



PITbUtils Specification

*Author: Per Larsson
pela@opencores.org*

Rev. 0.1
September 2, 2013

This page has been intentionally left blank.

Revision History

Rev	Date	Author	Description
.			
0.1	9/2/2013	Per Larsson	First draft

1

Introduction

Overview

PITbUtils 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.

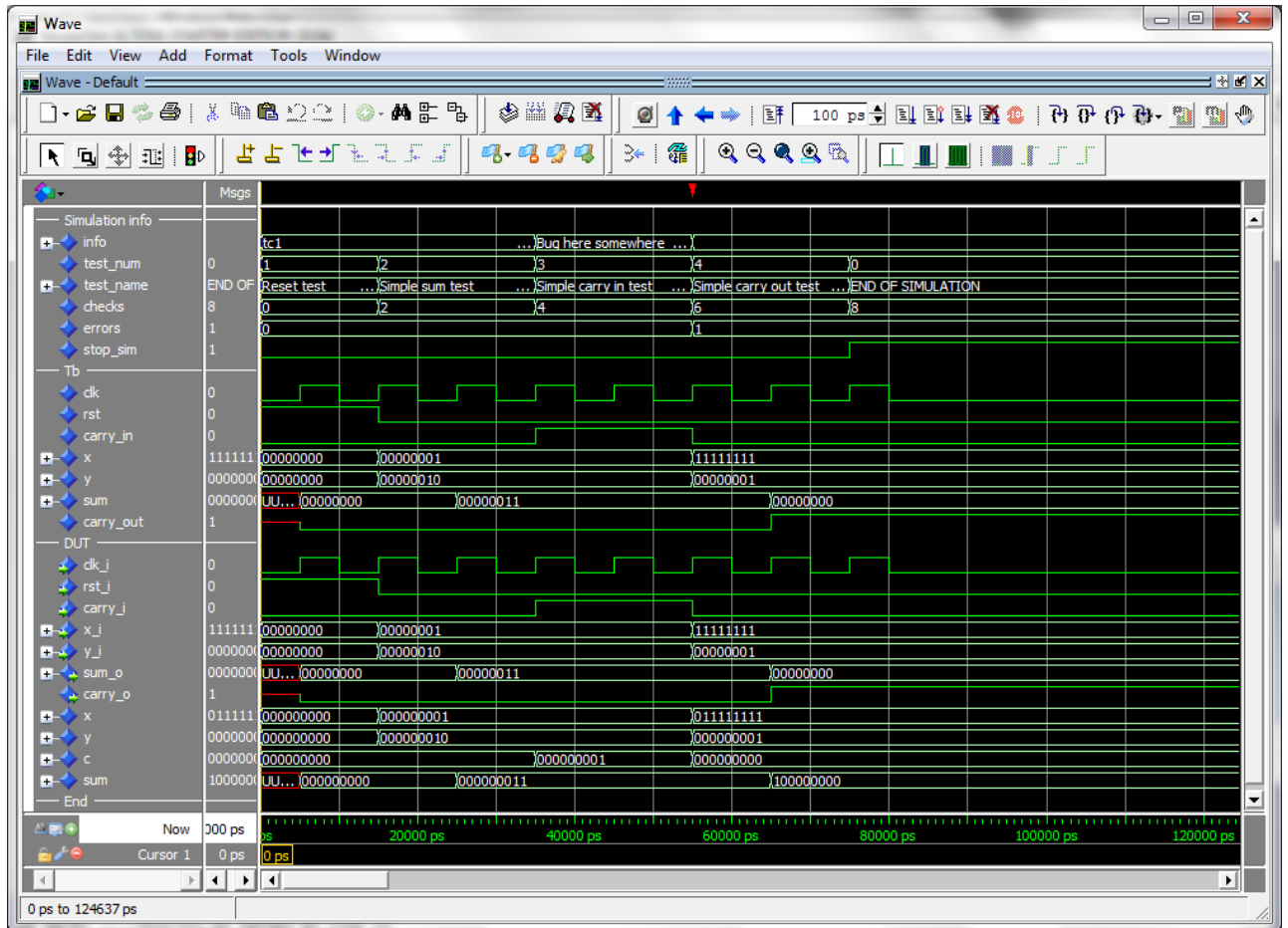
Features:

- 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.
- Reduces amount of code in tests, which makes them faster to write and easier to read.

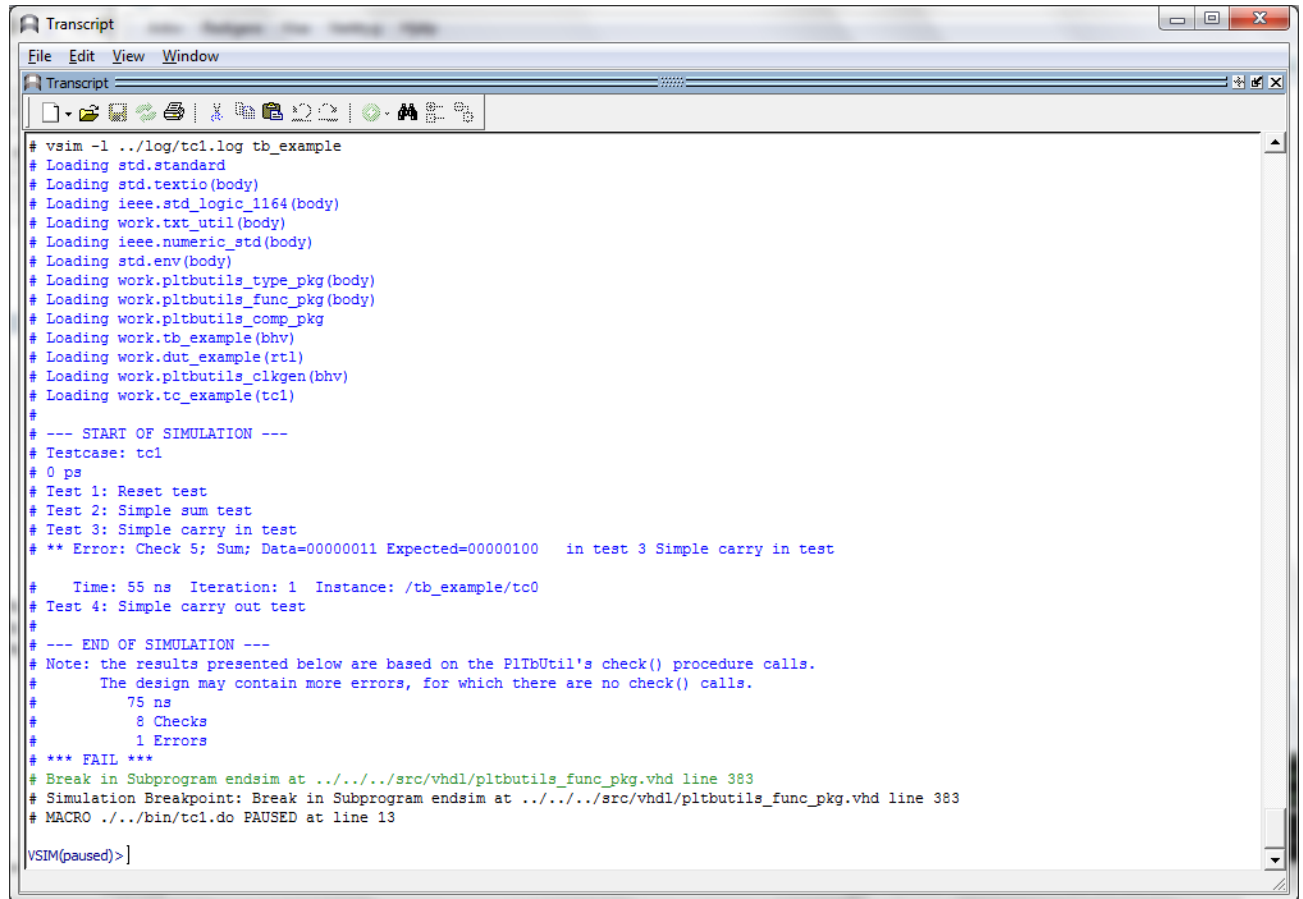
It is intended that PITbUtils will constantly expand by adding more and more functions, procedures and testbench components. Comments, feedback and suggestions are welcome to pela@opencores.org .

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

A quick look

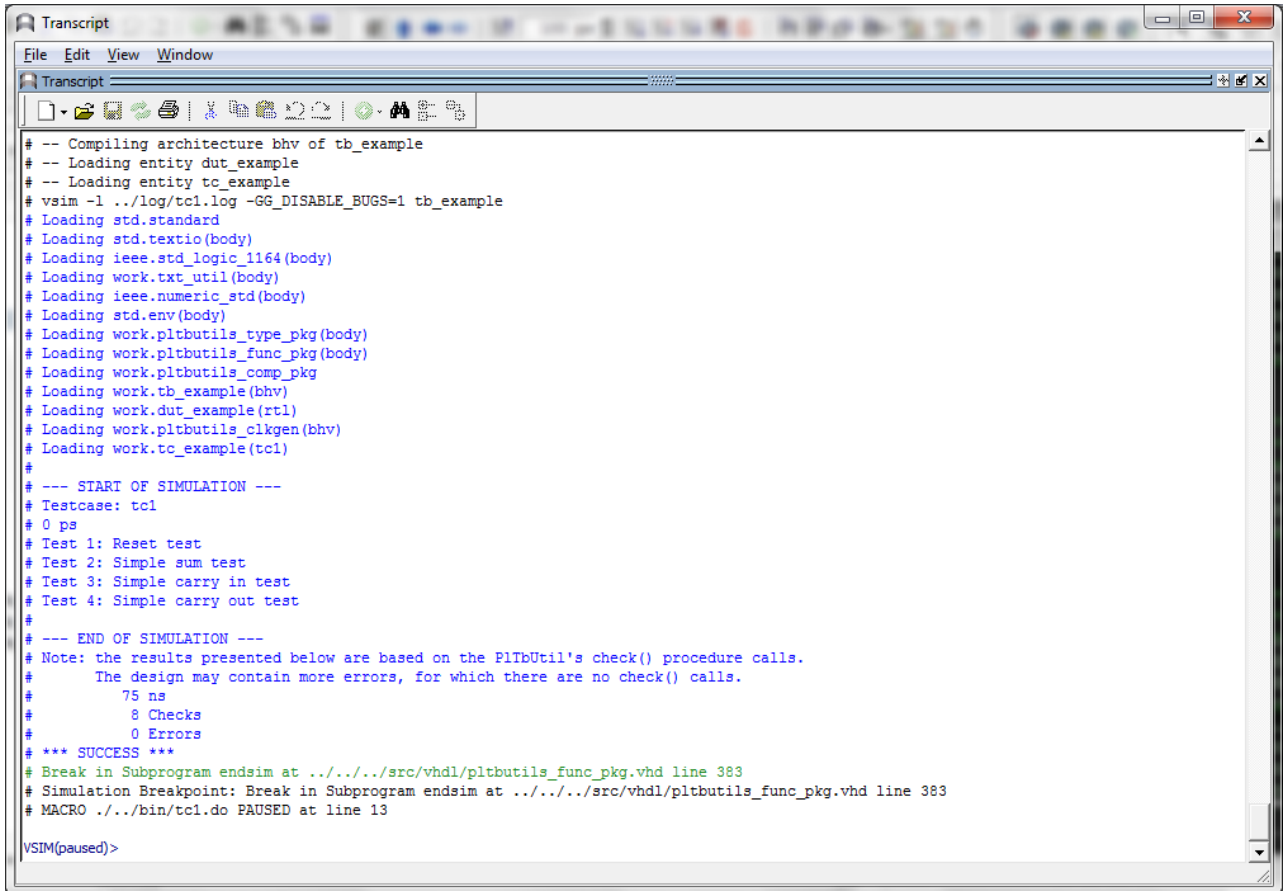


During a simulation, the waveform window shows current test number, test name, user-defined info, accumulated number of checks and errors. When the error counter increments, a bug has been found in that point in time.



```
# vsim -l ../log/tcl.log tb_example
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading work.txt_util(body)
# Loading ieee.numeric_std(body)
# Loading std.env(body)
# Loading work.pltbutils_type_pkg(body)
# Loading work.pltbutils_func_pkg(body)
# Loading work.pltbutils_comp_pkg
# Loading work.tb_example(bhv)
# Loading work.dut_example(rtl)
# Loading work.pltbutils_clkgen(bhv)
# Loading work.tc_example(tcl)
#
# --- START OF SIMULATION ---
# Testcase: tcl
# 0 ps
# Test 1: Reset test
# Test 2: Simple sum test
# Test 3: Simple carry in test
# ** Error: Check 5; Sum; Data=00000011 Expected=00000100   in test 3 Simple carry in test
#
#   Time: 55 ns   Iteration: 1   Instance: /tb_example/tc0
# Test 4: Simple carry out test
#
# --- END OF SIMULATION ---
# Note: the results presented below are based on the PITbUtil's check() procedure calls.
#       The design may contain more errors, for which there are no check() calls.
#           75 ns
#           8 Checks
#           1 Errors
# *** FAIL ***
# Break in Subprogram endsim at ../../src/vhdl/pltbutils_func_pkg.vhd line 383
# Simulation Breakpoint: Break in Subprogram endsim at ../../src/vhdl/pltbutils_func_pkg.vhd line 383
# MACRO ../../bin/tcl.do PAUSED at line 13
V$SIM(paused)>
```

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.



```

# -- Compiling architecture bhv of tb_example
# -- Loading entity dut_example
# -- Loading entity tc_example
# vsim -l ../log/tcl.log -GG_DISABLE_BUGS=1 tb_example
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading work.txt_util(body)
# Loading ieee.numeric_std(body)
# Loading std.env(body)
# Loading work.pltbutils_type_pkg(body)
# Loading work.pltbutils_func_pkg(body)
# Loading work.pltbutils_comp_pkg
# Loading work.tb_example(bhv)
# Loading work.dut_example(rtl)
# Loading work.pltbutils_clkgen(bhv)
# Loading work.tc_example(tcl)
#
# --- START OF SIMULATION ---
# Testcase: tcl
# 0 ps
# Test 1: Reset test
# Test 2: Simple sum test
# Test 3: Simple carry in test
# Test 4: Simple carry out test
#
# --- END OF SIMULATION ---
# Note: the results presented below are based on the PITbUtil's check() procedure calls.
#       The design may contain more errors, for which there are no check() calls.
#       75 ns
#           8 Checks
#           0 Errors
# *** SUCCESS ***
# Break in Subprogram endsim at ../../src/vhdl/pltbutils_func_pkg.vhd line 383
# Simulation Breakpoint: Break in Subprogram endsim at ../../src/vhdl/pltbutils_func_pkg.vhd line 383
# MACRO ../bin/tcl.do PAUSED at line 13
VSIM(paused)>
    
```

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.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.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 tcl of tc_example is
begin
  p_tcl : process
  begin
    startsim("tcl", pltbutils_sc);
    rst      <= '1';
    carry_in <= '0';
    x        <= (others => '0');
    y        <= (others => '0');

    testname(1, "Reset test", pltbutils_sc);
    waitclks(2, clk, pltbutils_sc);
    check("Sum during reset",      sum,      0, pltbutils_sc);
    check("Carry out during reset", carry_out, '0', pltbutils_sc);
    rst      <= '0';

    testname(2, "Simple sum test", pltbutils_sc);
    carry_in <= '0';
    x <= std_logic_vector(to_unsigned(1, x'length));
    y <= std_logic_vector(to_unsigned(2, x'length));
    waitclks(2, clk, pltbutils_sc);
    check("Sum",      sum,      3, pltbutils_sc);
    check("Carry out", carry_out, '0', pltbutils_sc);

    testname(3, "Simple carry in test", pltbutils_sc);
    print(pltbutils_sc, "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, pltbutils_sc);
    check("Sum",      sum,      4, pltbutils_sc);
    check("Carry out", carry_out, '0', pltbutils_sc);
```



```
print(pltbutils_sc, "");

testname(4, "Simple carry out test", pltbutils_sc);
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, pltbutils_sc);
check("Sum",      sum,      0, pltbutils_sc);
check("Carry out", carry_out, '1', pltbutils_sc);

endsim(pltbutils_sc, true);
wait;
end process p_tc1;
end architecture tc1;
```

2

Tutorial

We will demonstrate how to use PITbUtils 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 std.textio.all;
use work.txt_util.all;
use work.pltbutils_func_pkg.all;
use work.pltbutils_comp_pkg.all;

entity tb_example is
  generic (
    G_WIDTH          : integer := 8;
    G_CLK_PERIOD     : time := 10 ns;
    G_DISABLE_BUGS   : integer range 0 to 1 := 0
  );
end entity tb_example;

architecture bhv of tb_example is

  -- Simulation status- and control signals
  signal test_num      : integer;
  signal test_name     : string(pltbutils_sc.test_name'range);
  signal info          : string(pltbutils_sc.info'range);
  signal checks        : integer;
  signal errors        : integer;
  signal stop_sim      : std_logic;

  -- 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

    -- Simulation status and control for viewing in waveform window
    test_num <= pltbutils_sc.test_num;
    test_name <= pltbutils_sc.test_name;
    info      <= pltbutils_sc.info;
    checks    <= pltbutils_sc.chk_cnt;
    errors    <= pltbutils_sc.err_cnt;
    stop_sim  <= pltbutils_sc.stop_sim;

    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 => stop_sim
        );
end;
```

```
tc0 : entity work.tc_example
  generic map (
    G_WIDTH      => G_WIDTH
  )
  port map(
    clk          => clk,
    rst          => rst,
    carry_in     => carry_in,
    x            => x,
    y            => y,
    sum          => sum,
    carry_out    => carry_out
  );

end architecture bhv;
```

As the testbench example shows, the following packages are needed (in addition to the usual `std_logic_1164`, etc):

```
use std.textio.all;
use work.txt_util.all;
use work.pltbutils_func_pkg.all;
use work.pltbutils_comp_pkg.all;
```

`txt_util` contains useful text utilities, such as print procedures.

`pltbutils_func_pkg` contains functions and procedures for controlling stimuli and checking response.

`pltbutils_comp_pkg` contains component declarations for testbench components.

PITbUtils contain a number of hidden, global signals for controlling the simulation and keeping track of status. These signals are useful for viewing in the simulator's waveform window. To make them available for viewing, we declare a number of signals under the comment `Simulation status- and control signals`, and then make assignments to them under the `begin` statement.

The DUT is instansiated, as well as a clock generator component from PITbUtils. We also instansiate a testcase component (`tc_example`). 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;

entity tc_example is
  generic (
    G_WIDTH      : integer := 8
  );
  port (
    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_example;
```

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.

One possible testcase architecture could look as the following code.

-- 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_example is
begin
    p_tc1 : process
    begin
        startsim("tc1", pltbutils_sc);
        rst          <= '1';
        carry_in     <= '0';
        x            <= (others => '0');
        y            <= (others => '0');

        testname(1, "Reset test", pltbutils_sc);
        waitclks(2, clk, pltbutils_sc);
        check("Sum during reset",          sum,          0, pltbutils_sc);
        check("Carry out during reset",    carry_out, '0', pltbutils_sc);
        rst          <= '0';

        testname(2, "Simple sum test", pltbutils_sc);
        carry_in <= '0';
        x <= std_logic_vector(to_unsigned(1, x'length));
        y <= std_logic_vector(to_unsigned(2, x'length));
        waitclks(2, clk, pltbutils_sc);
        check("Sum",          sum,          3, pltbutils_sc);
        check("Carry out",    carry_out, '0', pltbutils_sc);

        testname(3, "Simple carry in test", pltbutils_sc);
        print(pltbutils_sc, "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, pltbutils_sc);
        check("Sum",          sum,          4, pltbutils_sc);
        check("Carry out",    carry_out, '0', pltbutils_sc);
        print(pltbutils_sc, "");

        testname(4, "Simple carry out test", pltbutils_sc);
        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, pltbutils_sc);
        check("Sum",          sum,          0, pltbutils_sc);
        check("Carry out",    carry_out, '1', pltbutils_sc);
    end
end
    
```



```
endsim(pltbutils_sc, true);  
wait;  
end process p_tcl;  
end architecture tcl;
```

The testcase process starts with calling the procedure `startsim()`. This process clears all hidden, global control- and status signals, 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 argument of `startsim()`, and to many other procedures in PITbUtils, is the name of the hidden, global status- and control signals. This argument must always look like this, with this name.

After initiating stimuli to the DUT, we call the procedure `testname()`, for setting a name and a number for the following test. `testname()` prints the test number and test name to the transcript and to the waveform window.

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. The purpose is the

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 `testname()`) is also printed. PITbUtils' check counter is incremented for every `check()` procedure call, and the error counter is incremented in case of error.

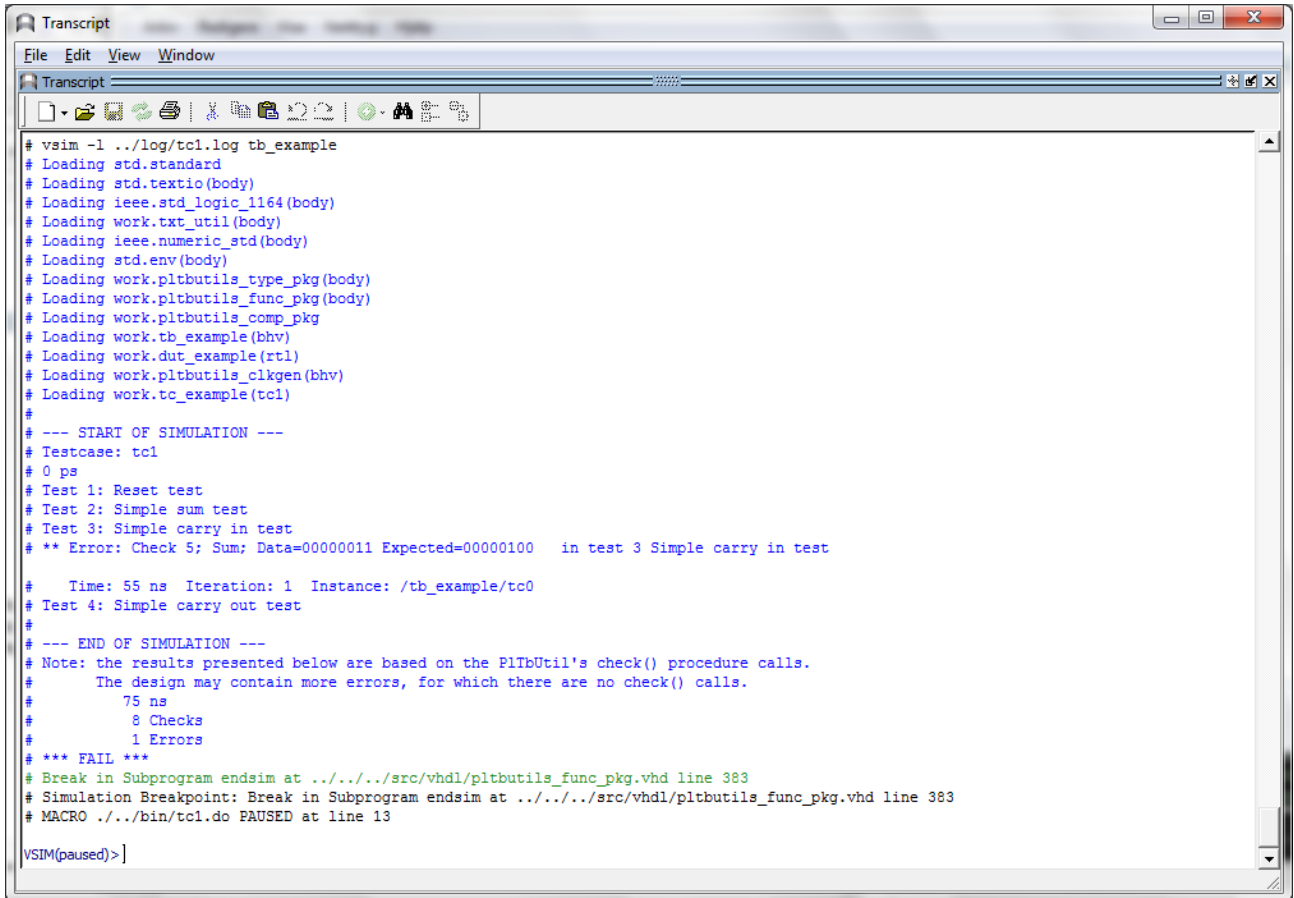
We make a number of different tests by calling `testname()`, setting stimuli, waiting for the DUT to react with `waitclks()` or some other means, and checking the outputs with the `check()` procedure.

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.

Now test to run the simulation. Start ModelSim, and in the ModelSim Gui select the menu item File->Change directory... Navigate to the PITbUtils directory `sim/example_sim/run/` and click Ok. Then, in the transcript window, type `do run_tcl.do`.

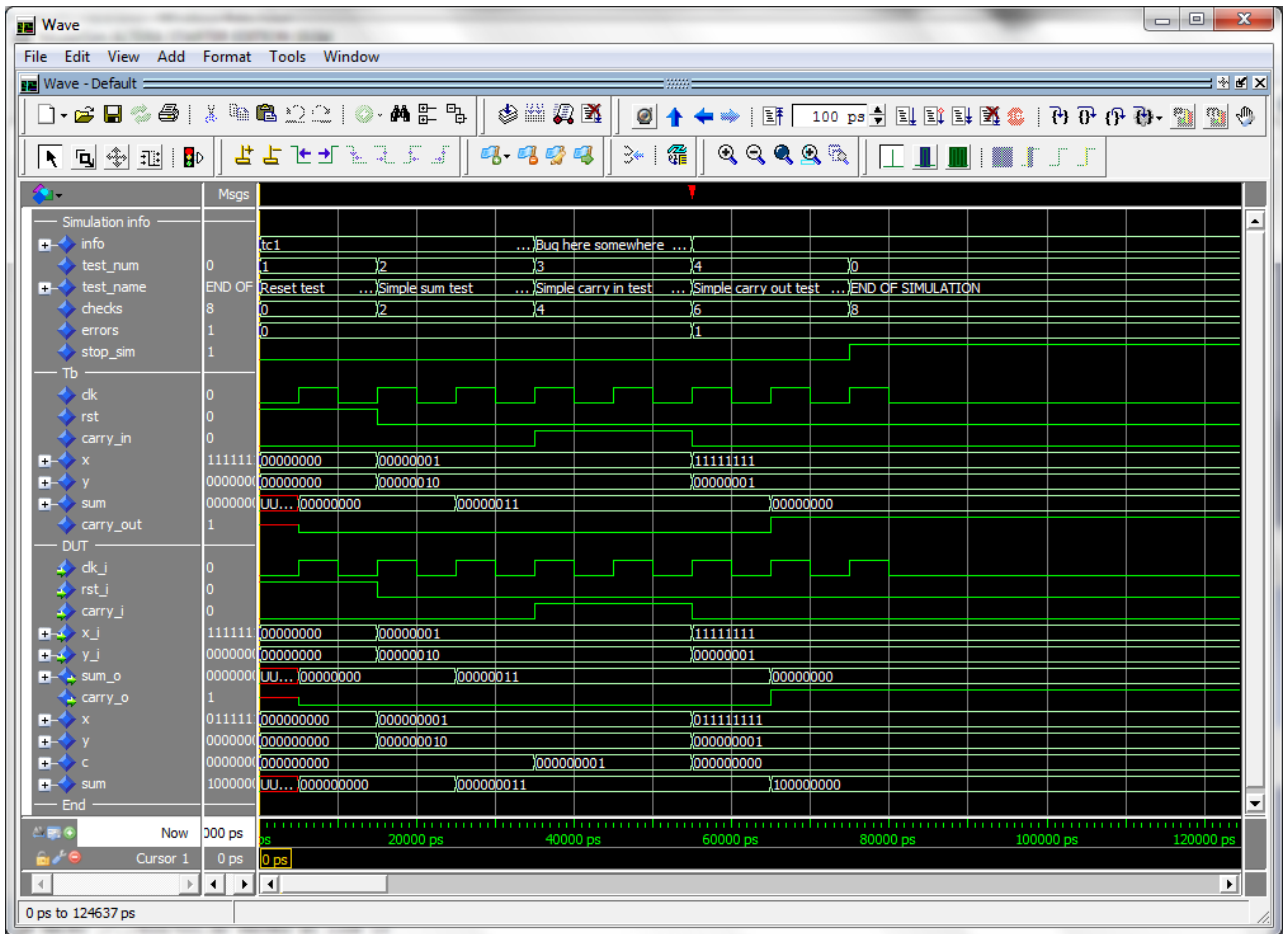
The simulation will start, and the transcript from the simulation looks as follows.



```
# vsim -l ../log/tcl1.log tb_example
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading work.txt_util(body)
# Loading ieee.numeric_std(body)
# Loading std.env(body)
# Loading work.pltbutils_type_pkg(body)
# Loading work.pltbutils_func_pkg(body)
# Loading work.pltbutils_comp_pkg
# Loading work.tb_example(bhv)
# Loading work.dut_example(rtl)
# Loading work.pltbutils_clkgen(bhv)
# Loading work.tc_example(tcl)
#
# --- START OF SIMULATION ---
# Testcase: tcl
# 0 ps
# Test 1: Reset test
# Test 2: Simple sum test
# Test 3: Simple carry in test
# ** Error: Check 5; Sum; Data=00000011 Expected=00000100 in test 3 Simple carry in test
#
# Time: 55 ns Iteration: 1 Instance: /tb_example/tc0
# Test 4: Simple carry out test
#
# --- END OF SIMULATION ---
# Note: the results presented below are based on the PITbUtil's check() procedure calls.
#       The design may contain more errors, for which there are no check() calls.
#       75 ns
#       8 Checks
#       1 Errors
# *** FAIL ***
# Break in Subprogram endsim at ../../src/vhdl/pltbutils_func_pkg.vhd line 383
# Simulation Breakpoint: Break in Subprogram endsim at ../../src/vhdl/pltbutils_func_pkg.vhd line 383
# MACRO ../bin/tcl1.do PAUSED at line 13
VSIM(paused)>]
```

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 when the error counter increments. Again, we can see that the error is found in test 3, the Simple carry in test.

The DUT is example/vhdl/dut_example.vhd . If we carefully study the code that involves carry in, we can see the following piece of code. It really looks suspicious.

```
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;
```

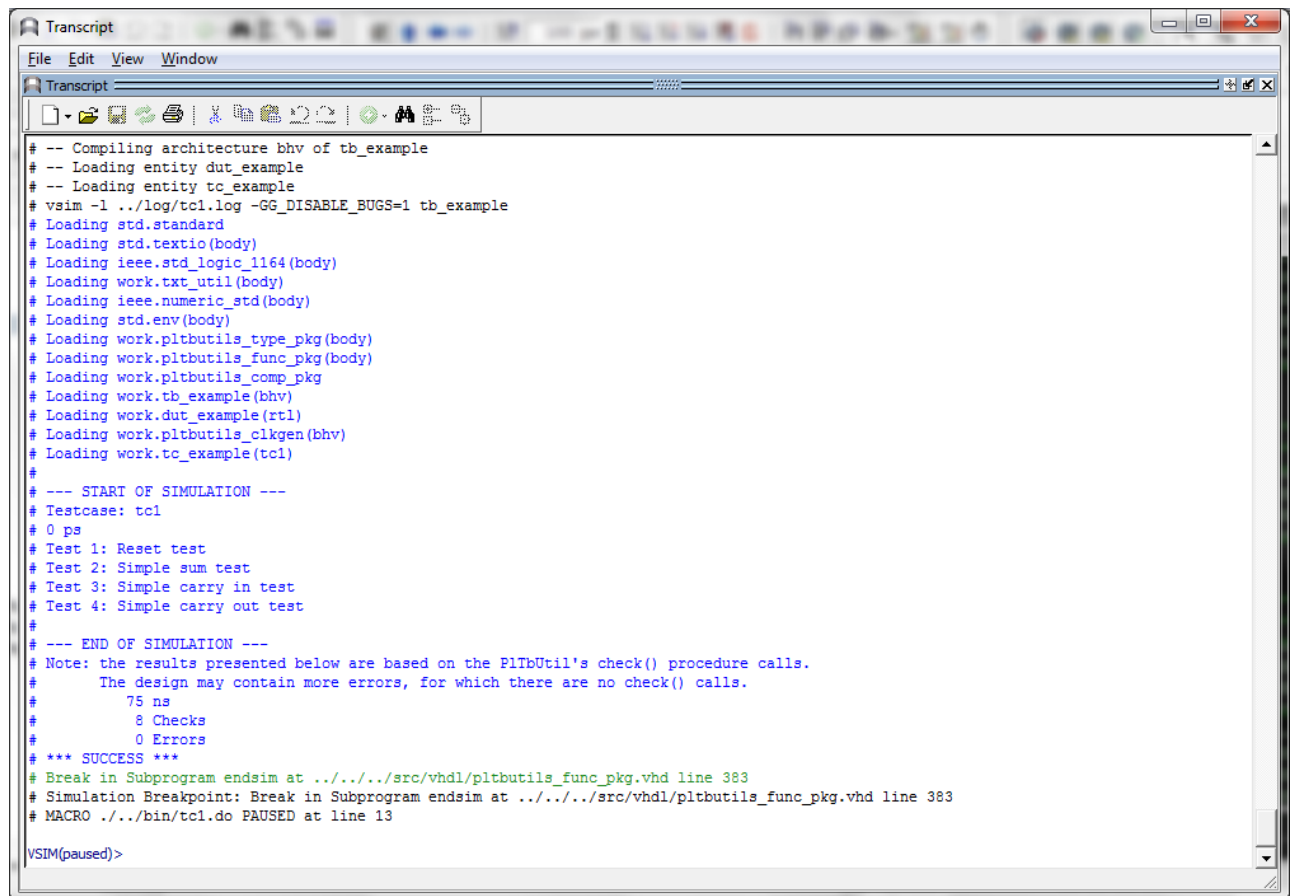
It looks like if the generic `G_DISABLE_BUGS` is not one, the carry input is not added to the sum. The simple way do disable this bug, is to set the generic `G_DISABLE_BUGS` to one. In this case, this can be done very easily, without any coding.

In the ModelSim transcript window, type

```
do run_tc1_bugfixed.do
```

This will run the test again, but now with the generic `G_DISABLE_BUGS` set to 1.

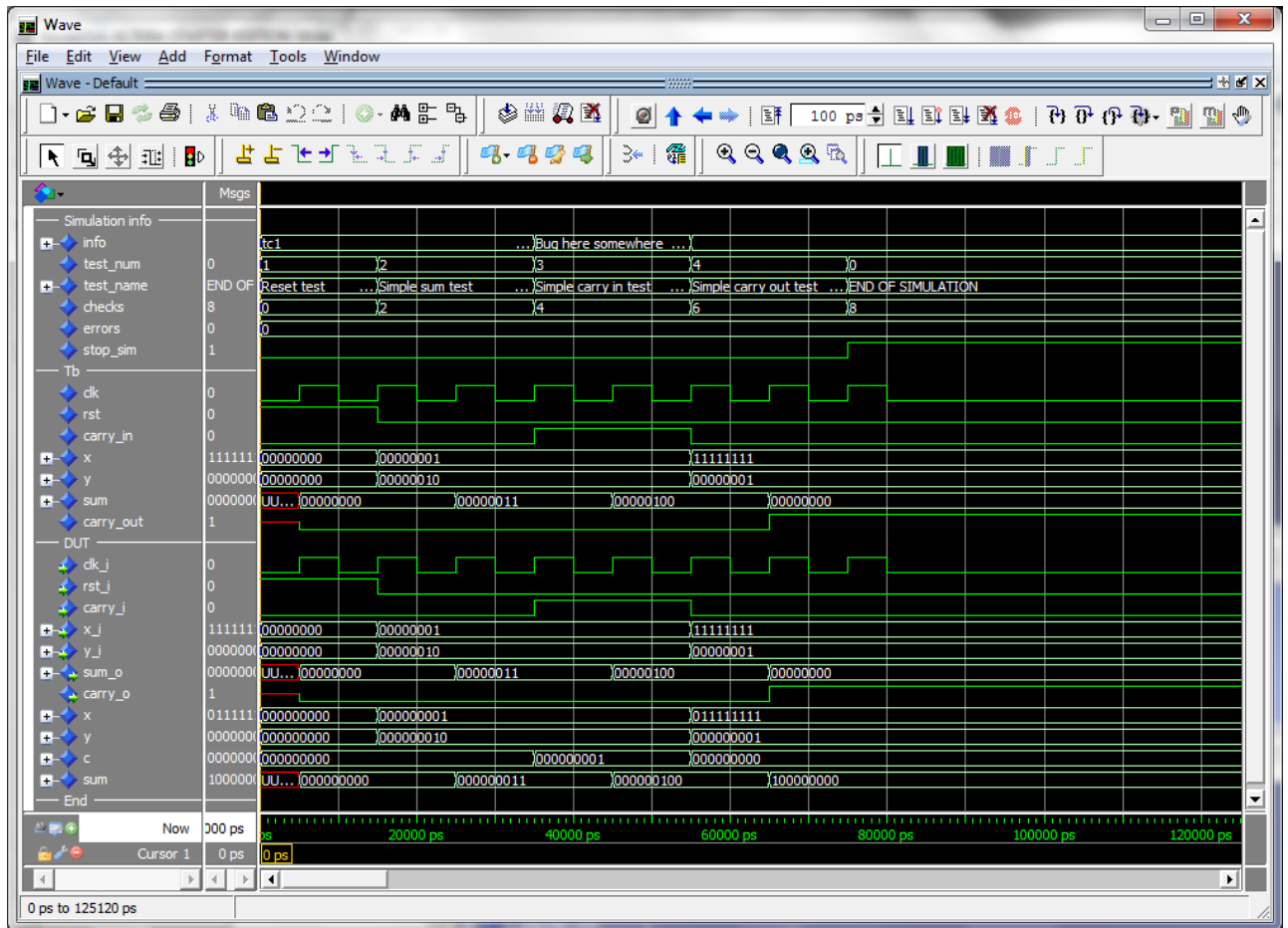
The transcript and waveform windows will now look like this:



The image shows a screenshot of a VSIM transcript window. The window title is "Transcript" and it has a menu bar with "File", "Edit", "View", and "Window". Below the menu bar is a toolbar with various icons. The main area of the window contains a text-based transcript of a simulation. The transcript starts with compilation and loading of various packages and entities, followed by a list of test cases. It concludes with simulation statistics and a success message. The transcript is as follows:

```
# -- Compiling architecture bhv of tb_example
# -- Loading entity dut_example
# -- Loading entity tc_example
# vsim -l ../log/tcl.log -GG_DISABLE_BUGS=1 tb_example
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading work.txt_util(body)
# Loading ieee.numeric_std(body)
# Loading std.env(body)
# Loading work.pltbutils_type_pkg(body)
# Loading work.pltbutils_func_pkg(body)
# Loading work.pltbutils_comp_pkg
# Loading work.tb_example(bhv)
# Loading work.dut_example(rtl)
# Loading work.pltbutils_clkgen(bhv)
# Loading work.tc_example(tcl)
#
# --- START OF SIMULATION ---
# Testcase: tcl
# 0 ps
# Test 1: Reset test
# Test 2: Simple sum test
# Test 3: Simple carry in test
# Test 4: Simple carry out test
#
# --- END OF SIMULATION ---
# Note: the results presented below are based on the PlTbUtil's check() procedure calls.
#       The design may contain more errors, for which there are no check() calls.
#       75 ns
#       8 Checks
#       0 Errors
# *** SUCCESS ***
# Break in Subprogram endsim at ../../src/vhdl/pltbutils_func_pkg.vhd line 383
# Simulation Breakpoint: Break in Subprogram endsim at ../../src/vhdl/pltbutils_func_pkg.vhd line 383
# MACRO ../bin/tcl.do PAUSED at line 13

VSIM(paused)>
```



The PITbUtils files are located in `src/vhdl/`. The files needed to be compiled are listed in `pltbutils_files.lst`.

See example code in `example/vhdl/`. This code can be simulated from `sim/example_sim/run/`.

Template code is available in `template/vhdl/`.

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

3

Reference

Functions and procedures

startsim

```
procedure startsim(  
    constant testcase_name    : in    string;  
    signal   pltbutils_sc     : out   pltbutils_sc_t  
)
```

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

Arguments:

testcase_name	Name of the test case, e.g. "tc1".
pltbutils_sc	PITbUtils' global status- and control signal. Must be set to pltbutils_sc.

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(
    signal   pltbutils_sc           : out pltbutils_sc_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:

pltbutils_sc	PITbUtils' global status- and control signal. Must be set to pltbutils_sc.
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);

```


testname

```
procedure testname(  
    constant num          : in    integer := -1;  
    constant name         : in    string;  
    signal  pltbutils_sc  : out   pltbutils_sc_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.
pltbutils_sc	PITbUtils' global status- and control signal. Must be set to pltbutils_sc.

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:

```
testname("Reset test", pltbutils_sc);  
testname(1, "Reset test", pltbutils_sc);
```

print printv print2

```
procedure print(  
    signal  s                : out  string;  
    constant txt            : in   string  
)  
  
procedure print(  
    signal  pltbutils_sc     : out  pltbutils_sc_t;  
    constant txt            : in   string  
)  
  
procedure printv(  
    variable s              : out  string;  
    constant txt            : in   string  
)  
  
procedure print2(  
    signal  s                : out  string;  
    constant txt            : in   string  
)  
  
procedure print2(  
    signal  pltbutils        : out  pltbutils_sc_t;  
    constant txt            : in   string  
)
```

`print()` prints text messages to a 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 `pltbutils_sc_t`. If the type is `pltbutils_sc_t`, the name can be no other than `pltbutils_sc`.

Arguments:

<code>s</code>	Signal or variable of type <code>string</code> to be printed to.
<code>txt</code>	The text.
<code>pltbutils_sc</code>	PITbUtils' global status- and control signal of type <code>pltbutils_sc_t</code> . The name must be no other than <code>pltbutils_sc</code> .

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(msg, "Hello, world"); -- Prints to signal msg
printv(v_msg, "Hello, world"); -- Prints to variable msg
print(pltbutils_sc, "Hello, world"); -- Prints to "info" in waveform
                                     -- window
print2(msg, "Hello, world"); -- Prints to signal and transcript window
print(pltbutils_sc, "Hello, world"); -- Prints to "info" in waveform and
                                     -- transcript windows
```

waitclks

```
procedure waitclks(  
    constant n                : in    natural;  
    signal  clk                : in    std_logic;  
    signal  pltbutils_sc      : out   pltbutils_sc_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.
pltbutils_sc	PITbUtils' global status- and control signal. Must be set to pltbutils_sc.
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, pltbutils_sc);  
waitclks(5, sys_clk, pltbutils_sc, true);  
waitclks(5, sys_clk, pltbutils_sc, 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;  
    signal  pltbutils_sc  : out   pltbutils_sc_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;  
    signal  pltbutils_sc  : out   pltbutils_sc_t  
)
```

```
procedure check(  
    constant rpt          : in    string;  
    constant expr         : in    boolean;  
    signal  pltbutils_sc  : out   pltbutils_sc_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, because 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 expression, not just equality.
pltbutils_sc	PITbUtils' global status- and control signal. Must be set to the name pltbutils_sc.

Examples:

```

check("dat_o after reset", dat_o, 0, pltbutils_sc);
-- With mask:
check("Status field in reg_o after start", reg_o, x"01", x"03",
      pltbutils_sc);
-- Boolean expression:
check("Counter after data burst", cnt_o > 10, pltbutils_sc);
    
```

Testbench components

pltbutils_clkgen

Creates a clock for use in a testbench. 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.

Port	Width	Direction	Description
clk_o	1	Output	Clock output
stop_sim_i	1	Input	When '1', stops the clock. This will normally stop the simulation.