# Lecture 5: Aldec Active-HDL Simulator

## 1. Objective

The objective of this tutorial is to introduce you to Aldec's Active-HDL 9.1 Student Edition simulator by performing the following tasks on a 4-bit adder design example:

- Create a new design or add .vhd files to your design
- Compile and debug your design
- Run Simulation

*Notes:*

- *This "lecture" is based on Lab 1 of EE-459/500 HDL Based Digital Design with Programmable Logic. We'll adapt this presentation to the Modelsim simulator in class. Principles are the same.*
- *Active-HDL is an alternative simulator to Xilinx's ISim (ISE Simulator) simulator. It is one of the most popular commercial HDL simulators today. It is developed by Aldec. In this course, we use the free student version of Active-HDL, which has some limitations (file sizes and computational runtime). You can download and install it on your own computer:* [http://www.aldec.com/en/products/fpga_simulation/active_hdl_student](http://www.aldec.com/en/products/fpga_simulation/active_hdl_student)

## 2. Introduction

Active-HDL is a Windows based integrated FPGA Design Creation and Simulation solution. Active-HDL includes a full HDL graphical design tool suite and RTL/gate-level mixed-language simulator. It supports industry leading FPGA devices, from Altera, Atmel, Lattice, Microsemi (Actel), Quicklogic, Xilinx and more.

The core of the system is an HDL simulator. Along with debugging and design entry tools, it makes up a complete system that allows you to write, debug and simulate VHDL code. Based on the concept of a workspace (think of it as of design), Active-HDL allows us to organize your VHDL resources into a convenient and clear structure.

## 3. Procedure

### *Creating the 1-bit full adder*

1. Start Aldec Active-HDL: Start->All Programs->Aldec->Active-HDL Student Edition
2. Select "Create New Workspace" and click OK
3. Enter **fall2012_aldec** as the name of the workspace and change the directory to where you want to save it (for example M:\UB\labs) and click OK
4. Select "Create an Empty Design" and click NEXT
5. Choose the block diagram configuration as "Default HDL Language" and default HDL language as "VHDL". Select the target technology as Xilinx for vendor and SPARTAN6 for technology. Click NEXT
6. Enter **fourbit_adder** as the name of the design as well as the name of the default working library. Click NEXT
7. Click FINISH

You should have now the Design Browser as a window showing current workspace and design contents.

8. Double-click on "Add New File" in the Design Browser window

9. Select “VHDL Source Code” and type in **full_adder** in the name field, click OK

The following is the VHDL code for the 1-bit full adder. Enter the code as seen below into the empty file.

```
-- 1-bit full adder
-- Declare the 1-bit full adder with the inputs and outputs
-- shown inside the port(). This adds two bits together (x,y)
-- with a carry in (cin) and outputs the sum (sum) and a
-- carry out (cout).

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder is
        port(x, y, cin: in std_logic;
        sum, cout: out std_logic);
end full_adder;

architecture my_dataflow of full_adder is

begin

sum <= (x xor y) xor cin;
cout  <= (x and y) or (x and cin) or (y and cin);

end my_dataflow;
```

10. Select the File menu and choose Save.
11. To check sintax of the newly created adder, right click on “full_adder.vhd” in the Design Browser window and select the Compile option. The code should compile without any problems and you should see a green check mark next to the full_adder.vhd file. If you get any errors, check the code that you have typed against the above code provided.

Once you have the source file (or all the source files of the entire design) compiled, the design can be simulated for functional correctness.

***Manual Simulation***

***Note:*** *This type of simulation should be done only for small designs with few inputs and outputs. As design size increases you should use testbenches – described later.*

1. Select menu Simulation->Initialize Simulation

After the simulation has been initialized, you have to open a new Waveform window.

2. Click the New Waveform toolbar button to invoke the Waveform window.

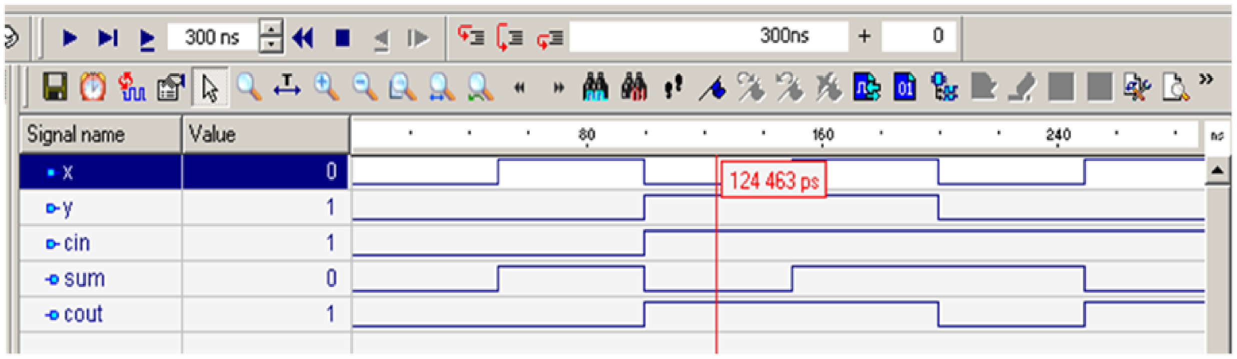
Now you need assign the stimulators to all the input signals.

3. In the Design Browser window select all signals (one by one by holding Control key pressed), then right click and choose Add to Wavefom

***Note:*** *To add signals to the simulator we could also use the drag and drop feature. In the Structure pane/tab of the Design Browser window, select the design and while holding down the left button, drag it to* *the right-section of the Waveform window and then release the mouse button. This is a standard drag-and-drop operation.*

4. Go to the left pane/tab of the Waveform Editor window and select the “x” signal. Press the right button to invoke a context menu, choose the “x” item from Stimulators… dialog; choose Clock for Type. Leave the Frequency at the default value of 10 MHz. Click APPLY and then CLOSE
5. Repeat step 4 for input “y”. Choose Clock for Type but this time place the mouse pointer in the Frequency box and set the value of 5 MHz. Click the APPLY button to assign the stimulator then CLOSE.
6. Repeat step 4 for input “cin”. Choose Formula for Type and when the dialog appears, type formula expression as follows: 0 0, 1 100000. Click APPLY and then CLOSE
7. Simulation->Run Until and enter 300ns
8. Finish simulation by selecting the Simulation->End Simulation option in the Simulation menu

At this time your Waveform viewer should look like this:

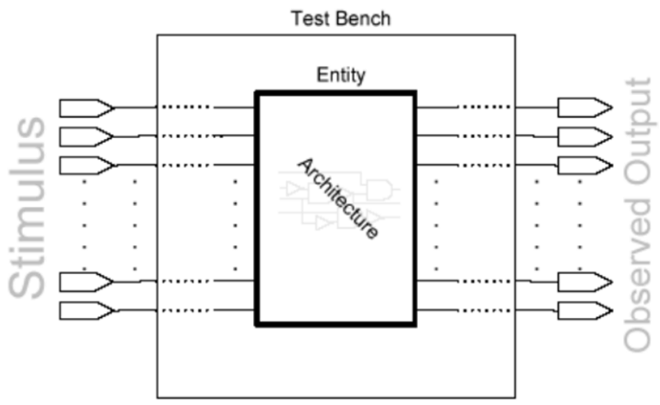


| Signal name | Value |
|---|---|
| x | 0 |
| y | 1 |
| cin | 1 |
| sum | 0 |
| cout | 1 |

Investigate the waveforms to verify that your full_adder works correctly.

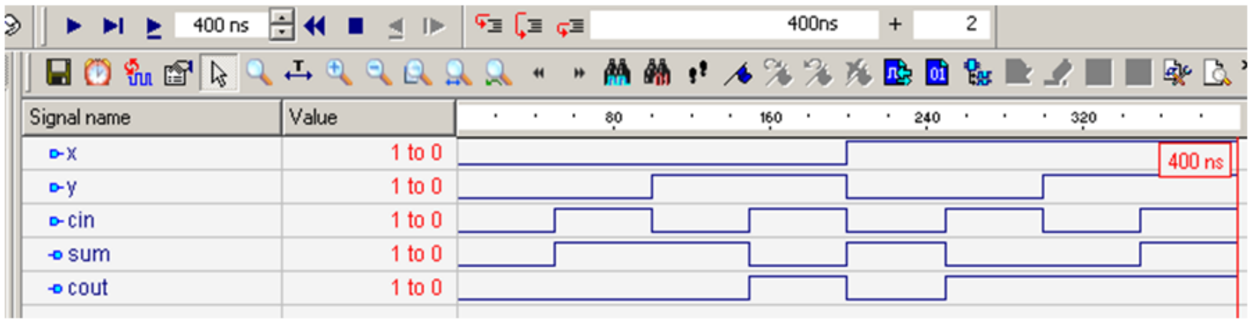
***Testbench Based Simulation***

The VHDL **testbench** is a VHDL program that describes simulation inputs in standard VHDL language. There is a wide variety of VHDL specific functions and language constructs designed to create simulation inputs. You can read the simulation data from a text file, create separate processes driving input ports, and more. The typical way to create a testbench is to create an additional VHDL file for the design that treats your actual VHDL design as a component (Design Under Test, DUT) and assigns specific values to this component input ports. It also monitors the output response of the DUT to verify correct operation.
The diagram below illustrates the relationship between the entity, architecture, and testbench:



1. Create a new file **full_adder_testbench.vhd** and save it under the current design's "src" directory (for example M:\UB\labs\fall2012_aldec\fourbit_adder\src). The content of this file is in the Appendix A at the end of this tutorial. You can create it using Aldec's editor or any other editor (e.g., even Notepad).
2. Select the Design menu and choose "Add Files to Design" and add the newly created full_adder_testbench.vhd to the design.
3. Right click on full_adder_testbench.vhd in the Design Browser window and select the Compile option.
4. Left click on the plus (+) next to full_adder_testbench.vhd. This will bring us the TEST_FULL_ADDER entity
5. Right click on the TEST_FULL_ADDER and choose Set as Top-Level
6. Select the File menu and choose the New option and pick New Waveform
7. In the Design Browser window select the Structure pane/tab at the bottom of the window
8. Select the Simulation menu and choose Initialize Simulation
9. Click on "+" next to TEST_FULL_ADDER (MY_TEST)
10. Click on U1:FULL_ADDER and drag all signals to the waveform window
11. Change the time for simulation to 400 ns by clicking on the up arrow
12. Select the Simulation menu and choose Run For
13. Inspect the simulation to verify that the 1-bit full adder functionality is indeed correct

At this time your Waveform viewer should look like this:

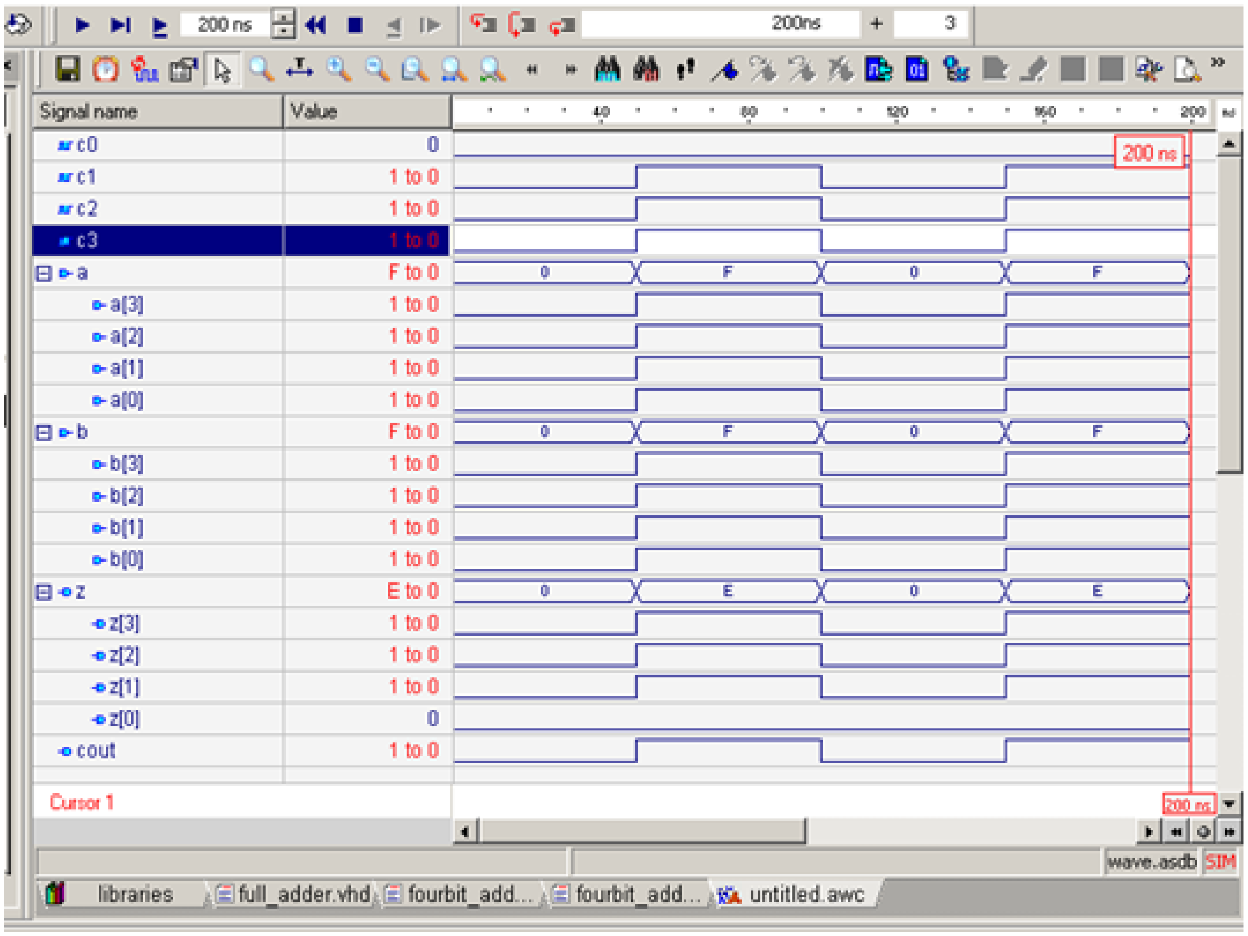


| Signal name | Value |
|---|---|
| x | 1 to 0 |
| y | 1 to 0 |
| cin | 1 to 0 |
| sum | 1 to 0 |
| cout | 1 to 0 |

***Creating and testing the 4-bit adder***

1. Add the following two files to the design: **fourbit_adder.vhd** and **fourbit_adder_testbench.vhd**. Their source code is in Apendices B and C at the end of this tutorial.
2. Compile both files and use the testbench (fourbit_adder_testbench.vhd) to simulate the design for say 200 ns
3. View the simulation to verify that the 4-bit adder functionality is correct.

At this time your Waveform viewer should look like this:



| Signal name | Value |
|---|---|
| c0 | 0 |
| c1 | 1 to 0 |
| c2 | 1 to 0 |
| c3 | 1 to 0 |
| a | F to 0 |
| a[3] | 1 to 0 |
| a[2] | 1 to 0 |
| a[1] | 1 to 0 |
| a[0] | 1 to 0 |
| b | F to 0 |
| b[3] | 1 to 0 |
| b[2] | 1 to 0 |
| b[1] | 1 to 0 |
| b[0] | 1 to 0 |
| z | E to 0 |
| z[3] | 1 to 0 |
| z[2] | 1 to 0 |
| z[1] | 1 to 0 |
| z[0] | 0 |
| cout | 1 to 0 |
| Cursor 1 | 200 ns |

## 4. Taking it further

While intuitive to use, Active-HDL has a lot of features. It is outside the scope of this tutorial to discuss all of them. You should spend some time searching and reading additional documentation on how to use Active-HDL. A few first examples:

- http://www.aldec.com/en/downloads/tutorials
- Once you launched Active-HDL tool select the Help menu and read stuff
- Google for "Active-HDL tutorial". You will find a lot of detailed tutorials (some written for older versions of the tool but a lot of concepts still apply), which have been kindly made public by the online community.

***Note:*** *As it is the case with most of the electronic design automation (EDA) tools, there are multiple ways of achieving or performing something. If by reading the documentation or other tutorials you learn how to accomplish any of the steps described in this tutorial in a different way - that is OK. You should learn and use the methods you like the most and are more comfortable with.*

Finally, while Active-HDL (of Aldec) and ModelSim (of Mentor Graphics) are arguably some of the most popular HDL simulators in industry, Xilinx has been improving their own simulator, ISim, which is part of the free ISE WebPack used in this course. You can read more about iSim here:

- http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/plugin_ism.pdf

## **Appendix A**: VHDL source code of **full_adder_testbench.vhd**

```
-- 1-bit full adder testbench
-- A testbench is used to rigorously tests a design that you have made.
-- The output of the testbench should allow the designer to see if
-- the design worked. The testbench should also report where the testbench
-- failed.

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Declare a testbench. Notice that the testbench does not have any input
-- or output ports.
entity TEST_FULL_ADDER is
end TEST_FULL_ADDER;

-- Describes the functionality of the tesbench.
architecture MY_TEST of TEST_FULL_ADDER is

        -- The object that we wish to test is declared as a component of
        -- the test bench. Its functionality has already been described elsewhere.
        -- This simply describes what the object's inputs and outputs are, it
        -- does not actually create the object.
        component FULL_ADDER
                port( x, y, cin : in  STD_LOGIC;
                sum, cout : out STD_LOGIC );
        end component;

        -- Specifies which description of the adder you will use.
        for U1: FULL_ADDER use entity WORK.FULL_ADDER(MY_DATAFLOW);

        -- Create a set of signals which will be associated with both the inputs
        -- and outputs of the component that we wish to test.
        signal X_s, Y_s        : STD_LOGIC;
        signal CIN_s           : STD_LOGIC;
        signal SUM_s           : STD_LOGIC;
        signal COUT_s          : STD_LOGIC;

        -- This is where the testbench for the FULL_ADDER actually begins.
        begin

        -- Create a 1-bit full adder in the testbench.
        -- The signals specified above are mapped to their appropriate
        -- roles in the 1-bit full adder which we have created.
        U1: FULL_ADDER port map (X_s, Y_s, CIN_s, SUM_s, COUT_s);

        -- The process is where the actual testing is done.
        process
        begin

                -- We are now going to set the inputs of the adder and test
                -- the outputs to verify the functionality of our 1-bit full adder.

                -- Case 0 : 0+0 with carry in of 0.

                -- Set the signals for the inputs.
                X_s <= '0';
                Y_s <= '0';
                CIN_s <= '0';

                -- Wait a short amount of time and then check to see if the
                -- outputs are what they should be. If not, then report an error
                -- so that we will know there is a problem.
                wait for 10 ns;
```

```
assert ( SUM_s = '0'  ) report "Failed Case 0 - SUM" severity error;
assert ( COUT_s = '0' ) report "Failed Case 0 - COUT" severity error;
wait for 40 ns;

-- Carry out the same process outlined above for the other 7 cases.

-- Case 1 : 0+0 with carry in of 1.
X_s <= '0';
Y_s <= '0';
CIN_s <= '1';
wait for 10 ns;
assert ( SUM_s = '1'  ) report "Failed Case 1 - SUM" severity error;
assert ( COUT_s = '0' ) report "Failed Case 1 - COUT" severity error;
wait for 40 ns;

-- Case 2 : 0+1 with carry in of 0.
X_s <= '0';
Y_s <= '1';
CIN_s <= '0';
wait for 10 ns;
assert ( SUM_s = '1'  ) report "Failed Case 2 - SUM" severity error;
assert ( COUT_s = '0' ) report "Failed Case 2 - COUT" severity error;
wait for 40 ns;

-- Case 3 : 0+1 with carry in of 1.
X_s <= '0';
Y_s <= '1';
CIN_s <= '1';
wait for 10 ns;
assert ( SUM_s = '0'  ) report "Failed Case 3 - SUM" severity error;
assert ( COUT_s = '1' ) report "Failed Case 3 - COUT" severity error;
wait for 40 ns;

-- Case 4 : 1+0 with carry in of 0.
X_s <= '1';
Y_s <= '0';
CIN_s <= '0';
wait for 10 ns;
assert ( SUM_s = '1'  ) report "Failed Case 4 - SUM" severity error;
assert ( COUT_s = '0' ) report "Failed Case 4 - COUT" severity error;
wait for 40 ns;

-- Case 5 : 1+0 with carry in of 1.
X_s <= '1';
Y_s <= '0';
CIN_s <= '1';
wait for 10 ns;
assert ( SUM_s = '0'  ) report "Failed Case 5 - SUM" severity error;
assert ( COUT_s = '1' ) report "Failed Case 5 - COUT" severity error;
wait for 40 ns;

-- Case 6 : 1+1 with carry in of 0.
X_s <= '1';
Y_s <= '1';
CIN_s <= '0';
wait for 10 ns;
assert ( SUM_s = '0'  ) report "Failed Case 6 - SUM" severity error;
assert ( COUT_s = '1' ) report "Failed Case 6 - COUT" severity error;
wait for 40 ns;

-- Case 7 : 1+1 with carry in of 1.
X_s <= '1';
Y_s <= '1';
CIN_s <= '1';
```

```
		wait for 10 ns;
		assert ( SUM_s = '1'  ) report "Failed Case 7 - SUM" severity error;
		assert ( COUT_s = '1' ) report "Failed Case 7 - COUT" severity error;
		wait for 40 ns;

	end process;
END MY_TEST;
```

## **Appendix B**: VHDL source code of **fourbit_adder.vhd**

```
-- 4-bit adder
-- Structural description of a 4-bit adder. This device
-- adds two 4-bit numbers together using four 1-bit full adders
-- described above.

-- This is just to make a reference to some common things needed.
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- This describes the black-box view of the component we are
-- designing. The inputs and outputs are again described
-- inside port(). It takes two 4-bit values as input (x and y)
-- and produces a 4-bit output (ANS) and a carry out bit (Cout).

entity fourbit_adder is
	port( a, b     : in    STD_LOGIC_VECTOR(3 downto 0);
	      z : out   STD_LOGIC_VECTOR(3 downto 0);
	      cout     : out   STD_LOGIC );
end fourbit_adder;

-- Although we have already described the inputs and outputs,
-- we must now describe the functionality of the adder (ie:
-- how we produced the desired outputs from the given inputs).

architecture MY_STRUCTURE of fourbit_adder is

-- We are going to need four 1-bit adders, so include the
-- design that we have already studied in full_adder.vhd.

component FULL_ADDER
	port( x, y, cin        : in  STD_LOGIC;
	      sum, cout        : out STD_LOGIC );
end component;

-- Now create the signals which are going to be necessary
-- to pass the outputs of one adder to the inputs of the next
-- in the sequence.
signal c0, c1, c2, c3 : STD_LOGIC;
begin

c0 <= '0';
b_adder0: FULL_ADDER port map (a(0), b(0), c0, z(0), c1);
b_adder1: FULL_ADDER port map (a(1), b(1), c1, z(1), c2);
b_adder2: FULL_ADDER port map (a(2), b(2), c2, z(2), c3);
b_adder3: FULL_ADDER port map (a(3), b(3), c3, z(3), cout);

END MY_STRUCTURE;
```

## **Appendix C**: VHDL source code of **fourbit_adder_testbench.vhd**

```
-- 4-bit Adder Testbench
-- A testbench is used to rigorously tests a design that you have made.
-- The output of the testbench should allow the designer to see if
-- the design worked.  The testbench should also report where the testbench
-- failed.

-- This is just to make a reference to some common things needed.

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Declare a testbench.  Notice that the testbench does not have any
-- input or output ports.

entity TEST_FOURBIT_ADDER is
end TEST_FOURBIT_ADDER;

-- Describes the functionality of the tesbench.

architecture MY_TEST of TEST_FOURBIT_ADDER is

        component fourbit_adder
                port( a, b     : in    STD_LOGIC_VECTOR(3 downto 0);
                      z        : out   STD_LOGIC_VECTOR(3 downto 0);
                      cout     : out   STD_LOGIC);
        end component;

        for U1: fourbit_adder use entity WORK.FOURBIT_ADDER(MY_STRUCTURE);
        signal a, b    : STD_LOGIC_VECTOR(3 downto 0);
        signal z       : STD_LOGIC_VECTOR(3 downto 0);
        signal cout    : STD_LOGIC;

        begin
        U1: fourbit_adder port map (a,b,z,cout);

                process
                begin

                -- Case 1 that we are testing.

                        a <= "0000";
                        b <= "0000";
                        wait for 10 ns;
                        assert ( z = "0000" ) report "Failed Case 1 - z" severity error;
                        assert ( Cout = '0' ) report "Failed Case 1 - Cout" severity error;
                        wait for 40 ns;

                -- Case 2 that we are testing.

                        a <= "1111";
                        b <= "1111";
                        wait for 10 ns;
                        assert ( z = "1110" )  report "Failed Case 2 - z" severity error;
                        assert ( Cout = '1' )   report "Failed Case 2 - Cout" severity error;
                        wait for 40 ns;

                end process;
END MY_TEST;
```