Diseño Digital I

VHDL para test y modelado funcional

Técnicas de simulación y verificación de circuitos complejos

En la realización de *test-benches* VHDL suelen utilizarse construcciones del lenguaje que rara vez se emplean para construir modelos sintetizables de circuitos

- Sentencias **assert**
- **Funciones** y **Procedimientos** definidos por el usuario
- **Ficheros**

Sirven para:

- La definición de **estímulos**:
  - Ficheros para la lectura de **estímulos generados automáticamente**
  - Procedimientos y funciones para el **modelado de alto nivel** del entorno de funcionamiento real del circuito y para la modularización de la asignación de estímulos en procesos
- La **verificación automática** de los resultados de las simulaciones
  - **Escritura de resultados** sobre ficheros para, posteriormente, procesarlos
  - Autoverificación en *test-bench* mediante **sentencias assert**

Sentencia ASSERT (I)

- Las sentencias **ASSERT** sirven para comprobar el cumplimiento de condiciones durante la ejecución de una simulación
- Las condiciones deben formularse como expresiones evaluables a un valor booleano
- Cuando la expresión es FALSE se reporta por consola un mensaje, acompañado por una indicación de la importancia del suceso –un valor del tipo predefinido **SEVERITY_LEVEL**
- Sintaxis:

```
ASSERT Condición Lógica
REPORT  String
SEVERITY Valor SEVERITY_LEVEL;
```

- Los valores del tipo **SEVERITY_LEVEL** son **NOTE**, **WARNING**, **ERROR** y **FAILURE**
- La sentencia se puede ejecutar secuencial o concurrentemente –si en la expresión de la condición lógica hay señales

Sentencia ASSERT (II)

```
architecture rtl of reg_file_M_Nbits is
  -- M: número de registros
  -- N: número de bits de cada registro
  type reg_file is array (NATURAL RANGE <>) of std_logic_vector(N-1 downto 0);
  signal reg_file_op: reg_file(M-1 downto 0);

begin

  assert conv_integer(Dir_RD) < M
    report "Direccion invalida de lectura de registro."
    severity warning;

  assert conv_integer(Dir_WR) < M
    report "Direccion invalida de escritura de registro."
    severity warning;

  process(clk, rst_n)
  begin
    if not rst_n then
      for i in reg_file_op'range loop
        reg_file_op(i) <= (others => '0');
      end loop;

    elsif clk'event and clk = '1' then
      if WR then
        if conv_integer(Dir_WR) < M then
          reg_file_op(conv_integer(Dir_WR)) <= Dato_in;
        end if;
      end if;
    end if;
   end process;

   Dato_out <= reg_file_op(conv_integer(Dir_RD)) when conv_integer(Dir_RD) < M
               else (others => 'X');

end rtl;
```

SENTENCIAS assert CONCURRENTES

Sentencias de comprobación de condiciones

Sintaxis completa de la Declaración de Entidad

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

use work.auxiliar.all;

entity reg_file_M_Nbits is
generic(N: in natural := 32;     -- Número de bits
        M: in natural := 32);    -- Número de registros

port(rst_n:       in  std_logic;
     clk:         in  std_logic;
     WR:          in  std_logic;
     Dir_WR:      in  std_logic_vector(ceil_log(M)-1 downto 0);
     Dir_RD:      in  std_logic_vector(ceil_log(M)-1 downto 0);
     Dato_in:     in  std_logic_vector(N-1 downto 0);
     Dato_out:    out std_logic_vector(N-1 downto 0));

type reg_file is array (NATURAL RANGE <>) of std_logic_vector(N-1 downto 0);

begin
  assert conv_integer(Dir_RD) < M
    report "Direccion invalida de lectura de registro."
    severity warning;

  assert conv_integer(Dir_WR) < M
    report "Direccion invalida de escritura de registro."
    severity warning;

end entity;
```

ZONA DE DECLARACIÓN

Tipos de datos, constantes, señales, subprogramas…

PROCESOS PASIVOS

Procesos que no asignan valor a señales

Sentencia ASSERT (III)

Las sentencias **ASSERT** pueden utilizarse también para definir fácilmente la finalización de las simulaciones

```
PROCESS
 BEGIN
 A <= '0';
 ……..
 ASSERT FALSE
 SEVERITY FAILURE;
END PROCESS;
```

Funciones y procedimientos

- Las **funciones** VHDL:
  - ✓ **no son** sentencias
  - ✓ devuelven un valor
  - ✓ dentro de ellas, no puede asignarse valor a señales o incluirse sentencias **WAIT**.

- Los **procedimientos** VHDL:
  - ✓ **son** sentencias
  - ✓ pueden ejecutarse secuencial o concurrentemente
  - ✓ pueden incluir sentencias **WAIT** y asignar valores a señales

- **Declaración** de funciones y procedimientos
  - ✓ Puede realizarse en cualquier zona declarativa: Declaración de Paquete, Zona de declaración de un Cuerpo de Arquitectura o de un proceso

- **Definición** de funciones y procedimientos
  - ✓ La definición del procedimiento puede realizarse por separado o en la propia declaración
  - ✓ Cuando la función se declara en una Declaración de Paquete, la definición se realiza en el Cuerpo del Paquete

Sintaxis de la declaración de funciones y procedimientos

```
FUNCTION    NOMBRE  (LISTA_DE_ARGUMENTOS) RETURN TIPO_DE_DATOS;
PROCEDURE   NOMBRE  (LISTA_DE_ARGUMENTOS);
```

La lista de argumentos es el conjunto de parámetros que devuelve o se pasan al subprograma:

- ✓ En las funciones todos los argumentos son de entrada
- ✓ En los procedimientos, los parámetros pueden ser: de entrada (IN), de salida (OUT) y bidireccionales (INOUT)
- ✓ Los argumentos pueden ser cualquier tipo de objeto
- ✓ En la declaración de un argumento puede omitirse la dirección y el tipo de objeto; en tal caso:
  - Se considerará por defecto que es un argumento de entrada
  - Si se omite el tipo de objeto se considerará que es una CONSTANTE, si es un argumento de entrada, y una VARIABLE, si es de salida o bidireccional
- ✓ Las funciones devuelven un valor del TIPO DE DATOS indicado en su declaración

Ejemplos de declaración de funciones y procedimientos

```
function retardo_en_tclk (constant retardo, tclk: in time) return natural;
```

```
function dentro_de_rango (signal a: in std_logic_vector;
                          constant sup, inf: integer) return boolean;
```

```
procedure tecleo ( signal      ena_cmd:      out std_logic;
                   signal      cmd_tecla:    out std_logic_vector(3 downto 0);
                   signal      clk:          in  std_logic;
                   constant    tecla:        in  std_logic_vector(3 downto 0));
```

Sintaxis de la definición de funciones y procedimientos

```
FUNCTION/PROCEDURE NOMBRE(LISTA_DE_ARGUMENTOS) [RETURN…] IS
    ZONA DE DECLARACIÓN
BEGIN
    ALGORITMO DE PROCESAMIENTO SECUENCIAL
END FUNCTION/PROCEDURE;
```

- ✓ En la zona de declaración del subprograma pueden declararse constantes, variables, tipos de datos e, incluso, otros subprogramas
- ✓ Todas las declaraciones son locales
- ✓ El algoritmo puede construirse con sentencias de ejecución secuencial
- ✓ En los procedimientos pueden realizarse asignaciones de valor a señal; en las funciones no se puede
- ✓ Las funciones deben devolver un valor, del tipo de datos indicado en su declaración, utilizando una sentencia **RETURN**

Sintaxis de las sentencias de funciones y procedimientos

NOMBRE (PROCEDURE) (LISTA_DE_ASOCIACIÓN);
OBJETO <= NOMBRE (FUNCIÓN) (LISTA_DE_ASOCIACIÓN);

- ✓ La lista de asociación puede construirse mediante conexión explícita o por posición
- ✓ Si se omite algún parámetro, es obligatorio que en la declaración del subprograma tenga asignado un valor por defecto
- ✓ Los procedimientos pueden ejecutarse secuencial o concurrentemente –si alguno de sus parámetros de entrada es una señal
- ✓ Una función puede estar en la parte derecha –asignación- de una sentencia concurrente si alguno de sus parámetros de entrada es una señal

EJEMPLO: **tecleo (ena_cmd, cmd_tecla, clk, X"F");**

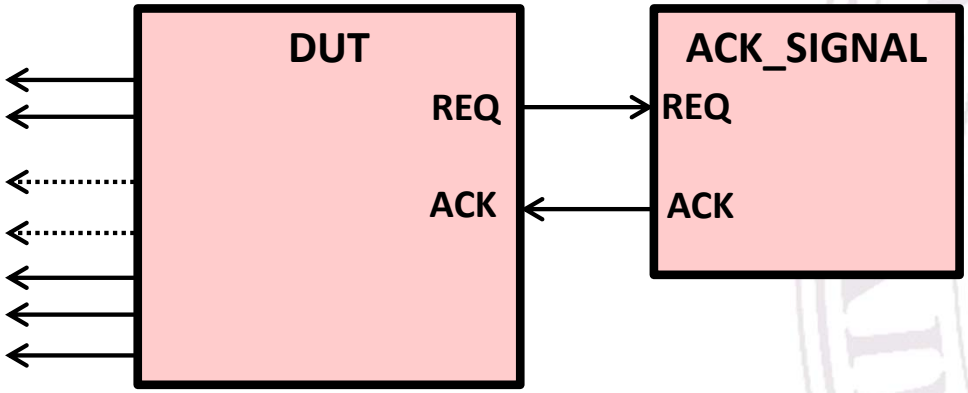
Ejemplo de definición de funciones

```
function hora_to_natural (hora: std_logic_vector(23 downto 0)) return natural is
    variable resultado: natural := 0;

  begin
    resultado := 10*conv_integer(hora(23 downto 20));
    resultado := resultado + conv_integer(hora(19 downto 16));
    resultado := resultado * 3600;
    resultado := resultado + 600*conv_integer(hora(15 downto 12));
    resultado := resultado + 60*conv_integer(hora(11 downto 8));
    resultado := resultado + 10*conv_integer(hora(7 downto 4));
    resultado := resultado + conv_integer(hora(3 downto 0));
    return resultado;

  end function;
```

Ejemplo de definición de procedimientos

- Los procedimientos pueden usarse para realizar modelos funcionales de sistemas para test

```
procedure ACK_signal(signal REQ: in  std_logic;
                     signal ACK: out std_logic;
                     signal CLK: in  std_logic) is
begin
  ACK <= '0';
  wait until clk'event and clk = '1' and REQ = '1';
  ACK <= '1';
  wait until clk'event and clk = '1';
  ACK <= '0';
  if REQ then
    wait until clk'event and clk = '1' and REQ = '0';
  end if;

end ACK_signal;
```



Paquetes VHDL: conceptos

- Cumplen funciones equivalentes a las de las librerías en los lenguajes de programación de alto nivel
- Contienen, fundamentalmente, tipos de datos y operadores definidos por el usuario, subprogramas y componentes
- Se construyen con dos unidades del lenguaje:
  - La **Declaración de Paquete**: es la vista pública del Paquete

```
package {nombre del paquete} is
  {zona de declaración}
end {nombre del paquete};
```

  - El **Cuerpo de paquete**, que contiene la definición de los operadores y subprogramas que aparecen en la Declaración del Paquete

```
package body {nombre del paquete} is
  {zona de declaración}
end {nombre del paquete};
```

Paquetes VHDL: ejemplo

```
package auxiliar is
 -- La función calc_log calcula el mínimo valor, n, que
 -- cumple que 2**n es mayor o igual que x
 function calc_log(x: in natural) return natural;

end package;

package body auxiliar is
function calc_log(x: in natural) return natural is
 begin

 for n in 1 to 16 loop
   if 2**n >= x then
     return n;
   end if;
 end loop;
 return 0; --error
end calc_log;

end package body;
```

DECLARACIÓN DE PAQUETE

Prototipo (declaración) de una función (**calc_log**) que realiza un cálculo matemático

SI EL PAQUETE ESTÁ ALMACENADO EN LA MISMA LIBRERÍA QUE LAS UNIDADES DE DISEÑO QUE LO USAN (EN LA LIBRERÍA WORK), PUEDE OBTENERSE VISIBILIDAD SOBRE ÉL SIN DECLARAR LA LIBRERÍA:

USE WORK.AUXILIAR.ALL;

CUERPO DE PAQUETE

definición de la función **calc_log**

SENTENCIA return

Indica el valor que devuelve la función

Ficheros (I)

- En VHDL pueden declararse y utilizarse ficheros
  - Los ficheros se utilizan, principalmente, en los Test-Benches: para almacenar los estímulos y/o los resultados de las pruebas
  - En ocasiones se utilizan también para la realización de modelos funcionales (de memorias, por ejemplo)
- Para utilizar un fichero hay que:
  - Declarar un tipo de datos que especifica el contenido del fichero
  - Declarar un objeto (fichero) del tipo creado
- Sintaxis de la declaración del tipo de fichero

```
TYPE Nombre_del_tipo_de_fichero IS FILE OF Tipo_de_Datos;
```

- Ejemplos:

```
TYPE Vectores IS FILE OF std_logic_vector;

TYPE Mensajes IS FILE OF string;
```

- En el paquete TEXTIO está definido el tipo TEXT (Fichero de String)

Ficheros (II)

- Sintaxis de la declaración de un fichero

  FILE Nombre_del_fichero: Tipo_de_Fichero OPEN Modo IS Nombre_lógico;

- Los únicos campos obligatorios son el nombre y el tipo de fichero
- El nombre lógico identifica el nombre del fichero en el Host
- La cláusula OPEN requiere que se indique el nombre lógico del fichero y abre dicho fichero de acuerdo con el valor indicado en el campo Modo
  - READ_MODE
  - WRITE_MODE
  - APPEND_MODE

- Ejemplos

```
FILE Fichero_Sim: Vectores;
FILE Fichero1: Vectores IS “Sim.dat”;
FILE Fichero2: Vectores OPEN READ_MODE IS “Mis_vectores.dat”;
FILE Fichero3: Resultados OPEN WRITE_MODE IS “Resultados.dat”;
```

Ficheros (III)

- Para abrir, cerrar, leer o escribir datos en ficheros se dispone de los siguientes procedimientos predefinidos:
  - **FILE_OPEN**

```
procedure FILE_OPEN(file F: TEXT;
                    File_Name: in STRING;
                    Open_Kind: in FILE_OPEN_KIND:=READ_MODE);

procedure FILE_OPEN(File_Status: out FILE_OPEN_STATUS;
                    file F: TEXT;
                    File_Name: in STRING;
                    Open_Kind: in FILE_OPEN_KIND:=READ_MODE);
```

  - **FILE_CLOSE**

```
procedure FILE_CLOSE (file F: TEXT);
```

  - **READ**

```
procedure READ (file F: TEXT; VALUE: out STRING);
```

  - **WRITE**

```
procedure WRITE (file F: TEXT; VALUE: in STRING);
```

  - **READLINE**

```
procedure READLINE (file F: TEXT; L: inout LINE);
```

  - **WRITELINE**

```
procedure WRITELINE (file F: TEXT; L: inout LINE);
```

Ficheros (III)

- Además se dispone de una función que permite detectar el fin de fichero
  - **ENDFILE** `function ENDFILE (file F: TEXT) return BOOLEAN;`
- En el paquete std_logic_textio de la librería ieee se definen procedimientos para leer o escribir datos de tipo std_logic.
- Algunos ejemplos:

```
procedure READ(L:inout LINE; VALUE:out STD_LOGIC_VECTOR);

procedure HREAD(L:inout LINE; VALUE:out STD_LOGIC_VECTOR; GOOD: out BOOLEAN);
```

(para caracteres en hexadecimal)

```
procedure WRITE(L:inout LINE; VALUE:in STD_LOGIC_VECTOR;
                JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
```

Modelado funcional con ficheros

MODELO FUNCIONAL DE UNA MEMORIA ROM

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

use std.textIO.all;
use ieee.std_logic_textio.all;

entity ROM_1KByte is
port(nCS:   in  std_logic;
     Add:   in  std_logic_vector(7 downto 0);
     Data:  out std_logic_vector(31 downto 0));
end entity;
```

PAQUETES

En el paquete **textIO** está definido el tipo **text** y hay funciones para leer y escribir en un fichero valores de los tipos predefinidos del lenguaje

En **std_logic_textio** hay funciones para leer y escribir valores de tipo **std_logic**

```
architecture sim of ROM_1KByte is
  type mem32b is array(natural range<>) of std_logic_vector(31 downto 0);
  file datos_rom:  text open read_mode is "datos.txt";
  shared variable memROM: mem32b(255 downto 0);

begin
  Data <= memROM(conv_integer(Add)) when not nCS else
          (others => 'Z');

  process
    variable linea: line;
    variable OK: boolean;
    variable valor: std_logic_vector(31 downto 0);
    variable add_ROM: integer range 0 to 255;

  begin
    add_ROM := 0;
    while not endfile(datos_ROM) loop
      readline(datos_ROM, linea);
      OK := TRUE;
      while OK loop
        hread(linea, valor, OK);
        if OK then
          memROM(add_ROM) := valor;
          add_ROM := add_ROM + 1;
        end if;
      end loop;
    end loop;
    wait;
  end process;
end sim;
```

VARIABLE GLOBAL

Modelado funcional con ficheros

MODELO FUNCIONAL DE UNA MEMORIA ROM

```
architecture sim of ROM_1KByte is
  type mem32b is array(natural range<>) of std_logic_vector(31 downto 0);
  file datos_rom:  text open read_mode is "datos.txt";
  shared variable memROM: mem32b(255 downto 0);

begin
  Data <= memROM(conv_integer(Add)) when not nCS else
          (others => 'Z');

  process
    variable linea: line;
    variable OK: boolean;
    variable valor: std_logic_vector(31 downto 0);
    variable add_ROM: integer range 0 to 255;

  begin
    add_ROM := 0;
    while not endfile(datos_ROM) loop
      readline(datos_ROM, linea);
      OK := TRUE;
      while OK loop
        hread(linea, valor, OK);
        if OK then
          memROM(add_ROM) := valor;
          add_ROM := add_ROM + 1;
        end if;
      end loop;
    end loop;
    wait;
  end process;
end sim;
```

DECLARACIÓN DE FICHERO

El tipo de fichero **text** está declarado en el paquete **std.textio**

DECLARACIÓN DE PUNTERO

LINE es un tipo declarado en el paquete **std.textIO**

```
type LINE is access STRING; -- A LINE is a pointer
                            -- to a STRING value.
```

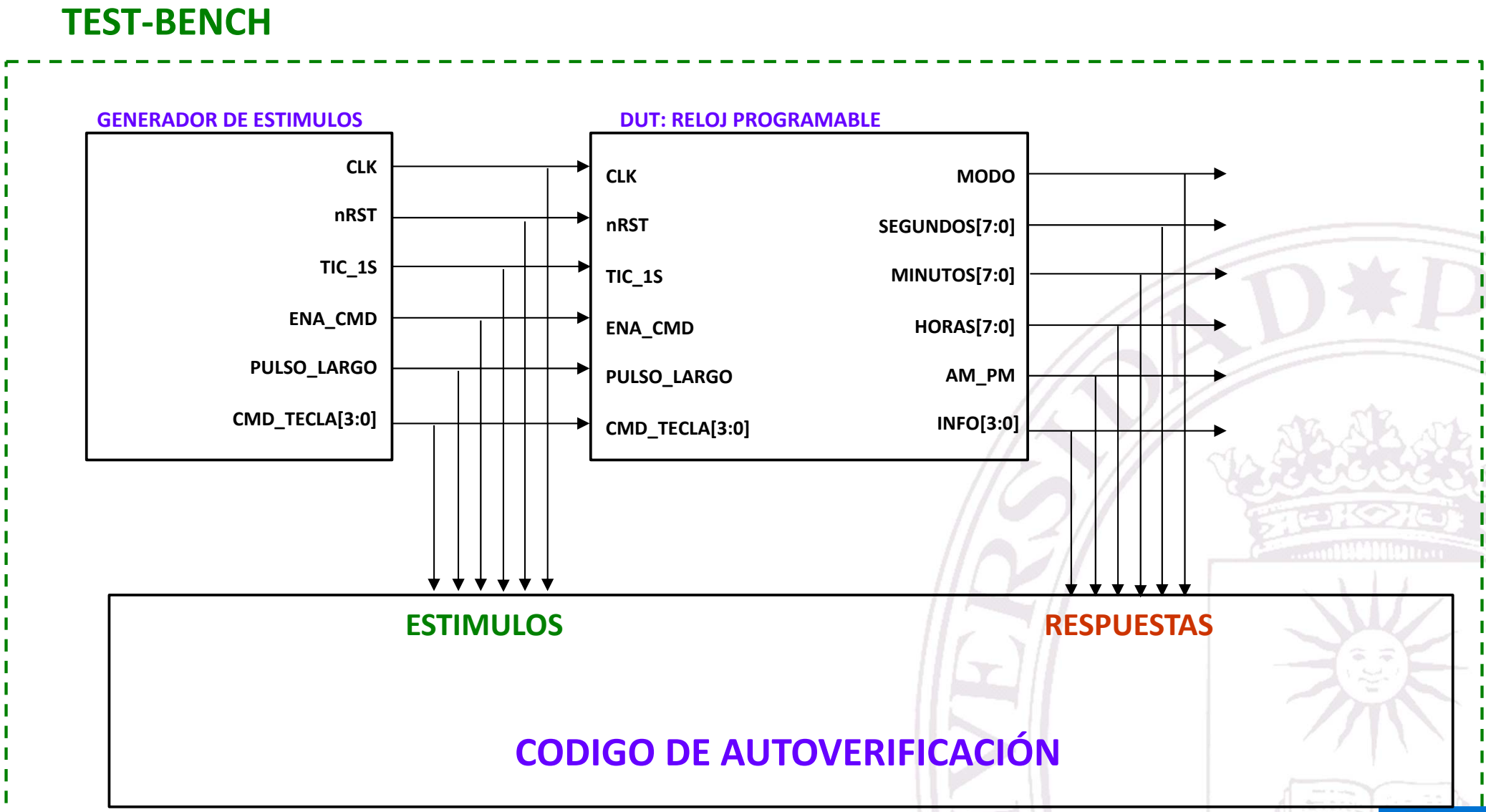
FUNCIÓN PREDEFINIDA

Devuelve un valor *true* cuando se alcanza el final del fichero

PROCEDIMIENTO DEFINIDO EN TEXTIO

PROCEDIMIENTO DEFINIDO EN STD_LOGIC_TEXTIO

Código de autoverificación: concepto



Código de autoverificación: ejemplo

```
 -- Verificación del comando de pasar a modo de programación de reloj
process(clk, nRst)
   variable cmd_tecla_T1:   std_logic_vector(3 downto 0);
   variable ena_assert:        boolean := false;
   variable pulso_largo_T1: std_logic;
   variable info_T1:           std_logic_vector(1 downto 0);

 begin
   if nRst'event and nRst = '0' then
     ena_assert := false;

   elsif nRst'event and nRst = '1' and nRst'last_value = '0' then
     ena_assert := true;

   elsif clk'event and clk = '1' and ena_assert then
     if pulso_largo_T1 = '1' and cmd_tecla_T1 = X"A" and info_T1 = 0 then
       assert  info = 2
       report "Error detectado por el monitor :-)"
       severity error;
     end if;

     cmd_tecla_T1 := cmd_tecla;
     pulso_largo_T1 := pulso_largo;
     info_T1 := info;

   end if;
 end process;
```

- Más información: https://blogs.upm.es/ue-upm/

- Contacto: comunidad.microelectronica@upm.es