Diseño Automático de Sistemas Fiables

Verificación: Sistemas combinacionales

UNIVERSIDAD NEBRIJA

Objetivos Clase

- Entender como verificar un sistema digital combinacional.
- Conocer la verificación formal.
- Poner en practica las técnicas de verificación.
- Conocer comandos específicos para simulación en VHDL

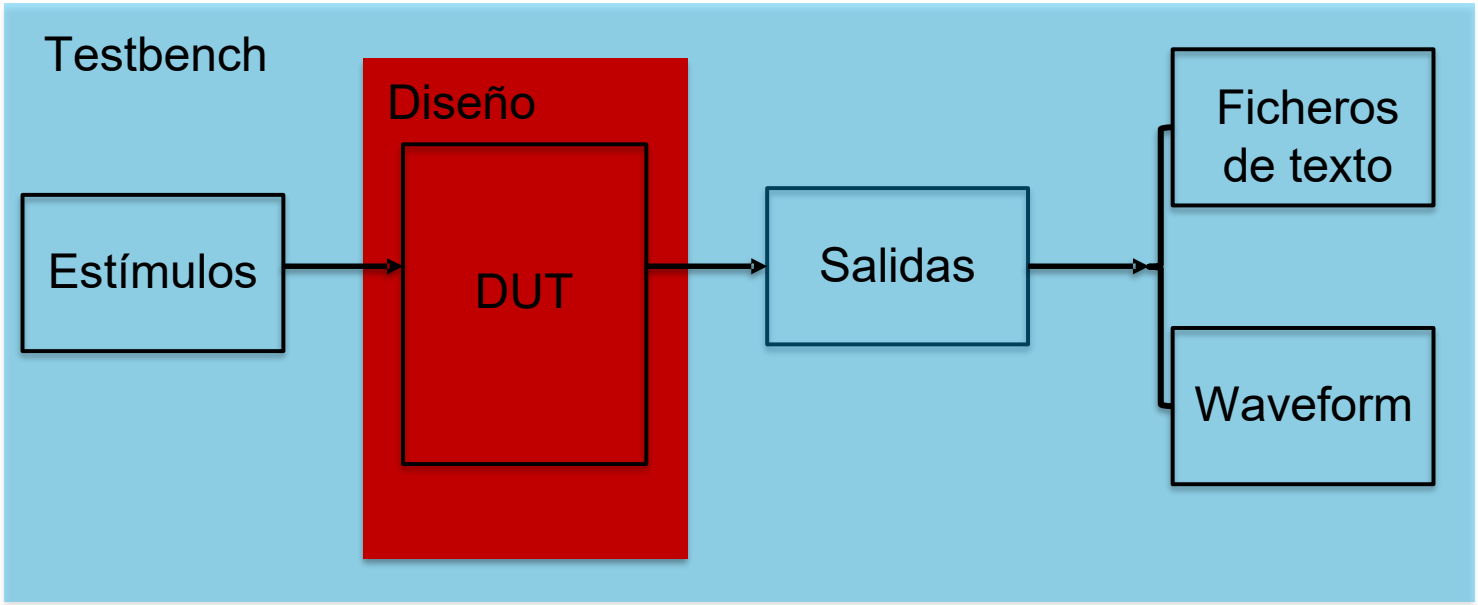
Contenidos

1. Verificación de sistemas digitales
2. Elementos en un testbench
3. Verificación formal
4. Verificación de circuitos combinacionales

Verificación de sistemas digitales

- Comprobar el correcto funcionamiento de un diseño digital es fundamental y es complejo.
- Es muy habitual que una persona diseñe y otra verifique.
- La complejidad de la verificación es en muchos casos casi equivalente al de diseño.
- Es muy habitual utilizar scripts (TCL, Python, Matlab) para automatizar el proceso.

Elementos de un testbench



Elementos de un testbench

- **Testbench:** Fichero que ejecuta las pruebas.
- **DUT:** Diseño bajo prueba por sus siglas en inglés.
- **Estimulos:** Estímulos o vector de estímulos: la serie de entradas que se van a utilizar para excitar el DUT.
- **Salidas:** Salidas o el vector de salidas: la serie de salidas para cada instante del circuito.
- **Ficheros de texto:** Una de las opciones para grabar y visualizar las señales de salida obtenidas.
- **Waveform:** La otra forma habitual de visualizar las salidas de un testbench.

¿Qué es verificar un sistema?

Habitualmente en un diseño nos centramos en verificar el comportamiento, pero también existen una serie de tests no funcionales. Para los tests de comportamiento (o funcionales) existen diversas maneras de hacerlo:

- Vector de pruebas, generado manual o automáticamente (útil en diseños combinacionales).
- Verificación formal, matemáticas aplicadas para expresar una serie de propiedades que se deben cumplir o comprobar la validez del modelo (útil en sistemas complejos).

Verificación formal

- Es un método de verificación que basado en la descripción de un sistema (**especificación formal**), compruebe su funcionamiento mediante un conjunto de propiedades normas o axiomas.
- Necesitamos modelar el sistema formalmente (especificación formal).
- Es un proceso paralelo a la implementación
- Tratamos de asegurar la **ausencia de errores** (frente a los tests que son búsqueda de errores)

Verificación formal

- **Objetivo:**

  Dado un modelo matemático de un sistema:

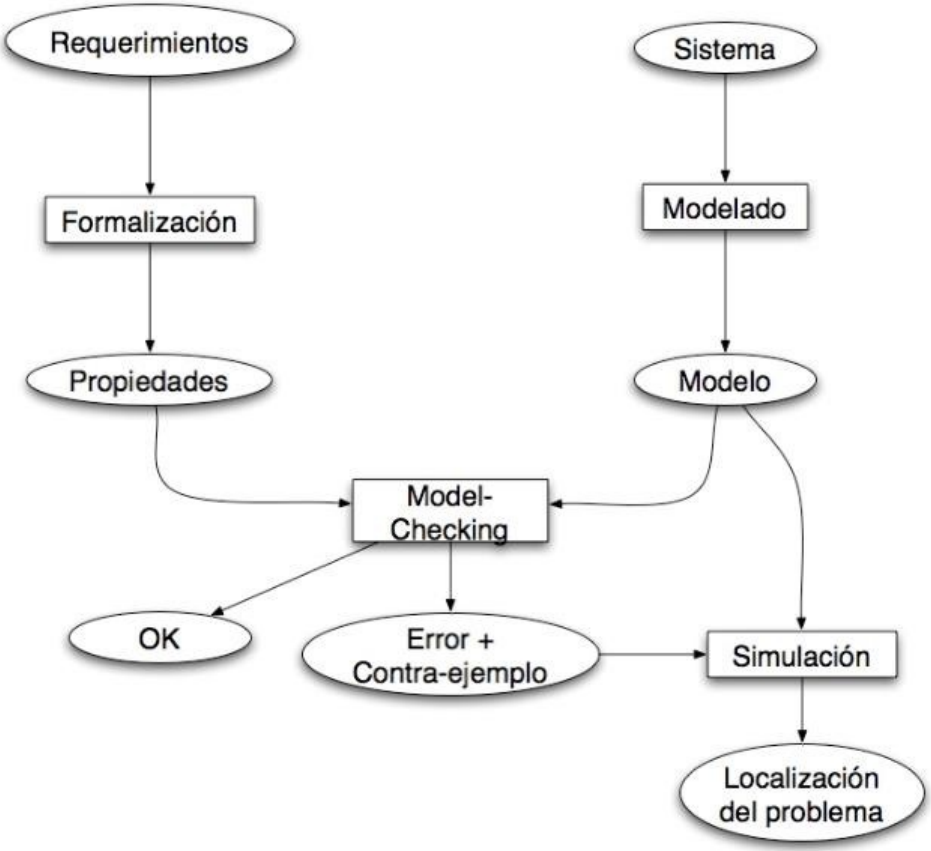
  - Probar que determinadas afirmaciones lógicas son ciertas (o falsas)
    - Propiedades sobre la especificación
    - Propiedades sobre la implementación
- **Técnicas:**
  - Prueba de teoremas (Theorem Proving)
  - Comprobación de modelos (Model Checking)

Verificación formal: Model Checking

- Se obtiene un modelo del sistema (diagrama de estados, ASM) mediante un lenguaje de modelado.
- Escribimos las propiedades.
- Verificamos que se cumplen las propiedades.

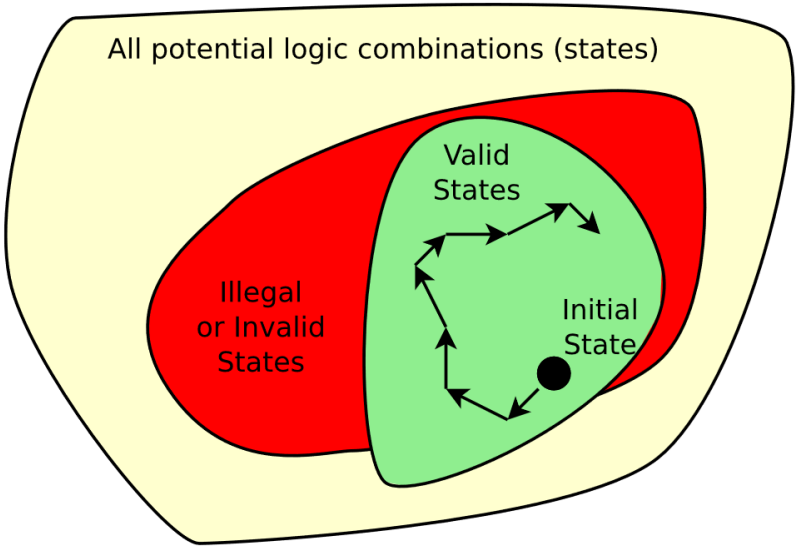
Actualmente Vivado no soporta la verificación formal, es habitual utilizar Modelsim o alternativas en abierto como GHDL + SymbiYosis.

Verificación formal: Model checking



Verificación formal vs tests

- Los tests solo buscan errores en ramas conocidas.
- No comprueban todas las condiciones de contorno.
- No son rigurosos.



Verificación de sistemas combinacionales

Dado un sistema combinacional (como un semisumador)

```
 6  entity half_adder is
 7    port (a, b : in std_logic;
 8          sum, carry : out std_logic
 9      );
10  end half_adder;
11
12  architecture arch of half_adder is
13  begin
14    sum <= a xor b;
15    carry <= a and b;
16  end arch;
```

Verificación de sistemas combinacionales

```
 6  entity half_adder is
 7    port (a, b : in std_logic;
 8          sum, carry : out std_logic
 9      );
10  end half_adder;
11
12  architecture arch of half_adder is
13  begin
14    sum <= a xor b;
15    carry <= a and b;
16  end arch;
```

La verificación la realizamos realizando todas las combinaciones de entrada posibles y comprobando las salidas.

Combinacionales: Test básico

Hasta ahora los tests les habíamos realizado de forma manual usando un código similar al siguiente:

```
a <= '0', '1' after 20 ns, '0' after 40 ns, '1' after 60 ns;
b <= '0', '1' after 40 ns;
```

Después comprobamos en el visor de ondas que el funcionamiento sea el esperado.

Problema: No sabemos cuando hay un error, tenemos que visualizar caso por caso cuando falla.

Combinacionales: Test con assert

Solución: Comprobamos automáticamente los valores con la instrucción *assert*.

```
tb1 : process
    constant period: time := 20 ns;
    begin
        a <= '0';
        b <= '0';
        wait for period;
        assert ((sum = '0') and (carry = '0'))  -- expected output
        -- error will be reported if sum or carry is not 0
        report "test failed for input combination 00" severity error;
```

Combinacionales: Test con assert

Cuando suceda un error se mostrará por consola:

```
Error: Test failed for combination 00
Time: 120 ns  Iteration: 0  Process: /tb_byteAdder/stimulus
File:
E:/git/FPGAs/byteAdder/byteAdder.srcs/sim_1/imports/Simulation/
tb_byteAdder.vhd
```

Problema: Tenemos poca información del error.

Combinacionales: Test con LUT

Solución: Escribimos una tabla (LUT) con las entradas y salidas esperadas.

Incluimos en el mensaje de error los valores de las entradas
y valor esperado de la salida.

```
type test_vector_array is array (natural range <>) of test_vector;
constant test_vectors : test_vector_array := (
    -- a, b, sum , carry   -- positional method is used below
    ('0', '0', '0', '0'), -- or (a => '0', b => '0', sum => '0', carry
    ('0', '1', '1', '0'),
    ('1', '0', '1', '0'),
    ('1', '1', '0', '1'),
    ('0', '1', '0', '1')  -- fail test
    );
```

Combinacionales: Test con LUT

A continuación mediante un bucle *for* recorremos la tabla y vamos comprobando los casos.

```
tb1 : process
begin
    for i in test_vectors'range loop
        a <= test_vectors(i).a;  -- signal a = i^th-row-value of t
        b <= test_vectors(i).b;

        wait for 20 ns;

        assert (
                    (sum = test_vectors(i).sum) and
                    (carry = test_vectors(i).carry)
                )

        -- image is used for string-representation of integer etc.
        report  "test_vector " & integer'image(i) & " failed " &
                " for input a = " & std_logic'image(a) &
                " and b = " & std_logic'image(b)
                severity error;
    end loop;
    wait;
end process;
```

Combinacionales: Test con LUT

Cuando suceda un error se mostrará por consola, esta vez con información sobre el caso:

```
Error: test_vector 0 failed for input a = 0 and b = 0
Time: 20 ns  Iteration: 0  Process: /tb_byteAdder/stimulus
File:
E:/git/FPGAs/byteAdder/byteAdder.srcs/sim_1/imports/Simulation/
tb_byteAdder.vhd
```

Problema: Estamos generando los casos manualmente, pueden faltar o sobrar y además es complicado de modificar

Combinacionales: Test lectura fichero

Solución: Vamos a leer los datos de un fichero externo,
Lo podemos escribir a mano o generar de forma procedural mediante un script (Python, Matlab, Pearl, …)

```
use std.textio.all;
use ieee.std_logic_textio.all; -- require for writing/reading std_logic etc

architecture tb of read_file_ex is
    signal a, b : std_logic;
    signal c : std_logic_vector(1 downto 0);

    -- buffer for storing the text from input read-file
    file input_buf : text;  -- text is keyword

tb1 : process
    variable read_col_from_input_buf : line;
```

Combinacionales: Test lectura fichero

Leer fichero: *file_open(input_buf, ruta al fichero, read_mode);*

```
while not endfile(input_buf) loop
  readline(input_buf, read_col_from_input_buf);
  read(read_col_from_input_buf, val_col1);
  read(read_col_from_input_buf, val_SPACE);          -- re
  read(read_col_from_input_buf, val_col2);
  read(read_col_from_input_buf, val_SPACE);          -- re
  read(read_col_from_input_buf, val_col3);

  -- Pass the read values to signals
  a <= val_col1;
  b <= val_col2;
  c <= val_col3;

  wait for 20 ns;  --  to display results for 20 ns
end loop;
```

Combinacionales: Escritura fichero

Escribir fichero: Escribir en un fichero de texto nos permite comprobar manualmente o mediante un script en otro lenguaje, el correcto funcionamiento.

1. Declarar un file

```
file output_buf : text;  -- text is keyword
```

2. Abrir el fichero

```
file_open(output_buf, "ruta",  write_mode);
```

3. Declarar una variable de tipo line

```
variable write_col_to_output_buf : line; -- line is keyword
```

4. Usar write para escribir en el buffer y write line para escribir en el fichero

```
write(write_col_to_output_buf, string'("a = "));
write(write_col_to_output_buf, a);
write(write_col_to_output_buf, string'(", b = "));
write(write_col_to_output_buf, b);
writeline(output_buf, write_col_to_output_buf);
```

Combinacionales: Escritura fichero

```
tb1 : process
    variable write_col_to_output_buf : line; -- line is keyword
    variable b : integer := 40;
    begin
        a <= '1';  -- assign value to a
        wait for 20 ns;

        -- if modelsim-project is created, then provide the relative
        -- input-file (i.e. read_file_ex.txt) with respect to main pr
        file_open(output_buf, "VHDLCodes/input_output_files/write_fil
        -- else provide the complete path for the input file as show
        --file_open(output_buf, "E:/VHDLCodes/input_output_files/writ

        write(write_col_to_output_buf, string'("Printing values"));
        writeline(output_buf, write_col_to_output_buf);  -- write in

        write(write_col_to_output_buf, string'("a = "));
        write(write_col_to_output_buf, a);
        write(write_col_to_output_buf, string'(", b = "));
        write(write_col_to_output_buf, b);
        writeline(output_buf, write_col_to_output_buf);     -- write i
```

Tests con lectura y escritura

- Podemos combinar la posibilidad de leer y escribir en ficheros para automatizar el proceso y mejorar la legibilidad de los resultados.
- Posibilidad de representar los resultados.
- Requiere un esfuerzo extra, pero una vez hemos creado nuestras funciones básicas podemos reutilizarlas tantas veces como queramos, modificándolas ligeramente.

Resumen

- Conocer que existe la verificación formal y sus ventajas.
- Conocer las estrategias de tests para circuitos combinacionales.