Generación de Estímulos y Verificación de Datos en Test Bench

Nota Técnica 11

Cristian Sisterna

1. Introducción

El esta nota técnica describiré solo lo relacionado a generación de estímulos y verificación de datos en un test bench. NO se describirá lo que es en sí un test bench ni su forma de escribirlo, ya hay mucho de esto en internet. Sin embargo hay poca información de cómo generar los datos/relojes/resets para estimular el dispositivo bajo test (usualmente conocido como DUT, device under test), y como verificar los resultados obtenidos a un determinado estimulo.

2. Utilidad y Composición del Test Bench

Básicamente un test bench sirve de estímulo y verificación de datos del componente o dispositivo que se desea verificar. El comportamiento del componente bajo el estímulo definido en el test bench puede verse en una pantalla del simulador usado a tal fin o puede verificarse con el comportamiento esperado usando instrucciones a tal fin en el mismo test bench o también puede guardar los datos resultados en archivos (este último caso no será detallado en este blog). Un diagrama ilustrativo de lo dicho se aprecia en la siguiente figura.

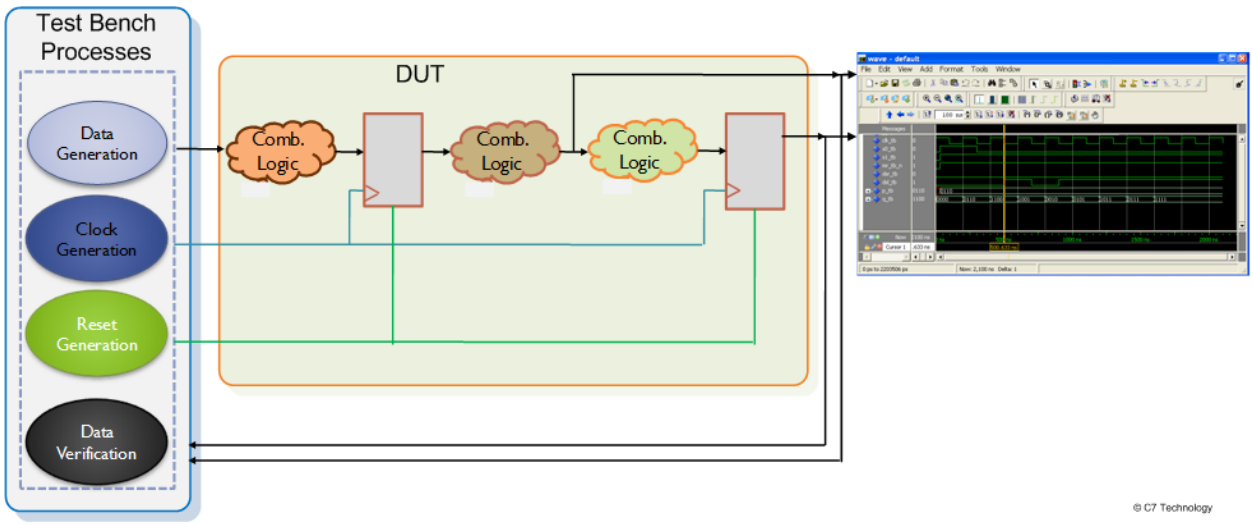

Figura 1- Esquema general de estímulo/respuesta de un DUT

Dentro del código VHDL del test bench podemos encontrar como ***mínimo*** los siguientes procesos:

- Generación de datos de estímulo para el DUT
- Generación de reloj/es de estímulo para el DUT
- Generación de señales de inicialización para el DUT
- Verificación de datos proveniente del DUT bajo un estímulo determinado

Se describe a continuación las características y manera de codificar cada uno de estos procesos.

2.1. Generación de Datos de Estímulo para el DUT

En cualquier sistema que se desee verificar se va necesitar generar datos que simulen el comportamiento de las entradas al DUT. De este modo se deberá, de algún modo, describir en VHDL el comportamiento de estas entradas. Este comportamiento debe estar basado en las especificaciones del sistema que se desea verificar; es decir hay que leer bien las especificaciones del diseño para saber de antemano cual es el comportamiento esperado de las entradas, para luego poder describir ese comportamiento en VHDL/Verilog. Por supuesto, y esto es importante, TAMBIÉN se debe verificar el comportamiento del DUT ante comportamiento no esperado de las entradas.

Se describen a continuación diferentes modos de generar distintos valores para ser asignados a las entradas del DUT.

2.1.1. Generación de Datos I

El siguiente código va cambiando los valores de entrada en el flanco de bajada de reloj. Los valores X1, y X2 son asignados a DATA1 y DATA2 en el flanco negativo del reloj.

```
data_gen_proc: process
while not (data_done) loop
   DATA1 <= X1;
   DATA2 <= X2;
   ...
   wait until falling_edge(clk);
 end loop;
end process data_gen_proc;
```

Figura 2 - Ejemplo de generación de datos 1

2.1.2. Generación de Datos II

Se pueden generar datos en función del tiempo de simulación. Existen dos modos de hacerlo uno es la generación usando tiempo relativos y la otra es la generación usando tiempos absolutos.

Tiempo Relativo: se van asignando valores a las señales en los tiempos de simulación con respecto al tiempo previo, de una manera acumulada.

```
architecture relative_timing of myTest is

     signal Add_Bus : std_logic_vector(7 downto 0);

begin
   patt_gen_proc: process
     begin
       Add_Bus <= "00000000";
       wait for 10 ns;

       Add_Bus <= "00000101";
       wait for 10 ns;

       Add_Bus <= "10101010";
       wait for 10 ns;
   end process patt_gen_proc;
. . .
end relative_timing;
```

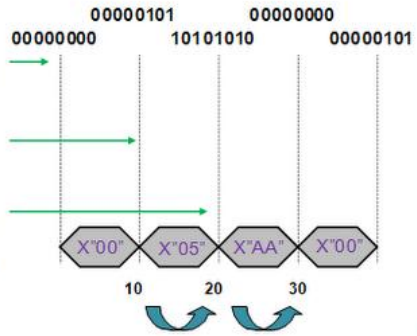

Figura 3 - Generación de datos con tiempo relativo

Tiempo Absoluto: se asignan valores a las señales en los tiempos absolutos de simulación con respecto al momento del comienzo de la simulación.

```
architecture absolute_timing of testbench is
     signal A_BUS : std_logic_vector(7 downto 0);

begin
       A_BUS    <=    "00000000",
                      "00000101"  after 10 ns,
                      "00001010"  after 20 ns;
                                  -- etc.
. . .
end absolute_timing;
```

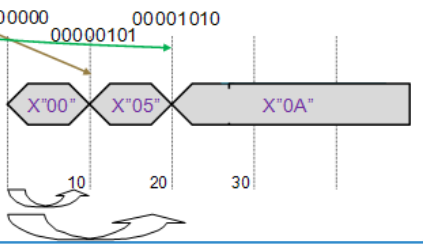

Figura 4 - Generación de datos con tiempo absoluto

2.1.3. Generación de Datos Usando Arreglos

Cuando se desea usar como estímulo ciertos valores de datos, los cuales van a ir cambiando en determinada secuencia de tiempo, el uso de arreglos de datos es bastante común, tal como se detalla en el siguiente ejemplo.

```
architecture array_usage of in_test_benches is

 signal Add_Bus : std_logic_vector(7 downto 0);
 type stimulus is array (0 to 4) of std_logic_vector (7 downto 0);

 constant DATA : stimulus :=
                        ("00000000",   -- declara el estimulo
                         "00000001",   -- como un array.
                         "00000010",   -- Estos datos seran
                         "00000011",   -- usados para estimular
                         "00000100");  -- las entradas

 begin
   stim_proc: process
   begin
    for i in 0 to 4 loop     -- for-loop que asigna
         Add_BUS <= DATA(i); -- a 'Add_Bus' un nuevo valor
         wait for 250 ns;    -- de estimulo cada 10ns
    end loop;
    wait;
 end process stim_proc;
. . .
end array_usage;
```

En el caso descrito en el código arriba, cada dato del arreglo DATA, definido como un arreglo de 5 datos de 8 bits cada uno, es asignado a la señal Add_BUS cada 10 nanosegundos. Este tipo de asignación es totalmente asincrónica, no señal de reloj es usada para nada.

Para una generación de estímulos sincrónica, usando el reloj del sistema bajo test, se puede usar el siguiente código.

```
1  architecture array_usage of in_test_benches is
2
3  -- mismas declaracion que el ejemplo anterior
4
5  begin
6   stim_proc: process
7   begin
8     for i in 0 to 4 loop
9        Add_BUS <= DATA(i);
10        for k in 1 to 7 loop
11           wait until rising_edge(clk);
12        end loop;
13        wait until falling_edge(clk);
14    end loop;
15    wait;
16  end process stim_proc;
17 end array_usage;
```

En este caso, cada dato del arreglo es asignado a Add_BUS en el flanco de bajada del reloj cada siete flancos de subida del reloj.

2.1.4. Generación de Reset

La generación de la señal de reset en bastante sencilla, ya que no es una señal que esté cambiando valores bastante seguido durante el tiempo de simulación. Un ejemplo se muestra en las siguientes líneas de código.

```
1  -- reset totalmente asincronico
2  reset_proc: process
3  begin
4   rst <= '1';
5   wait for 23 ns;
6   rst <= '0';
7   wait for 142 ns;
8   rst <= '1';
9   wait for 54 ns;
10  rst <= '0';
11  wait;
12 end process reset_proc;
```

Tal como se detalla en el código, en este caso se genera un reset asincrónico en la activación (assert) y en la desactivación (desassert). También se destaca la activación del reset en el 'medio' de la simulación, no olvidarse de este pequeño detalle que puede hacernos ver algún problema de inicialización cuando el sistema está en funcionamiento normal.

Como ya expliqué en un blog anterior, en realidad es conveniente que un sistema tenga un reset que se active asincrónicamente y se desactive sincrónicamente. Aquí el código del mismo.

```
1 -- desactivacion sincronica de reset
2 sreset_proc: process
3 begin
4   rst &lt;= '1';
5   for i in 1 to 5 loop
6     wait until clk = '1';
7   end loop;
8   rst <= '0';
9 end process sreset_proc;
```

2.1.5. Verificación de Datos

A fin de contrastar el valor obtenido como salida del DUT ante un cierto estímulo, se usa comúnmente la instrucción assert. De este modo se compara el valor esperado con el obtenido, y dependiendo del resultado de la comparación se lleva a cabo un reporte y se detiene o no la simulación.

En los test bench la instrucción assert también es usada para comprobar restricciones de tiempo del diseño, pero ese es otro tema y en nuestro caso solo la usaremos para verificación de datos.

La sintáxis de la instrucción assert es bastante sencilla: si la expresión booleana de la instrucción assert es falsa se ejecuta lo que está dentro del assert; normalmente es un cierto reporte, que debería detallar el motivo de la ejecución del assert, y un cierto grado de severidad que indica cuán importante es el error detectado con el assert. Un ejemplo de uso es el siguiente:

```
1  assert (out1 = "0110")
2     report"Valor esperado de la salida out1 no es igual a 0110"
3        severity ERROR;
```

En este caso se desea saber si en cierto instante de la simulación la señal *out1* tiene el valor *"0110".* Si la igualdad es falsa, es decir *out1* no es igual a *"0110",* se ejecuta el *assert* reportando el mensaje "*El valor de out no es igual a 0110*", y el nivel de severidad asociado es *ERROR*. En caso de que el problema detectado sea muy grave se puede detener directamente la simulación usando como grado de severidad, *severity*, *FAILURE*.

Un importante punto para usar el *assert* con una condición para saber si es correcto el resultado del DUT ante determinado estimulo, es saber en que momento de la simulación, o mejor en que tiempo de simulación se está preguntando por la condición deseada. Para ello se debería hacer un detallado diagrama de tiempo de los estímulos, y en función del tiempo de simulación de determinado estimulo hacer la pregunta del assert en ese preciso tiempo. Esto puede describirse en VHDL usando por ejemplo *wait for nnn ms*; hasta llegar al tiempo de simulación deseado y luego usar el *assert* respectivo. Otro modo de 'esperar' hasta un cierto valor del estimulo es usar un *wait until* tal como se usa en los test bench ejemplos descriptos a continuación.

El uso de verificación de datos dentro del test bench es una técnica que tiene muchas ventajas a pesar de ser una tarea complicada la escritura del mismo. Pero la automatización de la verificación ayuda mucho en casos que se tengan diferentes test benches, y se quiera verificar el comportamiento del DUT de manera mas rápida. Este procedimiento se usa mucho en lo que se llama "Regression Tests", en los cuales se corren diferentes test benches en modo batch y la respuestas de cada uno se da a conocer por medio de los resultados de los *asserts*.

Más detalle del uso de *assert* puedes encontrarlo en mi previo blog.

3. Conclusión

A modo de conclusión de este blog, a continuación se describen tres distintos test bench de un mismo componente (un decodificador 2:4). Cada TB tiene un nivel de complejidad mayor.

A. Primer Caso: Test Bench simple

```
1  -- Test Bench para verificar
2  -- el comportamiento de un deco 2:4
3  entity tb2_decode is
4  end tb2_decode;
5
6  architecture test_bench of tb2_decode is
7
8  type input_array is array(0 to 3) of std_logic_vector(1 downto 0);
9   constant input_vectors: input_array :=   ("00", "01", "10", "11");
10
11  signal in1  : std_logic_vector (1 downto 0);
12  signal out1 : std_logic_vector (3 downto 0);
13
14  component decode
15   port (
16          in1  : in  std_logic_vector(1 downto 0);
17         out1: out std_logic_vector(3 downto 0));
18  end component;
19 begin
20  decode_1: decode port map(
21                                      in1  => in1,
22                                     out1 => out1);
23
24  -- generacion de los estimulos
25  apply_inputs: process
26  begin
27    for j in input_vectors'range loop
28               In1 <= input_vectors(j);
29               wait for 50 ns;
30    end loop;
31   wait;
32  end process apply_inputs;
33
34  -- verificacion de datos
35  test_outputs: process
36  begin
37    wait until (in1 = "01");
38    wait for 25 ns;
```

```
39    assert (out1 = "0110")
40       report"Output not equal to 0110"
41         severity ERROR
42    -- check the other outputs
43    wait;
44  end process test_outputs;
45  -- ...
46 end test_bench;
```

B. Segundo Caso: Test Bench Elaborado I

```
 1 -- Test Bench elaborado para verificar
 2 -- el comportamiento de un deco 2:4
 3 entity tb3_decode is
 4 end tb3_decode;
 5
 6 architecture test_bench of tb3_decode is
 7
 8  type decoder_test is record
 9   in1  : std_logic_vector(1 downto 0);
10  out1: std_logic_vector(3 downto 0);
11 end record;
12
13  type test_array is array(natural range < >) of decoder_test;
14
15  constant test_data: test_array :=
16   ("00", "0000",
17    "01", "0010",
18    "10", "0100",
19    "11", "1000");
20
21 component decode
22    port (
23          in1 : in  std_logic_vector(1 downto 0);
24         out1: out std_logic_vector(3 downto 0));
25  end component;
26
27 begin
28   decode_1: decode port map(
```

```
29                              in1  => in1,
30                             out1 => out1);
31
32 apply_inputs_test_outputs: process
33  begin
34     for j in test_data'range loop
35           in1 <= test_data(j).in1);
36           wait for 50 ns;
37           assert (out1 = test_data(j).out1)
38              report "Output not equal to the expected value"
39                 severity ERROR;
40     end loop;
41  end process apply_inputs_test_outputs;
42 …
43  end test_bench;
```

En este caso, el arreglo es un poco más complicado, ya que se tienen dos columnas de datos de diferente tamaño cada una. La primer columna representa el estimulo, mientras que la segunda columna representa el valor esperado como respuesta del sistema. Al describir el estímulo de esta manera se puede realizar la comparación, usando assert, dentro del mismo *for-loop* usado para la generación de estímulos tal como se detalla en el código ejemplo de arriba.

Otro punto a destacar de este ejemplo es el hecho que el rango de test_array no ha sido definido en su declaración. Sin embargo cuando se declara la constante test_data el rango del arreglo es definido, de este modo el analizador de VHDL no tendrá problemas en asignarle el rango a la senal test_array durante el proceso de análisis. De esta manera se facilita el cambio del tamaño de la constante sin tener que cambiar la declaración del arreglo. Del mismo modo se explica el uso de 'range' en el for-loop, así el for-loop se ajustara al tamaño del arreglo, sin tener que modificarlo cada vez que se cambie el tamaño del arreglo.

C. Tercar Caso: Test Bench sincrónico

En este caso se debe tener en cuenta el reloj, por ello para se describe un proceso un poco complicado en el cual se genera el reloj, se generan los datos estímulos y también se verifica el resultado. A mi particularmente no me gusta mucho mezclar tanto, sino mas bien tener un proceso para cada tipo de datos, y uno o más para la verificación de datos tal como se detalla mas arriba, pero a modo de ejemplo para tener más idea de lo que se puede hacer, el código que se detalla es un buen ejemplo.

```
1  -- ...
2  begin
3  decode_1: decode port map( in1 =&gt; in1, out =&gt; out1);
4  apply_inputs: process
5  begin
6   for j in test_data'range loop
7           in1 <= test_data(j).in1);
8           clk <= '0';
9           wait for 5 ns;
10         clk <= '1';
11         wait for 5 ns;
12         assert (out1 = test_data(j).out1)
13             report "Output not equal to the expected value"
14               severity ERROR;
15       end loop;
16   wait;
17  end process apply_inputs;
18 -- ...
```

Espero que esta nota técnica sea de utilidad.