Advanced Digital Design
Lecture 6
Testbenches and Generic Parameters in VHDL

1 Introduction

In this lecture we will look at two important issues. Firstly, we will see how to construct a *testbench*, an environment (described in VHDL) in which you can apply test inputs to your design in order to find out if it functions as expected. The second issue is the description of devices that are n-bits wide.

2 Test benches

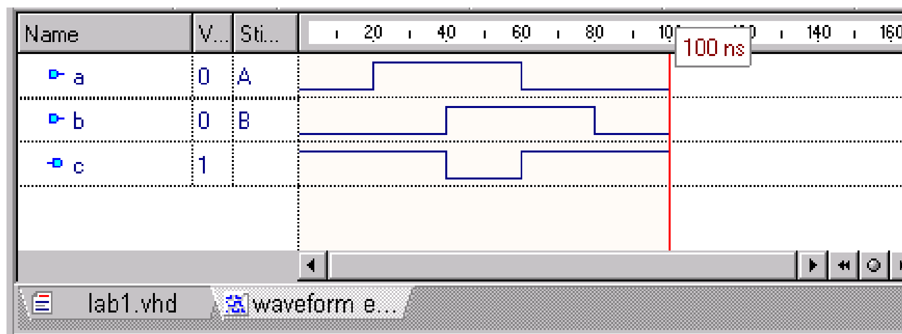
In lab 1, we used a simple example of describing a NAND gate, and then applying signals to its inputs. In order to apply the inputs, we used the features of the Active HDL simulator. This way of doing things has two disadvantages:

- It is time consuming to set up signals, and there are awkward limitations to how you can input values.
- It is non-portable. If you moved to a different VHDL simulator, there is no way to carry over the stimulator instructions that are specific to Active HDL.

For these reasons, it is advantageous to be able to describe your tests in VHDL. In the first section of this lecture, we will look through an example of how to do this. We will start off by re-visiting the NAND gate listing that we saw in lecture 2 and lab 1:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY nandgate IS
      PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC );
END ENTITY nandgate;

ARCHITECTURE simple OF nandgate IS
BEGIN
      c <= a NAND b;
END ARCHITECTURE simple;
```

Here are the test inputs that we want to apply, but we want to do this through VHDL, not through the *stimulator* feature of Active HDL:



2.1 Multiple assignments to signals

In one assignment statement, it is possible to make multiple assignments. The different values must take place at different times, and they are separated by commas. So we can write this:

```
in1 <= '0', '1' AFTER 20 NS, '0' AFTER 60 NS;
```

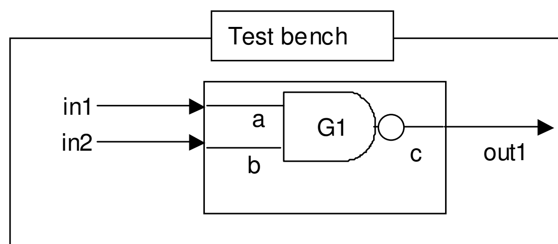
According to the normal rules of VHDL, this statement will run when a signal on its right hand side changes. But there aren't any signals there. A statement with no signals on its RHS will run exactly once at the very beginning of the simulation.

The corresponding statement for in2 is

```
in2 <= '0', '1' AFTER 40 NS, '0' AFTER 80 NS;
```

2.2 A test bench for the AND gate

What we now need to do is to apply our test inputs in1 and in2 to the NAND gate inputs a and b. It looks like this.



This illustrates a special type of description, called a *test bench*. A test bench contains an instance of the design that we want to test, together with local signals that describe the inputs that we want to apply to it. 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 generators are contained within the test bench as local signals.

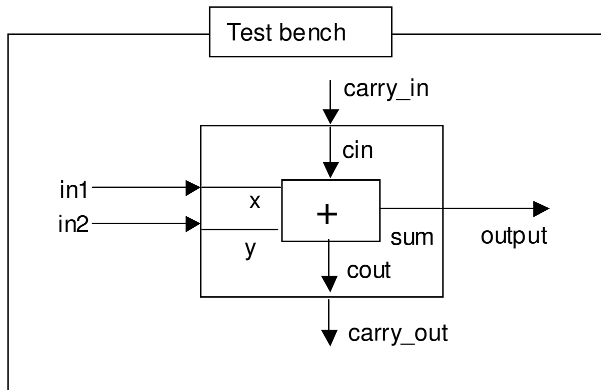
The description of this system looks like this

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mytestbench IS
END ENTITY mytestbench;

ARCHITECTURE test OF mytestbench IS
  SIGNAL in1, in2, out1: STD_LOGIC;
BEGIN
  G1: ENTITY work.nandgate(simple) PORT MAP (a=>in1, b=>in2, c=>out1);
  in1 <= '0', '1' AFTER 20 NS, '0' AFTER 60 NS;
  in2 <= '0', '1' AFTER 40 NS, '0' AFTER 80 NS;
END ARCHITECTURE test;
```

The `ENTITY` declaration for the test bench may look slightly weird, since it contains no port map. This is because it has no inputs or outputs. In order to fully describe a simulation in VHDL, it is necessary that the top level of our design has no inputs or outputs. (If it did have inputs and outputs, then we would need to think about some bigger system enclosing the design that was able to supply the required inputs and outputs.)

2.3 A test bench for our 4-bit adder

Here is another example of a test bench.



We create an instance of our 4 bit adder inside a test bench, and apply 4-bit inputs to it, and observe the results at the 4-bit outputs. Here is the code

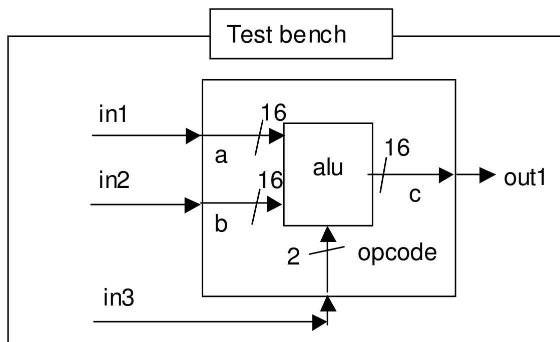
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY testbench IS
END ENTITY testbench;

ARCHITECTURE simple OF testbench IS
    SIGNAL in1, in2: STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL output: STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL carry_in, carry_out: STD_LOGIC;
BEGIN
    g1: ENTITY work.adder(structural)
        PORT MAP ( x=>in1, y=>in2, cin=>carry_in,
                   sum=>output, cout=>carry_out);

    in1 <= X"2",
           X"7" AFTER 30 NS,
           X"9" AFTER 60 NS;
    in2 <= X"5",
           X"1" AFTER 40 NS,
           X"4" AFTER 80 NS;
    carry_in <= '0';

END ARCHITECTURE simple;
```

- **A test bench for our ALU**

Now lets return to our ALU design that we introduced in lecture 2 and lab 1.

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

ENTITY alu IS
      PORT ( a, b: IN STD_LOGIC_VECTOR(15 DOWNTO 0);
             opcode: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
             c: OUT STD_LOGIC_VECTOR(15 DOWNTO 0) );
END ENTITY alu;

ARCHITECTURE simple OF alu IS
BEGIN
      c <= a + b WHEN opcode="00"
      ELSE a - b WHEN opcode="01"
      ELSE a OR b WHEN opcode="10"
      ELSE a AND b WHEN opcode="11";
END ARCHITECTURE simple;
```

We can now illustrate a simple testbench for the ALU circuit.




```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mytestbench IS
END ENTITY mytestbench;

ARCHITECTURE test OF mytestbench IS
  SIGNAL in1, in2, out1: STD_LOGIC_VECTOR (15 DOWNTO 0);
  SIGNAL in3: STD_LOGIC_VECTOR (1 DOWNTO 0);
BEGIN
  G1: ENTITY work.alu(simple)
      PORT MAP (a=>in1, b=>in2, opcode=>in3, c=>out1);
  in1 <= X"0001", X"0FAF" AFTER 20 NS, X"F000" AFTER 40 NS;
  in2 <= X"0100", X"7FFF" AFTER 10 NS, X"FFFF" AFTER 30 NS;
  in3 <= "00";
END ARCHITECTURE test;
```

We have set the opcode (in3) to "00" to test the addition function, and then fed in various values for in1 and in2. (N.B. this example is given just to show how we would set up the test bench. The values of numbers given aren't necessarily a good choice, and there are too few tests to be really confident that the design is functioning correctly.)

3 Generic parameters

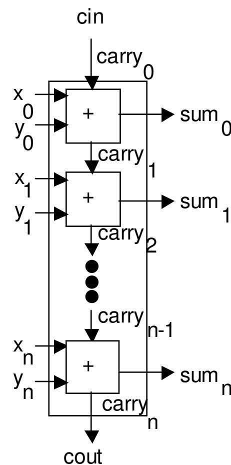
In lecture 4, we mentioned that synthesis tools have pre-designed libraries of circuits that accomplish important functions. So, for example, when a designer writes code like this

c <= a + b;

the synthesis tool simply retrieves an adder circuit from its library and connects a and b the inputs and c to the output. But some designers might be using 4-bit numbers, other 8-bit numbers, others might use 26. It would obviously be stupid if the library had to contain one 4-bit adder, one 5-bit adder, one 6-bit adder and so on up to the largest number a designer might want to use. What we need is a generic n-bit adder. Then when the adder is placed in a particular design, a value is assigned to n to make sure that it is the right width for the inputs a, b and the output c.

3.1 An n-bit adder

Here is the iterative adder extended to n-bits in length:



Here is the ENTITY for this n-bit adder

```
ENTITY nbit_adder IS
      GENERIC ( n: INTEGER );
      PORT ( x, y: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
             cin:  IN STD_LOGIC;
             sum: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0);
               cout: OUT STD_LOGIC);
END ENTITY nbit_adder;
```

This introduces a new feature: a GENERIC parameter. This is used when we have are creating a design that is in some sense generic. For this example, we are creating an adder that could be any number of bits wide.

Note that the GENERIC parameter must come first. If we had done this:

```
ENTITY wrong_adder IS
      PORT ( x, y: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
             cin:  IN STD_LOGIC;
             sum: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0);
             cout: OUT STD_LOGIC);
```

```
      GENERIC ( n: INTEGER );                           --This is wrong!
END ENTITY wrong_adder;
```

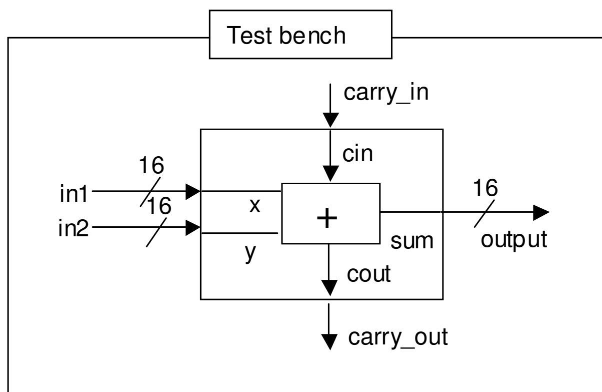
Then the description would not compile, because n has been used in the definition of x to specify its bitwidth, but n has not been declared yet.

The architecture of our n-bit adder is then a simple generalisation of the 4-bit iterative adder that we saw in lecture 4.

```
ARCHITECTURE iterative OF nbit_adder IS
    SIGNAL carry: STD_LOGIC_VECTOR(n DOWNTO 0);
BEGIN
    carry(0) <= cin;
    cout <= carry(n);
    gen1: FOR i IN 0 TO n-1 GENERATE
            g:    ENTITY work.fulladd(structural)
                PORT MAP (x(i),y(i),carry(i),sum(i),carry(i+1));
          END GENERATE gen1;
END ARCHITECTURE iterative;
```

3.2 Instantiating the n-bit adder

Of course, when we actually use the n-bit adder we have to say how many bits wide it is. So let's return to the test bench introduced in section 2.3, and place an n-bit adder, forcing the adder to be 16 bits wide.



The description is as follows:

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

ENTITY testbench IS
END ENTITY testbench;

ARCHITECTURE simple OF testbench IS
    SIGNAL in1, in2: STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL output: STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL carry_in, carry_out: STD_LOGIC;
BEGIN
    g1: ENTITY work.nbit_adder(structural)
        GENERIC MAP (16)
        PORT MAP ( x=>in1, y=>in2, cin=>carry_in,
                   sum=>output, cout=>carry_out);
```

```
    in1 <= X"2134",
           X"7A65" AFTER 30 NS,
           X"9D14" AFTER 60 NS;
    in2 <= X"5A10",
           X"1A56" AFTER 40 NS,
           X"4B32" AFTER 80 NS;
    carry_in <= '0';

END ARCHITECTURE simple;
```

When we place the adder in our design, not only do we have to say how its inputs and outputs are wired up (by giving it a PORT MAP). We also have to say what value its generic parameter should take (by giving it a GENERIC MAP).

3.3 Building in some safeguards

Let's return to the ENTITY of the generic n-bit adder

```
ENTITY nbit_adder IS
      GENERIC ( n: INTEGER );
      PORT ( x, y: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
             cin:  IN STD_LOGIC;
             sum: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0);
               cout: OUT STD_LOGIC);
END ENTITY nbit_adder;
```

When building library functions, like our generic n-bit adder, it is always wise to build in as many precautions as possible against misuse of the library. Suppose someone instantiated our design like this

```
    g1: ENTITY work.nbit_adder(structural)
        GENERIC MAP (-1)
        PORT MAP ( x=>in1, y=>in2, cin=>carry_in,
                   sum=>output, cout=>carry_out);
```

We would be in trouble. Apart from the fact that a –1 bit adder makes no sense, we also have illegal bounds on the declaration of x and y.

Instantiating the adder with a width of –1 may appear idiotic, and the library designer may be tempted to ignore this possibility, assuming that the users deserve what they get if they do something this stupid. However, library elements get used in the most unpredictable circumstances, and it is always wise to make them as robust as possible. In fact this error is hard to spot if the value to be fed into the generic map is computed by an expression, and inappropriate value are given to the expression.

Here is a safer version

```
ENTITY nbit_adder IS
      GENERIC ( n: POSITIVE:=4 );
      PORT ( x, y: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
             cin:  IN STD_LOGIC;
             sum: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0);
               cout: OUT STD_LOGIC);
END ENTITY nbit_adder;
```

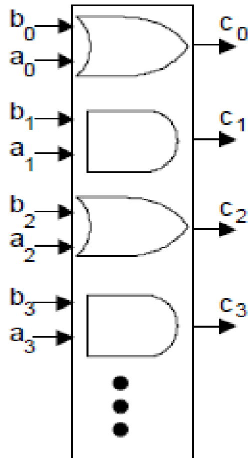
We have done two things to make the design robust:

- We have insisted that n must be a positive integer
- We have given a default width, so that if no generic parameter is specified, then a value of 4 will be used.

POSITIVE is a built-in type of VHDL that is a positive INTEGER.

- **Conditional generation**

We can, if we wish, use conditionals inside GENERATE blocks. Suppose we wanted to describe a device like this:



It is going to be n-bits wide, and every odd number position will have an AND gate, and every even position will have an OR gate. We could describe it like this:

```
ARCHITECTURE simple OF andorgate IS
BEGIN
      gen1: FOR i IN 0 TO N GENERATE

            evens: IF i MOD 2 = 0 GENERATE
                  c(i) <= a(i) OR b(i);
            END GENERATE evens;

            odds: IF i MOD 2 /= 0 GENERATE
                  c(i) <= a(i) AND b(i);
            END GENERATE odds;

      END GENERATE gen1;

END ARCHITECTURE simple;
```

Here we have used conditional generate blocks. (i MOD 2 means *the remainder when i is divided by 2*; the symbol /= means *not equal to*).
When the code is elaborated, the value of N must be known (otherwise elaboration will fail). Let's say that N=8. The code will be elaborated to

```
c(0) <= a(0) OR b(0);
c(1) <= a(1) AND b(1);
c(2) <= a(2) OR b(2);
c(3) <= a(3) AND b(3);
c(4) <= a(4) OR b(4);
```

```
c(5) <= a(5) AND b(5);
c(6) <= a(6) OR b(6);
c(7) <= a(7) AND b(7);
```

Summary
In this lecture we have seen at how to form test benches. We have also seen how to form generic devices that are n-bits wide.

You should now know...

How to form a test bench.
How to use generic parameters.
How to instantiate a device that uses generic parameters.