VHDL para síntesis.

Referencia básica.

Hipólito Guzmán Miranda
Departamento de Ingeniería Electrónica
Universidad de Sevilla
hguzman@us.es

- Cómo utilizar este documento
- Estructura de un fichero VHDL
  - La sección Library
  - La sección Entity
    - Generic (Parámetros)
    - Port (Puertos)
  - La seccion Architecture
    - Antes del begin
      - Declaración de señales
      - Definición de nuevos tipos de dato
      - Declaración de componentes
    - Después del begin
      - Sentencias concurrentes
      - Sentencia process
        - If … elsif … else
        - Case … when
        - Procesos combinacionales
        - Procesos síncronos
      - Instancias de componentes
- Ejemplos
  - Multiplexor 2 a 1
  - Contador de 8 bits
  - Contador de anchura parametrizable y entrada de sentido de cuenta
  - Multiplexor 4 a 1 instanciando 3 multiplexores 2 a 1

Cómo utilizar este documento

En este documento se plasma una referencia sencilla, basada en ejemplos, con los conocimientos mínimos para poder desarrollar circuitos combinacionales y secuenciales en VHDL.

Para poder entender este documento se requieren:

- Conocimientos básicos de circuitos combinacionales
- Conocimientos básicos de circuitos secuenciales

Se recomienda seguir los ejemplos con el software Xilinx ISE, instalado en los ordenadores del Centro de Cálculo. No obstante este documento no es una guia del software, sino del lenguaje. Se pueden encontrar guías del software aquí[1] y aquí[2].

Este documento no pretende ser una referencia exhaustiva del lenguaje, ya que para ello existen otros recursos como éste[3], éste[4] o éste[5].

Estructura de un fichero VHDL

Los ficheros VHDL suelen tener extensión `.vhd`, aunque en realidad son ficheros de texto plano. Se pueden editar con cualquier editor de texto[6], aunque nosotros generalmente los editaremos usando el editor integrado que trae Xilinx ISE.

En un fichero `.vhd` se describe una entidad o módulo, que viene a ser un componente de un circuito, con sus entradas y sus salidas. Un circuito sencillo normalmente estará descrito en un único fichero VHDL, mientras que un circuito o diseño complejo puede estar descrito en varios ficheros VHDL. Nótese que decimos "descrito" y no "programado", ya que no estamos ante un *lenguaje de programación*, sino ante un *lenguaje de descripción de hardware*. La diferencia principal es que un lenguaje de programación describe una *secuencia de instrucciones* y un lenguaje de descripción hardware describe la *estructura y comportamiento* de un circuito. En el primer caso las operaciones ocurren secuencialmente y en el segundo todo funciona en paralelo.

VHDL es *case-insensitive*, lo que significa que no es sensible a mayúsculas, es decir, que en este lenguaje `cuenta` y `CUENTA` representan lo mismo. Además, en VHDL se pueden separar la mayoría de las sentencias en diferentes líneas[7], lo que se usa principalmente para hacer más legibles las listas entre paréntesis (…).

---

[1] F. Muñoz. Guia Xilinx ISE para la tarjeta Basys-2
[2] F. Muñoz, H. Guzmán. Guía Xilinx ISE para la tarjeta LX9 Microboard
[3] Qualis Design Corporation, VHDL Quick Reference Card
[4] Brian Mealy, Fabrizio Tappero, Free Range VHDL
[5] M. A. Aguirre, J. Tombs, F. Muñoz, H. Guzmán, J. Nápoles, Diseño de Sistemas Digitales mediante Lenguajes de Descripcion Hardware
[6] Emacs, que es software libre, tiene un modo de indentación automática de VHDL que os puede ser útil antes de realizar entregas de código VHDL.
[7] Es más, se dice que VHDL es insensible al *whitespace* o espacios en blanco.

La sección Library

En la sección library se declaran las librerías y paquetes que vayamos a utilizar[8].

Las librería más común es la IEEE:

```
library IEEE;
```

Esta librería incluye distintos paquetes que proporcionan distintas funcionalidades:

IEEE.std_logic_1164.all; define los tipos de datos std_logic y std_logic_vector y permite realizar operaciones booleanas con ellos.

IEEE.numeric_std.all; define los tipos de datos signed y unsigned y permite realizar operaciones aritméticas con ellos.

Se incluyen los paquetes de una librería utilizando la sentencia use tras la llamada al library:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
```

Por defecto Xilinx ISE incluye las siguientes librerías si creamos una entidad VHDL seleccionado la opción “VHDL Module” en el asistente del menú “Project” -> “Add Source”:

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

Para una operación básica mientras estamos aprendiendo podemos utilizar el paquete ieee.std_logic_unsigned.all, que hace que automaticámente se interpreten los std_logic_vector como si fueran datos tipo unsigned (enteros sin signo). No obstante, en cuanto tengamos un poco de manejo se recomienda en su lugar utilizar las funciones de conversión de la librería numeric_std, ya que es la que se utiliza principalmente en la industria[9]

La sección Entity

La sección Entity es una descripción de ‘caja negra’ de nuestra entidad. Es decir, describe cómo se ‘ve’ el circuito desde fuera, sin entrar en cómo es por dentro. Esto tiene mucho sentido en diseño electrónico, donde es muy frecuente dividir el diseño en sistemas y subsistemas o módulos y submódulos.

La seccion Entity tiene dos subsecciones:

---

[8] Ya sean pertenecientes a estándares, propios o de terceros desarrolladores.

[9] En industria se prefiere, para operaciones aritméticas, el uso de la libreria numeric_std, de la cual se puede encontrar una referencia bien explicada en http://www.synthworks.com/papers/vhdl_math_tricks_mapld_2003.pdf

`Generic` (Parámetros)

La sección `Generic` es opcional. En esta sección se definen los `Generics`, que nos permiten parametrizar el circuito. Cada generic tiene un nombre, un tipo de dato y un valor por defecto. Por ejemplo, si describimos un contador genérico de N bits su sección `Generic` será:

```
Generic ( N : integer := 8 );
```

Si tenemos varios generic, se separan con comas, pero **el último no lleva coma**, ya que es una lista entre paréntesis:

```
Generic ( N : integer := 8, ACTIVE_RST_VALUE : std_logic := '1' );
```

`Port` (Puertos)

La sección `Port` describe los puertos de la entidad. Los puertos pueden ser de dirección `in` (entrada), `out` (salida) e `inout` (bidireccional). Cada puerto tiene un nombre, una dirección y un tipo de dato:

```
  Port ( clk    : in  std_logic;
         rst    : in  std_logic;
         enable : in  std_logic;
         salida : out std_logic
       );
```

Los puertos se separan con punto y coma, pero **el último no lleva separador**, ya que se trata de nuevo de una lista entre paréntesis.

La sección `entity` completa de un contador de cuatro bits podría ser:

```
entity cont_digito is
  Port ( clk    : in  std_logic;
         rst    : in  std_logic;
         cuenta : out std_logic_vector(3 downto 0)
       );
end cont_digito;
```

Y la de un contador de anchura parametrizable podría ser:

```
entity contador is
  Generic ( N : integer := 8 );
  Port ( clk    : in  std_logic;
         rst    : in  std_logic;
         cuenta : out std_logic_vector(N-1 downto 0)
       );
end contador;
```

Nótese que la anchura del puerto `cuenta` es de `N` bits en total.

La seccion Architecture

La sección Architecture describe cómo es el interior del Entity. Esta sección debe obligatoriamente contener la palabra reservada begin, que a su vez nos divide la arquitectura en dos secciones muy diferenciadas:

Antes del begin

Antes de la palabra reservada begin podemos definir elementos que vayamos a necesitar después. Estos elementos van a ser principalmente de tres tipos: *señales* (utilizadas para interconectar componentes), nuevos *tipos de datos* (utilizados principalmente para definir máquinas de estado) y *componentes* (entidades que tengamos ya descritas en otro fichero y queramos reutilizar en el fichero actual).

Declaración de señales

Para declarar señales se utiliza la sentencia signal:

```
signal nombre_senal: tipo_de_dato;
```

Se pueden declarar varias señales en una misma sentencia si las separamos por comas, siempre que sean del mismo tipo de dato:

```
signal cuenta, p_cuenta : integer range 0 to 255;
```

Hay que tener en cuenta que para los std_logic_vector especificamos el *número de bits*, sin embargo para los integer especificamos el *rango de valores* que puede tomar el dato (y el número de bits necesario para almacenar dicho rango será calculado automáticamente por el sintetizador). Por ejemplo, las siguientes declaraciones definen dos señales que se codificarán utilizando 8 bits:

```
signal dato1: integer range 0 to 255;
signal dato2: std_logic_vector (7 downto 0);
```

Definición de nuevos tipos de dato

Para realizar máquinas de estado definiremos un tipo de dato enumerado que podrá tomar como valores los estados posibles de la máquina de estados[10].

```
type tipo_estado is (reposo, cuenta_ciclos, cabecera, espera_datos,
procesa_datos);
```

Una vez definido el tipo de dato declararemos dos señales para el estado:

```
signal estado, p_estado : tipo_estado;
```

Declaración de componentes

Para poder reutilizar entidades que tengamos definidas en otros ficheros tendremos que declararlas como componente. La sentencia component tiene dos subsecciones que deben ser copia exacta de

---

[10] De esta forma, es el sintetizador el que se encarga de realizar la codificación de estados, es decir, la asignación de los valores en binario a cada uno de los estados posibles.

las subsecciones `Generic` y Port de la entidad que queramos reutilizar. El nombre del `component` debe coincidir con el nombre del `entity`:

```
component contador is
  Generic ( N : integer := 8 );
  Port ( clk    : in  std_logic;
         rst    : in  std_logic;
         cuenta : out std_logic_vector(N-1 downto 0)
       );
end component;
```

Por supuesto si la entidad que estamos reutilizando no tiene subsección Generic no será necesario que el component la tenga:

```
entity cont_digito is
  Port ( clk    : in  std_logic;
         rst    : in  std_logic;
         cuenta : out std_logic_vector(3 downto 0)
       );
end component;
```

Nótese que, a diferencia de la sección `entity`, la sentencia `component` no termina con "`end nombre_entidad`" sino con "`end component`".

Después del `begin`

Tras la palabra reservada `begin` podemos describir el funcionamiento del circuito. Todo lo que describa cualquier aspecto del funcionamiento u operación del circuito debe ir en esta parte del architecture y no antes.

Sentencias concurrentes

Podemos describir sentencias sencillas que se convertirán en lógica combinacional. Todas estas sentencias funcionarán dentro de nuestro circuito de manera concurrente, lo que simplemente significa que funciona todo a la vez. Esto tiene sentido ya que estamos hablando de hardware: las puertas lógicas están funcionando siempre todas a la vez.

```
c <= a AND b;

e <= (a AND b) OR (c AND d);
```

Existen además las sentencias `when … else` y `with … select` pero no las veremos aquí, ya que en general usarlas resulta más complejo que utilizar los equivalentes `if … elsif … else` y `case … when` dentro de un process.

Sentencia process

Describir un diseño complejo utilizando únicamente sentencias concurrentes es posible, pero no sería práctico ya que se podría convertir rápidamente en algo muy complicado de entender. La sentencia process nos permite describir una secuencia de instrucciones que podamos entender fácilmente y dejar que el sintetizador infiera el hardware que implementaría el circuito descrito por esta secuencia.

Es importante comentar que realmente no se implementa una secuencia de instrucciones, sino un circuito, pero el `process` hace más fácil entender la descripción del mismo.

La sintaxis de un process es:

```
nombre_proceso: process ( lista_de_sensibilidad )
begin
  sentencias_del_process;
end process
```

Donde `nombre_proceso` es una etiqueta opcional (si se elimina, se deben eliminar también los dos puntos tras `nombre_proceso`), y `lista_de_sensibilidad` es el conjunto de señales a las que es sensible el proceso.

Las sentencias disponibles para utilizar dentro de un `process` son las asignaciones, las operaciones booleanas (`AND`, `NAND`, `OR`, `NOR`, `XOR`, `XNOR`, `NOT`, etc), y las sentencias `if … elsif … else` y `case … when`.

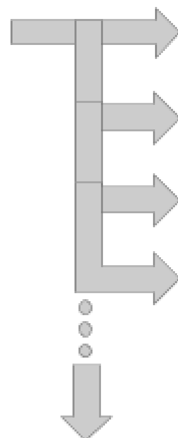
If … elsif … else

La sentencia `if … elsif … else` nos permite tomar decisiones en nuestros circuitos. El `if` es la única parte obligatoria de la sentencia: los `elsif` y el `else` son opcionales. Esta sentencia tiene *prioridad*, lo que significa que si se cumplen a la vez dos de las condiciones se entrará en la que ocurra antes en el código:

```
if (rst_sync = '1') then
  p_cont <= (others=>'0');
elsif (enable = '1') then
  p_cont <= cont + 1;
else
  p_cont <= cont;
end if;
```

Podemos tener más de un `elsif`, de forma que la sintaxis generalizada sería:

```
if (cond1) then
 <sentencias>
elsif (cond2) then
 <sentencias>
<... más elsif ...>
else
 <sentencias>
end if;
```



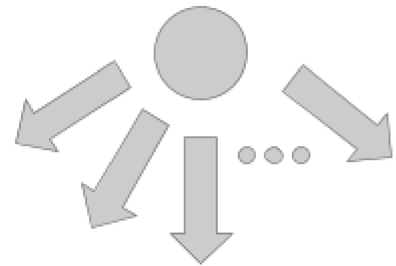
Case … when

La sentencia `case … when` es similar a la sentencia `if`, pero en lugar de evaluar condiciones (que pueden ser únicamente `true` o `false`), se evalúa el valor de un objeto (señal o puerto):

```
case state is
    when idle =>
      <sentencias>
    when count =>
      <sentencias>
    when header =>
     <sentencias>
    when others =>
      <sentencias>
end case;
```

Esta sentencia no tiene prioridad, ya que no puede ocurrir que el objeto (señal o puerto) tenga un valor y otro diferente a la vez. La sintaxis generalizada del `case … when` sería:

```
case obj1 is
    when value1 =>
      <sentencias>
    when value2 =>
      <sentencias>
    when value3 =>
     <sentencias>
    <... más when ...>
    when others =>
      <sentencias>
end case;
```



El "`when others =>`" actúa de caso por defecto (es decir, si no se cumple ninguno de los anteriores) y generalmente el sintetizador exige que se añada para evitar warnings.

El sintetizador avisará si falta alguno de los valores por evaluar, por ejemplo:

> *Enumerated value reposo is missing in case.*

Y tampoco nos permitirá considerar dos veces el mismo valor:

> *Choice reposo duplicated in case.*

Procesos combinacionales

Los procesos combinaciones son aquellos que al sintetizarse generarán una descripción en puertas lógicas, sin elementos de memoria. Ya que las puertas lógicas son sensibles a todas sus entradas, los procesos combinacionales son sensibles a todas las señales que leen. Las señales que son leídas en un proceso son aquellas que están dentro de una condición en un `if` / `elsif`, aquellas que son evaluadas en una sentencia `case`, y aquellas que aparecen a la derecha de una asignación (`<=`).

Por ejemplo, el proceso combinacional de un contador de 4 bits con señal de habilitación podría ser:

```
comb:process(cont, enable)
  begin
    if (enable='1') then
      if (cont="1001") then
        p_cont<="0000";
      else
        p_cont<=cont+1;
      end if;
    else
      p_cont<=cont;
    end if;
  end process;
```

Ya que las puertas lógicas no tienen memoria, es muy importante asignar las salidas del proceso en todos los casos posibles[11]. Si dejamos algún caso sin asignar el sintetizador de Xilinx nos dará el siguiente warning:

> *WARNING:Xst:737 - Found 4-bit latch for signal <p_cont>. Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing problems.*

El cual significa que ha insertado un elemento de memoria activo por nivel (Latch), en este caso de 4 bits. Estos elementos harán que nuestro circuito no funcione adecuadamente por lo que deberemos arreglar el proceso combinacional para eliminar el latch.

Procesos síncronos

Los procesos síncronos son un poco más complicados de entender, pero tienen la ventaja de que todos tienen la misma estructura si diseñamos con dos procesos, tomando las decisiones en los procesos combinacionales y almacenando los estados internos en los procesos síncronos.

Toda señal que aparezca a la izquierda de una asignación (“`<=`”) dentro del “`elsif rising_edge(clk)`” hará que el sintetizador infiera biestables activos por flanco (*Flip-flops*) para almacenar el valor de la misma. Es buena práctica de diseño que todos los biestables de nuestros diseños tengan reset[12], por lo que por regla general toda señal que sea asignada dentro del “`elsif rising_edge(clk)`” debería ser asignada también dentro del “`if (rst=’1’)`”.

---

[11] Para entender ésto, puede ser útil preguntarse: ¿qué pasa si le pido a las puertas lógicas un caso que no está contemplado?

[12] Al menos, mientras estamos aprendiendo el lenguaje. En determinadas arquitecturas se puede incumplir esta regla si tenemos claro lo que estamos haciendo.

```
sinc:process(clk,rst)
  begin
    if rst='1' then
      cuenta <= (others => '0');
    elsif rising_edge(clk) then
      cuenta <= p_cuenta;
    end if;
  end process;
```

Los procesos síncronos sólo son sensibles al reloj y al reset, ya que los Flip-flops sólo pueden actualizarse en los flancos de reloj (por mucho que cambie el dato a la entrada del flip-flop, la salida de éste no cambiará hasta que no llegue el flanco de reloj).

La expresión `(others => '0')` sirve para poner todos los bits de un vector a cero, independientemente del tamaño de éste.

Instancias de componentes

Para instanciar un componente que tengamos declarado, utilizaremos la siguiente sintaxis:

```
nombre_instancia: nombre_component
  generic map(
    generic1 => valor,
    generic2 => valor,
    generic3 => valor,
    ...
    genericN => valor
  )
  port map(
    port_del_component1 => objeto_top1,
    port_del_component2 => objeto_top2,
    port_del_component3 => objeto_top3,
    port_del_component4 => objeto_top4,
    ...
    port_del_componentN => objeto_topN
  );
```

Nótese que ambos `generic map` y `port map` son listas entre paréntesis, por lo cual **el último elemento de cada lista no lleva separador**. Nótese además que el `generic map` no está terminado en punto y coma[13].

Si la entidad que estamos instanciando no tiene generics, la instancia no debe incluir el `generic map`.

Por ejemplo si hemos declarado un contador como el siguiente componente:

```
component cont_digito
Port ( clk    : in  std_logic;
```

[13] El punto y coma señala el final de la instancia, por lo que si nos equivocamos y terminamos el `generic map` con un punto y coma, el efecto que tendrá será que ignorará la sección `port map`, resultando en warnings del tipo "el puerto `nombre_puerto` no está conectado".

```
       reset  : in  std_logic;
       salida : out std_logic_vector(3 downto 0);
       enable : in  std_logic;
       Sat    : out std_logic
     );
end component;
```

La instancia será de esta manera:

```
inst_cont_digito: cont_digito
  port map(
    enable => s_sat,
    clk    => clk,
    reset  => reset,
    sat    => open,
    salida => s_cuenta
  );
```

Ejemplos

Multiplexor 2 a 1

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

entity mux2a1 is
  port (a   : in  std_logic;
        b   : in  std_logic;
        sel : in  std_logic;
        o   : out std_logic);
end mux2a1;

architecture Behavioral of mux2a1 is

begin

  comb : process (a, b, sel)
  begin
    if (sel = '0') then
      o <= a;
    else
      o <= b;
    end if;
  end process;

end Behavioral;
```

Contador de 8 bits

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

entity cont8bit is
  port (rst    : in  std_logic;
        clk    : in  std_logic;
        enable : in  std_logic;
        count  : out std_logic_vector (7 downto 0));
end cont8bit;

architecture Behavioral of cont8bit is

  signal cuenta, p_cuenta : unsigned (7 downto 0);

begin

  comb : process (cuenta, enable)
  begin
    if (enable = '1') then
      p_cuenta <= cuenta + 1;
    else
      p_cuenta <= cuenta;
    end if;
  end process;

  sinc : process (clk, rst)
  begin
    if (rst = '1') then
      cuenta <= (others => '0');
    elsif (rising_edge(clk)) then
      cuenta <= p_cuenta;
    end if;
  end process;

  count <= std_logic_vector(cuenta);

end Behavioral;
```

Contador de anchura parametrizable y entrada de sentido de cuenta

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

entity contparam is
  generic (N : integer := 8);
  port (rst    : in  std_logic;
        clk    : in  std_logic;
        enable : in  std_logic;
        updown : in  std_logic;
        count  : out std_logic_vector (N-1 downto 0));
end contparam;

architecture Behavioral of contparam is

  signal cuenta, p_cuenta : unsigned (N-1 downto 0);

begin

  comb : process (cuenta, enable, updown)
  begin
    if (enable = '1') then
      if (updown = '1') then
        p_cuenta <= cuenta + 1;
      else
        p_cuenta <= cuenta - 1;
      end if;
    else
      p_cuenta <= cuenta;
    end if;
  end process;

  sinc : process (clk, rst)
  begin
    if (rst = '1') then
      cuenta <= (others => '0');
    elsif (rising_edge(clk)) then
      cuenta <= p_cuenta;
    end if;
  end process;

  count <= std_logic_vector(cuenta);

end Behavioral;
```

Multiplexor 4 a 1 instanciando 3 multiplexores 2 a 1

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

entity mux4a1 is
  port (a   : in  std_logic;
        b   : in  std_logic;
        c   : in  std_logic;
        d   : in  std_logic;
        sel : in  std_logic_vector (1 downto 0);
        o   : out std_logic);
end mux4a1;

architecture Behavioral of mux4a1 is

  signal o1, o2 : std_logic;

  component mux2a1 is
    port (a   : in  std_logic;
          b   : in  std_logic;
          sel : in  std_logic;
          o   : out std_logic);
  end component;

begin

  muxab : mux2a1
    port map (
      a   => a,
      b   => b,
      sel => sel(0),
      o   => o1);

  muxcd : mux2a1
    port map (
      a   => c,
      b   => d,
      sel => sel(0),
      o   => o2);

  muxo1o2 : mux2a1
    port map (
      a   => o1,
      b   => o2,
      sel => sel(1),
      o   => o);

end Behavioral;
```