



Advanced Logic Synthesis for Electronics
<http://www.alse-fr.com>

© ALSE- B. Cuzeau - Aug 2009, ver 1.1a

Writing Test Benches

A Free Application Note

Introduction

Preamble

Writing test Benches is an Essential Activity, and more and more so !

Today, all our Industry agrees on the fact that Verification efforts are now forming the bulk of the investments in a Digital Design. Typical numbers oscillate between 70% and 95% depending on the type of project. It's no surprise that ASIC projects are the "worst" (highest Verification efforts) for obvious reasons, but FPGA designs have also moved very significantly in the same direction.

In other words: less RTL coding, more Verification efforts.

It's not the place here to go within all the details behind the rationale, but the purpose of this Application Note is to teach the very Basics of the Verification process: how to write test benches, practically and efficiently.

A fundamental Coding Style Rule says that **Every RTL Entity** must have it's **Unitary Test Bench**.

We will focus on **Black Box Testing** which is usually seen as the most daunting task (as compared with Unitary "glass-box" testing).

So we are going to start with a with a practical and realistic example and go through these steps :

- Define the Verification Framework.
- Code the test bench skeleton.
- Add in the basic Stimuli (clock reset etc)
- Find or Design, then integrate a Behavioral Model
- Add some code for Self-Testing.
- Create a Tcl script to automate the simulation.
- Run the simulation and debug both the UUT and the Test Bench !.

This seems a pretty ambitious program and difficult tasks, but we'll see it's not the case provided you know your HDL basics (that's what our Training courses are for !).

This Application will use VHDL, but it maps directly to (System)Verilog.

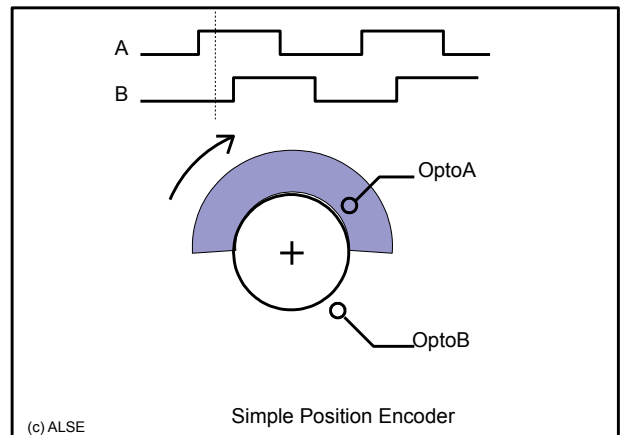
Practical Example

Practical Example : the Quadrature Encoder

We will use a Quadrature Encoder as an example.

The figure on the right reminds the principle: a two sectors coding wheel and two sensors placed at 90° angle can accurately monitor the position and the direction of turn of the wheel (with a resolution of 90° indeed). Actual coding wheels (like in mice) have more sectors, and the sensors are shifted by a half-sector angle.

The position encoder uses the signals from the A & B sensors to issue the output information as **Position** (8 bits vector in our case for -128 to +127) and **Direction** of the last step.



This is described in more details inside our Application Note about Motor Control Basics.

You can download the code for this encoder in the “Free IPs” section of our Web site.

Note: This ApNote is about considering the **Unit Under Test** (UUT) as a “black box” and exercise it, while it's in fact a single unitary module. In real life, our test bench would rather be part of unitary verification. In a “glass box” context, we would try to exercise all the code and verify all the features coded inside (like the initial deadtime).

For the sake of this Application Note, we will not examine the code inside, but only consider the encoder as a “system”, or a “black box”.

Here is the Entity declaration for the function we want to test. We do not need the architecture ! In fact, in the “black box” methodology, we should not even have access to the architecture...

```

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

-- -----
-- Entity QUAD is
-- -----
port( Rst : in  std_logic; -- Asynchronous Reset
      Clk : in  std_logic; -- System Clock
      A,B : in  std_logic; -- Quadrature inputs (resynch'ed internally)
      Cnt : out std_logic_vector(7 downto 0); -- (un)signed 8 bits position !
      Dir : out std_logic; -- Clockwise information (of last step)
end entity QUAD;
```

You can **open the source code** (quad.vhd), **copy the above piece of code** and **save it in your new test bench file**.

Important : make sure you have a Text Editor you're reasonably familiar with, and that this editor has a **column mode** and supports **VHDL syntax-coloring**. If you don't have this, consider the small and free Text Editor “Crimson”.

Building the Test Bench

Creating the Test bench - Step 1

- Create a new file named **QUAD_TB.vhd** and paste the Entity declaration (preceded by the libraries) from the original file **QUAD.vhd**, as seen previous page.
Remove the comment lines, but keep the comments after the ports.
We have outlined in bold characters the code added or modified from this original piece.
- Add a Header with the file name and minimal information
- Add the two library statements in bold for **STD.TEXTIO** and **IEEE.STD_LOGIC_TEXTIO**.
- Add the Test bench Entity declaration, in one line (there is no port in a Test Bench).
- Add the test bench Architecture with the Begin and End architecture declarations.
- Remove the *end entity quad;* line
- Copy Again the Entity declaration and paste it between the Begin and End of the test bench architecture.
- Delete Entity QUAD is port below Architecture (in italic below).

Your code should now look like this :

```
-- QUAD_TB.vhd
-- Test Bench for QUAD

use STD.TEXTIO.all;
Library IEEE;
use IEEE.std_logic_TEXTIO.all;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

Entity QUAD_TB is end;

Architecture TEST of QUAD_TB is
    Entity QUAD is
        port( Rst : in  std_logic; -- Asynchronous Reset
              Clk : in  std_logic; -- System Clock
              A,B : in  std_logic; -- Quadrature inputs (resynch'ed internally)
              Cnt : out std_logic_vector; -- Position is an unconstrained vector
              Dir : out std_logic); -- Clockwise information (of last step)

    Begin -- Architecture

    Entity QUAD is
        port( Rst : in  std_logic; -- Asynchronous Reset
              Clk : in  std_logic; -- System Clock
              A,B : in  std_logic; -- Quadrature inputs (resynch'ed internally)
              Cnt : out std_logic_vector(7 downto 0); -- (un)signed 8 bits position !
              Dir : out std_logic); -- Clockwise information (of last step)

    End Architecture TEST;
```

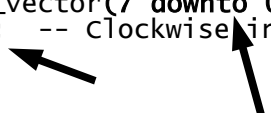
Noe the use of an unconstrained vector for the position information (Cnt). This is a very nice feature in VHDL. In our case, the actual size of the counter inside the encoder will be determined by the instantiation in the Test bench (we'll use a size of eight bits).

Step 2 : Create the Signals

- Switch your Text Editor in column mode (Alt-C with Crimson),
- Select the area in front of port lines and type “ **signal** ”
- Remove the port direction vertical block (in, out)
- Remove the last closing parenthesis “)” but remember you need the last semi-column.
- Add the constants for Clock Frequency, Period, and Done boolean.
Don't forget the initial value for Clk (and other inputs if you want).

You code should look now like :

```
Architecture TEST of QUAD_TB is
  constant Fclock : positive := 10E6; -- 10 MHz
  constant Period : time := 1 sec / Fclock;
  signal Done : boolean;
  signal Rst : std_logic; -- Asynchronous Reset
  signal Clk : std_logic := '0'; -- System Clock
  signal A,B : std_logic; -- Quadrature inputs (resynch'ed internally)
  signal Cnt : std_logic_vector(7 downto 0); -- (un)signed 8 bits position !
  signal Dir : std_logic; -- Clockwise information (of last step)
Begin -- Architecture
```



Note that we decide to use an eight-bits vector for Cnt.

Step 3 : Instantiate the UUT

- Edit the second copy of the entity to transform it into a Direct Instantiation as below.
You can again use the editor's column mode to cut and paste the list of ports:

```
Begin -- Architecture
UUT: Entity work.QUAD port map (
  Rst => Rst,
  Clk => Clk,
  A  => A ,
  B  => B ,
  Cnt => Cnt,
  Dir => Dir );
End Architecture TEST;
```

Note : every designer should know how to quickly write a test bench without help, but some tools can do the above steps for you, like the free Text Editor Emacs with its sophisticated VHDL mode written by Reto Zimmermann. There is not much intrinsic value in the steps above, very mechanical.

The Stimuli !

Step 4 : Create the Stimuli

For Clk and Rst the stimuli are typically always the same :

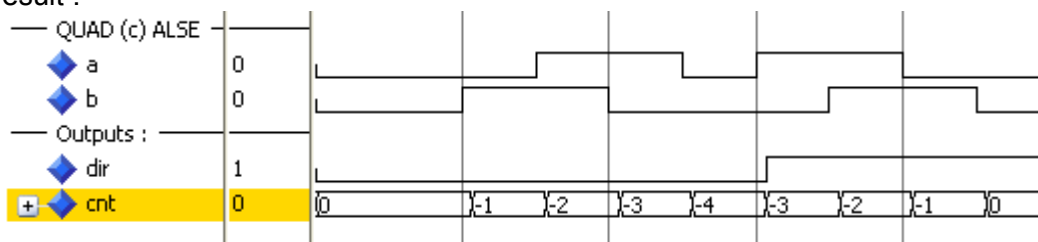
```
Rst <= '1', '0' after Period;
Clk <= '0' when Done else not Clk after Period / 2;
```

Now we need to create the stimuli for A and B.

The first idea could be to follow directly the waveform patterns from the documentation (specification) in a linear (sequential) process. This is indeed possible, but that would create a lot of repetitive code, and not in a very flexible nor subtle and even less re-usable style :

```
-- Simple process for A & B
process begin
  A <= '0'; B <= '0'; wait for 20 * Period; -- wait & make sure BootDly has expired
  A <= '1'; wait for 10 * Period;
  B <= '1'; wait for 10 * Period;
  A <= '1'; wait for 10 * Period;
  B <= '0'; wait for 10 * Period;
  A <= '0'; wait for 10 * Period;
  A <= '1'; wait for 10 * Period;
  B <= '1'; wait for 10 * Period;
  A <= '0'; wait for 10 * Period;
  B <= '0'; wait for 10 * Period;
  Done <= true;
  wait;
end process;
```

Simulation result :



For a simple Test bench and quick verification, we could stop here.

Better Stimuli (procedural & self-testing)

We will replace the crude code above by a simple behavioral Model for the Position sensor and add a procedure to “turn” it by N steps in any direction.

The Model will maintain the current Angular position (an integer), and the modulo 4 of this angle will be 0, 1, 2, or 3 corresponding to 00 01 11 10 for A & B.

The procedure will take an argument being the signed number of steps.

If we have done a good job and if the UUT (Unit Under Test) is working properly, the internal Angle of the behavioral model should match the position encoder output “Posit” (in the -128 .. +127 range at least). Why wouldn't our model check this automatically ?

Good idea ! This is enhancing our Test bench into a *Self-Testing* test bench. We do not have to rely solely on the manual inspection of the waveforms to verify that our UUT works fine.

Contrarily to the previous steps, **creating stimuli and self-testing code** is where the designer's (tester's) creativity and know-how will make a difference !

The final code for the A & B Stimuli can be :

```

Process -- Encoder Model + Self-Testing Stimuli
type Phase_t is array (0 to 3) of std_logic_vector(0 to 1);
constant Phase_Table : Phase_t := ("00", "10", "11", "01");
variable Phase : integer := 0;
variable L : line;
procedure Turn (N : integer) is
    variable J : integer := N;
begin
    while J /= 0 loop
        if J < 0 then
            J := J+1;
            Phase := Phase - 1;
        else
            J := J-1;
            Phase := Phase + 1;
        end if;
        (A,B) <= Phase_Table(Phase mod 4);
        wait for 20 * Period;
        -- Self Test section :
        assert signed(Cnt)=Phase
            report "Cnt error, expected = "&integer'image (Phase)
            severity Error;
        assert (N<0 and Dir='0') or (N>0 and Dir='1') or N=0
            report "Error in Dir !"
            severity Error;
    end loop;
end procedure Turn;
Begin -- process

```

and the test sequence is then :

```

A <= '0'; B <= '0';
wait for 20 * Period; -- wait & make sure BootDly has expired
write(L,now);
write (L, HT&"Incrementing now Phase 4 times");
writeline (output,L);
Turn(4);
write(L,now);
write (L, HT&"Decrementing now Phase 8 times");
writeline (output,L);
Turn(-8);
A <= '1'; B <= '1';
write(L,now);
write (L, HT&"Impossible phase skip"&HT);
writeline (output,L);
wait for 20 * Period;
A <= '0'; B <= '0';
write(L,now);
write (L, HT&"Impossible phase skip"&HT);
writeline (output,L);
wait for 20 * Period;
Turn(1);
Turn(-1);
write (L, "End of Simulation"&HT);
writeline (output,L);
Done <= true;
wait;
end process;

```

Simulation Transcript :

```

# 2000 ns Incrementing now Phase 4 times
# 10000 ns Decrementing now Phase 8 times
# 26000 ns Impossible phase skip
# 28000 ns Impossible phase skip
# End of Simulation

```

With ModelSim, if you double-click in the transcript, the cursor will automatically jump to the expected point in time (that's our motivation to report the current simulation time, **now**).

Simulation Automation

Step 5 : Automate

By definition, we are probably going to run the same simulation many times during the development phase, and probably also later (like for non-regression testing). In both cases, it is important to not rely on the user to drive all the steps involved, from creating the working library to compiling the proper files, to loading the right test bench, running the simulation and collate the results.

During the development, the issue is the productivity.

Later, the issue is to be able to reliably re-run a specific test without having to remember or know a lot.

A simple solution is to write a script like this one :

```
# simquad.do
puts "-- Simulation script (c) ALSE for QUAD encoder --"

vlib work

vcom -93 quad.vhd
vcom -93 quad_tb.vhd

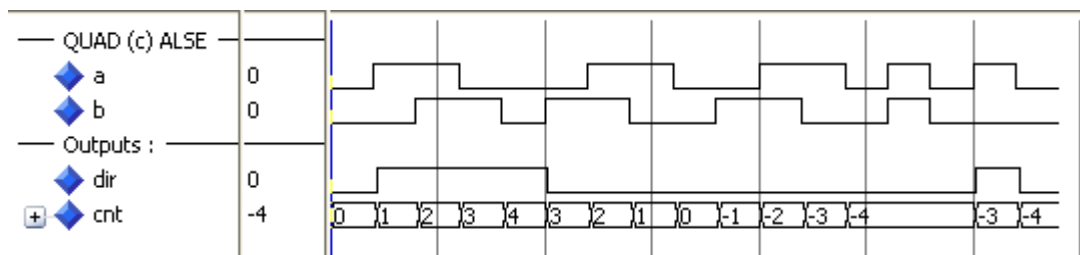
vsim quad_tb

add wave -divider "QUAD (c) ALSE"
add wave A
add wave B
add wave -divider "Outputs:"
add wave Dir
add wave -radix decimal Cnt

run -a
```

Using the script with ModelSim is trivial :

- Change to the right directory (Menu or "cd" command)
- Type the command :
do simquad.do
- Enjoy !



The Design documentation would simply :

- Indicate what simulation script to run
- Reproduce the expected transcript output.

Any operator, even not knowing the design, would be able to run the simulation and get some insurance about the behavior of the UUT.

Conclusion

Conclusion

After spending more than 15 years advocating for the use of simulation in FPGA design, I hope this Application Note will encourage young (and less young !) FPGA Designers into spending more time upfront for creating solid test benches... and indeed thus saving a lot of time later chasing problems in hardware.

I hope you enjoyed this Application Note !

If you did or if you have feedback or suggestions, an e-mail is always welcome ☺

Bertrand CUZEAU
Technical Manager A.L.S.E.
Write to : info at alse-fr dot com