[](http://www.opencores.org/)

PlTbUtils  
Specification

*Author: Per Larsson*

*pela.opencores@gmail.com*

**Rev. 0.6**

**Februrary 2, 2015**

*This page has been intentionally left blank.*

**Revision History**

| **Rev.** | **Date** | **Author** | **Description** |
| --- | --- | --- | --- |
| 0.1 | 9/2/2013 | Per Larsson | First draft |
| 0.2 | 11/10/2013 | Per Larsson | Added sections Acknowledgements and Language. Added reference section on waitsig().  Updated reference section on print() and pltbutils\_clkgen. |
| 0.3 | 1/5/2013 | Per Larsson | Added sections User Configuration, Configuring Simulation Halt, Configuring Messages for Integration Environments. In reference section added starttest, endtest, removed testname. Updated figures and feature bullets. |
| 0.4 | 1/9/2013 | Per Larsson | Updates for alpha0006: Text modified in numerous places to reflect that PlTbUtils is now using the variable pltbv and the signal pltbs for control and status, instead of the previous shared variable and global signals. |
| 0.5 | 13/1/2014 | Per Larsson | Updates for alpha0007: added example testbench where the testcase process is instansiated in the testbench top (tb\_example1). The old example where the testcase process is located in a VHDL component of its own, is now called example\_tb2. |
| 0.6 | 2/2/2015 | Per Larsson | Updates for beta0002: added description of to\_ascending(), to\_descending(), hxstr(), functions and procedures in txt\_util.vhd. |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Introduction

### Overview

PlTbUtils makes it easy to create automatic, self-checking simulation testbenches, and to locate bugs during a simulation. It is a collection of functions, procedures and testbench components that simplifies creation of stimuli and checking results of a device under test.

Features:

* Simulation status printed in transcript windows as well as in waveform window (error count, checks count, number and name of current test, etc).
* Check procedures which output meaningful information when a check fails.
* Clear SUCCESS/FAIL message at end of simulation.
* Easy to locate point in time of bugs, by studying increments of the error counter in the waveform window.
* User-defined information messages in the waveformwindow about what is currently going on.
* Transcript outputs prepared for parsing by scripts, e.g. in regression tests.
* Configurable status messages for use in continous integration environments, e.g. TeamCity.
* Reduces amount of code in tests, which makes them faster to write and easier to read.

It is intended that PlTbUtils will constantly expand by adding more and more functions, procedures and testbench components. Comments, feedback and suggestions are welcome to [pela.opencores@gmail.com](mailto:pela.opencores@gmail.com?subject=PlTbUtils) .

The project page on the web is <http://opencores.org/project,pltbutils> .

Subversion repository URL is <http://opencores.org/ocsvn/pltbutils/pltbutils/trunk>

Subversion export command is  
 svn export http://opencores.org/ocsvn/pltbutils/pltbutils/trunk pltbutils

### Acknowledgements

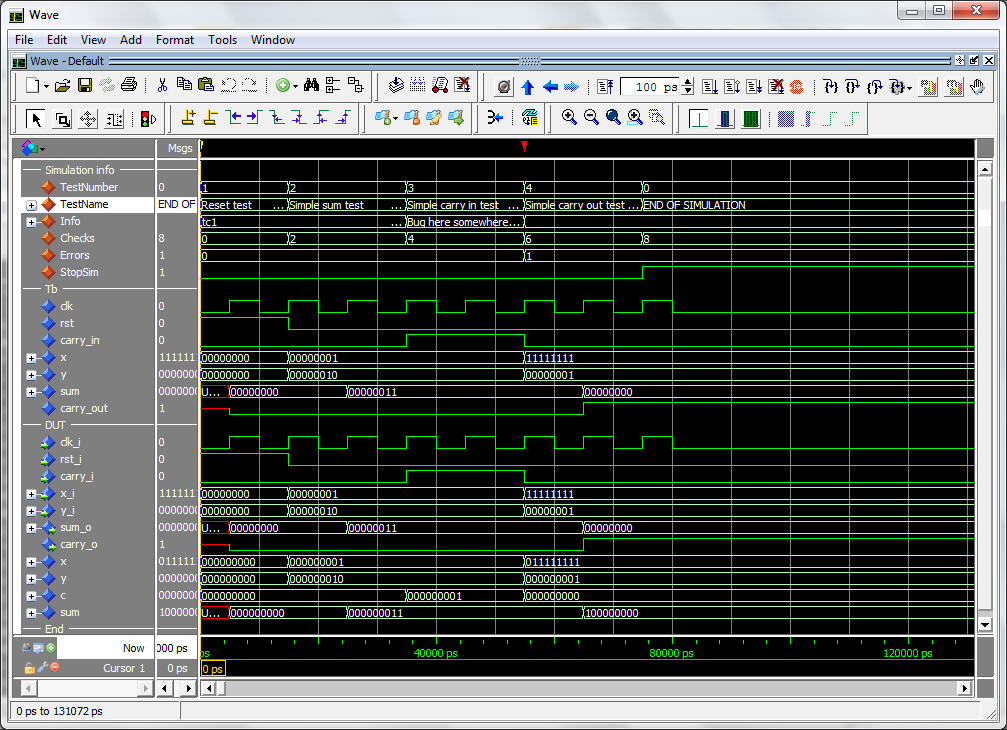
PlTbUtils contains the file txt\_util.vhd by Stefan Doll and James F. Frenzel.

### Language

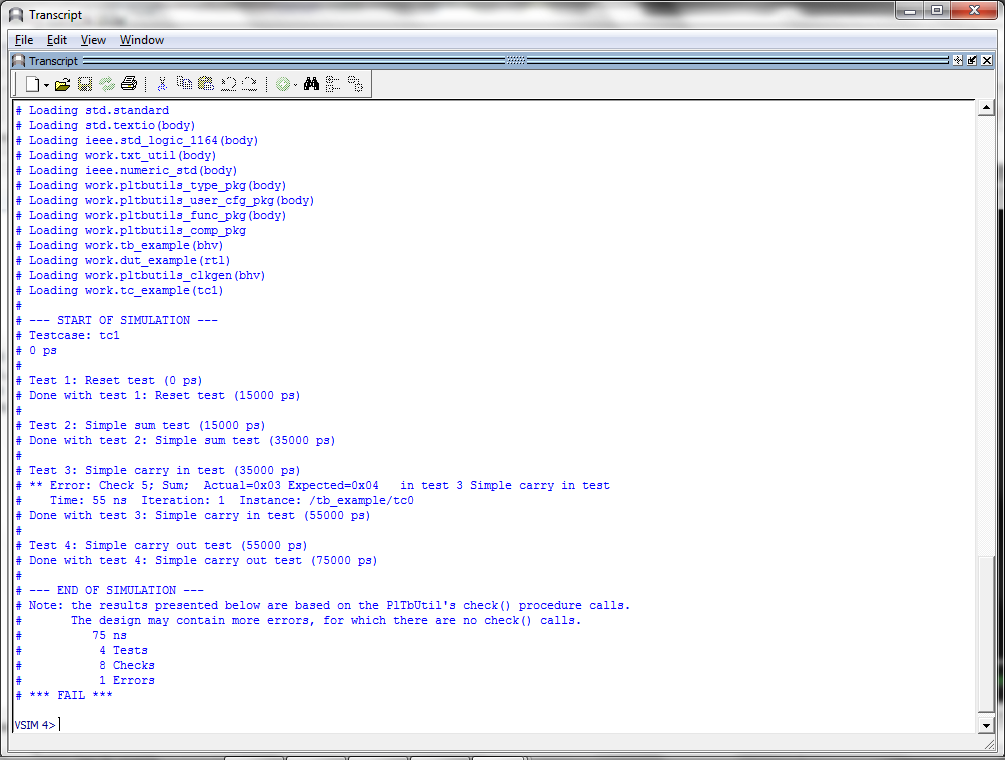
PlTbUtils complies with VHDL-1993, so it works with most VHDL simulators.

However, it is possible to configure the way a simulation stops, by taking advantage of the VHDL-2008 keywords stop and finish. If your simulator supports stop and/or finish, see Configuring Simulation Halt on page 19.

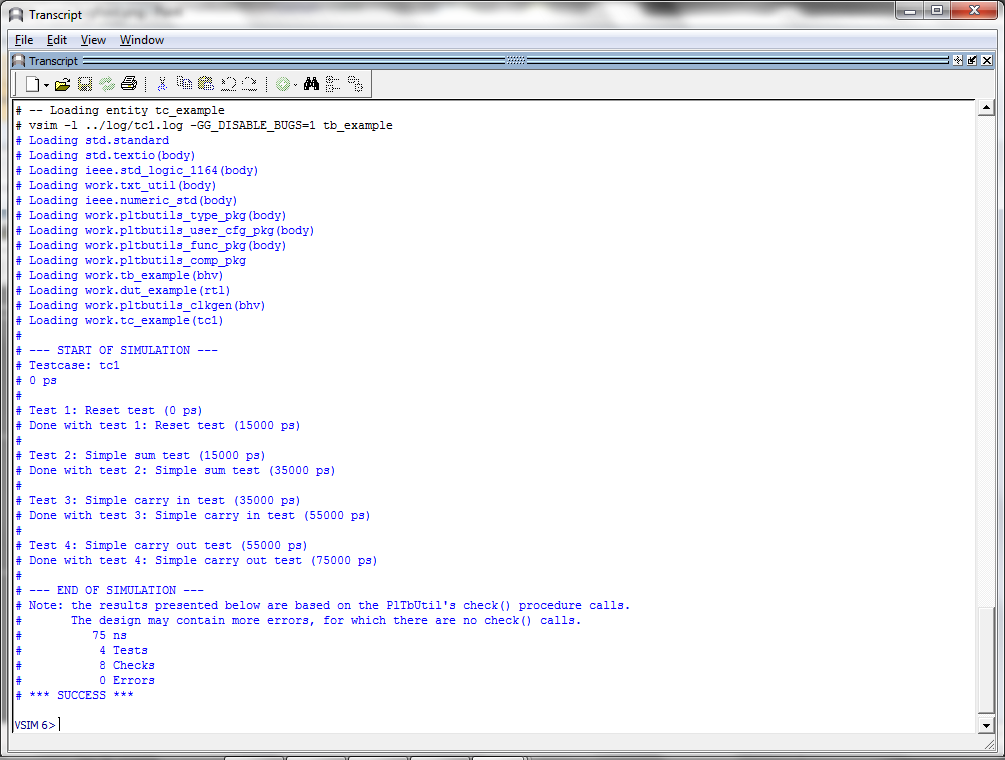
### A quick look



During a simulation, the waveform window shows current test number, test name, user-defined info, accumulated number och checks and errors. When the error counter increments, a bug has been found in that point in time.



The transcript window clearly shows points in time where the simulation starts, ends, and where errors are detected. The simulation stops with a clear SUCCESS/FAIL message, specifically formatted for parsing by scripts.



The testcase code is compact and to the point, which results in less code to write, and makes the code easier to read, as in the following example.

-- NOTE: The purpose of the following code is to demonstrate some of the

-- features in PlTbUtils, not to do a thorough verification.

p\_tc1 : process

variable pltbv : pltbv\_t := C\_PLTBV\_INIT;

begin

startsim("tc1", pltbv, pltbs);

rst <= '1';

carry\_in <= '0';

x <= (others => '0');

y <= (others => '0');

starttest(1, "Reset test", pltbv, pltbs);

waitclks(2, clk, pltbv, pltbs);

check("Sum during reset", sum, 0, pltbv, pltbs);

check("Carry out during reset", carry\_out, '0', pltbv, pltbs);

rst <= '0';

endtest(pltbv, pltbs);

starttest(2, "Simple sum test", pltbv, pltbs);

carry\_in <= '0';

x <= std\_logic\_vector(to\_unsigned(1, x'length));

y <= std\_logic\_vector(to\_unsigned(2, x'length));

waitclks(2, clk, pltbv, pltbs);

check("Sum", sum, 3, pltbv, pltbs);

check("Carry out", carry\_out, '0', pltbv, pltbs);

endtest(pltbv, pltbs);

starttest(3, "Simple carry in test", pltbv, pltbs);

print(G\_DISABLE\_BUGS=0, pltbv, pltbs, "Bug here somewhere");

carry\_in <= '1';

x <= std\_logic\_vector(to\_unsigned(1, x'length));

y <= std\_logic\_vector(to\_unsigned(2, x'length));

waitclks(2, clk, pltbv, pltbs);

check("Sum", sum, 4, pltbv, pltbs);

check("Carry out", carry\_out, '0', pltbv, pltbs);

print(G\_DISABLE\_BUGS=0, pltbv, pltbs, "");

endtest(pltbv, pltbs);

starttest(4, "Simple carry out test", pltbv, pltbs);

carry\_in <= '0';

x <= std\_logic\_vector(to\_unsigned(2\*\*G\_WIDTH-1, x'length));

y <= std\_logic\_vector(to\_unsigned(1, x'length));

waitclks(2, clk, pltbv, pltbs);

check("Sum", sum, 0, pltbv, pltbs);

check("Carry out", carry\_out, '1', pltbv, pltbs);

endtest(pltbv, pltbs);

endsim(pltbv, pltbs, true);

wait;

end process p\_tc1;

Tutorial

### Basics

We will demonstrate how to use PlTbUtils by showing an example. In this example, we have a DUT (Device Under Test / Design Under Test) with the following entity.

entity dut\_example is  
 generic (  
 G\_WIDTH : integer := 8;  
 G\_DISABLE\_BUGS : integer range 0 to 1 := 1  
 );  
 port (  
 clk\_i : in std\_logic;  
 rst\_i : in std\_logic;  
 carry\_i : in std\_logic;  
 x\_i : in std\_logic\_vector(G\_WIDTH-1 downto 0);  
 y\_i : in std\_logic\_vector(G\_WIDTH-1 downto 0);  
 sum\_o : out std\_logic\_vector(G\_WIDTH-1 downto 0);  
 carry\_o : out std\_logic  
 );   
end entity dut\_example;

As you can see, it has a clock- and a reset input port (clk\_i and rst\_i), three other input ports (x\_i, y\_i, and carry\_i), and two output ports (sum\_o and carry\_o). There is also a generic, G\_WIDTH, which sets the number of bits in x\_i, y\_i and sum\_o. The second generic, G\_DISABLE\_BUGS, is very unusual in real designs, but it is useful in this example. We will reveal the purpose of this strange generic later, although some may already be able to guess what it is for.

To verify this DUT, we want the testbench to apply different stimuli to the input ports, and check the response of the output ports. The following code is an example of such a testbench. We will first show all of the code, and then explain parts of it.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

use work.txt\_util.all;

use work.pltbutils\_func\_pkg.all;

use work.pltbutils\_comp\_pkg.all;

entity tb\_example1 is

generic (

G\_WIDTH : integer := 8;

G\_CLK\_PERIOD : time := 10 ns;

G\_DISABLE\_BUGS : integer range 0 to 1 := 0

);

end entity tb\_example1;

architecture bhv of tb\_example1 is

-- Simulation status- and control signals

-- for accessing .stop\_sim and for viewing in waveform window

signal pltbs : pltbs\_t := C\_PLTBS\_INIT;

-- DUT stimuli and response signals

signal clk : std\_logic;

signal rst : std\_logic;

signal carry\_in : std\_logic;

signal x : std\_logic\_vector(G\_WIDTH-1 downto 0);

signal y : std\_logic\_vector(G\_WIDTH-1 downto 0);

signal sum : std\_logic\_vector(G\_WIDTH-1 downto 0);

signal carry\_out : std\_logic;

begin

dut0 : entity work.dut\_example

generic map (

G\_WIDTH => G\_WIDTH,

G\_DISABLE\_BUGS => G\_DISABLE\_BUGS

)

port map (

clk\_i => clk,

rst\_i => rst,

carry\_i => carry\_in,

x\_i => x,

y\_i => y,

sum\_o => sum,

carry\_o => carry\_out

);

clkgen0 : pltbutils\_clkgen

generic map(

G\_PERIOD => G\_CLK\_PERIOD

)

port map(

clk\_o => clk,

stop\_sim\_i => pltbs.stop\_sim

);

-- Testcase process

-- NOTE: The purpose of the following code is to demonstrate some of the

-- features of PlTbUtils, not to do a thorough verification.

p\_tc1 : process

variable pltbv : pltbv\_t := C\_PLTBV\_INIT;

begin

startsim("tc1", pltbv, pltbs);

rst <= '1';

carry\_in <= '0';

x <= (others => '0');

y <= (others => '0');

starttest(1, "Reset test", pltbv, pltbs);

waitclks(2, clk, pltbv, pltbs);

check("Sum during reset", sum, 0, pltbv, pltbs);

check("Carry out during reset", carry\_out, '0', pltbv, pltbs);

rst <= '0';

endtest(pltbv, pltbs);

starttest(2, "Simple sum test", pltbv, pltbs);

carry\_in <= '0';

x <= std\_logic\_vector(to\_unsigned(1, x'length));

y <= std\_logic\_vector(to\_unsigned(2, x'length));

waitclks(2, clk, pltbv, pltbs);

check("Sum", sum, 3, pltbv, pltbs);

check("Carry out", carry\_out, '0', pltbv, pltbs);

endtest(pltbv, pltbs);

starttest(3, "Simple carry in test", pltbv, pltbs);

print(G\_DISABLE\_BUGS=0, pltbv, pltbs, "Bug here somewhere");

carry\_in <= '1';

x <= std\_logic\_vector(to\_unsigned(1, x'length));

y <= std\_logic\_vector(to\_unsigned(2, x'length));

waitclks(2, clk, pltbv, pltbs);

check("Sum", sum, 4, pltbv, pltbs);

check("Carry out", carry\_out, '0', pltbv, pltbs);

print(G\_DISABLE\_BUGS=0, pltbv, pltbs, "");

endtest(pltbv, pltbs);

starttest(4, "Simple carry out test", pltbv, pltbs);

carry\_in <= '0';

x <= std\_logic\_vector(to\_unsigned(2\*\*G\_WIDTH-1, x'length));

y <= std\_logic\_vector(to\_unsigned(1, x'length));

waitclks(2, clk, pltbv, pltbs);

check("Sum", sum, 0, pltbv, pltbs);

check("Carry out", carry\_out, '1', pltbv, pltbs);

endtest(pltbv, pltbs);

endsim(pltbv, pltbs, true);

wait;

end process p\_tc1;

end architecture bhv;

As the testbench example shows, the following packages are needed (in addition to the usual std\_logic\_1164, etc):

use work.txt\_util.all;  
use work.pltbutils\_func\_pkg.all;  
use work.pltbutils\_comp\_pkg.all;

txt\_util contains functions and procedures for handling strings.

pltbutils\_func\_pkg contains type definitions, functions and procedures for controlling stimuli and checking response.

pltbutils\_comp\_pkg contains component declarations for testbench components.

PlTbUtils uses a variable called pltbv, and a signal called pltbs, for controlling the simulation and keeping track of status. The pltbs signal is useful for viewing in the simulator’s waveform window. pltbs is a record containing a number of members which show various information. Expand pltbs in the simulator’s waveform window to expose the members. To make it prettier, you can make use of ModelSim’s Combine Signals feature. Each member of the pltbs record can be set to be its own Combined Signal, see the waveform images in this document. Other simulators usually have similar features.

The DUT is instansiated in the testbench, as well as a clock generator component from PlTbUtils.

There is also a testcase process, which feeds the DUT with stimuli, and checks the results.

The testcase process starts with calling the procedure startsim(). This procedure clears pltbv and pltbs, and outputs a message to the transcript and to the waveform window to inform that the simulation now starts. The first argument to startsim is the name of the testcase.

The last arguments of startsim(), and to many other procedures in PlTbUtils, are pltbv and pltbs.

After initiating stimuli to the DUT, we call the procedure starttest() with the number and name for the first test. starttest() prints the test number and test name to the transcript and to the waveform window, and updates pltbv and pltbs.

Then we need to wait until the DUT has reacted to the stimuli. We do this by calling the procedure waitclks(), which waits a specified number of cycles of the specified clock.

After this, we start checking the results, by examining the outputs from the DUT. To do this, we use the check() procedure. The first argument is a text string that specifies what we check, the second argument is the signal or variable that we want to examine, and the third is the expected value of the signal or variable. If the examined signal holds the expected value, nothing is printed. But if the value is incorrect, the string in the first argument is printed, together with the actual and expected values of the signal. The number and name of the test (as specified with starttest() ) is also printed. PlTbUtils’ check counter is incremented for every check() procedure call, and the error counter is incremented in case of error.

After the test, we call endtest().

We make a number of different tests by calling starttest(), setting stimuli, waiting for the DUT to react with waitclks() or some other means, and checking the outputs with the check() procedure, and calling endtest().

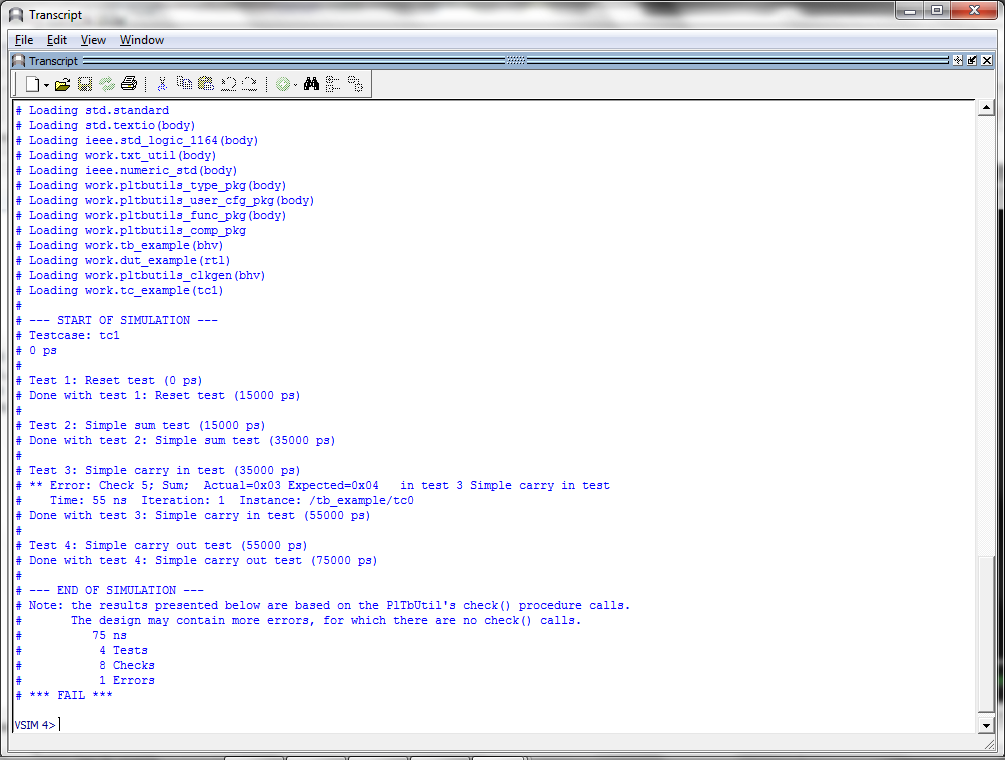
Finally, we call the endsim() procedure, which prints an end-of-simulation message to the transcript, and presents the results, including a SUCCESS or FAIL message.

The start-of-simulation message, end-of-simulation message, and SUCCESS/FAIL messages are unique, to make them easy to search for by scripts. This simplifies collection of simulation status for regression tests with a lot of different simulations.

Try it out in your simulator! The pltbutils files that need to be compiled are located in src/vhdl/, and they are listed in compile order in pltbutils\_files.lst . The example DUT file is located in examples/vhdl/rtl\_example/, and the example testbench files are located in examples/vhdl/example1/. The files are listed in compile order in example\_dut.lst and tb\_example1\_files.lst .

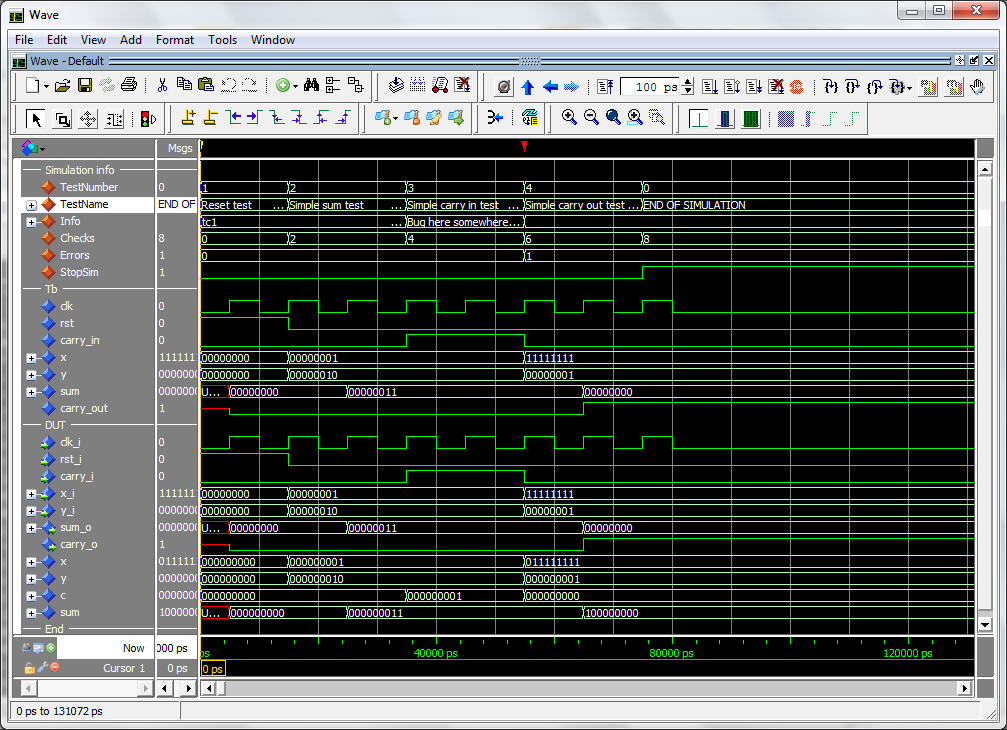
If you are a ModelSim user, there are .do files available in sim/modelsim\_tb\_example1/run/ .  
To use them, start Start ModelSim, and in the ModelSim Gui select the menu item File->Change directory... . Navigate to the PlTbUtils directory sim/modelsim\_tb\_example1/run/ and click Ok. Then, in the transcript window, type   
do run.do .

The simulation will start, and the transcript from the simulation looks as follows.



The transcript says that one error has been found at 55 ns, in test 3; Simple carry in test.

The waveform window looks like this.



Here we can see the error detected at the point in time where the error counter increments from 0 to 1. Again, we can that the error is found in test 3, the Simple carry in test.

Have a look at the DUT code in examples/vhdl/rtl\_example/dut\_example.vhd . It looks as follows.

x <= resize(unsigned(x\_i), G\_WIDTH+1);  
 y <= resize(unsigned(y\_i), G\_WIDTH+1);  
 c <= resize(unsigned(std\_logic\_vector'('0' & carry\_i)), G\_WIDTH+1);  
   
 p\_sum : process(clk\_i)  
 begin  
 if rising\_edge(clk\_i) then  
 if rst\_i = '1' then  
 sum <= (others => '0');  
 else  
 if G\_DISABLE\_BUGS = 1 then  
 sum <= x + y + c;  
 else  
 sum <= x + y;  
 end if;  
 end if;  
 end if;  
 end process;

The code really looks suspisious. If the generic G\_DISABLE\_BUGS is not one, the carry input is not added to the sum. But we need the carry input to be added to the sum!

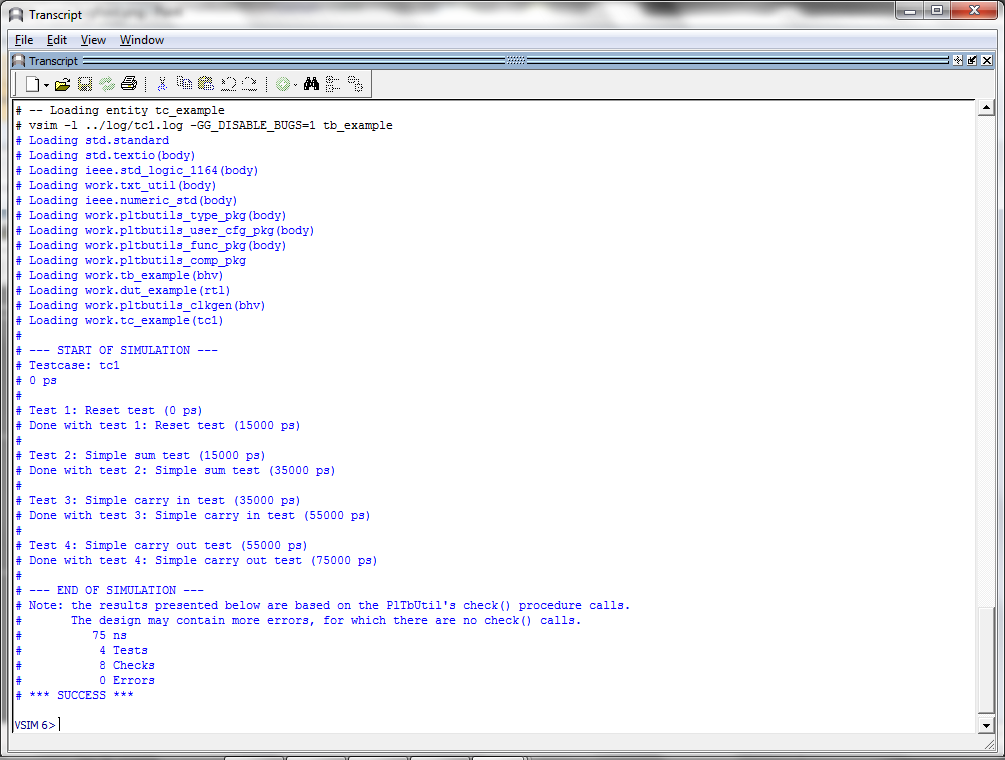
A simple way do disable this bug, is to set the generic G\_DISABLE\_BUGS to one. In this case, this can be done very easily, without any modifying and code.

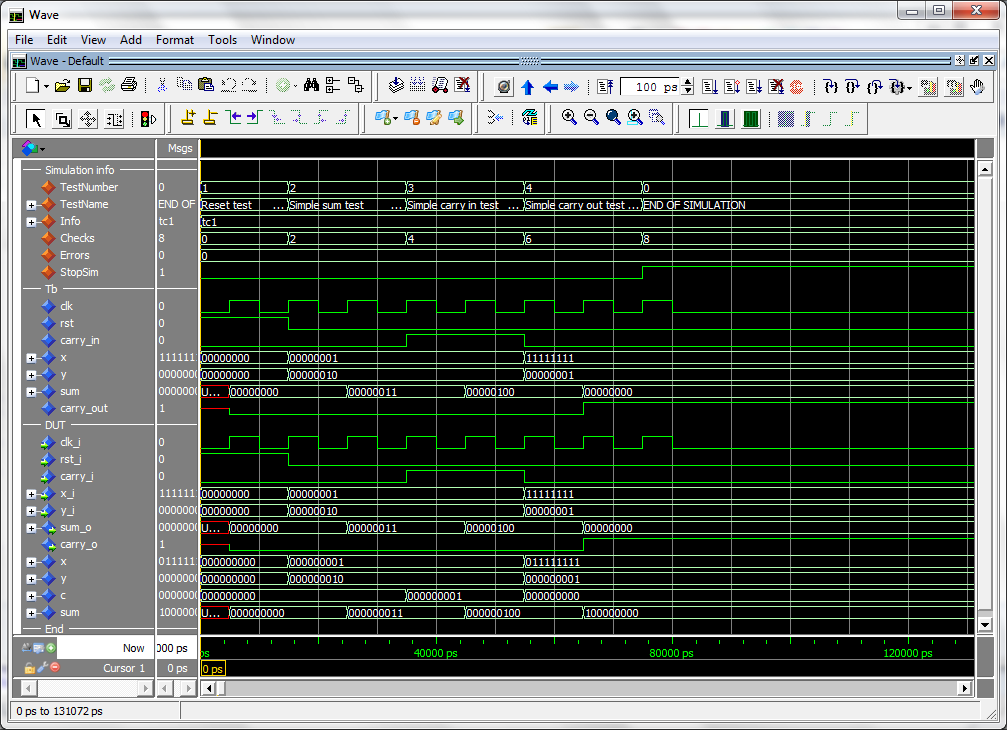
In the ModelSim transcript window, type

do run\_bugfixed.do

This will run the test again, but now with the generic G\_DISABLE\_BUGS set to 1.

The transcript and waveform windows will now look like the following images.





This tutorial has shown some of the available procedures and testbench components in PlTbUtils. For a complete list, see the reference section.

When you want to make your own testbenches with PlTbUtils, have a look at the template files in templates/vhdl/template1/ .

### Testbench with multiple testcases

In some cases, it is more convenient to not include the testcase process in the testbench top. Instead, we can put the testcase process in its own VHDL component. Then we can have alternative architectures for this component, with different testcase processes.

This is practial for large testbenches with a lot of testbench components and other code, with a requirement for multiple testcases. Then we don’t have to write a new testbench for each testcase.

The following is an example of such a testbench.

library ieee;

use ieee.std\_logic\_1164.all;

use work.pltbutils\_func\_pkg.all;

use work.pltbutils\_comp\_pkg.all;

entity tb\_example2 is

generic (

G\_WIDTH : integer := 8;

G\_CLK\_PERIOD : time := 10 ns;

G\_DISABLE\_BUGS : integer range 0 to 1 := 0

);

end entity tb\_example2;

architecture bhv of tb\_example2 is

-- Simulation status- and control signals

-- for accessing .stop\_sim and for viewing in waveform window

signal pltbs : pltbs\_t := C\_PLTBS\_INIT;

-- DUT stimuli and response signals

signal clk : std\_logic;

signal rst : std\_logic;

signal carry\_in : std\_logic;

signal x : std\_logic\_vector(G\_WIDTH-1 downto 0);

signal y : std\_logic\_vector(G\_WIDTH-1 downto 0);

signal sum : std\_logic\_vector(G\_WIDTH-1 downto 0);

signal carry\_out : std\_logic;

begin

dut0 : entity work.dut\_example

generic map (

G\_WIDTH => G\_WIDTH,

G\_DISABLE\_BUGS => G\_DISABLE\_BUGS

)

port map (

clk\_i => clk,

rst\_i => rst,

carry\_i => carry\_in,

x\_i => x,

y\_i => y,

sum\_o => sum,

carry\_o => carry\_out

);

clkgen0 : pltbutils\_clkgen

generic map(

G\_PERIOD => G\_CLK\_PERIOD

)

port map(

clk\_o => clk,

stop\_sim\_i => pltbs.stop\_sim

);

tc0 : entity work.tc\_example2

generic map (

G\_WIDTH => G\_WIDTH,

G\_DISABLE\_BUGS => G\_DISABLE\_BUGS

)

port map(

pltbs => pltbs,

clk => clk,

rst => rst,

carry\_in => carry\_in,

x => x,

y => y,

sum => sum,

carry\_out => carry\_out

);

end architecture bhv;

Instead of a testcase process, we instansiate a testcase component (tc\_example2). This testcase component has an entity defined in one file, and the architecture defined in another file. This makes it possible to have several different testcases for the same testbench. Just compile the testcase architecture that you want to use for a specific simulation run.

The entity declaration for the testcase looks as follows.

library ieee;

use ieee.std\_logic\_1164.all;

use work.pltbutils\_func\_pkg.all;

entity tc\_example2 is

generic (

G\_WIDTH : integer := 8;

G\_DISABLE\_BUGS : integer range 0 to 1 := 0

);

port (

pltbs : out pltbs\_t;

clk : in std\_logic;

rst : out std\_logic;

carry\_in : out std\_logic;

x : out std\_logic\_vector(G\_WIDTH-1 downto 0);

y : out std\_logic\_vector(G\_WIDTH-1 downto 0);

sum : in std\_logic\_vector(G\_WIDTH-1 downto 0);

carry\_out : in std\_logic

);

end entity tc\_example2;

The ports of the testcase components are the same as for the DUT, but the mode (direction) of the ports are the opposite, so the testcase component can drive the inputs of the DUT, and detect the values of the output of the DUT. The only exception to this rule is the clock, which is an input, just as for the DUT.

There is also an output port for pltbs, because pltbs is driven from the tc architecture.

The entity is stored in its’ own file.

The architecture contains the testcase process. There can be several different architecture files. The architecture looks as follows.

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

use work.txt\_util.all;

use work.pltbutils\_func\_pkg.all;

-- NOTE: The purpose of the following code is to demonstrate some of the

-- features in PlTbUtils, not to do a thorough verification.

architecture tc1 of tc\_example2 is

begin

p\_tc1 : process

variable pltbv : pltbv\_t := C\_PLTBV\_INIT;

begin

startsim("tc1", pltbv, pltbs);

rst <= '1';

carry\_in <= '0';

x <= (others => '0');

y <= (others => '0');

starttest(1, "Reset test", pltbv, pltbs);

waitclks(2, clk, pltbv, pltbs);

check("Sum during reset", sum, 0, pltbv, pltbs);

check("Carry out during reset", carry\_out, '0', pltbv, pltbs);

rst <= '0';

endtest(pltbv, pltbs);

starttest(2, "Simple sum test", pltbv, pltbs);

carry\_in <= '0';

x <= std\_logic\_vector(to\_unsigned(1, x'length));

y <= std\_logic\_vector(to\_unsigned(2, x'length));

waitclks(2, clk, pltbv, pltbs);

check("Sum", sum, 3, pltbv, pltbs);

check("Carry out", carry\_out, '0', pltbv, pltbs);

endtest(pltbv, pltbs);

starttest(3, "Simple carry in test", pltbv, pltbs);

print(G\_DISABLE\_BUGS=0, pltbv, pltbs, "Bug here somewhere");

carry\_in <= '1';

x <= std\_logic\_vector(to\_unsigned(1, x'length));

y <= std\_logic\_vector(to\_unsigned(2, x'length));

waitclks(2, clk, pltbv, pltbs);

check("Sum", sum, 4, pltbv, pltbs);

check("Carry out", carry\_out, '0', pltbv, pltbs);

print(G\_DISABLE\_BUGS=0, pltbv, pltbs, "");

endtest(pltbv, pltbs);

starttest(4, "Simple carry out test", pltbv, pltbs);

carry\_in <= '0';

x <= std\_logic\_vector(to\_unsigned(2\*\*G\_WIDTH-1, x'length));

y <= std\_logic\_vector(to\_unsigned(1, x'length));

waitclks(2, clk, pltbv, pltbs);

check("Sum", sum, 0, pltbv, pltbs);

check("Carry out", carry\_out, '1', pltbv, pltbs);

endtest(pltbv, pltbs);

endsim(pltbv, pltbs, true);

wait;

end process p\_tc1;

end architecture tc12;

Try this too in your simulator. The example testbench files are located in examples/vhdl/example2/. The files are listed in compile order in tb\_example2\_files.lst .

If you are a ModelSim user, there are .do files available in sim/modelsim\_tb\_example2/run/ .  
To use them, start Start ModelSim, and in the ModelSim Gui select the menu item File->Change directory... . Navigate to the PlTbUtils directory sim/modelsim\_tb\_example2/run/ and click Ok. Then, in the transcript window, type   
do run\_tc1.do .

Also try

do run\_tc1\_bugfixed.do

Template files for this type of testbench is available in templates/vhdl/template2/ .

### User Configuration

It is possible to configure some aspects of PlTbUtils’s behaviour, by modifying the package file pltbutils\_user\_cfg.pkg.

It is recommended NOT to modify the file directly. Instead, copy it to another directory and modify the copy. Make the simulator read the modified copy instead of the original. This makes it easier to update pltbutils to a later version without destroying the modifications. After updating, check if antyhing has changed in the file, and change your modified copy accordingly.

### Configuring Simulation Halt

When calling endsim(), the signal stop\_sim is set to ‘1’. When set, all clock generators etc in the testbench and the DUT should stop, so there will be no further events in the simulation. The simulator will detect that nothing more will happen, and stops the simulation.

In some cases, it is not possible to stop clock generators, PLL models etc. In that case, endsim() can force a simulaton halt, by setting the force argument to true.

The declaration of endsim() is

procedure endsim(  
 signal pltbutils\_sc : out pltbutils\_sc\_t;  
 constant show\_success\_fail : in boolean := false;  
 constant force : in boolean := false  
 );

so to force a simulation halt, call endsim with

endsim(pltbutils\_sc, true, true);

This stops the simulationg using an assert-failure. This works in all versions of VHDL, but it is an ugly way of doing it, since it outputs a failure message for something which isn’t a failure.

You can change the way the simulation stops when the force flag is set in your copy of pltbutils\_user\_cfg.vhd.

Change the constant C\_PLTBUTILS\_USE\_CUSTOM\_STOPSIM to true, and modify the behaviour of the procedure custom\_stopsim(). In VHDL-2008 the new keywords stop and finish was introduced. Try one of them, for example.

### Configuring Messages for Integration Environments

It is possible adapt the status messages to suit various continous integration environments, e.g. TeamCity, by specifying what the messages should look like.

You can create your own messages printed when starting and stopping a simulation, starting and stopping a test, for checking, etc.

In your copy of pltbutils\_user\_cfg\_pkg.vhd, set one or more of the message constants to true, and modify the associated procedure.

The constants are

C\_PLTBUTILS\_USE\_CUSTOM\_STARTSIM\_MSG  
 C\_PLTBUTILS\_USE\_CUSTOM\_ENDSIM\_MSG  
 C\_PLTBUTILS\_USE\_CUSTOM\_STARTTEST\_MSG  
 C\_PLTBUTILS\_USE\_CUSTOM\_ENDTEST\_MSG  
 C\_PLTBUTILS\_USE\_CUSTOM\_CHECK\_MSG  
 C\_PLTBUTILS\_USE\_CUSTOM\_ERROR\_MSG

The corresponding procedures already contain examples for TeamCity. Modify if you use another environment.

You can disable the standard messages by setting the standard constants to false (C\_PLTBUTILS\_USE\_STD\_STARTSIM\_MSG etc).

Reference

### PlTbUtils files

The PlTbUtils files are located in src/vhdl/ .

The files needed to be compiled are listed in compile order in pltbutils\_files.lst .

See example testbenches using PlTbUtils in examples/vhdl/ .

This code can be simulated from sim/modelsim\_tb\_example1/run/ and sim/modelsim\_tb\_example2/run/ .

Template code is available in templates/vhdl/ .

### Functions and procedures

#### startsim

procedure startsim(

constant testcase\_name : in string;

variable pltbv : inout pltbv\_t;

signal pltbs : out pltbs\_t

)

Displays a message at start of simulation message, and initializes PlTbUtils' status and control variable and -signal. Call startsim() only once.

Arguments:

testcase\_name Name of the test case, e.g. "tc1".

pltbv, pltbs PlTbUtils' status- and control variable and -signal.

The start-of-simulation message is not only intended to be informative for humans. It is also intended to be searched for by scripts, e.g. for collecting results from a large number of regression tests.

Example:

startsim("tc1", pltbutils\_sc);

#### endsim

procedure endsim(

variable pltbv : inout pltbv\_t;

signal pltbs : out pltbs\_t;

constant show\_success\_fail : in boolean := false;

constant force : in boolean := false

)

Displays a message at end of simulation message, presents the simulation results, and stops the simulation. Call endsim() it only once.

Arguments:

pltbv, pltbs PlTbUtils' status- and control variable and -signal.

show\_success\_fail If true, endsim() shows "\*\*\* SUCCESS \*\*\*", "\*\*\* FAIL \*\*\*",   
 or "\*\*\* NO CHECKS \*\*\*". Optional, default is false.

force If true, forces the simulation to stop using an assert failure  
 statement. Use this option only if the normal way of stopping the  
 simulation doesn't work (see below). Optional, default is false.

The testbench should be designed so that all clocks stop when endsim() sets the signal stop\_sim to '1'. This should stop the simulator.

In some cases, that doesn't work, then set the force argument to true, which causes a false assert failure, which should stop the simulator.

The end-of-simulation messages and success/fail messages are not only intended to be informative for humans. They are also intended to be searched for by scripts, e.g. for collecting results from a large number of regression tests.

Examples:

endsim(pltbutils\_sc);

endsim(pltbutils\_sc, true);

endsim(pltbutils\_sc, true, true);

#### starttest

procedure starttest(

constant num : in integer := -1;

constant name : in string;

variable pltbv : inout pltbv\_t;

signal pltbs : out pltbs\_t

)

Sets a number (optional) and a name for a test. The number and name will be printed to the screen, and displayed in the simulator's waveform window.

The test number and name is also included if there errors reported by the check() procedure calls.

Arguments:

num Test number. Optional, default is to increment the current test  
 number.

name Test name.

pltbv, pltbs PlTbUtils' status- and control variable and -signal.

If the test number is omitted, a new test number is automatically computed by incrementing the current test number. Manually setting the test number may make it easier to find the test code in the testbench code, though.

Examples:

starttest("Reset test", pltbv, pltbs);

starttest(1, "Reset test", pltbv, pltbs);

#### endtest

procedure endtest(

variable pltbv : inout pltbv\_t;

signal pltbs : out pltbs\_t

**)**

Prints an end-of-test message to the screen.

Arguments:

pltbv, pltbs PlTbUtils' status- and control variable and -signal.

Example:

endtest(pltbv, pltbs);

#### print printv print2

*Defined in txt\_util.vhd:*

procedure print(

constant txt : in string

)

procedure print(

constant active : in boolean;

constant txt : in string

)

*Defined in pltbutils\_func\_pkg.vhd:*

procedure print(

signal s : out string;

constant txt : in string

)

procedure print(

constant active : in boolean;

signal s : out string;

constant txt : in string

)

procedure print(

variable pltbv : inout pltbv\_t;

signal pltbs : out pltbs\_t;

constant txt : in string

)

procedure print(

constant active : in boolean;

variable pltbv : inout pltbv\_t;

signal pltbs : out pltbs\_t;

constant txt : in string

)

procedure printv(

variable s : out string;

constant txt : in string

)

procedure printv(

constant active : in boolean;

variable s : out string;

constant txt : in string

)

procedure print2(

signal s : out string;

constant txt : in string

)

procedure print2(

constant active : in boolean;

signal s : out string;

constant txt : in string

)

procedure print2(

variable pltbv : inout pltbv\_t;

signal pltbs : out pltbs\_t;

constant txt : in string

)

procedure print2(

constant active : in boolean;

variable pltbv : inout pltbv\_t;

signal pltbs : out pltbs\_t

constant txt : in string

)

print() without a signal as argument prints text messages to the transcript window.  
print() with a signal as argument prints text messages to that signal for viewing in the simulator's waveform window.  
printv() does the same thing, but to a variable instead.  
print2() prints both to a signal and to the transcript window.

The type of the output can be string or pltbv+pltbs.

Arguments:

s Signal or variable of type string to be printed to.

txt The text.

active The text is only printed if active is true. Useful for debug switches,   
 etc.

pltbv, pltbs PlTbUtils' status- and control variable and -signal.

If the string txt is longer than the signal s, the text will be truncated. If txt is shorter, s will be padded with spaces.

Examples:

print("Hello, world"); -- Prints to transcript window

print(msg, "Hello, world"); -- Prints to signal msg

print(G\_DEBUG, msg, "Hello, world"); -- Prints to signal msg if   
 -- generic G\_DEBUG is true

printv(v\_msg, “Hello, world”); -- Prints to variable msg

print(pltbv, pltbs, “Hello, world”); -- Prints to “info” in waveform  
 -- window

print2(msg, “Hello, world”); -- Prints to signal and transcript window

print(pltbv, pltbs, “Hello, world”); -- Prints to “info” in waveform and

-- transcript windows

#### waitclks

procedure waitclks(

constant n : in natural;

signal clk : in std\_logic;

variable pltbv : inout pltbv\_t;

signal pltbs : out pltbs\_t;

constant falling : in boolean := false;

constant timeout : in time := C\_PLTBUTILS\_TIMEOUT

)

Waits specified amount of clock cycles of the specified clock. Or, to be more precise, a specified number of specified clock edges of the specified clock.

Arguments:

n Number of rising or falling clock edges to wait.

clk The clock to wait for.

pltbv, pltbs PlTbUtils' status- and control variable and -signal.

falling If true, waits for falling edges, otherwise rising edges.  
 Optional, default is false.

timeout Timeout time, in case the clock is not working.  
 Optional, default is C\_PLTBUTILS\_TIMEOUT.

Examples:

waitclks(5, sys\_clk, pltbv, pltbs);

waitclks(5, sys\_clk, pltbv, pltbs, true);

waitclks(5, sys\_clk, pltbv, pltbs, true, 1 ms);

#### waitsig

procedure waitsig(  
 signal s : in   
 integer|std\_logic|std\_logic\_vector|unsigned|signed;  
 constant value : in  
 integer|std\_logic|std\_logic\_vector|unsigned|signed;  
 signal clk : in std\_logic;  
 variable pltbv : inout pltbv\_t;

signal pltbs : out pltbs\_t;

constant falling : in boolean := false;  
 constant timeout : in time := C\_PLTBUTILS\_TIMEOUT)

Waits until a signal has reached a specified value after specified clock edge.

Arguments:

s The signal to test.   
 Supported types: integer, std\_logic, std\_logic\_vector, unsigned,  
 signed.

value Value to wait for.  
 Same type as data or integer.

clk The clock.

pltbv, pltbs PlTbUtils' status- and control variable and -signal.

falling If true, waits for falling edges, otherwise rising edges.   
 Optional, default is false.

timeout Timeout time, in case the clock is not working.  
 Optional, default is C\_PLTBUTILS\_TIMEOUT.

Examples:

waitsig(wr\_en, '1', sys\_clk, pltbv, pltbs);

waitsig(rd\_en, 1, sys\_clk, pltbv, pltbs, true);

waitclks(full, '1', sys\_clk, pltbv, pltbs, true, 1 ms);

#### check

procedure check(

constant rpt : in string;

constant data : in integer |  
 std\_logic | std\_logic\_vector |  
 unsigned | signed;

constant expected : in integer |  
 std\_logic | std\_logic\_vector |  
 unsigned | signed;

variable pltbv : inout pltbv\_t;

signal pltbs : out pltbs\_t

)

procedure check(

constant rpt : in string;

constant data : in std\_logic\_vector;

constant expected : in std\_logic\_vector;

constant mask : in std\_logic\_vector;

variable pltbv : inout pltbv\_t;

signal pltbs : out pltbs\_t

)

procedure check(

constant rpt : in string;

constant expr : in boolean;

variable pltbv : inout pltbv\_t;

signal pltbs : out pltbs\_t

)

Checks that the value of a signal or variable is equal to expected. If not equal, displays an error message and increments the error counter.

Arguments:

rpt Report message to be displayed in case of mismatch.  
 It is recommended that the message is unique and that it contains  
 the name of the signal or variable being checked. The message  
 should NOT contain the expected value, becase check() prints that  
 automatically.

data The signal or variable to be checked.  
 Supported types: integer, std\_logic, std\_logic\_vector, unsigned,  
 signed.

expected Expected value. Same type as data, or integer.

mask Bit mask and:ed to data and expected before comparison.  
 Optional if data is std\_logic\_vector. Not allowed for other types.

expr boolean expression for checking.  
 This makes it possible to check any kind of expresion,   
 not just equality.

pltbv, pltbs PlTbUtils' status- and control variable and -signal.

Examples:

check(“dat\_o after reset”, dat\_o, 0, pltbv, pltbs);

-- With mask:  
check(“Status field in reg\_o after start”, reg\_o, x”01”, x”03”,  
 pltbv, pltbs);

-- Boolean expresson: check(“Counter after data burst”, cnt\_o > 10, pltbv, pltbs);

#### to\_ascending

function to\_ascending(

constant s : std\_logic\_vector | unsigned | signed

) return std\_logic\_vector | unsigned | signed;

Converts a vector to ascending range ("to-range").   
The argument s can have ascending or descending range.   
E.g. an argument defined as a std\_logic\_vector(3 downto 1)  
will be returned as a std\_logic\_vector(1 to 3).

Arguments:

s Constant, signal or variable to convert

Return value: Converted value

Examples:

ascending\_sig <= to\_ascending(descending\_sig);

ascending\_var := to\_ascending(descending\_var);

#### to\_descending

function to\_descending(

constant s : std\_logic\_vector | unsigned | signed

) return std\_logic\_vector | unsigned | signed;

Converts a vector to descending range ("downto-range").   
The argument s can have ascending or descending range.   
E.g. an argument defined as a std\_logic\_vector(1 to 3)  
will be returned as a std\_logic\_vector(3 downto 1).

Arguments:

s Constant, signal or variable to convert

Return value: Converted value

Examples:

descending\_sig <= to\_descending(ascending\_sig);

descending\_var := to\_descending(ascending\_var);

#### hxstr

function hxstr(

constant s : std\_logic\_vector | unsigned | signed;

constant prefix : string := "";

constant postfix : string := ""

) return string;

Converts a vector to a string in hexadecimal format.  
An optional prefix can be specified, e.g. "0x", as well as a suffix.

The input argument can have ascending range ( "to-range" ) or descending range ("downto-range"). There is no vector length limitation.

Arguments:

s Constant, signal or variable to convert

Return value: Converted value

Examples:

print("value=" & hxstr(s));

print("value=" & hxstr(s, "0x"));

print("value=" & hxstr(s, "16#", "#"));

#### More functions and procedures in txt\_util.vhd

-- converts std\_logic into a character  
 function chr(sl: std\_logic) return character;  
  
 -- converts std\_logic into a string (1 to 1)  
 function str(sl: std\_logic) return string;  
  
 -- converts std\_logic\_vector into a string (binary base)  
 function str(slv: std\_logic\_vector) return string;  
  
 -- converts boolean into a string  
 function str(b: boolean) return string;  
  
 -- converts an integer into a single character  
 -- (can also be used for hex conversion and other bases)  
 function chr(int: integer) return character;  
  
 -- converts integer into string using specified base  
 function str(int: integer; base: integer) return string;  
  
 -- converts integer to string, using base 10  
 function str(int: integer) return string;  
  
 -- convert std\_logic\_vector into a string in hex format  
 *-- NOTE: Argument limited to 32 bits. Consider hxstr(), see p 37.* function hstr(slv: std\_logic\_vector) return string;  
  
  
 -- functions to manipulate strings  
 -----------------------------------  
  
 -- convert a character to upper case  
 function to\_upper(c: character) return character;  
  
 -- convert a character to lower case  
 function to\_lower(c: character) return character;  
  
 -- convert a string to upper case  
 function to\_upper(s: string) return string;  
  
 -- convert a string to lower case  
 function to\_lower(s: string) return string;  
   
 -- checks if whitespace (JFF)  
 function is\_whitespace(c: character) return boolean;  
   
 -- remove leading whitespace (JFF)  
 function strip\_whitespace(s: string) return string;  
   
 -- return first nonwhitespace substring (JFF)  
 function first\_string(s: string) return string;

-- finds the first non-whitespace substring in a string and (JFF)   
 -- returns both the substring and the original with the substring  
 -- removed   
 procedure chomp(variable s: inout string; variable shead:   
 out string);  
   
  
   
   
 -- functions to convert strings into other formats  
 --------------------------------------------------  
   
 -- converts a character into std\_logic  
 function to\_std\_logic(c: character) return std\_logic;   
   
 -- converts a hex character into std\_logic\_vector (JFF)  
 function chr\_to\_slv(c: character) return std\_logic\_vector;   
   
 -- converts a character into int (JFF)  
 function chr\_to\_int(c: character) return integer;  
   
 -- converts a binary string into std\_logic\_vector  
 function to\_std\_logic\_vector(s: string) return std\_logic\_vector;   
   
 -- converts a hex string into std\_logic\_vector (JFF)  
 function hstr\_to\_slv(s: string) return std\_logic\_vector;  
   
 -- converts a decimal string into an integer (JFF)  
 function str\_to\_int(s: string) return integer;   
  
   
 -- file I/O  
 -----------  
   
 -- read variable length string from input file  
 procedure str\_read(file in\_file: TEXT;   
 res\_string: out string);  
   
 -- print string to a file and start new line  
 procedure print(file out\_file: TEXT;  
 new\_string: in string);  
   
 -- print character to a file and start new line  
 procedure print(file out\_file: TEXT;  
 char: in character);

### Testbench components

#### pltbutils\_clkgen

Creates a clock for use in a testbech. The clock stops when input port stop\_sim goes ‘1’. This makes the simulator stop (unless there are other infinite processes running in the simulation).

| **Generic** | **Width** | **Type** | **Description** |
| --- | --- | --- | --- |
| G\_PERIOD | 1 | time | Clock period. |
| G\_INITVALUE | 1 | std\_logic | Initial value of the non-inverted clock output. |

| **Port** | **Width** | **Direction** | **Description** |
| --- | --- | --- | --- |
| clk\_o | 1 | Output | Non-inverted clock output. Use this output for single ended or differential clocks. |
| clk\_n\_o | 1 | Output | Inverted clock output.  Use if a differential clock is needed, leave open if single-ended clock is needed. |
| stop\_sim\_i | 1 | Input | When ‘1’, stops the clock. This will normally stop the simulation. |