DISEÑO DIGITAL I

Ejercicios sobre tipos de datos, atributos, sentencias *assert*, subprogramas y modelos paramétricos

DD1-A3

Introducción

Esta actividad consta de una serie de ejercicios que se irán intercalando durante las presentaciones de teoría. Algunos cuentan con una breve introducción teórica y en ellos se plantean cuestiones de análisis de código o que requieren búsqueda de información y, en otros casos, es necesario completar modelos VHDL y realizar simulaciones en Questa para verificarlos.

La búsqueda de información para completar la actividad es, en sí misma, una parte importante de la misma.

PARTE I: Tipos de datos

Los diseñadores del lenguaje VHDL tomaron la decisión de que los tipos de datos predefinidos (BIT, INTEGER, REAL, etc.) y las operaciones de entrada-salida sobre ficheros se declararan en dos Paquetes VHDL: los Paquetes STANDARD y TEXTIO de la librería STD. Debido a que la visibilidad sobre el primero de estos paquetes es automática, puede trabajarse con los tipos de datos que contiene sin emplear las cláusulas **library** o **use**.

Todos los entornos de simulación o síntesis VHDL proporcionan, además de la librería STD, la librería IEEE. La librería IEEE contiene:

- El paquete estandarizado **std_logic_1164** –en el que se definen los tipos estándar para síntesis (**std_logic** y **std_logic_vector**), así como la declaración de un conjunto de operaciones lógicas básicas para ellos.

- Cuatro paquetes, desarrollados originalmente por *Synopsys* (una empresa de CAD electrónico), que han sido durante muchos años el estándar *de facto* para la construcción de modelos sintetizables. Dos de ellos, los paquetes **std_logic_unsigned** y **std_logic_signed**, en los que se declaran operaciones que interpretan los valores de tipo **std_logic_vector** como números en binario natural o complemento a dos, son los que se han venido utilizando comúnmente para la realización de modelos sintetizables. En un tercer paquete, denominado **std_logic_arith**, se definen dos tipos de datos **unsigned** y **signed**, y operaciones aritméticas para operar con ellos: está orientado a la realización de modelos sintetizables en los que se mezclan datos en binario natural y complemento a dos. El cuarto paquete, **std_logic_textio** tiene definidas operaciones de entrada-salida para leer y escribir valores de tipo **std_logic** y **std_logic_vector** sobre ficheros.

- Un conjunto adicional de paquetes, estandarizados por IEEE, que desarrollan distintas interpretaciones semánticas de los datos definidos en el paquete **std_logic_1164**. De este grupo forman parte, entre otros de menor interés, los paquetes **numeric_std** y **numeric_std_unsigned** que equivalen y pueden sustituir, aproximada y respectivamente, a los paquetes **std_logic_arith** y **std_logic_unsigned** de *Synopsys*.

El paquete más comúnmente utilizado para la construcción de modelos sintetizables –al margen, obviamente del paquete **std_logic_1164**- es el **std_logic_unsigned**, que puede ser sustituido en la mayor parte de los casos por el paquete **numeric_std_unsigned**.

Nota: El contenido de los paquetes estandarizados por IEEE puede ser consultado en la pestaña 'Library' de QuestaSim

Ejercicio 1.1.

En VHDL pueden definirse tipos y subtipos de datos. ¿Qué diferencias existen entre ambos?

Ejercicio 1.2.

Los tipos y subtipos predefinidos en el lenguaje VHDL se encuentran declarados en el paquete STANDARD, de la librería STD –esta librería forma parte del entorno predefinido del lenguaje y contiene, además, el paquete TEXTIO; su contenido puede revisarse en cualquier herramienta de simulación VHDL (como Questa, por ejemplo).

a. Haga una lista que enumere todos los tipos y subtipos escalares predefinidos en el lenguaje VHDL, indicando para cada uno su naturaleza (tipo o subtipo).

b. ¿Qué valores forman parte del tipo BIT? ¿Cuáles son las operaciones predefinidas en VHDL para los valores de tipo BIT?

c. El tipo TIME es un tipo físico. ¿Qué es un tipo físico VHDL?

Ejercicio 1.3.

Revise el contenido de la declaración del paquete **std_logic_1164** para contestar las siguientes cuestiones.

a. ¿Cuántos tipos y subtipos de datos se declaran en este paquete? Indique el nombre de cada uno y agrúpelos, atendiendo al modo en que se definen, en las categorías: **ENUMERADOS**, con función de **RESOLUCIÓN**, como **ARRAY** o mediante una **RESTRICCIÓN** de rango.

b. ¿En qué se diferencia el subtipo **std_logic** del tipo **std_ulogic**?

PARTE II: Atributos

Los atributos permiten obtener información de un tipo de datos (escalar o array) o de un objeto (señal o constante). A continuación, se presentan algunos de los atributos predefinidos del lenguaje donde **T** es cualquier tipo, **A** es cualquier tipo array y **S** representa cualquier señal.

T'left	devuelve el valor más a la izquierda del tipo T.
A'left	devuelve el subíndice del elemento más a la izquierda del array A.
T'right	devuelve el valor más a la derecha del tipo T.
A'right	devuelve el subíndice del elemento más a la derecha del array A.
T'high	devuelve el valor más grande del tipo T.
A'high	devuelve el subíndice más alto del array A.
T'low	devuelve el valor más pequeño del tipo T.
A'low	devuelve el subíndice más bajo del array A.
T'ascending	devuelve ***true*** si el rango de T está definido ***(n to N)***.
T'image(X)	devuelve un ***string*** con la representación del valor X que debe ser del tipo T.
T'value(X)	devuelve el valor del ***string*** X convertido al tipo T.
T'pos(X)	devuelve un ***integer*** que indica la posición de X en el tipo T.
T'val(X)	devuelve el valor del tipo T en la posición entera X.
T'succ(X)	devuelve el valor del tipo T que va después de X.
T'pred(X)	devuelve el valor del tipo T que va antes de X.
T'leftof(X)	devuelve el valor del tipo T que está a la izquierda de X.
T'rightof(X)	devuelve el valor del tipo T que está a la derecha de X.
A'range	devuelve el rango *A'left to A'right* o *A'left downto A'right*.
A'reverse_range	devuelve el rango de A con ***to*** y ***downto*** invertidos.
A'length	devuelve un ***integer*** que es el número de elementos del array A.
A'ascending	devuelve ***true*** si el rango del array A está definido con ***to***.
S'delayed(t)	devuelve la señal S retrasada un tiempo t (t es un objeto de tipo ***time***).
S'stable	devuelve ***true*** si la señal S no ha cambiado de valor en el ciclo de simulación actual.
S'stable(t)	devuelve ***true*** si la señal S no ha cambiado de valor en un tiempo t.
S'quiet	devuelve ***true*** si no se han proyectado cambios en la señal S en el ciclo de simulación actual.
S'quiet(t)	devuelve ***true*** si no se han proyectado cambios en la señal S en un tiempo t.
S'event	devuelve ***true*** si la señal S ha tenido un evento en el ciclo de simulación actual.
S'active	devuelve ***true*** si la señal S está activa en el ciclo de simulación actual.
S'last_event	devuelve el tiempo desde el último evento en la señal S.
S'last_active	devuelve el tiempo desde que la señal S se activó por última vez.
S'last_value	devuelve el valor anterior de la señal S.

En la documentación adjunta dispone del fichero **atributos.vhd**. Observe que la declaración de entidad no tiene puertos y, a diferencia de los *test-benches* que ha venido realizando, no instancia ningún modelo para simular. Sin embargo, puede compilar y ejecutar la simulación del mismo para familiarizarse con algunos de los atributos. Hay sentencias completas y otras -que aparecen comentadas- que debe completar.

PARTE III: Sentencias ASSERT

En la documentación adjunta dispone del fichero **aserciones.vhd** que incluye el modelo de un *flip-flop*. Dispone también de un *test-bench* (**aserciones_tb.vhd**) que, además de generar las señales de reset, reloj y entrada, incluye tres sentencias **assert**. Analice el código de las dos últimas y escriba el mensaje que debe visualizarse en la consola, cuando no se cumplen.

```
process(D_in, clk)
  begin
    if clk'event and clk = '1' then
      assert D_in'stable(2 ns)
        report "Escriba el mensaje adecuado al error"
        severity warning;
    end if;

    if D_in'event and clk = '1' then
      assert clk'last_event > 1 ns
        report "Escriba el mensaje adecuado al error"
        severity warning;
    end if;
end process;
```

Ejecute la simulación para comprobar el funcionamiento de las sentencias **assert**.

PARTE IV: Subprogramas

Además de los operadores predefinidos, el lenguaje VHDL permite el uso de subprogramas para el diseño de operaciones de computación y el modelado hardware. El lenguaje soporta dos tipos de subprogramas: las **funciones** y los **procedimientos**, que son semejantes a las existentes en los lenguajes de programación de alto nivel, diferenciándose por tanto entre sí en que:

- La invocación de una función es una expresión (un operador que devuelve un valor), mientras que la de un procedimiento es una sentencia.

- Los parámetros de las funciones son siempre "de entrada" y por defecto se consideran constantes, mientras que los de los procedimientos pueden ser "de entrada", "de salida" o bidireccionales (por defecto los parámetros de entrada se consideran constantes y los demás variables).

Para hacer uso de un subprograma hay que definir su interfaz, que consta de un nombre identificativo y de una lista de parámetros, en una **Declaración de Subprograma** y su funcionalidad en un **Cuerpo del Subprograma**, donde se describe la operación del mismo mediante sentencias secuenciales.

Los subprogramas suelen encapsularse en **Paquetes VHDL**, de manera que la interfaz se declara en la **Declaración de Paquete** –la vista pública– y el cuerpo del subprograma se define en el **Cuerpo de Paquete**; en general pueden definirse en la zona de declaración de cualquier construcción VHDL (en una Declaración de Entidad, en un Cuerpo de Arquitectura, en los procesos, en la zona de declaración de otro subprograma, etc.).

Por ejemplo, la declaración de la función **CONV_INTEGER** en la Declaración del Paquete **std_logic_unsigned**, de la librería **IEEE**, se realiza de la siguiente forma:

```
function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER;
```

Como puede verse, la función toma como argumento un objeto o valor de tipo **std_logic_vector** y devuelve un valor entero: el correspondiente a su decodificación en **binario natural**. La información que proporciona la declaración del subprograma resulta suficiente para emplearlo en la realización de modelos o *test-benches*.

El código de la definición de la función **CONV_INTEGER** está en el Cuerpo del Paquete **std_logic_unsigned**, y es el siguiente:

```
function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER is
  variable result    : UNSIGNED(ARG'range);
begin
  result    := UNSIGNED(ARG);
  return   CONV_INTEGER(result);
end;
```

Para descifrar cómo opera esta función hay que tener en cuenta que el cuerpo del paquete **std_logic_unsigned** va precedido por cláusulas de visibilidad que le permiten hacer uso del contenido de los paquetes **std_logic_1164** y **std_logic_arith**. Sabiendo esto:

- En la función **CONV_INTEGER** se declara una variable, **result**, de tipo **UNSIGNED**, un tipo declarado en el paquete **std_logic_arith**, de la librería **IEEE**, para representar números codificados en binario natural.
- La variable **result** se declara con un rango igual al del parámetro de tipo **std_logic_vector** con que se llama a la función, rango que se obtiene aplicando al argumento de la función el atributo **'range** (**ARG'range**).
- Para calcular el valor devuelto por la función se hace uso, primero, de una operación de conversión de tipos[1] (estas conversiones sólo pueden aplicarse a los tipos que el lenguaje VHDL denomina "*closely related types*" y los tipos **std_logic_vector** y **unsigned** lo son porque su definición es idéntica), para convertir el argumento de tipo **std_logic_vector** a **unsigned** y poder asignarlo a la variable **result**:

```
result    := UNSIGNED(ARG);
```

Después se obtiene el valor **integer** de vuelta llamando a una función, que tiene el mismo nombre, **CONV_INTEGER**, que la que estamos analizando (recuerde que en VHDL se puede "sobrecargar" el nombre de subprogramas y operaciones), pero que está definida en el paquete **std_logic_arith** y calcula el valor entero de un parámetro de tipo **UNSIGNED**. El código de su definición es el siguiente:

```
function CONV_INTEGER(ARG: UNSIGNED) return INTEGER is
  variable result: INTEGER;
  variable tmp: STD_ULOGIC;
-- synopsys built_in SYN_UNSIGNED_TO_INTEGER
-- synopsys subpgm_id 366
begin
-- synopsys synthesis_off
  assert ARG'length <= 31
  report "ARG is too large in CONV_INTEGER"
  severity FAILURE;
  result := 0;
  for i in ARG'range loop
    result := result * 2;
    tmp := tbl_BINARY(ARG(i));
    if tmp = '1' then
      result := result + 1;
    elsif tmp = 'X' then
      assert false
      report "There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic ….."
      severity warning;
      assert false
      report "CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in…"
      severity WARNING;
      return 0;
    end if;
  end loop;
  return result;
-- synopsys synthesis_on
end;
```

[1] La conversión de tipos se realiza como una llamada a una función, cuyo nombre es el del tipo al que se convierte (en el ejemplo **UNSIGNED**), a la que se le pasa como argumento el nombre del objeto cuyo tipo se desea convertir (en el ejemplo, **ARG**).

- En esta función se declaran dos variables para la materialización del algoritmo de conversión, **result** y **tmp**, de tipo **integer** y **std_ulogic**, respectivamente.

- El código de la función comprueba, en primer lugar y empleando una sentencia **ASSERT**, que el argumento tiene menos de 32 bits, ya que presupone que los enteros se representan en complemento a 2 con 32 bits y, por tanto, el número positivo más grande que se puede representar es 2^{31}-1. Si la comprobación falla, se reporta como error catastrófico (**FAILURE**).

- A continuación comienza la conversión de binario natural a decimal, que se realiza evaluando, mediante un bucle, la expresión:

  $$x_n \times 2^n + x_{n-1} \times 2^{n-1} + \ldots + x_2 \times 2^2 + x_1 \times 2^1 + x_0$$

  Donde x_i es el bit de peso i del array **ARG**.

  La expresión se evalúa testeando si el bit i es '0' ó '1', como:

  $$x_n \times 2^n + x_{n-1} \times 2^{n-1} + \ldots + x_2 \times 2^2 + x_1 \times 2^1 + x_0 = (((x_n \times 2) + x_{n-1}) \times 2 + x_{n-2}) \times 2 + \ldots + x_0$$

  Durante la ejecución del bucle, y a medida que se van extrayendo los valores de los bits del vector **ARG** a la variable **tmp**, se comprueba, mediante una sentencia **ASSERT**, que todos ellos son '1's ó '0's; en caso contrario se genera un aviso en el que se informa de la presencia de un valor en el vector que no puede reducirse a un '0' ó '1' lógico, y la función devuelve un cero.

  Resulta muy interesante analizar el mecanismo elegido para extraer y comprobar los valores de los bits de **ARG**; para ello se utiliza la sentencia:

  ```
  tmp := tbl_BINARY(ARG(i));
  ```

  donde **tbl_BINARY** es una constante de tipo **tbl_TYPE**, un tipo definido en el propio Cuerpo del Paquete **std_logic_arith**. La definición de este tipo puede resultarle curiosa:

  ```
  type tbl_type is array (STD_ULOGIC) of STD_ULOGIC;
  ```

  Es un array que se indexa, a diferencia de lo que suele ser normal (con una secuencia de enteros), con valores de tipo **std_ulogic**, es decir, con los valores 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H' y '-'. Este tipo de arrays permiten construir tablas que materializan relaciones entre dos conjuntos de valores.

  En nuestro ejemplo la constante **tbl_BINARY** se define como:

  ```
  constant tbl_BINARY : tbl_type := ('X', 'X', '0', '1', 'X', 'X', '0', '1',
  'X');
  ```

  Por tanto, los elementos **tbl_BINARY**('0') y **tbl_BINARY**('L') valen '0', **tbl_BINARY**('1') y **tbl_BINARY**('H') valen '1' y el resto (**tbl_BINARY**('U'), **tbl_BINARY**('X') , **tbl_BINARY**('Z'),

tbl_BINARY('W')y **tbl_BINARY**('-')) valen 'X'. La constante se utiliza, por tanto, para construir una tabla que establece una relación entre todos los valores de tipo **std_ulogic** (los índices del array) y los valores '0', '1' y 'X'.

Ejercicio

El análisis de la función **CONV_INTEGER** que se acaba de presentar ha permitido comprobar distintas aplicaciones de algunos de los elementos del lenguaje que se han estudiado en las últimas actividades (tipos de datos, atributos y sentencias **ASSERT**). Realice un análisis análogo de la función **TO_INTEGER** del paquete **numeric_std**[2] de la librería **IEEE**, que toma como argumento un objeto de tipo **UNSIGNED** y devuelve un valor de tipo **NATURAL**, para contestar las cuestiones que se formulan a continuación.

```
function TO_INTEGER (ARG: UNSIGNED) return NATURAL is
    constant ARG_LEFT: INTEGER := ARG'LENGTH-1;
    alias XXARG: UNSIGNED(ARG_LEFT downto 0) is ARG;
    variable XARG: UNSIGNED(ARG_LEFT downto 0);
    variable RESULT: NATURAL := 0;
  begin
    if (ARG'LENGTH < 1) then
      assert NO_WARNING
          report "NUMERIC_STD.TO_INTEGER: null detected, returning 0"
          severity WARNING;
      return 0;
    end if;
    XARG := TO_01(XXARG, 'X');
    if (XARG(XARG'LEFT)='X') then
      assert NO_WARNING
          report "NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0"
          severity WARNING;
      return 0;
    end if;
    for I in XARG'RANGE loop
      RESULT := RESULT+RESULT;
      if XARG(I) = '1' then
        RESULT := RESULT + 1;
      end if;
    end loop;
    return RESULT;
  end TO_INTEGER;
```

a. **XXARG** es un **alias** de **ARG**. ¿Qué es un **alias** en VHDL?

b. ¿Qué operación realiza la función[3] **TO_01**? ¿Cómo influye el segundo parámetro que se le pasa en el resultado que devuelve dicha función?

c. La función incluye dos sentencias **ASSERT**. ¿Bajo qué condiciones se activa cada una de ellas? ¿A qué objeto, tipo de datos y valor hace referencia[4] la etiqueta **NO_WARNING**? ¿Dónde está definida?

[2] Puede descargar el código de todos los paquetes de la librería ieee libremente desde: http://standards.ieee.org/downloads/1076/1076-2008/. También en QuestaSim.
[3] Para contestar esta pregunta tendrá que revisar el código del paquete **numeric_std**.
[4] Para contestar esta pregunta tendrá que revisar el código del paquete **numeric_std**.

d. ¿En qué se diferencia el tipo de datos del argumento de la función, **ARG**, y el de la variable **XARG**?

e. ¿Cuántos elementos tiene el array **XARG**?

f. Explique la operación del segmento de código de la función **TO_INTEGER** que se muestra a continuación

```
for I in XARG'RANGE loop
  RESULT := RESULT+RESULT;
  if XARG(I) = '1' then
    RESULT := RESULT + 1;
  end if;
end loop;
return RESULT;
```

PARTE V. Sentencias Generate

Deduzca el circuito digital que se obtendría de la síntesis del siguiente modelo VHDL.

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

entity mult_1xN is
generic(N: in natural := 4);

port(bit_in:   in     std_logic;
     Dato_in:  in     std_logic_vector(N-1 downto 0);
     Dato_out: buffer std_logic_vector(N-1 downto 0)
    );
end entity;

architecture estructural of mult_1xN is
begin
 G1: for i in 0 to N-1 generate
       Dato_out(i) <= bit_in and Dato_in(i);
     end generate;

end estructural;
```

En la documentación adjunta dispone del fichero **multiplicador.vhd** que incluye el modelo anterior y de un *test-bench* (**multiplicador_tb.vhd**) que debe completar para verificar el funcionamiento del circuito.

Realice un modelo equivalente al anterior, en un segundo cuerpo de arquitectura, empleando alguna de las construcciones incorporadas a VHDL-2008 que hagan el código más simple.