Testbenchs avanzados: momento de evolucionar

Creador: David Rubio G.

Entrada: https://soceame.wordpress.com/2024/12/12/testbenchs-avanzados-momento-de-evolucionar/

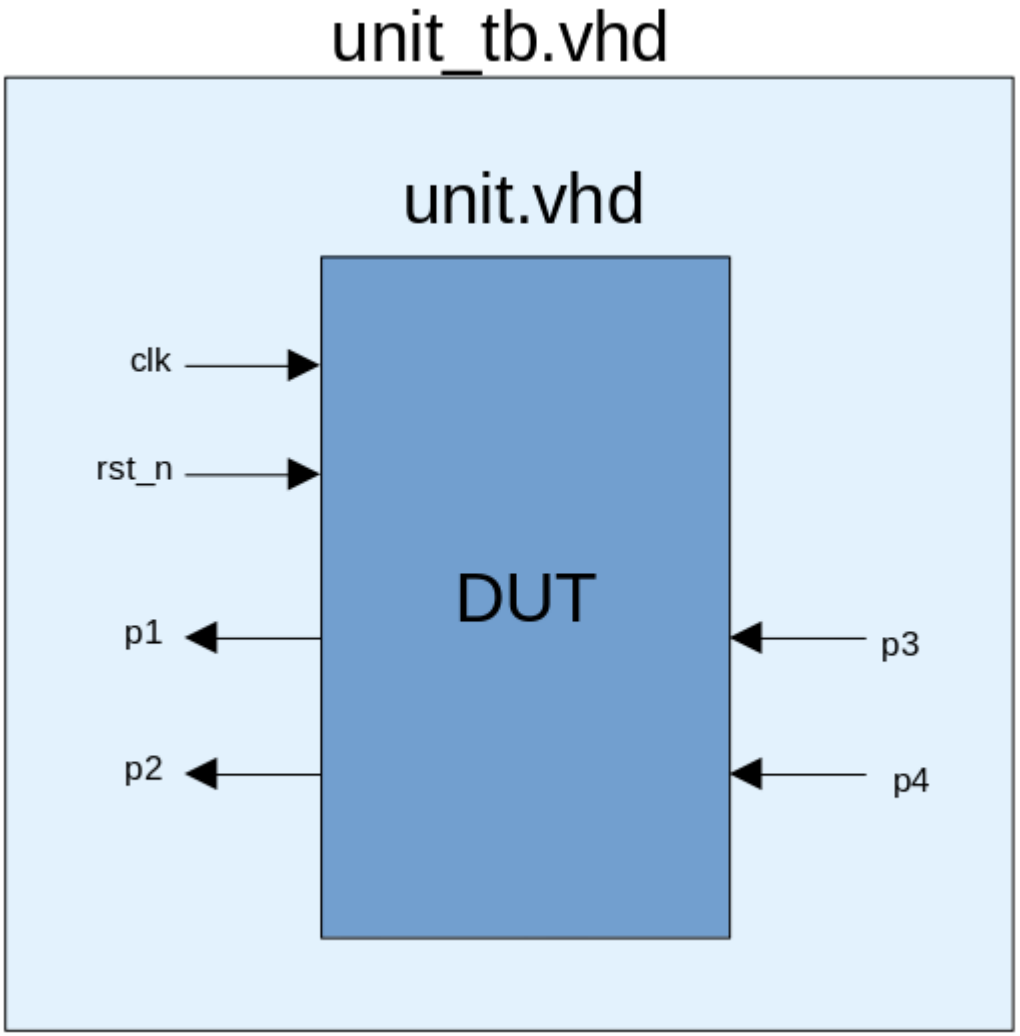
Blog: https://soceame.wordpress.com/

GitHub: https://github.com/DRubioG

Fecha última modificación: 23/02/2025

Introducción

La forma clásica de desarrollar testbench es de la siguiente forma, bien pues esta forma presenta diferentes inconvenientes: la **escalabilidad es compleja**, porque todo el código de test está en un único fichero, entonces expandir el fichero te obliga a continuar con las estructuras ya implementadas; los **ficheros pueden ser inmensos**, te puedes encontrar con testbenchs de más de 10.000 líneas para simular toda la funcionalidad, y como aparezca un nuevo requisito o puerto se puede volver muy complejo de manejar o actualizar el testbench; y sobre todo **la reusabilidad** entre proyectos, para poder hacer simulaciones entre ficheros muchas veces se tiene que estar haciendo un copia-pega entre testbenchs para hacer lo mismo en dos simulaciones.



Testbenchs avanzados

Bien, pues al igual que no desarrollas todo un proyecto FW en un solo módulo, con los testbenchs pasa lo mismo.

NOTA*: el sistema que te estoy comentando no es más fácil que un testbench normal, pero sí que garantiza la posibilidad de escalar un sistema, además, de poder desarrollar un testbench de forma conjunta entre varias personas porque permite que cada uno desarrolle un módulo independiente, y debido a que internamente se desarrollan los submódulos, y estos se puede reutilizar en diferentes proyectos. Además de poder reutilizar bloques de simulación entre simulaciones.*

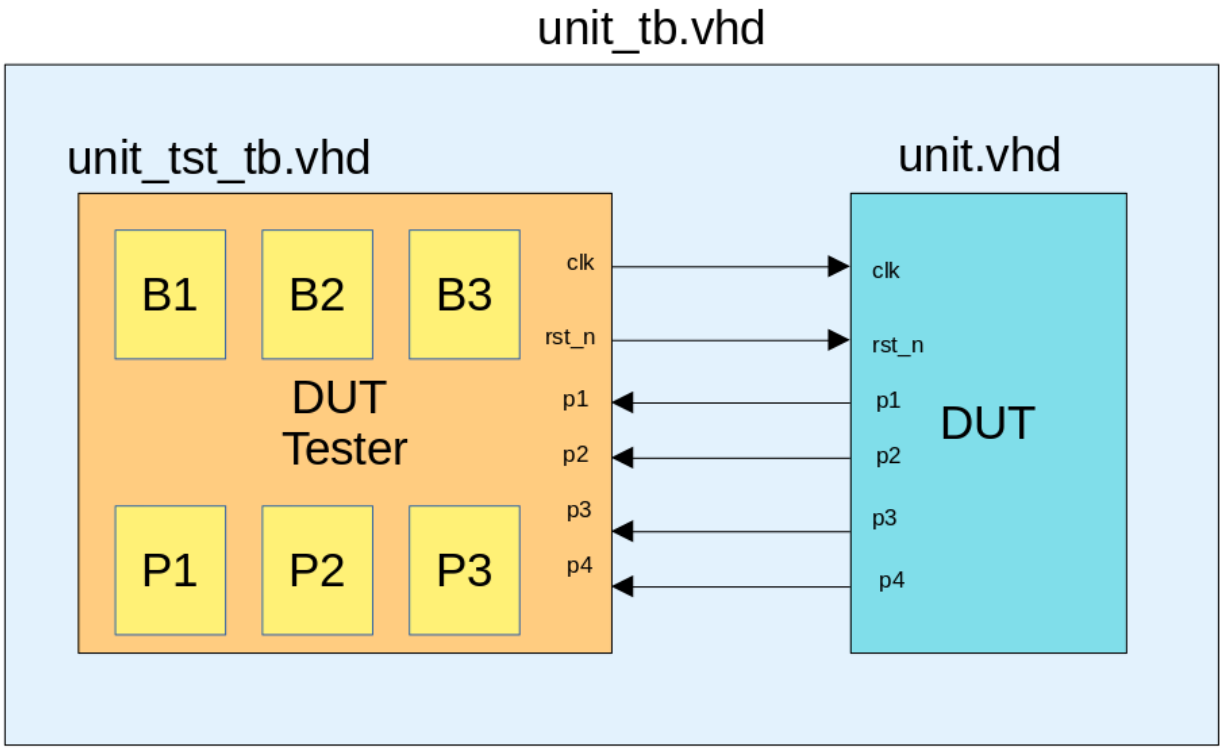
En vez de tener un fichero testbench que contenga solo el módulo a simular, incluyes un nuevo módulo que es el que va a realizar la generación de estímulos y el análisis del funcionamiento del sistema. Este nuevo módulo va a estar subdividido internamente para garantizar que haya partes que se encarguen de unas tareas y otros de analizar la simulación *(muchos de estos módulos que se encargan de hacer otras tareas pueden ser reutilizados de otros proyectos o simulaciones)*.

Además, se pueden reutilizar los bloques de simulación de un proyecto según se van escalando las simulaciones, porque un bloque de simulación de un fichero *bottom* se puede utilizar en ficheros *top* de éste. Y también, se pueden utilizar submódulos del FW funcional dentro del bloque de testeo del DUT (*DUT_Tester*), gracias a ello podemos confrontar partes del propio FW contra sí mismas para ver cómo responden, y garantizar su funcionamiento correcto.

Otra cosa que se puede hacer gracias a esta estructura es diseñar la simulación antes que el FW funcional, para que cuando se diseñe o implemente el FW funcional se tenga unas simulaciones de gran precisión.

NOTA: *esto NO es una simulación con diferentes módulos del código fuente en un solo fichero testbench (cosa que no es recomendable en desarrollo FW). Esto es añadir un fichero nuevo de testbench dentro del testbench principal, que es el que se va a encargar internamente de hacer las simulaciones.*

NOTA 2: *este sistema es mejor que la utilización de procedures porque puedes utilizar señales en el tiempo, porque los submódulos internamente pueden utilizar relojes o cualquier señal, como lo haría en un testbench normal. Solo que con este sistema se encapsulan los desarrollos.*



Recordemos que para las simulaciones no es necesario seguir un esquema sintetizable, porque no se va a generar FW de un testbench, entonces, puedes hacer todo cuanto el simulador te deje.

Para la reutilización se puede utilizar tanto *procedures* que estén en *packages* separados, como de módulos que estén bien diferenciados. Pongo un ejemplo para facilitar el entendimiento.

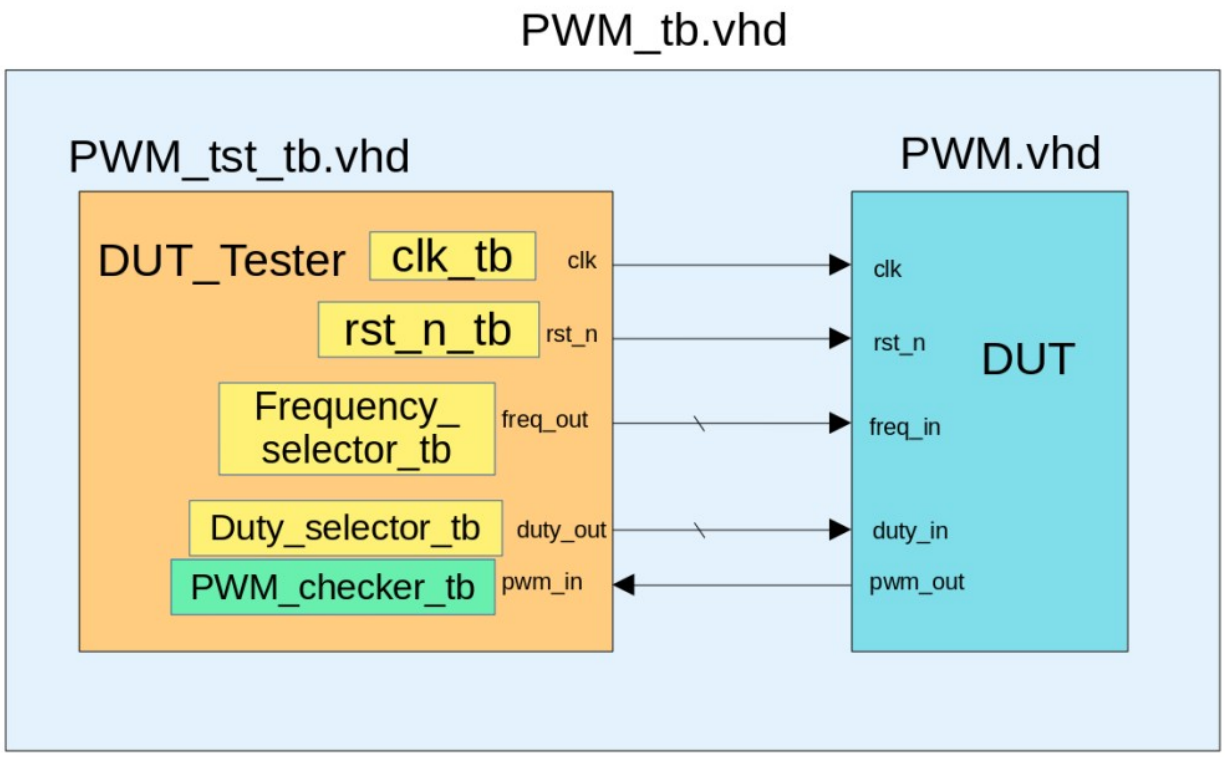
Ejemplo

Vamos a diseñar un pequeño ejemplo para ilustrar la nueva forma de estructurar los testbenchs.

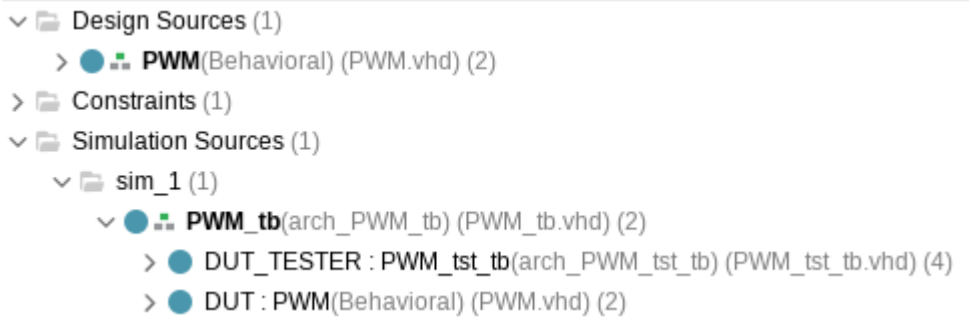
Este pequeño ejemplo representará como se tendría que crear un nuevo tipo de testbench para un módulo que ejecuta un PWM.

La PWM a simular tiene la capacidad de seleccionar la frecuencia y el ciclo de trabajo.

El diseño a implementar es el siguiente *(para comprobar que la PWM es correcta se puede acoplar un PWM_checker, que compruebe la PWM de salida)*:



La estructura del testbench quedaría de la siguiente forma:



- **clk_tb**: Este es el módulo generador de reloj de la PWM.
  Se puede ver que se ha generado utilizando principios de FW funcional, debido a que se utilizan puertos para sacar la señal de reloj al mundo exterior *(pero es un puerto de salida, si lo intentas sintetizar, el sintetizador se zampa ese puerto, por lo que es un módulo de SOLO simulación)*.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity clk_tb is
    generic(
        CLK_TIME : time := 5ns
    );
    Port (
        clk_o : out std_logic
    );
end clk_tb;

architecture arch_clk_tb of clk_tb is

signal clk_i : std_logic := '0';

begin

    clk_i <= not clk_i after CLK_TIME;
    clk_o <= clk_i;

end architecture;
```

- **rst_n_tb**: Este es el módulo que genera el reset asíncrono a nivel bajo. Este módulo hace uso de un puerto exterior para sacar el reset.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity rst_n_tb is
    generic(
        TIME_RESET : time := 50ns
    );
    Port (
        rst_n_o : out std_logic
    );
end rst_n_tb;

architecture arch_rst_n_tb of rst_n_tb is

begin

    rst_n_o <= '0', '1' after TIME_RESET;


end arch_rst_n_tb;
```

- **frequency_selector_tb**: Este módulo es el encargado de cambiar el valor al divisor de frecuencia interno. Se puede ver que tiene una señal de reset de entrada que es la misma que se genera en el otro submódulo, para provocar que la señal este a ‘0’ cuando el reset esté activo al comienzo.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity frequency_selector_tb is
    Port (
        rst_n : in std_logic;
        freq_tst_out : out std_logic_vector(15 downto 0)
    );
end frequency_selector_tb;

architecture Behavioral of frequency_selector_tb is

begin

    process begin
        freq_tst_out <= (others=>'0');
        wait until rst_n = '1';
        freq_tst_out <= x"00FF";
        wait for 20us;
        freq_tst_out <= x"FF00";
        wait;
    end process;

end Behavioral;
```

- **Duty_selector_tb**: Este módulo es el encargado de seleccionar el ciclo de trabajo de la PWM. Este módulo tiene un reset para detectar el comienzo del sistema, y también en caso de que se produzca un reset al terminar la operativa volver a estado inicial.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Duty_selector_tb is
    Port (
        rst_n : in std_logic;
        duty_tst_out : out std_logic_vector(7 downto 0)
    );
end Duty_selector_tb;

architecture Behavioral of Duty_selector_tb is

begin


    process begin
        duty_tst_out <= (others=>'0');
        wait until rst_n = '1';
        duty_tst_out <= x"0F";
        wait for 40us;
        duty_tst_out <= x"F0";
        wait until rst_n = '0';
    end process;

end Behavioral;
```

Una vez creados los submódulos, se crean los módulos de los testbenchs.

- **PWM_tst_tb**: Este es el módulo encargado de generar todos los estímulos al bloque FW generado. Para ello instancia todos los bloques de simulación necesarios. También puede generar las señales necesarias para comprobar que el DUT funciona como debería.

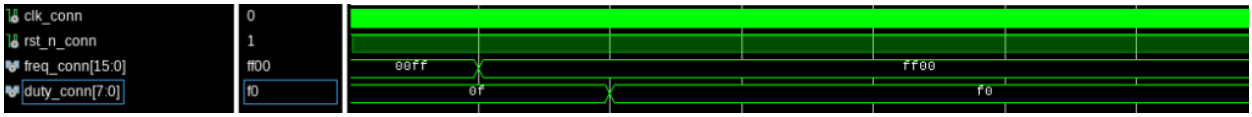
```
library ieee;
use ieee.std_logic_1164.all;

entity PWM_tst_tb is
    port (
        clk : out std_logic;
        rst_n : out std_logic;
        freq_out : out std_logic_vector(15 downto 0);
        duty_out : out std_logic_vector(7 downto 0);
        pwm_in : in std_logic
    );
end entity;

architecture arch_PWM_tst_tb of PWM_tst_tb is

component clk_tb is
    generic(
        CLK_TIME : time := 5ns
    );
    Port (
        clk_o : out std_logic
    );
end component;

component rst_n_tb is
    generic(
        TIME_RESET : time := 50ns
    );
    Port (
        rst_n_o : out std_logic
    );
end component;

component Duty_selector_tb is
    Port (
        rst_n : in std_logic;
        duty_tst_out : out std_logic_vector(7 downto 0)
    );
end component;

component frequency_selector_tb is
    Port (
        rst_n : in std_logic;
        freq_tst_out : out std_logic_vector(15 downto 0)
    );
end component;

signal  rst_n_aux : std_logic;

begin

CLK_GEN : clk_tb
    generic map(
        CLK_TIME => 5ns
    )
    Port map (
        clk_o => clk
    );

rst_n <= rst_n_aux;
RESET_GEN : rst_n_tb
    generic map(
        TIME_RESET => 50ns
    )
    Port map(
        rst_n_o => rst_n_aux
    );

DUTY_GEN : Duty_selector_tb
    Port map (
        rst_n => rst_n_aux,
        duty_tst_out => duty_out
    );


FREQ_GEN : frequency_selector_tb
    Port map (
        rst_n => rst_n_aux,
        freq_tst_out => freq_out
    );
end architecture;
```


- **PWM_tb**: Este es el testbench del DUT de la PWM, en este módulo hay dos instanciaciones, uno del **DUT** y otro del **DUT_Tester**.

Este módulo siempre permanece limpio e inalterable a menos que cambien los puertos de salida del *DUT*.

```
library ieee;
use ieee.std_logic_1164.all;

entity PWM_tb is

end entity;

architecture arch_PWM_tb of PWM_tb is

component PWM is
    port (
        clk : out std_logic;
        rst_n : out std_logic;
        freq_in : out std_logic_vector(15 downto 0);
        duty_in : out std_logic_vector(7 downto 0);
        pwm_out : in std_logic
    );
end component;

component PWM_tst_tb is
    port (
        clk : out std_logic;
        rst_n : out std_logic;
        freq_out : out std_logic_vector(15 downto 0);
        duty_out : out std_logic_vector(7 downto 0);
        pwm_in : in std_logic
    );
end component;

signal clk_conn : std_logic;
signal rst_n_conn : std_logic;
signal freq_conn : std_logic_vector(15 downto 0);
signal duty_conn : std_logic_vector(7 downto 0);
signal pwm_conn : std_logic;

begin

DUT_TESTER : PWM_tst_tb
    port map (
        clk =>clk_conn,
        rst_n => rst_n_conn,
        freq_out => freq_conn,
        duty_out => duty_conn,
        pwm_in => pwm_conn
    );

DUT : PWM
    port map (
        clk => clk_conn,
        rst_n => rst_n_conn,
        freq_in => freq_conn,
        duty_in => duty_conn,
        pwm_out => pwm_conn
    );


end architecture;
```


Si simulamos todos los estímulos tenemos la forma siguiente, se puede ver que la simulación de solo estímulos para la PWM es correcta.



Como se puede ver en los módulo anteriores, el *reset* o el *reloj* son módulos genéricos que se pueden reutilizar, al igual que el *frequency_selector* o el *Duty_selector* que en determinadas circunstancias también se pueden reutilizar.

Además, el módulo **PWM_tb** permanece imperturbable entre simulaciones, porque a menos que cambien los puertos de salida del DUT, este módulo no se toca.

El módulo del **DUT_Tester**, es el único módulo que junto con el testbench principal pertenecen a este proyecto y por lo tanto **NO son reutilizables**, el resto son reutilizables.
Y en caso de que se quiera cambiar la configuración solo sería necesario tocar el *DUT_Tester.*
Además, puedes meter estímulos complejos desde este módulo como señales que varían en el tiempo o cualquier cosa que puedas necesitar en tu desarrollo para simularlo.
También, todos los submódulos se pueden llevar a simulación de un fichero top.

Recomendación

Este tipo de esquemas de testbench son totalmente necesarios en testbenchs que van a ser de grandes dimensiones o de múltiples funcionalidades, también para aquellos que manejen muchos mini-módulos de FW, porque van a tener montones de ficheros de simulación. Y sobre todo para aquellos que trabajen en equipos, para subdividir los trabajos y evitar el copia-pega.
Para otro tipo de simulaciones simples no se recomienda utilizar esta estructura porque es demasiado compleja para la utilidad real que se requiere.

Nota final

Esto se puede asemejar al OVVM, pero esto es un estándar de trabajo demasiado enrevesado para trabajarlo en VHDL y lo mejor es manejar tu propio estándar basado en lo que comento en esta entrada.

https://soceame.wordpress.com/2025/02/09/que-son-las-metodologias-de-verificacion-y-como-crear-la-tuya-propia/