University of Pennsylvania
Department of Electrical and Systems Engineering
ESE171 - Digital Design Laboratory

VHDL Test Bench Tutorial

Purpose

The goal of this tutorial is to demonstrate how to automate the verification of a larger, more complicated module with many possible input cases through the use of a VHDL test bench.

Background Information

Test bench waveforms, which you have been using to simulate each of the modules you have designed so far, are quick to create and easy to use: you merely need to click on a graphical waveform to set inputs, and after running a simulation, the output values also appear on waveforms. This form of simulation has a few obvious limitations that create additional difficulty the engineer:

- You are required to validate that the output is correct yourself from a waveform, which is easy for simple circuits like a full adder, but would be more difficult for something complex like a floating-point multiplier, and is very prone to human error (or, more likely, laziness).

- You are required to manually set test cases for the inputs, which is fine for a simple combinational circuit, but for even a simple 4-bit adder, you have 8 total input bits (2 inputs, each 4 bits), corresponding to 2^8 = 256 possible input cases, and it is very time-consuming to test each one.

An option that is more commonly used among engineers working with a HDL (VHDL, Verilog) is called a "test bench". A test bench is essentially a "program" that tells the simulator (in our case, the *Xilinx ISE Simulator*, which will be referred to as ISim) what values to set the inputs to, and what outputs are expected for those inputs. Thanks to standard programming constructs like loops, iterating through a large set of inputs becomes much easier for the engineer trying to test their design.

For the purposes of this tutorial, the following VHDL elements will be discussed in the context of simulation and verification. For the impatient, an actual example of how to use all of these together is provided below, as most of these statements are fairly intuitive in practice. Note that anything surrounded in brackets is optional and does not need to be provided for the statement to be syntactically correct.

1. The **wait** statement: **wait** [sensitivity] [condition];

   This statement can take many forms. The most useful in this context is to wait for a specific amount of time, which is accomplished by the following example:

   ```
    wait for 10ns;
   ```

   This statement instructs the simulator to simulate the behavior of the module for 10ns. When testing combinational logic, you generally need to insert a "wait for X;" statement (where X is some duration of time, e.g. 10ns) in order for the simulator to calculate outputs based on the value of the inputs. Note that "wait;" by itself will just pause the simulator until the user tells it to continue manually. Additionally, for our purposes, the length of time you choose to wait does not matter, as this simulation does not account for any delay from inputs to outputs.

   There are many other ways to use the **wait** statement: one important use is clocking for sequential circuits, which will be covered in Lab 6.

2. The **report** statement: **report** string [**severity** type];

   This statement simply prints out the specified string to the Xilinx console (at the bottom of the screen). Strings are, as with many programming languages like C and Java, represented in double quotes. The line below

   ```
    report "Test completed";
   ```

   will print the string "Test completed" to the terminal (without quotes). The severity argument merely allows you to specify whether what you are reporting is a NOTE, WARNING, ERROR, or FAILURE.

3. The **assert** statement: **assert** condition [**report** string] [**severity** type];

   This statement serves as the core of our test benches. It allows us to test that values match what we expect them to be, and when they are not (the condition is false) we can also have a report statement give the user an ERROR or WARNING.

Consider the following example from a tester for a full adder (inputs of A and B, output of S and Co) that will set the inputs to A = '0' and B = '1', wait for the simulator to update the output, and then assert that both outputs are correct.

```
A <= '0';
B <= '1';
wait for 10ns;
assert S = '1' report "Error, expected 1 for S" severity ERROR;
assert Co = '0' report "Error, expected 0 for Co" severity ERROR;
```

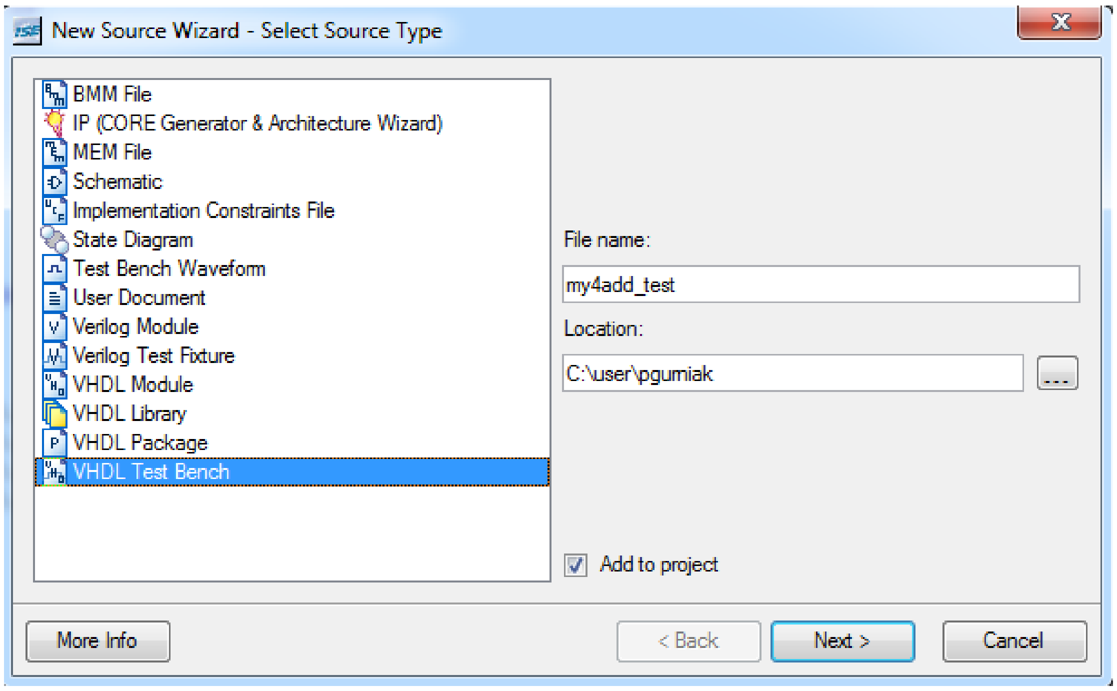
The following tutorial procedure will demonstrate how to use these statements to develop test benches for your own modules.

Tutorial Procedure

The best way to learn to write your own VHDL test benches is to see an example. For the purposes of this tutorial, we will create a test bench for the **four-bit adder** used in **Lab 4**.

*For the impatient, actions that you need to perform have key words in **bold**.*

1. With the project containing your four-bit adder open in the Xilinx ISE, right click under the sources window and select **New Source…**
2. From the list of options given, select "VHDL Test Bench". Name the new module appropriately, and select **Next**. See the image below for an example.

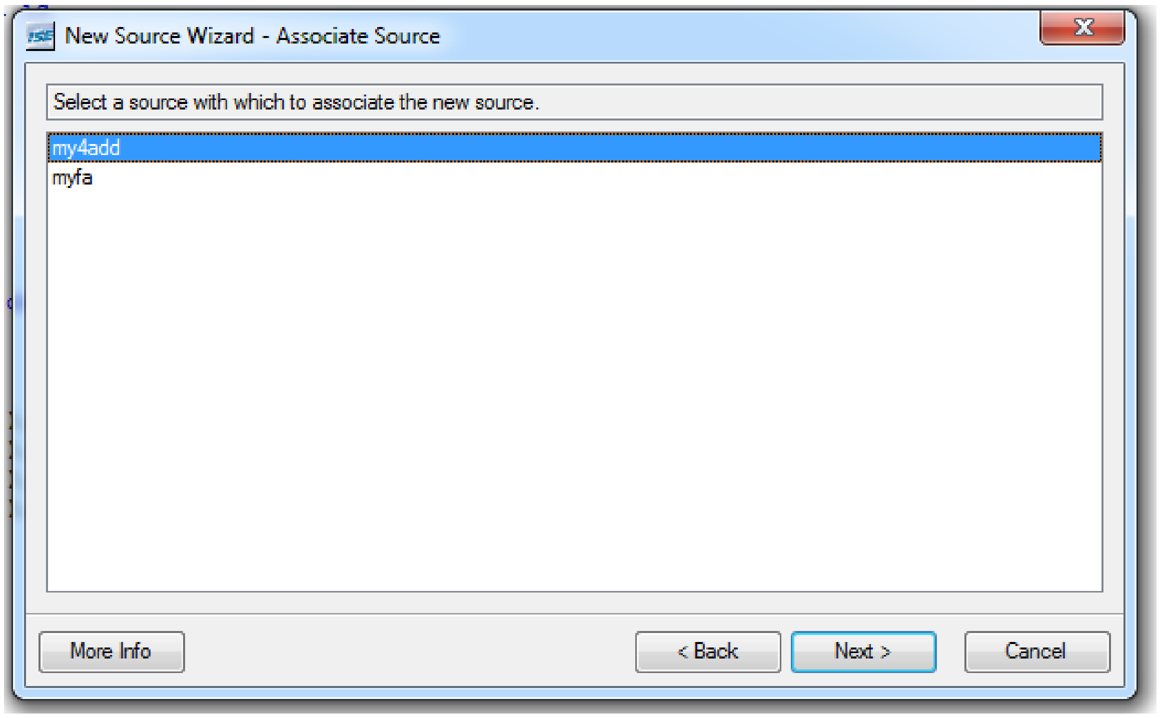


File name: my4add_test

Location: C:\user\pgumiak

Add to project ✓

3. In the next screen, select the module for your four-bit adder from the list selected, and select **Next**. In this case, the name of the four-bit adder is "my4add". See the image below for an example.



Select a source with which to associate the new source.

my4add
myfa

4. Select **Finish** on the next screen to create the test bench.
5. Take a minute to skim through the VHDL file that was created for you. It is *not* expected that you understand most of this file: quite the contrary, as you will not cover most of the elements in the auto-generated file until later in the course. You should be able to recognize a few things, though. The next few steps of this tutorial will highlight the elements of this file that are important to you now.
6. Under the *architecture* section (between ARCHITECTURE and BEGIN), you should see a declaration for the module you are trying to test. This should look something like the following:

```
ARCHITECTURE behavior OF my4add_test IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT my4add
    PORT(
         A : IN  std_logic_vector(3 downto 0);
         B : IN  std_logic_vector(3 downto 0);
         Ci : IN  std_logic;
         S : OUT  std_logic_vector(3 downto 0);
         Co : OUT  std_logic
        );
    END COMPONENT;
```

The auto-generated VHDL you see above is called a *component declaration*. When you want to use instances of other lower-level modules (either schematic or HDL) in a VHDL file, you need to explicitly declare the inputs and outputs (*ports*) in this manner.

A good analogue for a component declaration is the creation of a *schematic symbol* when using instances of your own modules in a schematic. One thing to note here is that in VHDL, a component declaration like the one shown above is *all* you need. You *do not* need to create a schematic symbol to include one of your own modules in a VHDL file (which you would need to do with schematic entry).

7. Under the behavioral description (lines following BEGIN), you should see some VHDL similar to the following:

```
BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: my4add PORT MAP (
          A => A,
          B => B,
          Ci => Ci,
          S => S,
          Co => Co
        );
```

The above code is an example of how to actually create an instance of our 4-bit adder. You will cover exactly what this does in more detail later in Lab 6. For now, suffice to say that you have access to signals A, B, and Ci that correspond to the *inputs* to the 4-bit adder, and signals S and Co that are the *outputs* from the 4-bit adder.

Also, note that the acronym **UUT** is a very common testing and validation term that stands for **unit under test**, which is the module being evaluated by the given tester.

8. Immediately after the above instantiation code, you should see short code block like the following.

```
-- *** Test Bench - User Defined Section ***
tb : PROCESS
BEGIN
   WAIT; -- will wait forever
END PROCESS;
-- *** End Test Bench - User Defined Section ***
```

This is an example of a VHDL *process*, which, for the purpose of this tutorial, will contain all of your VHDL code to simulate the four-bit adder. We will cover VHDL processes in more detail in Lab 6.

9. Add simple **wait for 100ms** and **report** commands to the Test Bench process in between the BEGIN and END PROCESS lines as shown in the following example:

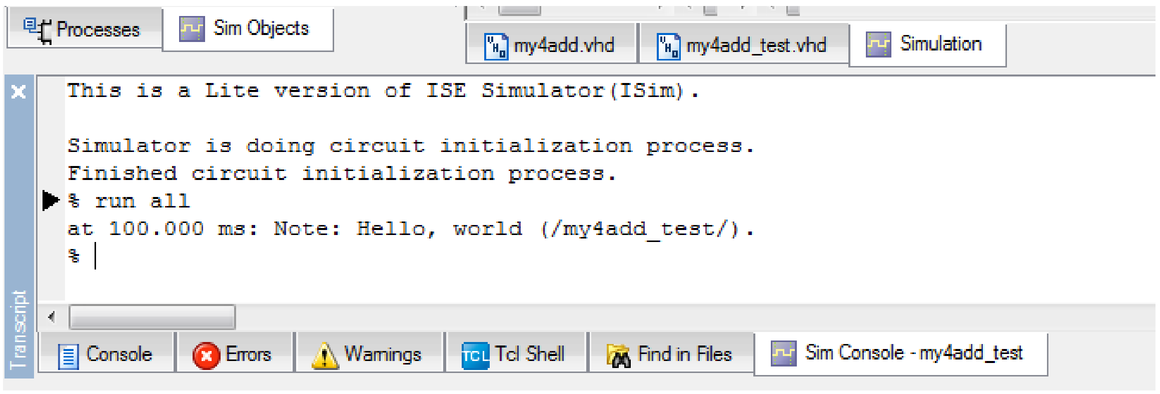
```
-- *** Test Bench - User Defined Section ***
tb : PROCESS
BEGIN
   -- Hold reset state for 100ms
   wait for 100ms;
   report "Hello, world";

   WAIT; -- will wait forever
END PROCESS;
-- *** End Test Bench - User Defined Section ***
```

10. Much like regular VHDL modules, you also have the ability to check the syntax of a VHDL test bench. With your test bench module highlighted, select **Behavioral Check Syntax** under the processes tab.

11. Now, it's time to actually execute the VHDL test bench. To do this, select **Simulate Behavioral Model** under the processes tab.

12. This will open the simulator window. Note that the first 1000ns of your simulation have been executed automatically. In order to run the remainder, click in the Sim Console at the bottom of the screen, **type** "run all" (without quotes) and press **enter**. You should see the following output:



```
This is a Lite version of ISE Simulator(ISim).

Simulator is doing circuit initialization process.
Finished circuit initialization process.
% run all
at 100.000 ms: Note: Hello, world (/my4add_test/).
%
```

13. Now it's time to make the simulator do something interesting. In order to accomplish this, we will be using a **for** loop in combination with the wait, report, and assert statements.

Add a loop similar to the following to your test bench, and re-run it.

```
-- *** Test Bench - User Defined Section ***
tb : PROCESS
BEGIN
   -- Initialize values (very important!)
   A <= "0000";
   B <= "0000";
   Ci <= '0';

   -- Loop over all values of A and check sum
   for I in 0 to 15 loop
      -- Wait for output to update
      wait for 10ns;
      -- Test output.  B is held at 0, so S = A+0 = A
      assert S = A report "Error, sum incorrect!" severity Error;
      -- Increment to next value of A
      A <= A + "0001";
   end loop;

   -- Echo to user that test is finished
   report "Test completed";

   WAIT; -- will wait forever
END PROCESS;
-- *** End Test Bench - User Defined Section ***
```

Note that the variable "I" in the VHDL above does *not* have to be declared elsewhere. Also, make note of the syntax of are the "end loop;" which terminates the loop, and the use of the '+' operator for addition of signals (other arithmetic operators such as '*' work as well!).

Warning: *do not* use a signal name (in this case, "A", "B", etc.) as the variable name in a *for* loop (in this case, "I")!

14. As with any VHDL module you write, you should now run **Behavioral Check Syntax** again to validate your changes. This will ***fail***, with an error about not knowing what to do with the "+" operator in this context. In order to fix this, we need to include a library that knows how the "+" operator works on `STD_LOGIC_VECTOR`'s. **Add the line** "`USE ieee.std_logic_unsigned.all;`" to the top of your test bench, as shown below:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
-- ADD THE FOLLOWING LINE
USE ieee.std_logic_unsigned.all;

LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;
```

Note: if *Check Syntax* does not give an error, add this line anyway!

15. Run **Behavioral Check Syntax** again to make sure the problem was fixed.

16. To ensure that this does in fact test the output of the adder (and it isn't just a script that does nothing), change the **assert** line above to: `assert S = B` … and rerun your test bench. You should now get an error on the console. Once you do, change the **assert** statement back to normal (S = A).

Now, modify your VHDL test bench code to cycle through all possible input values of **A and B** and check that the sum is correct for each case.

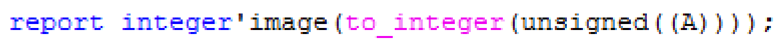
Hint: use a second "nested" for loop inside the one provided above to cycle through all 16 values of B for each value of A.

While you should try to do this on your own, a working example is provided on the next page should you get stuck.

17. *(Optional – not required)* It is important to note that there is one key feature our test bench is missing. Suppose that an error *did* occur in our module: we would be told there was an error, but we would have no idea what input conditions caused it, or what the incorrect output was, so we would have no idea how to fix it! In order to remedy this, we will modify the report line to print out the values of A and B, in addition to the incorrect sum S, when there is an error.

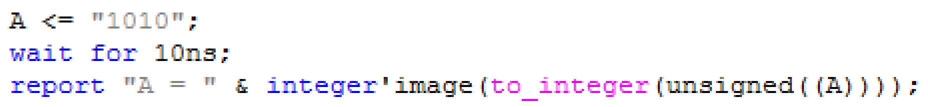
Unfortunately, printing out the value of a vector in VHDL is not straightforward. The IEEE libraries define the function **image** for most VHDL types (`std_logic`, `integer`, etc.), which generates a text string that corresponds to the printed value of that variable, but unfortunately they do not do so for the `std_logic_vector` type.

In order to get around this limitation, we will convert our `std_logic_vector` signals to `unsigned integers` using the `unsigned` and `to_integer` functions, and then use `image` function on the resulting `integer` to generate a text string. For an example of how this works, see the following line, which reports the value of the `std_logic_vector` signal A:



```
report integer'image(to_integer(unsigned((A))));
```

You can also **concatenate** strings using the "&" operator. The result is a string composed of the strings on both sides of the operator combined. For example, the following will print out "A = 10" as a note:



```
A <= "1010";
wait for 10ns;
report "A = " & integer'image(to_integer(unsigned((A))));
```

You can use these two techniques (concatenation and the `image` function) to make the error reports of your test bench as detailed as you would like.

You are *not* required to include detailed reporting code like this example in your own tests.

Completed example of a VHDL test bench for the four-bit adder:

```
-- *** Test Bench - User Defined Section ***
tb : PROCESS
BEGIN
   -- Initialize input signals
   A <= "0000";
   B <= "0000";
   Ci <= '0';

   -- Loop over all values of A
   for I in 0 to 15 loop
      -- Loop over all values of B
      for J in 0 to 15 loop
         -- Wait for output to update
         wait for 10ns;
         -- Check value of Sum
         assert (S = A + B) report "Expected sum of " &
            integer'image(to_integer(unsigned((A + B)))) & " for A = " &
            integer'image(to_integer(unsigned((A)))) & " and B = " &
            integer'image(to_integer(unsigned((B)))) & ", but was " &
            integer'image(to_integer(unsigned((S)))) severity ERROR;
         -- Increment to the next value of B
         B <= B + "0001";
      end loop;
      -- Increment to next value of A
      A <= A + "0001";
   end loop;

   -- Echo to user that test is finished
   report "Test completed";
   wait;
END PROCESS;
-- *** End Test Bench - User Defined Section ***
```