Advanced Digital Design
Lecture 13
Design Verification Overview

1 Introduction

In this lecture we will look in detail at the use of simulation to verify whether the design works as required.

2 Test benches

A test bench is a *simulation* model of the entire universe around a design. You place your design (through an instantiation statement) into a testbench. The testbench supplies the appropriate inputs, and the simulator displays the resulting outputs. Often this is regarded as a sufficient definition of a testbench.

However, a proper testbench has an additional feature. Throughout the simulation, it should know what the correct outputs ought to be, compare the outputs from your design with the correct outputs, and notify you of any discrepancies. The general appearance of a VHDL test bench is as follows:

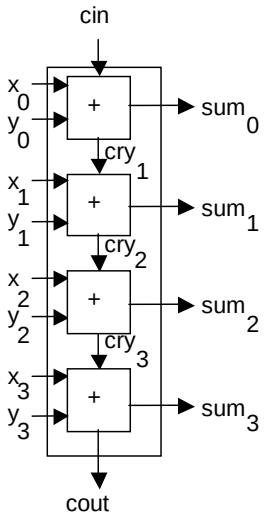
```
ENTITY testbench IS
END ENTITY testbench;

ARCHITECTURE tb OF testbench IS
  Declare all the test signals that will be connect to
  the inputs and outputs of the device we are testing
BEGIN

  Place one copy of the design under test, and wire
  its inputs and outputs up to test signals

  Generate test waveforms that are applied to the inputs

  Observe the outputs from the design and compare them with
  the required results.
  Report if any discrepancies are found.

END ARCHITECTURE tb;
```

The test bench represents “the entire universe” around our design, so the test bench does not have any inputs or outputs, and the test signal are declared within the test bench as local signals. The `ENTITY` declaration therefore contains no port map.

The style of VHDL that is used in a test bench is often quite different from what would be used within the design of a piece of hardware. This is because the hardware description will ultimately be fed to a synthesis tool, and the design must use only the features of the VHDL language that will synthesize well. By contrast, the test bench model of the outside world is never going to be synthesized, so all of the many complicated features of VHDL language can be used.

2.1 An example to be verified

As an example, consider the 4-bit adder circuit that we used in lecture 4.




```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY adder IS
  PORT ( x, y: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
           cin:  IN STD_LOGIC;
           sum: OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
          cout: OUT STD_LOGIC);
END ENTITY adder;

ARCHITECTURE structural OF adder IS
  SIGNAL cry: STD_LOGIC_VECTOR(4 DOWNTO 0);
BEGIN
  c0:  entity work.fulladd(dataflow)
       PORT MAP (x(0),y(0),cin,sum(0),cry(1));
  c1:  entity work.fulladd(dataflow)
       PORT MAP (x(1),y(1),cry(1),sum(1),cry(2));
  c2:  entity work.fulladd(dataflow)
       PORT MAP (x(2),y(2),cry(2),sum(2),cry(3));
  c3:  entity work.fulladd(dataflow)
       PORT MAP (x(3),y(3),cry(3),sum(3),cout);
END ARCHITECTURE structural;
```

The design of the adder has already been compiled to the library element

```
work.adder(structural)
```

We want to find out if the design is has the correct function, i.e. whether the output truly is the sum of the inputs.

2.2 The ASSERT statement

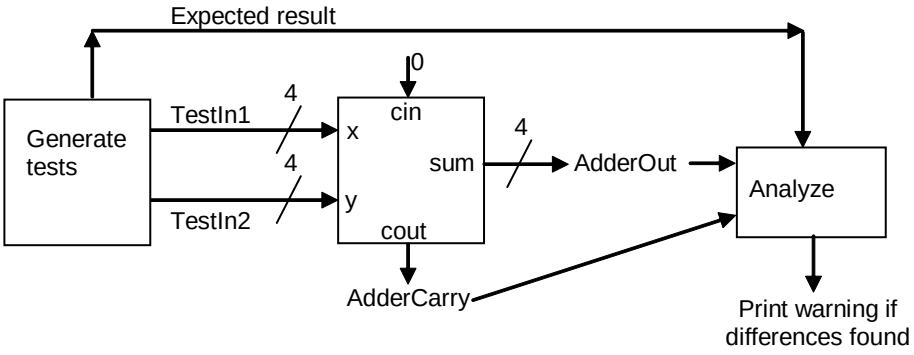
The ASSERT statement provides a way to tell the VHDL tools what conditions we believe ought to be true if the design is functioning correctly. It has the syntax

```
ASSERT condition REPORT message SEVERITY severity.
```

The *condition* shows what should be happening. If the condition is false, then a message is printed, and an error is generated of the stated severity. The severity can be *NOTE*, *WARNING*, *ERROR*, or *FAILURE*. An error of severity *ERROR* or *FAILURE* will cause termination of the simulation. *NOTE* and *WARNING* are simply reported on the output console and simulation continues

2.3 A test bench for the 4-bit adder

Now we will develop a test bench for the 4-bit adder that was designed in earlier lectures. The general idea is illustrated in the diagram below.



Two 4-bit inputs will be generated called *TestIn1* and *TestIn2*. These will apply every possible combination of 4-bit numbers to the *x* and *y* inputs of the adder. The value of the *sum* and *carry* outputs of the adder will be compared to our knowledge of what the outputs should be if the adder is functioning correctly. Any discrepancies will be reported during the simulation. The code for this arrangement is as follows

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY addtest IS
END ENTITY addtest;

ARCHITECTURE tb OF addtest IS
  SIGNAL clock: std_logic:='0';
  --Declarations of test inputs and outputs
  SIGNAL TestIn1: STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL TestIn2: STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL AdderOut: STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL AdderCarry: STD_LOGIC;
  SIGNAL ExpectedResult: STD_LOGIC_VECTOR(4 DOWNTO 0);
BEGIN

clock <= NOT clock AFTER 10 NS;

-- Place one instance of test generation unit
TG:  ENTITY work.test_generator(tb)
     PORT MAP ( clock=>clock, TestIn1=>TestIn1, TestIn2=>TestIn2,
                ExpectedResult=>ExpectedResult);

-- Place one instance of the Unit Under Test
UUT:  ENTITY work.adder(structural)
      PORT MAP ( x=>TestIn1, y=>TestIn2, cin=>'0',
                 sum=>AdderOut, cout=>AdderCarry );

-- Place one instance of the result analyzer
RA:  ENTITY work.result_analyzer(tb)
     port map (clock=>clock, TestIn1=>TestIn1, TestIn2=>TestIn2,
     ExpectedResult=>ExpectedResult, ActualAdd=>AdderOut,
     ActualCarry=>AdderCarry);

END ARCHITECTURE tb;
```

The test generator is as follows:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY test_generator IS
  PORT ( clock: IN STD_LOGIC;
       TestIn1:  OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
       TestIn2: OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
       ExpectedResult: OUT STD_LOGIC_VECTOR(4 DOWNTO 0));
END ENTITY test_generator;

ARCHITECTURE tb OF test_generator IS
BEGIN
  PROCESS
  BEGIN
```

```
    FOR I IN 0 TO 15 LOOP
      FOR J IN 0 TO 15 LOOP
        -- Set the inputs to the adder
        TestIn1 <= CONV_STD_LOGIC_VECTOR(i,4);
        TestIn2 <= CONV_STD_LOGIC_VECTOR(j,4);
        -- Calculate what the output of the adder should be
        ExpectedResult <= CONV_STD_LOGIC_VECTOR(i+j,5);
        -- Wait until adder output has settled
        WAIT until rising_edge(clock);
      END LOOP;
    END LOOP;
    WAIT;
  END PROCESS;
END ARCHITECTURE tb;
```

A nested pair of loops[1] will cause two integer variables i and j to take values ranging from 0 to 15. These values will be converted into 4-bit STD_LOGIC_VECTORs using a conversion function that is found in the STD_LOGIC_ARITH package and these values are applied to the adders inputs through the signals *TestIn1* and *TestIn2*.

The expected output is calculated using the following line of code:

```
ExpectedResult <= CONV_STD_LOGIC_VECTOR(i+j,5);
```

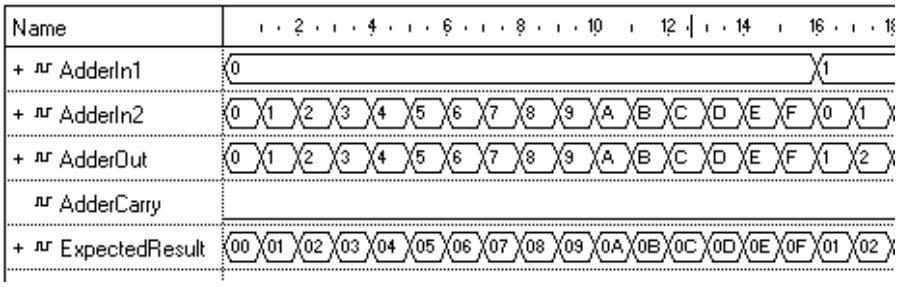
The values of i and j are added together and converted to a 5-bit STD_LOGIC_VECTOR. Bit 4 of ExpectedResult tells us what the carry output from the adder should be and bits 3 DOWNTO 0 tell us what the sum output should be. The result analyzer looks like this:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY result_analyzer IS
  PORT ( clock: IN STD_LOGIC;
       TestIn1:  IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       TestIn2: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       ExpectedResult: IN STD_LOGIC_VECTOR(4 DOWNTO 0);
       ActualAdd: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       ActualCarry: IN STD_LOGIC);
END ENTITY result_analyzer;

ARCHITECTURE tb OF result_analyzer IS
BEGIN

  PROCESS(clock)
  BEGIN
    IF rising_edge(clock) THEN
      -- Check whether adder output matches expectation
      ASSERT   ExpectedResult(3 DOWNTO 0) =  ActualAdd
               and ExpectedResult(4) = ActualCarry
      REPORT   "Adder output is incorrect"
      SEVERITY WARNING;
    END IF;
  END PROCESS;
END ARCHITECTURE tb;
```

[1] In VHDL it is not necessary to declare the type of a loop index (i and j in this example). Instead, their type is inferred automatically from the range (0 TO 15), which dictates that they must be integers.

At each rising edge of the clock, the expected results will be compared with the actual results. If any discrepancy is found, a warning will be issued.

If we run the code through the simulator, the result for the first 18 test inputs is as shown below.



In total 256 test inputs will be applied. The simulator shows how the expected output compares with the adder output. During simulation if any discrepancies are found between the expected values and the observed values a warning message is generated.

3 Code coverage

When we have run the test bench simulation, we know whether or not the results were as expected. However, it may be that for our particular choice of inputs for the test bench, some parts of our VHDL code were not executed. Any bugs in these portions of the VHDL would not have been detected. In order to address this problem, many simulators provide reports on *code coverage*, and show which portions of the code were executed during the simulation and which were not.

However, even identifying which lines of code were not exercised by the simulator is still not sufficient. Suppose we have a line such as

```
c <='1' WHEN a>b ELSE '0';
```

It could be that our set of test inputs always causes the condition a>b to be true. As a result, *even though this line of code has been executed*, we still have not checked for bugs that might manifest themselves only when this line of code sets c to '0'. A simulator that provides *expression coverage* reports would be able to inform the user that this line of code had only been executed with the WHEN clause evaluating to true, and not with the clause evaluating to false.

8 Summary

In this lecture we have looked

- Functional verification
- The ASSERT statement and Assertion based verification
- Code coverage and expression coverage.