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

PlTbUtils  
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 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

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.

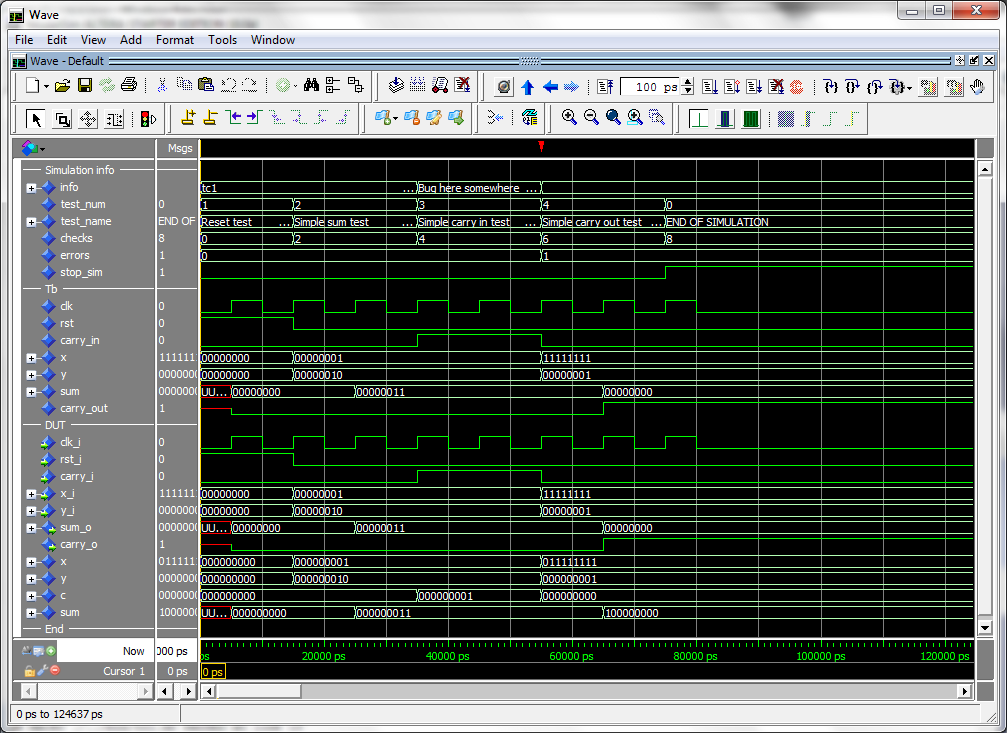
Features:

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

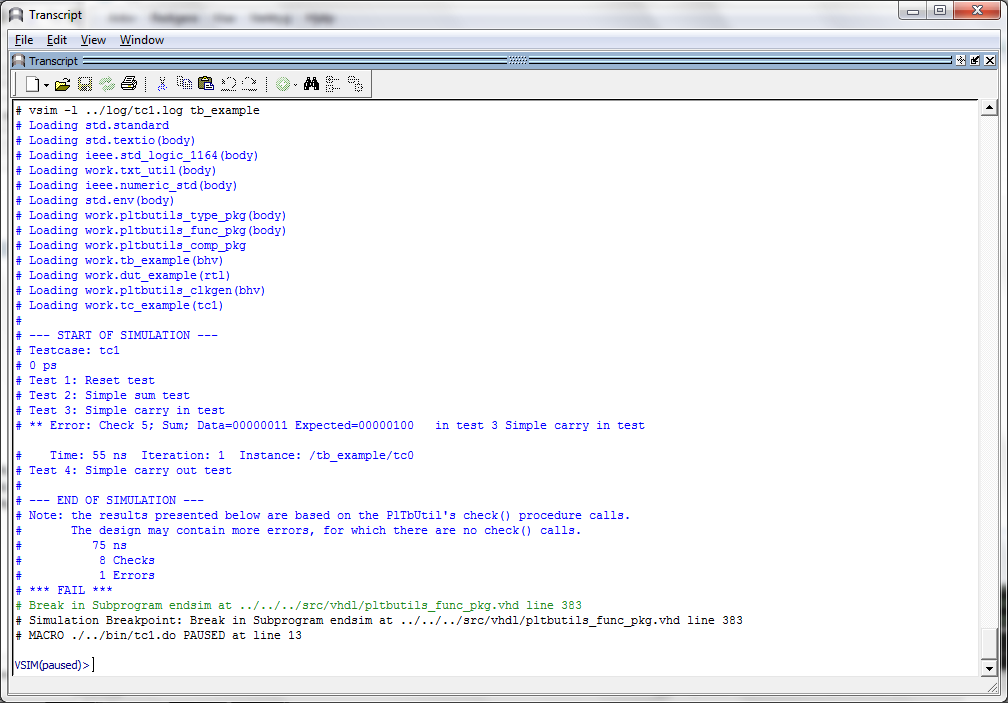
It is intended that PlTbUtils will constantly expand by adding more and more functions, procedures and testbench components. Comments, feedback and suggestions are welcome to [pela@opencores.org](mailto:pela@opencores.org) .

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

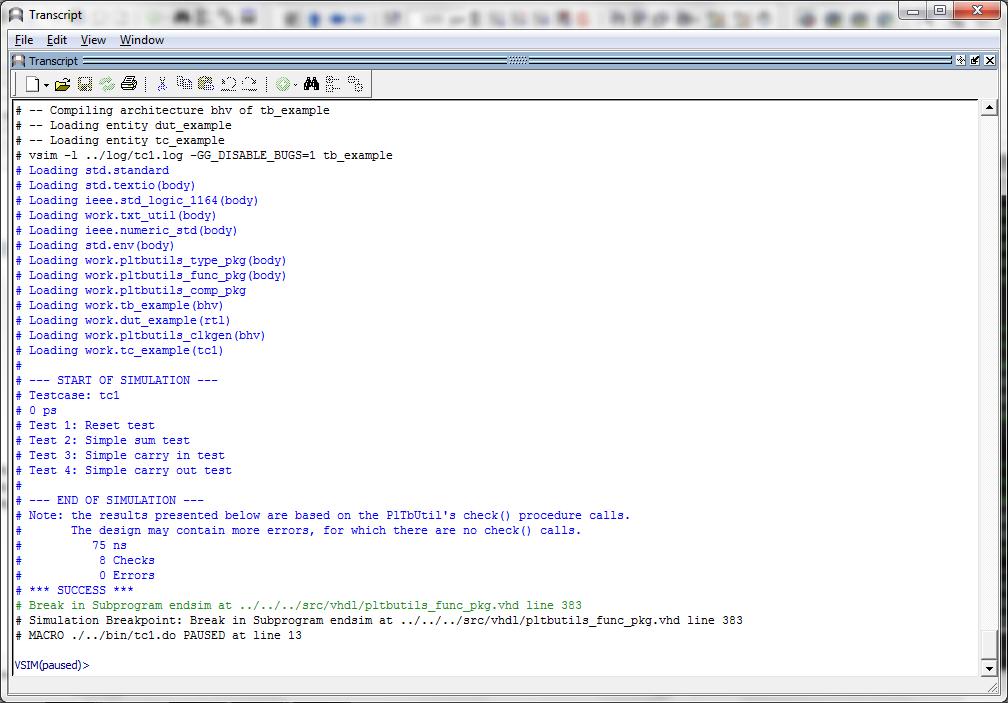
### A quick look



During a simulation, the waveform window shows current test number, test name, user-defined info, accumulated number 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.

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 demostrate 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);

endsim(pltbutils\_sc, true);

wait;

end process p\_tc1;

end architecture tc1;

Tutorial

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 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

);

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.

PlTbUtils 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 assignements to them under the begin statement.

The DUT is instansiated, as well as a clock generator component from PlTbUtils. 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 demostrate 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);

endsim(pltbutils\_sc, true);

wait;

end process p\_tc1;

end architecture tc1;

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 PlTbUtils, 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. PlTbUtils’ check counter is incremented for every check() procedure call, and the error counter is incremented in case of error.

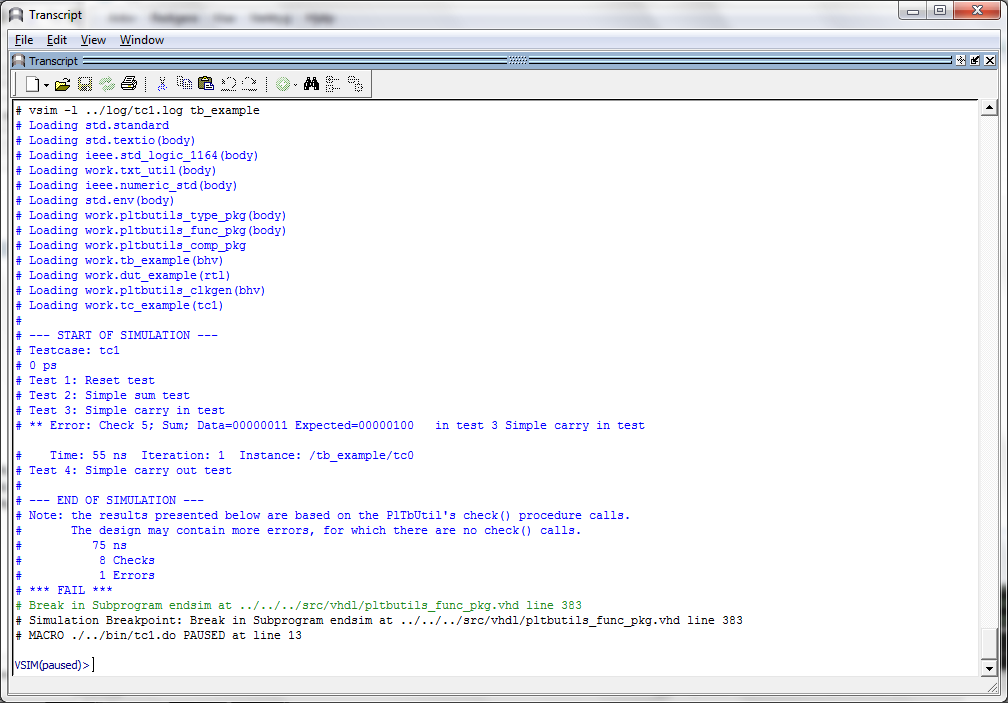
We make a number of different tests by calling testname(), setting stimuli, waiting for the DUT to react with waitclks() or some other means, and checking the outputs with the check() procedure.

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.

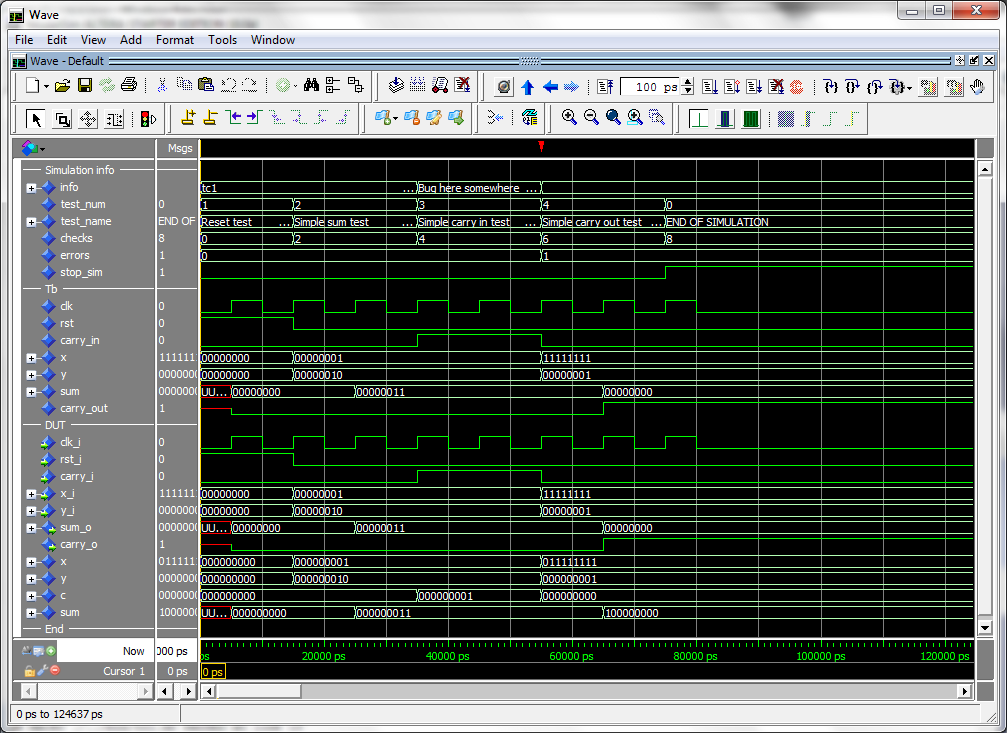
Now test to run the simulation. Start ModelSim, and in the ModelSim Gui select the menu item File->Change directory... Navigate to the PlTbUtils directory sim/example\_sim/run/ and click Ok. Then, in the transcript window, type   
do run\_tc1.do .

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 when the error counter increments. Again, we can 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 suspisious.

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;

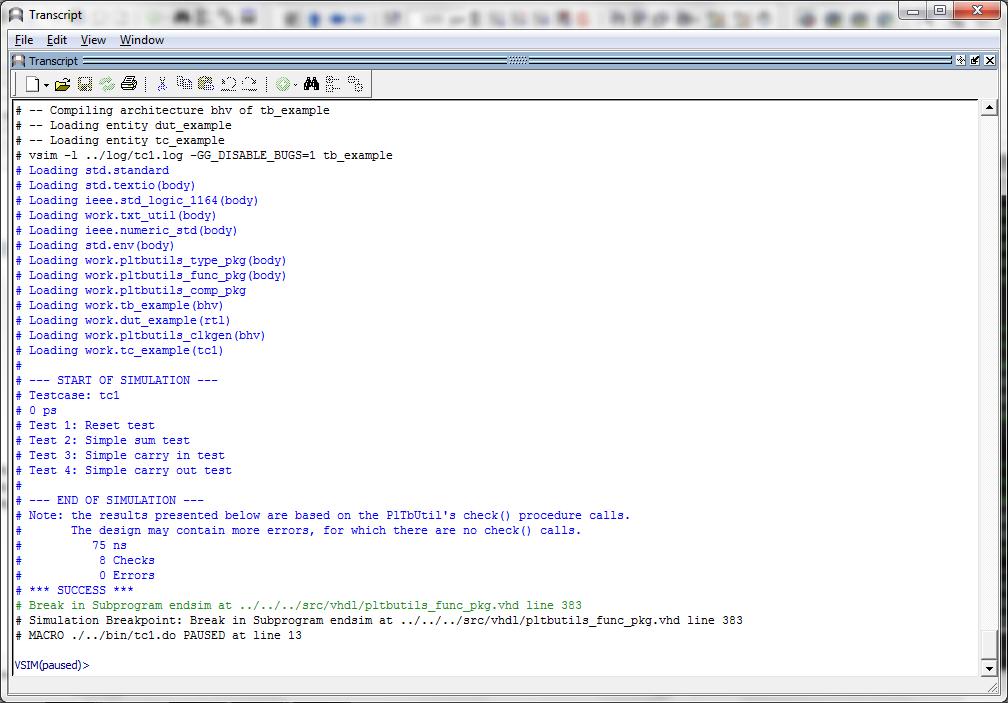
It looks like if the generic G\_DISABLE\_BUGS is not one, the carry input is not added to the sum. The simple way do disable this bug, is to set the generic G\_DISABLE\_BUGS to one. In this case, this can be done very easily, without any coding.

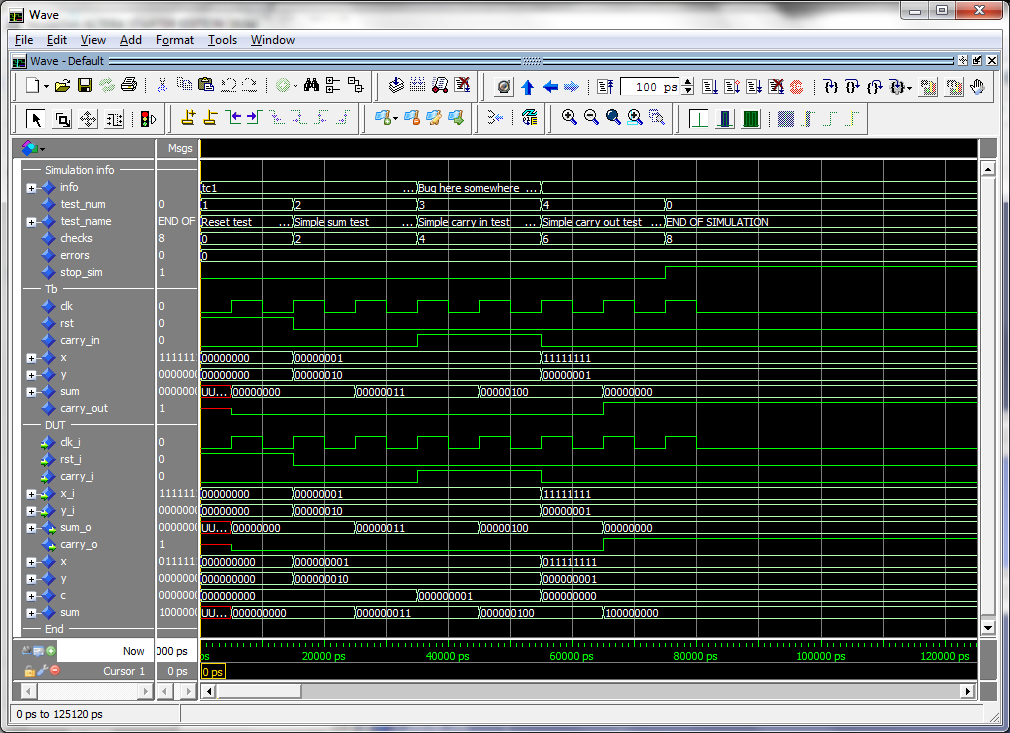
In the ModelSim transcript window, type

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 PlTbUtils 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 PlTbUtils. For a complete list, see the reference section.

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 PlTbUtils' global status and control signal. Call startsim() only once.

Arguments:

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

pltbutils\_sc PlTbUtils' 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 PlTbUtils' 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 PlTbUtils' 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:

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

txt The text.

pltbutils\_sc PlTbUtils' global status- and control signal of type pltbutils\_sc\_t.  
 The name must be no other than pltbutils\_sc.

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 PlTbUtils' 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, 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.

pltbutils\_sc PlTbUtils' 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 expresson: check("Counter after data burst", cnt\_o > 10, pltbutils\_sc);

### 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. |

| **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. |