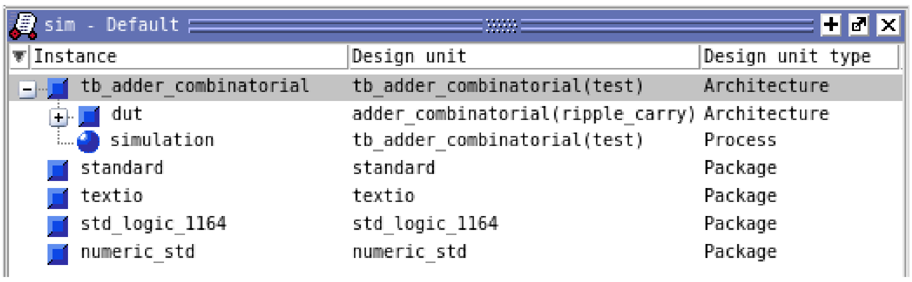
Name	Type
work	Library
tb_adder_sequential	Entity
tb_adder_combinatorial	Entity
full_adder	Entity
adder_sequential	Entity
adder_combinatorial	Entity
vital2000	Library
verilog	Library
twentynm_ver	Library
twentynm_hssi_ver	Library

Design Unit(s): work.tb_adder_combinatorial

Resolution: default

Figure 7: Start simulation

4. In the `sim` tab, click on `tb_adder_combinatorial` (Figure 8).

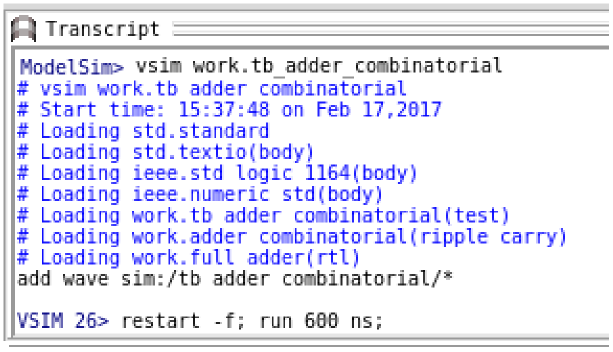


Instance	Design unit	Design unit type
tb_adder_combinatorial	tb_adder_combinatorial(test)	Architecture
dut	adder_combinatorial(ripple_carry)	Architecture
simulation	tb_adder_combinatorial(test)	Process
standard	standard	Package
textio	textio	Package
std_logic_1164	std_logic_1164	Package
numeric_std	numeric_std	Package

Figure 8: Simulation tab

5. Go to `Add > To Wave > All items in region`.

6. Finally, in the simulation transcript, type `restart -f; run 600 ns;` (Figure 9). A waveform window should appear with the result of the simulation (Figure 10).



```
Transcript
ModelSim> vsim work.tb_adder_combinatorial
# vsim work.tb adder combinatorial
# Start time: 15:37:48 on Feb 17,2017
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std logic 1164(body)
# Loading ieee.numeric std(body)
# Loading work.tb adder combinatorial(test)
# Loading work.adder combinatorial(ripple carry)
# Loading work.full adder(rtl)
add wave sim:/tb adder combinatorial/*

VSIM 26> restart -f; run 600 ns;
```

Figure 9: Transcript

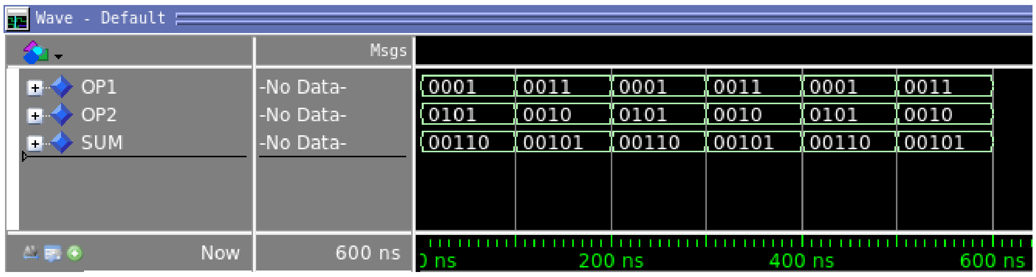


Figure 10: Waveform (repetitive input feeding)

The results look correct, i.e. $1+5=6$ and $3+2=5$, so at least the circuit is behaving correctly with these 2 input vectors. However, notice that the simulation is "looping" and is applying the two input vectors 3 times. Recall how processes work in VHDL. A process executes all its statements sequentially, then restarts. In the testbench, the simulation process uses 2 `wait` statements for a total of $200\,\text{ns}$ of simulation time. Since we asked for a $600\,\text{ns}$ simulation, the process had enough time to restart 3 times.

5.3.2 Automatic simulation termination

In Figure 9, we saw that we had to manually provide the simulation interval, $600\,\text{ns}$, to ModelSim. Manually selecting the time interval is error-prone, as one may accidentally supply an interval too short for all test vectors to pass through the DUT, or too long and unnecessarily re-executing test vectors which have already passed through the DUT.

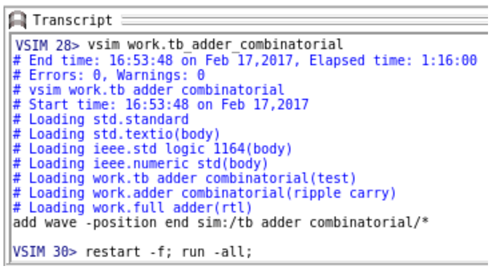
It would be nice if the simulator could let the simulation run for as long as needed, i.e. until all test vectors have passed through the DUT, then automatically halt the simulation. ModelSim has a command specifically for this purpose: `run -all`. Instead of specifying a time interval, the `-all` flag instructs ModelSim to continue simulating as long as events are scheduled for simulation. Therefore, one way to halt the simulation is to cause processes *not* to restart once they have finished all their statements. This can be achieved by placing an *indefinite* `wait` statement at the end of the simulation process, after all the test vectors have passed through the DUT.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```vhdl
entity tb_adder_combinatorial is
end tb_adder_combinatorial;

architecture test of tb_adder_combinatorial is

    -- "Time" that will elapse between test vectors we submit to the component.
    constant TIME_DELTA : time := 100 ns;

    -- adder_combinatorial GENERICS
    constant N_BITS : positive range 2 to positive'right := 4;

    -- adder_combinatorial PORTS
    signal OP1 : std_logic_vector(N_BITS - 1 downto 0);
    signal OP2 : std_logic_vector(N_BITS - 1 downto 0);
    signal SUM : std_logic_vector(N_BITS downto 0);

begin

    -- Instantiate DUT
    dut : entity work.adder_combinatorial
    generic map(N_BITS => N_BITS)
    port map(OP1 => OP1,
             OP2 => OP2,
             SUM => SUM);

    -- Test adder_combinatorial
    simulation : process
    begin
        -- Assign values to circuit inputs.
        OP1 <= "0001"; -- 1
        OP2 <= "0101"; -- 5

        -- OP1 and OP2 are NOT yet assigned. We have to wait for some time
        -- for the simulator to "propagate" their values. Any infinitesimal
        -- period would work here since we are testing a combinatorial
        -- circuit.
        wait for TIME_DELTA;

        -- Assign values to circuit inputs.
        OP1 <= "0011"; -- 3
        OP2 <= "0010"; -- 2

        -- OP1 and OP2 are NOT yet assigned. We have to wait for some time
        -- for the simulator to "propagate" their values. Any infinitesimal
        -- period would work here since we are testing a combinatorial
        -- circuit.
        wait for TIME_DELTA;

        -- Make this process wait indefinitely (it will never re-execute from
        -- its beginning again).
        wait;
    end process simulation;
end architecture test;
```

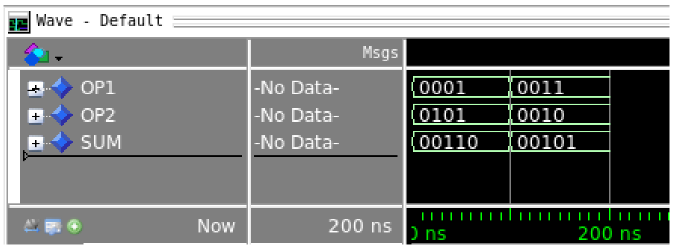
Listing 6: Indefinite `wait` statement added to simulation process

If we now recompile and relaunch the simulation, we should obtain the output shown in Figure 11b.



```
Transcript
VSIM 28> vsim work.tb_adder_combinatorial
# End time: 16:53:48 on Feb 17,2017, Elapsed time: 1:16:00
# Errors: 0, Warnings: 0
# vsim work.tb adder combinatorial
# Start time: 16:53:48 on Feb 17,2017
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std logic 1164(body)
# Loading ieee.numeric std(body)
# Loading work.tb adder combinatorial(test)
# Loading work.adder combinatorial(ripple carry)
# Loading work.full adder(rtl)
add wave -position end sim:/tb adder combinatorial/*

VSIM 30> restart -f; run -all;
```

(a) Transcript



(b) Waveform

Figure 11: Automatic simulation termination

5.3.3 Avoiding code duplication

Listing 6 is quite simple, but we can already see that there is a fair amount of code duplication going on. Every additional test vector requires copy-pasting the code responsible for the operand assignments and the `wait` statement. Although not exhaustive, we want to test the DUT with a potentially large number of test vectors to have more confidence in its correctness. However, a large number of test vectors would cause a huge increase in the simulation process' code size, therefore making it hard to read.

We handle this issue by refactoring the code used to feed a test vector into a *procedure* called `check_add`. Notice the type of the input arguments provided to the `check_add` procedure. We provide the operands in `natural` format instead of in `std_logic_vector`. This makes it easy to create & add test vectors, and also makes the code more readable.

As a side-effect, we increased the count of test vectors from 2 to 10 for illustration purposes.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_adder_combinatorial is
end tb_adder_combinatorial;

architecture test of tb_adder_combinatorial is

    -- "Time" that will elapse between test vectors we submit to the component.
    constant TIME_DELTA : time := 100 ns;

    -- adder_combinatorial GENERICS
    constant N_BITS : positive range 2 to positive'right := 4;

    -- adder_combinatorial PORTS
    signal OP1 : std_logic_vector(N_BITS - 1 downto 0);
    signal OP2 : std_logic_vector(N_BITS - 1 downto 0);
    signal SUM : std_logic_vector(N_BITS downto 0);

begin

    -- Instantiate DUT
    dut : entity work.adder_combinatorial
    generic map(N_BITS => N_BITS)
    port map(OP1 => OP1,
             OP2 => OP2,
             SUM => SUM);

    -- Test adder_combinatorial
    simulation : process

        procedure check_add(constant in1 : in natural;
                            constant in2 : in natural) is
        begin
            -- Assign values to circuit inputs.
            OP1 <= std_logic_vector(to_unsigned(in1, OP1'length));
            OP2 <= std_logic_vector(to_unsigned(in2, OP2'length));

            -- OP1 and OP2 are NOT yet assigned. We have to wait for some time
            -- for the simulator to "propagate" their values. Any infinitesimal
            -- period would work here since we are testing a combinatorial
            -- circuit.
            wait for TIME_DELTA;
        end procedure check_add;

    begin

        -- Check test vectors
        check_add(12, 8);
        check_add(10, 6);
        check_add(4, 1);
        check_add(11, 7);
        check_add(10, 13);
        check_add(8, 7);
        check_add(1, 9);
        check_add(7, 3);
        check_add(1, 4);
        check_add(8, 0);

        -- Make this process wait indefinitely (it will never re-execute from
        -- its beginning again).
        wait;
    end process simulation;

end architecture test;
```

Listing 7: Refactored test vector feeding code into a *procedure* called `check_add`

Recompiling and relaunching the simulation (`restart -f; run -all;`) results in a waveform with many more test vectors being shown (Figure 12).

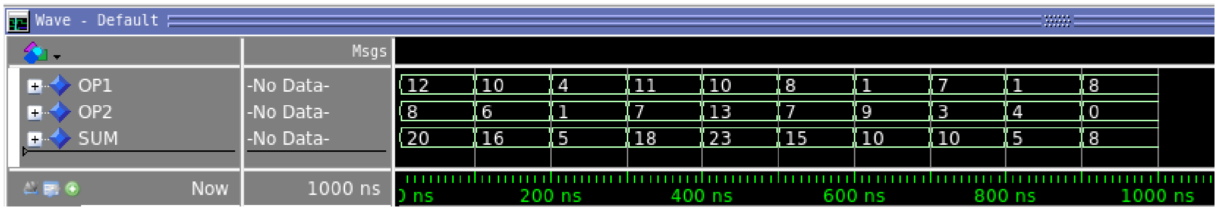

Wave - Default
Msgs
OP1 -No Data- 12 10 4 11 10 8 1 7 1 8
OP2 -No Data- 8 6 1 7 13 7 9 3 4 0
SUM -No Data- 20 16 5 18 23 15 10 10 5 8
Now 1000 ns
) ns 200 ns 400 ns 600 ns 800 ns 1000 ns

Figure 12: Waveform (many test vectors). Note that, by default, ModelSim displays signals in binary on a waveform. You can modify the display radix of various signals by selecting the signals and going to `Wave > Format > Radix`. The `Unsigned` radix is used in this waveform.

5.3.4 Self-checking testbench

Listing 7 is easy to read and to modify, but there is one last thing which is quite cumbersome: verifying the DUT's outputs. With the current system, a human must visually check the DUT's outputs for correctness on the simulation waveform. Although not a difficult task, it is quite annoying and error-prone to have someone manually verify a waveform containing hundreds, perhaps thousands, of test vectors.

It would be great if the user could modify the definition of a test vector to not only include DUT inputs, but also its *expected* output. We could then modify the testbench such that it automatically compares the output of the DUT with the expected output provided in the test vector. Nobody would then have to manually check the results for correctness. This type of testbench is commonly called a *self-checking* testbench. We apply the idea in Listing 8.

Notice how we use an assertion to compare the DUT output with the expected output provided in the test vector. The VHDL `assert` statement is followed by:

1. A `report` statement and a string describing the error.
2. A `severity` statement and a severity level. VHDL supports 4 severity levels (`note`, `warning`, `error`, `failure`) and simulators can be configured to react to each severity level in different ways. We do not use this feature, but it is good to be aware of.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_adder_combinatorial is
end tb_adder_combinatorial;

architecture test of tb_adder_combinatorial is

    -- "Time" that will elapse between test vectors we submit to the component.
    constant TIME_DELTA : time := 100 ns;

    -- adder_combinatorial GENERICS
    constant N_BITS : positive range 2 to positive'right := 4;

    -- adder_combinatorial PORTS
    signal OP1 : std_logic_vector(N_BITS - 1 downto 0);
    signal OP2 : std_logic_vector(N_BITS - 1 downto 0);
    signal SUM : std_logic_vector(N_BITS downto 0);
```

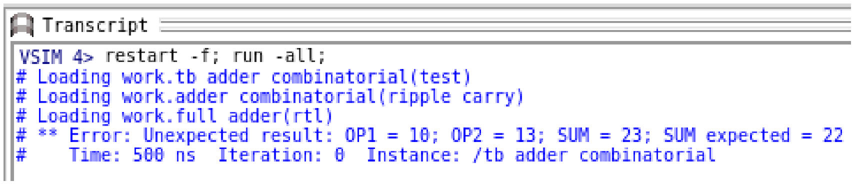
```
begin
    -- Instantiate DUT
    dut : entity work.adder_combinatorial
    generic map(N_BITS => N_BITS)
    port map(OP1 => OP1,
             OP2 => OP2,
             SUM => SUM);

    -- Test adder_combinatorial
    simulation : process

        procedure check_add(constant in1          : in natural;
                            constant in2          : in natural;
                            constant res_expected : in natural) is
            variable res : natural;
        begin
            -- Assign values to circuit inputs.
            OP1 <= std_logic_vector(to_unsigned(in1, OP1'length));
            OP2 <= std_logic_vector(to_unsigned(in2, OP2'length));

            -- OP1 and OP2 are NOT yet assigned. We have to wait for some time
            -- for the simulator to "propagate" their values. Any infinitesimal
            -- period would work here since we are testing a combinatorial
            -- circuit.
            wait for TIME_DELTA;

            -- Check output against expected result.
            res := to_integer(unsigned(SUM));
            assert res = res_expected
            report "Unexpected result: " &
                   "OP1 = " & integer'image(in1) & "; " &
                   "OP2 = " & integer'image(in2) & "; " &
                   "SUM = " & integer'image(res) & "; " &
                   "SUM_expected = " & integer'image(res_expected)
            severity error;
        end procedure check_add;

    begin

        -- Check test vectors against expected outputs
        check_add(12, 8, 20);
        check_add(10, 6, 16);
        check_add(4, 1, 5);
        check_add(11, 7, 18);
        check_add(10, 13, 23);
        check_add(8, 7, 15);
        check_add(1, 9, 10);
        check_add(7, 3, 10);
        check_add(1, 4, 5);
        check_add(8, 0, 8);

        -- Make this process wait indefinitely (it will never re-execute from
        -- its beginning again).
        wait;
    end process simulation;

end architecture test;
```

Listing 8: Self-checking testbench: updated `check_add` procedure to automatically compare DUT output against expected output provided in test vector

With this new updated code, any error detected in the simulation will result in an entry being written to the simulation transcript. Let's try to see what the transcript would look like when an error is detected:

The RTL code of the combinatorial adder we provide in the template is correct, so it is impossible to obtain an "incorrect" result to trigger the assertion failure. However, for testing purposes, we can artificially supply an incorrect `res_expected` argument to the `check_add` procedure to trigger the assertion failure and see what the description on the transcript looks like.

For example, let's change the line `check_add(10, 13, 23)` with `check_add(10, 13, 22)`. If you now recompile and run the testbench, you should get the following output on the simulation transcript:



```
Transcript
VSIM 4> restart -f; run -all;
# Loading work.tb adder combinatorial(test)
# Loading work.adder combinatorial(ripple carry)
# Loading work.full adder(rtl)
# ** Error: Unexpected result: OP1 = 10; OP2 = 13; SUM = 23; SUM expected = 22
#    Time: 500 ns  Iteration: 0  Instance: /tb adder combinatorial
```

Figure 13: Assertion failure in ModelSim transcript

You can then double-click on `Time: 500ns` in the simulation transcript to make ModelSim automatically display a cursor at the specified location in the waveform. You should get something like this:

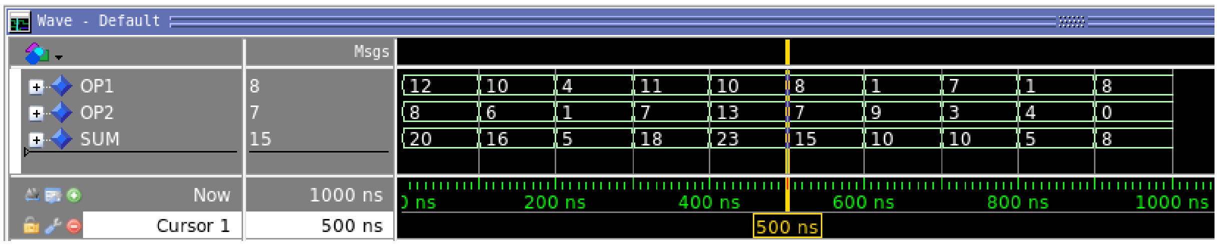


Figure 14: Cursor displayed at error location `500 ns`

Of course, the actual result obtained in the waveform is correct as we have artificially triggered the assertion by providing an incorrect expected result here.

6 Testing the sequential adder

We will now write, from scratch, the testbench for the sequential adder. All code presented in this section is to be written in `testbench/tb_adder_sequential.vhd`.

6.1 Instantiating the DUT

Let's start by instantiating the DUT and wiring it into the testbench. We use the same algorithm described in Section 5.2 to obtain Listing 9.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_adder_sequential is
end tb_adder_sequential;

architecture test of tb_adder_sequential is

    -- adder_sequential GENERICS
    constant N_BITS : positive range 2 to positive'right := 4;

    -- adder_sequential PORTS
    signal CLK   : std_logic;
    signal RST   : std_logic;
    signal START : std_logic;
    signal OP1   : std_logic_vector(N_BITS - 1 downto 0);
    signal OP2   : std_logic_vector(N_BITS - 1 downto 0);
    signal SUM   : std_logic_vector(N_BITS downto 0);
    signal DONE  : std_logic;

begin

    -- Instantiate DUT
    dut : entity work.adder_sequential
    generic map(N_BITS => N_BITS)
    port map(CLK   => CLK,
             RST   => RST,
             START => START,
             OP1   => OP1,
             OP2   => OP2,
             SUM   => SUM,
             DONE  => DONE);

end architecture test;
```

Listing 9: Instantiate DUT

6.2 Generating a clock signal

The sequential adder is a synchronous component as it is sensitive to a clock. We saw in Section 5.3.1 that, for simulation purposes, one can assign values to signals by using a VHDL process. We do the same in Listing 10 specifically for generating the `CLK` input of the DUT.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_adder_sequential is
end tb_adder_sequential;

architecture test of tb_adder_sequential is

    constant CLK_PERIOD : time := 100 ns;

    -- adder_sequential GENERICS
    constant N_BITS : positive range 2 to positive'right := 4;

    -- adder_sequential PORTS
    signal CLK   : std_logic;
    signal RST   : std_logic;
    signal START : std_logic;
    signal OP1   : std_logic_vector(N_BITS - 1 downto 0);
    signal OP2   : std_logic_vector(N_BITS - 1 downto 0);
    signal SUM   : std_logic_vector(N_BITS downto 0);
    signal DONE  : std_logic;

begin

    -- Instantiate DUT
    dut : entity work.adder_sequential
    generic map(N_BITS => N_BITS)
    port map(CLK   => CLK,
             RST   => RST,
             START => START,
             OP1   => OP1,
             OP2   => OP2,
             SUM   => SUM,
             DONE  => DONE);

    -- Generate CLK signal
    clk_generation : process
    begin
        CLK <= '1';
        wait for CLK_PERIOD / 2;
        CLK <= '0';
        wait for CLK_PERIOD / 2;
    end process clk_generation;

end architecture test;
```

Listing 10: Add process for generating `CLK` signal

Note the absence of any *indefinite* `wait` statement in the `clk_generation` process. It is important that one *not* use such an indefinite `wait` statement, otherwise the process would only generate a single clock pulse before halting instead of continuously generating clock pulses each time the process restarts.

A consequence of this design is that one cannot use the `run -all` command in ModelSim, but must instead explicitly provide a simulation duration such as `run 1000 ns`.

6.3 Feeding inputs to the DUT

6.3.1 Default DUT inputs

In Section 5.3.1, we used a *single* process for feeding inputs to the *combinatorial* adder. Could we envision doing the same thing for the *sequential* adder (i.e. use a *single* process for generating the clock signal and the non-clock signals)?

A clock signal is *periodic*, whereas non-clock signals are generally *non-periodic*. Recall the execution model of VHDL processes: processes execute in *parallel* among each other, but the statements within each process execute in *sequence*. Given this execution model, there is no way to describe the periodic behaviour of a clock parallely to the sequential behaviour of the other input signals in a *single* process.

However, it is possible to do so with *multiple* processes. One process can periodically generate a clock signal, while another process can sequentially generate the non-clock signals which are to be fed to the DUT. The 2 processes execute in parallel, so we can correctly model the periodic and non-periodic behaviour of our 2 signal categories (clock and non-clock).

Therefore, we update our testbench with a second process called `simulation` which is responsible for generating the non-clock inputs of the DUT. Our first attempt is shown in Listing 11 where the testbench just assigns default values to the DUT's non-clock signals.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_adder_sequential is
end tb_adder_sequential;

architecture test of tb_adder_sequential is

    constant CLK_PERIOD : time := 100 ns;

    -- adder_sequential GENERICS
    constant N_BITS : positive range 2 to positive'right := 4;

    -- adder_sequential PORTS
    signal CLK   : std_logic;
    signal RST   : std_logic;
    signal START : std_logic;
    signal OP1   : std_logic_vector(N_BITS - 1 downto 0);
    signal OP2   : std_logic_vector(N_BITS - 1 downto 0);
    signal SUM   : std_logic_vector(N_BITS downto 0);
    signal DONE  : std_logic;

begin

    -- Instantiate DUT
    dut : entity work.adder_sequential
    generic map(N_BITS => N_BITS)
    port map(CLK   => CLK,
             RST   => RST,
                START => START,
                OP1   => OP1,
                OP2   => OP2,
                SUM   => SUM,
                DONE  => DONE);

    -- Generate CLK signal
    clk_generation : process
    begin
        CLK <= '1';
        wait for CLK_PERIOD / 2;
        CLK <= '0';
        wait for CLK_PERIOD / 2;
    end process clk_generation;

    -- Test adder_sequential
    simulation : process
    begin

        -- Default values
        OP1   <= (others => '0');
        OP2   <= (others => '0');
        RST   <= '0';
        START <= '0';
        wait for CLK_PERIOD;
    end process simulation;

end architecture test;
```

Listing 11: Add `simulation` process for generating non-clock signals to the DUT. This first version only assigns default values to the non-clock signals.

6.3.2 Resetting the DUT

The DUT is a synchronous circuit and has an *asynchronous* reset input, `RST`. When testing a component, it is a good practice to supply it with a reset pulse before feeding it any other inputs. This guarantees the DUT is in a valid state before we do anything. We update the `simulation` process with an `async_reset` *procedure* for this.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_adder_sequential is
end tb_adder_sequential;

architecture test of tb_adder_sequential is

    constant CLK_PERIOD : time := 100 ns;

    -- adder_sequential GENERICS
    constant N_BITS : positive range 2 to positive'right := 4;

    -- adder_sequential PORTS
    signal CLK   : std_logic;
    signal RST   : std_logic;
    signal START : std_logic;
    signal OP1   : std_logic_vector(N_BITS - 1 downto 0);
    signal OP2   : std_logic_vector(N_BITS - 1 downto 0);
    signal SUM   : std_logic_vector(N_BITS downto 0);
    signal DONE  : std_logic;

begin

    -- Instantiate DUT
    dut : entity work.adder_sequential
    generic map(N_BITS => N_BITS)
    port map(CLK   => CLK,
             RST   => RST,
             START => START,
             OP1   => OP1,
             OP2   => OP2,
             SUM   => SUM,
             DONE  => DONE);

    -- Generate CLK signal
    clk_generation : process
    begin
        CLK <= '1';
        wait for CLK_PERIOD / 2;
        CLK <= '0';
        wait for CLK_PERIOD / 2;
    end process clk_generation;

    -- Test adder_sequential
    simulation : process

        procedure async_reset is
        begin
            wait until rising_edge(CLK);
            wait for CLK_PERIOD / 4;
            RST <= '1';

            wait for CLK_PERIOD / 2;
            RST <= '0';
        end procedure async_reset;

    begin

        -- Default values
        OP1   <= (others => '0');
        OP2   <= (others => '0');
        RST   <= '0';
        START <= '0';
        wait for CLK_PERIOD;

        -- Reset the circuit.
        async_reset;
    end process simulation;

end architecture test;
```

Listing 12: Add *asynchronous* reset

6.3.3 Self-checking test vectors

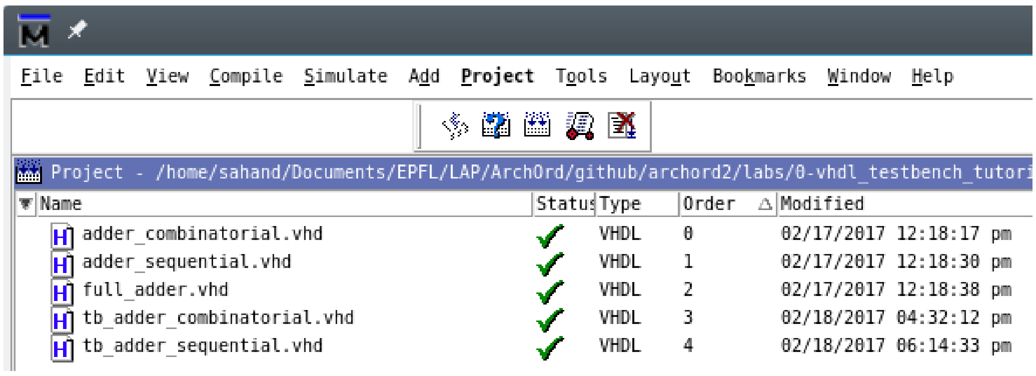
Now that the DUT is in a valid initial state, we can start feeding it test vectors and automatically check if the output is correct. We create a `check_add` *procedure* for this purpose, similarly as for the combinatorial adder.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_adder_sequential is
end tb_adder_sequential;

architecture test of tb_adder_sequential is

    constant CLK_PERIOD : time := 100 ns;

    -- adder_sequential GENERICS
    constant N_BITS : positive range 2 to positive'right := 4;

    -- adder_sequential PORTS
    signal CLK   : std_logic;
    signal RST   : std_logic;
    signal START : std_logic;
    signal OP1   : std_logic_vector(N_BITS - 1 downto 0);
    signal OP2   : std_logic_vector(N_BITS - 1 downto 0);
    signal SUM   : std_logic_vector(N_BITS downto 0);
    signal DONE  : std_logic;

begin

    -- Instantiate DUT
    dut : entity work.adder_sequential
    generic map(N_BITS => N_BITS)
    port map(CLK   => CLK,
             RST   => RST,
             START => START,
             OP1   => OP1,
             OP2   => OP2,
             SUM   => SUM,
             DONE  => DONE);

    -- Generate CLK signal
    clk_generation : process
    begin
        CLK <= '1';
        wait for CLK_PERIOD / 2;
        CLK <= '0';
        wait for CLK_PERIOD / 2;
    end process clk_generation;

    -- Test adder_sequential
    simulation : process

        procedure async_reset is
        begin
            wait until rising_edge(CLK);
            wait for CLK_PERIOD / 4;
            RST <= '1';

            wait for CLK_PERIOD / 2;
            RST <= '0';
        end procedure async_reset;

        procedure check_add(constant in1          : in natural;
                            constant in2          : in natural;
                            constant res_expected : in natural) is
            variable res : natural;
        begin
            -- Our circuit is sensitive to the rising edge of the CLK, so we
            -- need to be sure to assign signal values such that they are stable
            -- at the next rising edge of the CLK.
            wait until rising_edge(CLK);

            -- Assign values to circuit inputs.
            OP1   <= std_logic_vector(to_unsigned(in1, OP1'length));
            OP2   <= std_logic_vector(to_unsigned(in2, OP2'length));
            START <= '1';

            -- OP1, OP2 and START are NOT yet assigned. We have to wait for some
            -- time for the simulator to "propagate" their values. Any
            -- infinitesimal period would work for the simulator to "propagate"
            -- the values. However, our circuit is a sequential circuit
            -- sensitive to the rising edge of CLK, so we need to hold our
            -- signal assignments until the next rising edge of CLK so the
            -- circuit can see them.
            wait until rising_edge(CLK);

            -- Remove values from circuit inputs. The circuit works with a PULSE
            -- on its START input, which means that data on the inputs only
            -- needs to be valid when START is high.
            OP1   <= (others => '0');
            OP2   <= (others => '0');
            START <= '0';

            -- The circuit informs us it has finished by asserting DONE, so we
            -- can wait until we receive the signal before proceeding. DONE is
            -- asserted at the rising edge of CLK, so we (the test system) can
            -- sample the data and check its correctness.
            wait until DONE = '1';

            -- Check output against expected result.
            res := to_integer(unsigned(SUM));
            assert res = res_expected
            report "Unexpected result: " &
                   "OP1 = " & integer'image(in1) & "; " &
                   "OP2 = " & integer'image(in2) & "; " &
                   "SUM = " & integer'image(res) & "; " &
                   "SUM_expected = " & integer'image(res_expected)
            severity error;

            -- Wait for the circuit to go back into the IDLE state.
            wait until DONE = '0';
        end procedure check_add;

    begin

        -- Default values
        OP1   <= (others => '0');
        OP2   <= (others => '0');
        RST   <= '0';
        START <= '0';
        wait for CLK_PERIOD;

        -- Reset the circuit.
        async_reset;

        -- Check test vectors against expected outputs
        check_add(12, 8, 20);
    end process simulation;

end architecture test;
```

Listing 13: Add test vectors

Let's simulate our current testbench and see what we get.

1. Click on the icon to compile all sources. You should get something similar to Figure 15. All files should successfully compile as they now all contain valid VHDL.

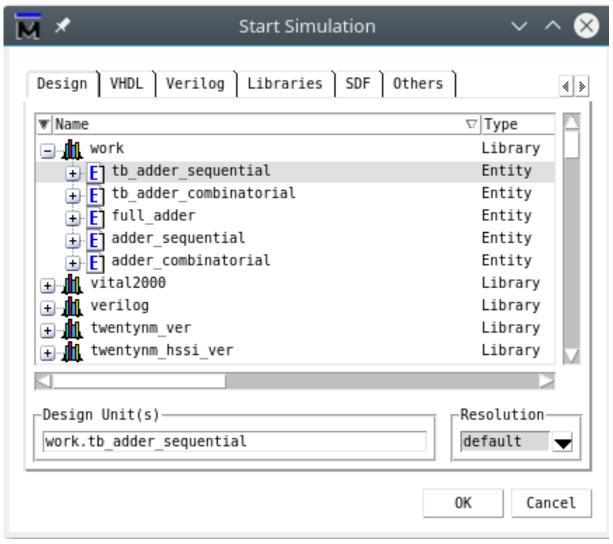

File Edit View Compile Simulate Add Project Tools Layout Bookmarks Window Help

```
Project - /home/sahand/Documents/EPFL/LAP/ArchOrd/github/archord2/labs/0-vhdl_testbench_tutori
```

Name	Status	Type	Order	Modified
adder_combinatorial.vhd	✓	VHDL	0	02/17/2017 12:18:17 pm
adder_sequential.vhd	✓	VHDL	1	02/17/2017 12:18:30 pm
full_adder.vhd	✓	VHDL	2	02/17/2017 12:18:38 pm
tb_adder_combinatorial.vhd	✓	VHDL	3	02/18/2017 04:32:12 pm
tb_adder_sequential.vhd	✓	VHDL	4	02/18/2017 06:14:33 pm

Figure 15: Compiling all testbenches

2. Go to `Simulate > Start Simulation...`
3. Select `work > tb_adder_sequential`, then press `OK` (Figure 16).

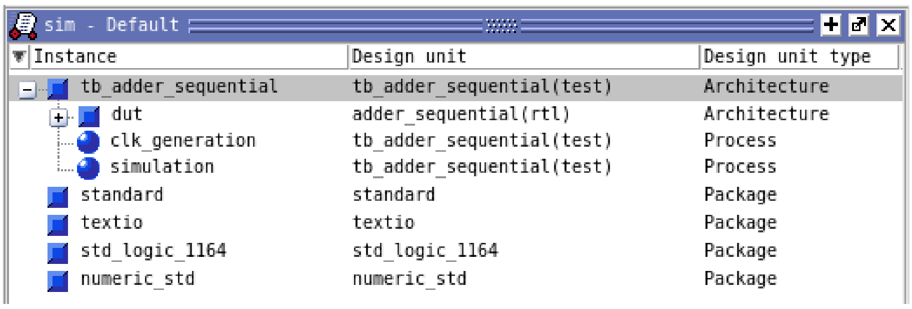


Name	Type
work	Library
tb_adder_sequential	Entity
tb_adder_combinatorial	Entity
full_adder	Entity
adder_sequential	Entity
adder_combinatorial	Entity
vital2000	Library
verilog	Library
twentynm_ver	Library
twentynm_hssi_ver	Library

Design Unit(s): work.tb_adder_sequential

Figure 16: Start simulation

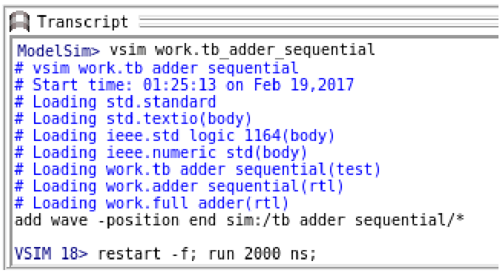
4. In the `sim` tab, click on `tb_adder_sequential` (Figure 17).



Instance	Design unit	Design unit type
tb_adder_sequential	tb_adder_sequential(test)	Architecture
dut	adder_sequential(rtl)	Architecture
clk_generation	tb_adder_sequential(test)	Process
simulation	tb_adder_sequential(test)	Process
standard	standard	Package
textio	textio	Package
std_logic_1164	std_logic_1164	Package
numeric_std	numeric_std	Package

Figure 17: Simulation tab

5. Go to `Add > To Wave > All items in region`.
6. Finally, in the simulation transcript, type `restart -f; run 2000 ns;` (Figure 18). A waveform window should appear with the result of the simulation (Figure 19).



```
ModelSim> vsim work.tb_adder_sequential
# vsim work.tb adder sequential
# Start time: 01:25:13 on Feb 19,2017
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std logic 1164(body)
# Loading ieee.numeric std(body)
# Loading work.tb adder sequential(test)
# Loading work.adder sequential(rtl)
# Loading work.full adder(rtl)
add wave -position end sim:/tb adder sequential/*

VSIM 18> restart -f; run 2000 ns;
```

Figure 18: Transcript

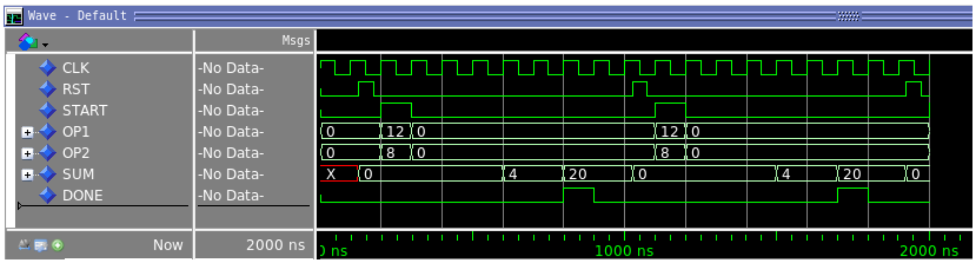


Figure 19: Waveform (repetitive input feeding)

The results look correct:

(a) The circuit seems to be reset as the `SUM` output changes as soon as `RST` is high (recall it is an *asynchronous* reset, so various registers should be reset as soon as the reset signal is high, and *not* at the next rising edge of the clock signal).

(b) The test vector $12 + 8 = 20$ is correct.

However, we suffer from the "loopy" execution of the `simulation` process again, as was previously seen for the combinatorial adder. We will address this issue next so we can take advantage of ModelSim's practical automatic simulation termination command, `run -all`.

6.3.4 Automatic simulation termination

For the combinatorial adder, we saw how to stop a process from re-executing by means of an *indefinite* `wait` statement. We would like to use the same technique to automatically halt the simulation with ModelSim's `run -all` command once all test vectors have gone through the DUT. Unfortunately, unlike for the combinatorial adder, simply adding an indefinite `wait` statement to the `simulation` process will not be sufficient for the simulation to automatically terminate.

Recall the semantics of the `run -all` command: ModelSim terminates the simulation when there no longer exists any scheduled signal assignment in the testbench. If we add an indefinite `wait` statement to the `simulation` process, then this process will indeed never re-execute, causing our test vectors to only go through the DUT once (which is what we are looking for). However, the `clk_generation` process still keeps restarting, so there always will be scheduled signal assignments in the testbench. The end result is that the `run -all` command will never stop the simulation!

To address this issue, we would need the `clk_generation` process to also stop scheduling any new signal assignments once all test vectors have gone through the DUT in the `simulation` process. We can achieve this by introducing a boolean `sim_finished` signal. The `sim_finished` signal is written by the `simulation` process once all test vectors have gone through the DUT, and is continuously read by the `clk_generation` process to know when to stop scheduling any new signal assignments. If the `sim_finished` signal is `true`, then we can use an indefinite `wait` statement in both processes to avoid any new signal assignments from occuring, therefore allowing ModelSim to automatically terminate the simulation.

Listing 14 shows how to use this technique.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_adder_sequential is
end tb_adder_sequential;
```

```
architecture test of tb_adder_sequential is

    constant CLK_PERIOD : time := 100 ns;

    -- Signal used to end simulator when we finished submitting our test cases
    signal sim_finished : boolean := false;

    -- adder_sequential GENERICS
    constant N_BITS : positive range 2 to positive'right := 4;

    -- adder_sequential PORTS
    signal CLK   : std_logic;
    signal RST   : std_logic;
    signal START : std_logic;
    signal OP1   : std_logic_vector(N_BITS - 1 downto 0);
    signal OP2   : std_logic_vector(N_BITS - 1 downto 0);
    signal SUM   : std_logic_vector(N_BITS downto 0);
    signal DONE  : std_logic;

begin

    -- Instantiate DUT
    dut : entity work.adder_sequential
    generic map(N_BITS => N_BITS)
    port map(CLK   => CLK,
             RST   => RST,
             START => START,
             OP1   => OP1,
             OP2   => OP2,
             SUM   => SUM,
             DONE  => DONE);

    -- Generate CLK signal
    clk_generation : process
    begin
        if not sim_finished then
            CLK <= '1';
            wait for CLK_PERIOD / 2;
            CLK <= '0';
            wait for CLK_PERIOD / 2;
        else
            wait;
        end if;
    end process clk_generation;

    -- Test adder_sequential
    simulation : process

        procedure async_reset is
        begin
            wait until rising_edge(CLK);
            wait for CLK_PERIOD / 4;
            RST <= '1';

            wait for CLK_PERIOD / 2;
            RST <= '0';
        end procedure async_reset;

        procedure check_add(constant in1          : in natural;
                            constant in2          : in natural;
                            constant res_expected : in natural) is
            variable res : natural;
        begin
            -- Our circuit is sensitive to the rising edge of the CLK, so we
            -- need to be sure to assign signal values such that they are stable
            -- at the next rising edge of the CLK.
            wait until rising_edge(CLK);

            -- Assign values to circuit inputs.
            OP1   <= std_logic_vector(to_unsigned(in1, OP1'length));
            OP2   <= std_logic_vector(to_unsigned(in2, OP2'length));
            START <= '1';

            -- OP1, OP2 and START are NOT yet assigned. We have to wait for some
            -- time for the simulator to "propagate" their values. Any
            -- infinitesimal period would work for the simulator to "propagate"
            -- the values. However, our circuit is a sequential circuit
            -- sensitive to the rising edge of CLK, so we need to hold our
            -- signal assignments until the next rising edge of CLK so the
            -- circuit can see them.
            wait until rising_edge(CLK);

            -- Remove values from circuit inputs. The circuit works with a PULSE
            -- on its START input, which means that data on the inputs only
            -- needs to be valid when START is high.
            OP1   <= (others => '0');
            OP2   <= (others => '0');
            START <= '0';

            -- The circuit informs us it has finished by asserting DONE, so we
            -- can wait until we receive the signal before proceeding. DONE is
            -- asserted at the rising edge of CLK, so we (the test system) can
            -- sample the data and check its correctness.
            wait until DONE = '1';

            -- Check output against expected result.
            res := to_integer(unsigned(SUM));
            assert res = res_expected
            report "Unexpected result: " &
                   "OP1 = " & integer'image(in1) & "; " &
                   "OP2 = " & integer'image(in2) & "; " &
                   "SUM = " & integer'image(res) & "; " &
                   "SUM_expected = " & integer'image(res_expected)
            severity error;
```

```
            -- Wait for the circuit to go back into the IDLE state.
            wait until DONE = '0';
        end procedure check_add;

    begin

        -- Default values
        OP1   <= (others => '0');
        OP2   <= (others => '0');
        RST   <= '0';
        START <= '0';
        wait for CLK_PERIOD;

        -- Reset the circuit.
        async_reset;

        -- Check test vectors against expected outputs
        check_add(12, 8, 20);
        check_add(10, 6, 16);
        check_add(4, 1, 5);
        check_add(11, 7, 18);

        -- Instruct "clk_generation" process to halt execution.
        sim_finished <= true;

        -- Make this process wait indefinitely (it will never re-execute from
        -- its beginning again).
        wait;
    end process simulation;

end architecture test;
```

Listing 14: Automatic simulation termination

Finally, compiling () and simulating (`restart -f; run -all;`) the code shown in Listing 14 should yield the waveform shown in Figure 20. Note that only 4 test vectors are shown here to prevent the figure from becoming too small and illegible.

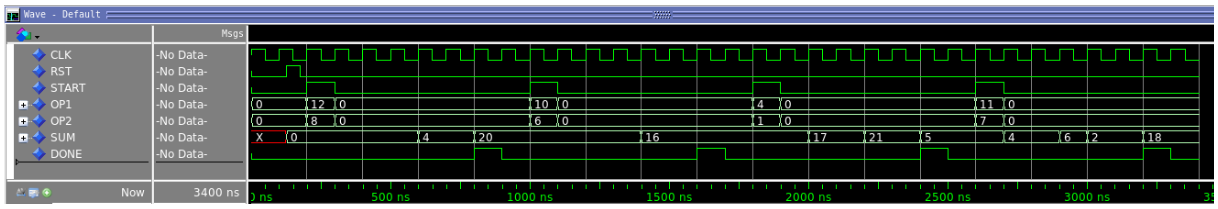

Wave - Default
Msgs
CLK -No Data-
RST -No Data-
START -No Data-
OP1 -No Data- 0 12 0 10 0 4 0 11 0
OP2 -No Data- 0 8 0 6 0 1 0 7 0
SUM -No Data- X 0 4 20 16 17 21 5 4 6 2 18
DONE -No Data-
Now 3400 ns
0 ns 500 ns 1000 ns 1500 ns 2000 ns 2500 ns 3000 ns

Figure 20: Automatic termination