Lab-exercise

Lab6:
Reuseable Testprocedures

Introduction

The verification of hardware through simulation is key in the design of integrated circuits. ForASICs requiring a first-time-right design because of the high mask cost this is obvious. However,even if the high mask cost does not exist for FPGA's, simulation can be the only way to get a good insight in the correct functioning of the system because you can observe signals inside the system.
In practice, even for FPGA design, a set of test benches is written in VHDL to test the system's functionality. These test benches use the full potential of VHDL. The hardware itself can only make use of the synthesizable subset of VHDL.

Goals

The goal of this module is to show students the necessity for

- Re-useable code to write efficient, readable and maintainable testbenches fast;
- Self-checking testbenches. During the debug process the testbenches are rerun many times before the system is functionally correct. Checking all the outputs graphically each time the simulation has run is a lot of error-prone work. Ideally a testbench should be written such that when no error messages are generated, the functionality is ok.

Knowledge background

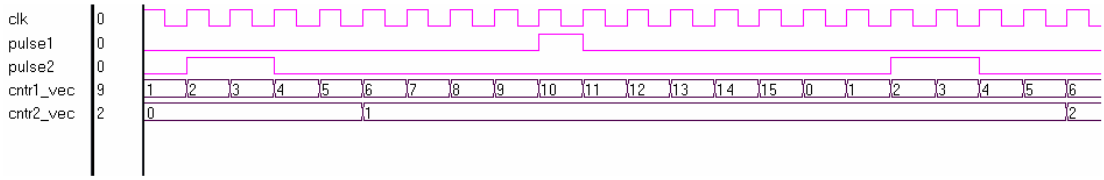
A basic knowledge of VHDL is necessary.

Classification

Difficulty on a scale of 1 to 5 : 3
Time needed (without support): 3 hours

The test environment

To write a number of widely useable testbench procedures, this lab starts from a simple, predesigned system consisting of 2 counters. A first counter counts from 0 to 15 and back to 0 on the rising clock edge. A second counter is incremented on the edge the first counter counts up from 5 to 6.
The first counter was chosen to be 5 bits wide. It's MSB hence will remain 0. The second counter is a 3-bit counter.
Using the value of the first counter two pulse signals are generated. Signal pulse1 should be high when the counter is equal to 10, whereas signal puls2 should be high when the counter is equal to 2 or 3.
We hence get a system generating outputs as can be seen in the following figure. The clock clk is the only input to the system. The other signals are outputs.



We call this small system TestHardware. The entity and architecture of TestHardware are given and can be found in the file called TestHardware.vhd.

```
entity TestHardware is
port(
clk : in std_logic;
arst : in std_logic;
pulse1 : out std_logic;
pulse2 : out std_logic;
cntr1_vec: out std_logic_vector(4 downto 0);
cntr2_vec: out std_logic_vector(2 downto 0)
);
end TestHardware;
```

Re-useable procedures

If you want to verify this system, you can of course make a testbench with a VHDL process to drive the clock, then run a simulation and check the complete output after each change.

As mentioned in the goals of this module, this is not the way to go. To simulate this example we can already define a few reusable procedures.

- GenClock : concurrent procedure driving clk
- CheckPulsWidth : concurrent procedure verifying the width of a pulse
- CheckDist : concurrent procedure verifying the time between two pulses
- CheckIfValueOccursInSim : concurrent procedure verifying if a certain value occurs on a bus

Given

- General_TB_pack.vhd : package to which the testbench procedures and functions should be added
- TestHardware.vhd : the circuit that should be verified
- TB_TestHardware.vhd : testbench with which to test TestHardware

Assignment

Starting from the file General_TB_pack.vhd, holding a package called General_TB_pack with a few widely useable procedures already, the student is supposed to write a few extra of these procedures. The testbench file

instantiating TestHardware and using a number of procedures from the General_TB_pack is also given. This allows the student to quickly check what he has written.

1. Write the body of the GenClock procedure

```
procedure GenClock ( signal clk : out std_logic;
constant startlevel: in std_logic; -- '0' | '1'
constant CLKPeriod : in time;
signal EndOfSim : in boolean);
```

This procedure generates a clock (clk) starting form an initial level with a period of CLKPeriod. When the EndOfSim signal input becomes true, the clock should halt. Once it is halted the clock should not be started up again.

As you can see in the declaration there are 2 kinds of inputs to the procedure. One kind is constant and the other kind is signal. A constant input's value is given to the procedure at the moment the procedure is called. If CLKPeriod or startlevel change during the execution of the procedure, these changes will not be seen by the procedure. It is still going to use the values that it was given when it was called.

A signal input on the contrary, can be compared to passing a reference in C. A change of this input during the execution of the procedure is also seen inside the procedure.

A concurrent procedure in VHDL in fact is equivalent to a VHDL process of which all signal inputs are in its sensitivity list. In the procedure you can also write wait statements like "wait for some_time" or "wait until rising_edge(some_input_signal). A concurrent procedure hence is kind of a special process that has as well a sensitivity list as possible wait statements. You should also know that a concurrent procedure, just as any other process, is brought into life at the start of a simulation, independent of which signals are in its sensitivity list.

To use the GenClock procedure to drive a clock clk, you have to use a loop in the body of the procedure in which the clock is first set to a start level and then, after half a period, is inverted. That way we get a procedure that never stops. We however want it to stop at the end of our simulation. Hence you have to include a check on the EndOfSim Boolean signal inside the loop. When EndOfSim becomes true, you should jump out of the loop using an exit statement.

The procedure is concurrently called in the testbench.

```
architecture behav of Testbench is
….
signal clk : std_logic;
signal EndOfSim : boolean:=false;
begin
U1 : TestHardware
port map(
…..
```

```
);
GenClock(Clk,'0',10 ns,EndOfSim); -- <<<<<<<<<<<<<<<
main : process
begin
…..
…..
EndOfSim <= true;
wait;
end process;
end behav;
```

Questions:

- Why is there a wait statement at the end of the process labeled main?

- Why is there no wait statement needed at the end of the concurrent

GenClock procedure?

2. Write the body of the CheckPulseWidth procedure

```
procedure CheckPulsWidth( constant name : in string;
signal sig : in std_logic;
constant level : in string; -- high | low
constant wid : in time;
constant optional_message : in string :="");
```

During the whole simulation this procedure checks if the width of a pulse on a signal (sig) has
the correct length. The string constant “name” is the name of the signal. It is used inside the procedure when reporting a message to the screen if a pulse is too short or too wide. The optional_message constant allows to write some extra information to the screen and in this lab is only used to debug the procedure. In case the pulse does not have the correct length, this is the statement to use to write information to the user.

```
report " Measured pulse width of " & name & " = " & ……..
& " instead of the expected " & …….
& optional_message
```

The first set of dots is a placeholder for the simulated length of the pulse, the second set stands for the expected length.
There are two ways to implement this procedure. The easiest method is using a loop like in the GenClock procedure. The simulation will enter the loop at the start and will never jump out of it again.
As the procedure is only looking at values and not generating any, it will still allow for a clean end of simulation.
The second possibility is not using a loop, but making use of the fact that a concurrently called procedure is equivalent to a process with signal sig in its sensitivity list. At the end of the process, it will only be started again at a value change on sig.

Option: implement a procedure called CheckPulsWidthV2 with the same parameters als heckPulsWidth, but without using a loop statement. The procures(s) has(have) to be called concurrently in the testbench.

```
architecture behav of Testbench is
….
signal clk : std_logic;
signal EndOfSim : boolean:=false;
begin
U1 : TestHardware
port map(
…..
);
GenClock(Clk,'0',10 ns,EndOfSim);
main : process
begin
…..
end process;
CheckPulsWidth ("pulse2",pulse2,"high",20 ns); --
<<<<<
CheckPulsWidth ("pulse1",pulse1,"low",100 ns,”Critical
problem”); --
<<<<<
-- option
CheckPulsWidthV2("pulse2",pulse2,"high",20 ns); --
<<<<<
CheckPulsWidthV2("pulse1",pulse1,"low",100 ns,”Critical
problem”); --
<<<<<
end behav;
```

3. Look at the TB_TestHardware.vhd file. Via the library en use statement the procedures and functions in General_TB_pack are visible in the testbench.

```
LIBRARY General_TB_lib;
use General_TB_lib.General_TB_pack.all;
```

In this file TestHardware is instantiated. The GenClock procedure is called to drive the clock of TestHardware as well as a number of CheckPulsWidth procedures to verify the length of pulses on the outputs of TestHardware. The ones necessare are the following.

CheckPulsWidth("pulse2",pulse2,"high",2*10 ns);
CheckPulsWidth("pulse2",pulse2,"low",14*10 ns);

If all is well there should be no warnings during simulation. But, because we want to test the functionality of the procedures themselves, we are going to call them with erroneous values.

CheckPulsWidth("pulse2",pulse2,"high",1*10 ns);
CheckPulsWidth("pulse2",pulse2,"high",3*10 ns);

CheckPulsWidth("pulse2",pulse2,"low",13*10 ns);
CheckPulsWidth("pulse2",pulse2,"low",16*10 ns);
This way the simulator will generate messages, allowing us to see how the procedures function.

4. Compilation and simulation

Start Modelsim and go to the directory where your package is located. Run the following
commands.

 Remark: there is a difference in usage of the modelsim.ini file (settings for modelsim) between a windows and Linux or unix OS.
When running your simulations on a linux or unix OS you can copy a modelsim.ini file to your local work directory. The command vmap will then update this local modelsim.ini file. This command vmap makes the link between a vhdl library and the directory in which this is compiled.
When running your simulations on a windows OS it is better to create a project in your work directory at the startup of modelsim. In that case the modelsim.ini file stored in the installation directory of modelsim will be automatically copied into your project file <your_project>.mpf . If you do not create a project in modelsim the modelsim.ini file in the installation directory of modelsim will be directly used and modified.

```
 vlib work
 vlib General_TB_lib
 vmap General_TB_lib General_TB_lib
 vcom -work General_TB_lib General_TB_pack.vhd
 vcom TestHardware.vhd
 vcom TB_TestHardware.vhd
 vsim work.tb_testhardware
 add wave *
 run -all
```

The simulation should end automatically at 310 or 320ns. If this is not the case, you have an error in your procedures. The simulation output should look like this.

```
# ** Error: Measured pulse width of pulse2 = 20 ns instead
of the
expected 30 ns <<<….
# Time: 45 ns Iteration: 2 Instance: /tb_testhardware
# ** Error: Measured pulse width of pulse2 = 20 ns instead
of the
expected 10 ns <<<….
# Time: 45 ns Iteration: 2 Instance: /tb_testhardware
# ** Error: Measured pulse width of pulse2 = 140 ns instead
of the
expected 160 ns <<<….
```

```
# Time: 185 ns Iteration: 2 Instance: /tb_testhardware
# ** Error: Measured pulse width of pulse2 = 140 ns instead
of the
expected 130 ns <<<….
# Time: 185 ns Iteration: 2 Instance: /tb_testhardware
# ** Error: Measured pulse width of pulse2 = 20 ns instead
of the
expected 30 ns <<<<….
# Time: 205 ns Iteration: 2 Instance: /tb_testhardware
# ** Error: Measured pulse width of pulse2 = 20 ns instead
of the
expected 10 ns <<<….
# Time: 205 ns Iteration: 2 Instance: /tb_testhardware
# ** Note: *************** no !@@@@@@@@ should be in the
messages!
# Time: 310 ns Iteration: 1 Instance: /tb_testhardware
```

If you don't see any messages in the log window check the Modelsim settings. Under simulation options you can select which kinds of messages you want to or don't want to be sent to the window.

Hints

- The VHDL report statement can only send strings to the screen. To be able to include values of integer, time and std_logic_vector type a number of ToString functions have been added to the General_TB_pack package. Notice that the same function name can be used with different arguments (overloading). It is also possible to use the 'image attribute but this is not demonstrated in this lab.

- As functions only operate on types, the vector type arguments can not have a range specification (it is not part of the type definition).

You can however write a commonly useable function. This is also true for procedures. Inside the body of the function/procedure you can use attributes to find out what the actual bus ranges and limits are. The example below shows the use of 'length and 'range. Suppose a std_logic_vector(3 downto 0) is the actual for Vecval. Vecval 'length is then equal to 4 and Vecval'range is equal to (3 downto 0)..

```
function ToString(constant Vecval : Std_Logic_vector)
return string is
variable OutString : string(1 to Vecval'length);
variable k : natural;
begin
k := 1;
for i in Vecval'range loop
OutString(k) := ToChar(Vecval(i));
```

```
k := k+1;
end loop;
return OutString;
end ToString;
```

- You can jump out of an endless loop with an "exit when some_boolean=true" statement.
- Remember that the result of a division on an object of type time is chopped to the resolution of the simulator. E.g. if the resolution is set to to nanoseconds (ns) then 15ns/2=7ns.
- Don't forget to add a test in the code that checks if the actual string value for level is indeed "high" or "low" and nothing else.
- The "now" function returns the current simulation time.

Example : current_time := now;

- Use the numeric_std package to perform arithmetic operations on std_logic_vectors or to convert them to integers. DON'T use the std_logic_arith package! It's not standard VHDL. You can find the full source of the numeric_std package in this file, available in the modelsim installation folder: vhdl_src/ieee/mti_numeric_std.vhd. You can eg convert an integer to an unsigned vector using the TO_UNSIGNED function.
Usage: TO_UNSIGNED(integer_value, amount_of_bits_in_the_vector).
An unsigned vector can easily be 'casted' to an std_logic_vector.

```
use IEEE.numeric_std.all;
…
…
some_std_logic_vector:=std_logic_vector(TO_UNSIGNED(int_v
al,16);
```

Extra exercises

1. Write the CheckDist procedure

```
procedure CheckDist ( constant sig1_name : in string;
signal sig1 : in std_logic;
constant sig1_edge : in string;
-- rising | falling
constant sig2_name : in string;
signal sig2 : in std_logic;
constant dist : in time;
-- distance sig1 edge -> sig2 change
constant optional_message : in string :="");
```

This concurrent procedure can measure the distance between the rising and falling edge of signal sig1 and a change on signal sig2 during the simulation.

The figure below shows what distance is checked for rising edges of sig1 (the sig1_edge actual parameter value is rising). Notice that sig2 does not have to change. But when it does change, it has to be at a correct distance with respect to the edge on sig1.

Use the following statement to report a message to the screen if the distance is not correct.

```
report "Measured distance of " & sig1_name & " " & sig1_edge
&" = " & ……
& " instead of the expected " & ……
& optional_message;
```

The first set of dots is a placeholder for the value of the period between sig1 and sig2. The second set is a placeholder for the expected period. The following is an example of how to call the procedure in the testbench.

```
CheckDist("pulse2",pulse2,"falling","pulse1",pulse1,6*20
ns, "optional ");
```

2. write the CheckIfValueOccursInSim procedure

```
procedure CheckIfValueOccursInSim
( constant signame : in string;
signal sig : in std_logic_vector;
constant checkvalue : in std_logic_vector;
constant check : in string; -- must_occur | should_not_occur
signal EndOfSim : in boolean;
constant optional_message : in string :="");
```

Using this concurrent procedure you can check whether a certain vector value is occurring during the simulation. It allows you to e.g. verify the correct wrapping value of a counter.