Simulación de sistemas digitales mediante lenguajes descriptores de hardware

Juan Antonio Martínez Carrascal

PID_00170602

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

Si recapitulamos lo que hemos visto hasta ahora, veremos que hemos evolucionado desde las placas de silicio hasta el diseño de sistemas digitales de complejidad creciente.

Académicamente, podríamos pensar que con estos conceptos y una dosis de experiencia sería abordable la creación de circuitos digitales arbitrarios. Desde un punto de vista práctico, sin embargo, esta realización no es viable. Menos aún si pensamos que cada vez más se piden circuitos más fiables y con tiempos de desarrollo más cortos.

En este escenario, se plantea la utilidad de los lenguajes descriptores de hardware. Sería deseable disponer de un conjunto de herramientas que nos permitiera describir lo que hace nuestro circuito, aplicarle un conjunto de entradas, ver las salidas y comprobar que todo sea correcto. En definitiva, simular el funcionamiento del circuito antes de enviarlo al complicado –y caro– proceso de fabricación. Estamos hablando de simular por hardware nuestro diseño.

Pues bien, esto es precisamente lo que veremos a continuación. ¿Cómo podemos describir y modelar nuestro diseño con el fin de poder simularlo? ¿Cómo podemos ver los resultados de la simulación? ¿Funciona nuestro circuito como esperamos? Todas estas preguntas las podremos resolver con la ayuda de un lenguaje descriptor de hardware.

En nuestro caso, utilizaremos el lenguaje VHDL (acrónimo de VHSIC *hardware description language*). Aunque existen otros, éste tiene el apoyo de la industria, es de uso generalizado y, como veremos, será posible disponer de herramientas reales que realicen todas las funciones descritas.

VHSIC

VHSIC es el acrónimo de *very high speed integrated circuits*.

Con esto, tendremos nuestro diseño simulado. Nos faltará un paso más, que será la implementación real del diseño. VHDL también nos dará herramientas para hacerlo. De momento, sin embargo, nos centraremos en la simulación. Los módulos siguientes cerrarán el círculo y nos llevarán hasta la implementación real de los circuitos que queremos.

Objetivos

Al acabar este módulo deberemos ser capaces de lo siguiente:

1. Entender el concepto de lenguaje descriptor de hardware.
2. Conocer la filosofía del lenguaje VHDL.
3. Conocer la sintaxis básica del lenguaje VHDL.
4. Crear bancos de pruebas para simular los circuitos.
5. Diseñar circuitos digitales y simularlos mediante VHDL.
6. Profundizar en el diseño digital de sistemas complejos mediante máquinas de estados.
7. Diseñar sistemas secuenciales síncronos.

1. Diseño digital: de las puertas lógicas a la descripción del hardware

Hoy en día, podemos decir sin exagerar que nuestra vida sería impensable sin la electrónica. Prácticamente cualquiera de nuestras actividades cotidianas tiene el soporte de un dispositivo electrónico –habitualmente, digital. Una característica inherente a este tipo de dispositivos es su evolución constante, y el hecho de que les pidamos cada vez más funcionalidades y que se hagan de forma más rápida.

Es precisamente esta evolución constante y rápida –junto con la complejidad inherente de los diseños– uno de los motores que provoca que las herramientas de diseño digital tengan una importancia fundamental. Igual que en la actualidad cualquier proyecto arquitectónico tiene el soporte de una herramienta CAD[1] detrás, no hay circuito electrónico de cierta complejidad que se implemente sin un diseño mediante un lenguaje descriptor de hardware o, más genéricamente, de una herramienta de descripción de diseño electrónico (EDA[2]).

[(1)]CAD es la sigla de *computer aided design*.

[(2)]EDA es la sigla de *electronic design automation*.

Un lenguaje descriptor de hardware (HDL) describe mediante código el funcionamiento de un circuito.

Y es ésta la filosofía con la que, en los años ochenta, se desarrollaron los lenguajes descriptores de hardware. Inicialmente, con la filosofía de describir y simular el comportamiento de los circuitos. Con posterioridad, sin embargo, los diferentes fabricantes se dieron cuenta de que se podía aprovechar la potencia del lenguaje para efectuar implementaciones prácticas de los diseños.

Aunque nosotros centraremos nuestro estudio en el lenguaje VHDL, cabe decir que existen otros. Nosotros nos centraremos en el VHDL por tres motivos fundamentales:

- Dispone del apoyo del IEEE y de los principales fabricantes.
- Es, junto con Verilog, el lenguaje más utilizado.
- Dispone de herramientas que permiten crear todo el ciclo de diseño.

Web recomendada

En http://en.wikipedia.org/wiki/Hardware_description_language podréis encontrar una recopilación de lenguajes descriptores de hardware.

Antes de entrar en la modelización de circuitos con VHDL, es importante entender todo el flujo de diseño de un circuito. La figura 1 muestra las etapas que constituyen el proceso.

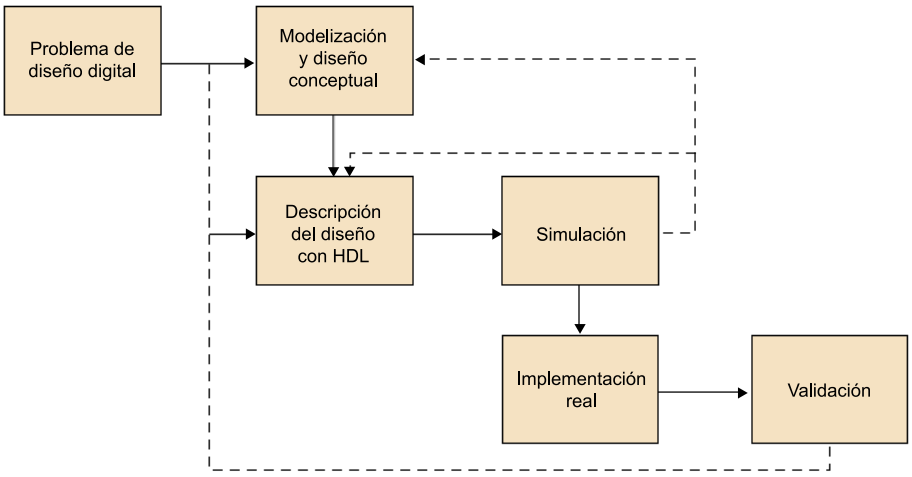

Figura 1. Proceso de diseño con un lenguaje descriptor de hardware

Fijémonos en que el proceso consta de seis subprocesos que podemos dividir en tres grandes bloques:

1) Un primer bloque de diseño conceptual. Este diseño conceptual nos permite llegar a una descripción de comportamiento de un circuito que resuelve una determinada problemática. Partiendo de unas especificaciones, llegamos a un diagrama de bloques (o incluso a una implementación con puertas) de un determinado circuito. De hecho, es lo que hemos hecho en el módulo anterior de la asignatura.

2) Un segundo bloque en el que el diseño que hemos creado se modeliza mediante VHDL. Esta modelización nos permite realizar una simulación. En el caso de que la simulación no proporcione los resultados deseados, habrá que revisar o el diseño o la codificación con el lenguaje descriptor.

3) El tercer bloque, si el resultado de la simulación es el esperado, consiste en la implementación real del circuito. No olvidemos que, hasta el momento, nuestro circuito se encuentra dentro de un simulador. La etapa final lleva a la realización del prototipo y, si todo es correcto, a una posible fabricación masiva. Aunque hemos efectuado la simulación, hay que realizar siempre una simulación del funcionamiento del dispositivo real. En el caso de que aparezca algún tipo de error no detectado en el diseño y la simulación, habrá que volver a repetir el proceso.

Empecemos, pues, a describir y simular el funcionamiento de los circuitos que ya sabemos diseñar mediante VHDL.

2. Descripción de sistemas digitales con VHDL

2.1. ¿Qué es VHDL?

Aunque podemos decir que la idea de describir mediante un lenguaje el funcionamiento de la electrónica es tan antigua como la electrónica en sí, las bases de VHDL son de finales de la década de 1980. En concreto, el IEEE lo definió como el estándar 1076 en el año 1987.

VHDL es un lenguaje que permite describir la estructura y el funcionamiento de un circuito digital, de tal manera que con esta descripción se pueda simular y, finalmente, pueda ser implementado en un dispositivo de lógica programable.

Obviamente, el estándar fue evolucionando. Por este motivo en algunas referencias podréis encontrar VHDL 87 –y en general VHDL y un año–, que hace referencia a la versión concreta de VHDL. A pesar de la evolución, las bases del lenguaje continúan siendo las mismas. En el fondo, lo hemos de imaginar tal como imaginamos otros lenguajes de programación de otros entornos que han ido evolucionando, pero mantienen el grueso de su sintaxis.

VHDL fue un lenguaje pensado originalmente para describir y simular el circuito. Fue su evolución la que permitió que, adicionalmente y con el apoyo de los fabricantes, pudiera dar lugar también a la implementación real de circuitos.

Como lenguaje, dispone de un entorno de programación o, mejor dicho, de descripción. De la misma manera que el lenguaje C presenta múltiples compiladores, el lenguaje VHDL también presentará múltiples entornos. En principio, todo lo que corresponde a la descripción y simulación del circuito será común y nos valdrá para cualquiera de los entornos. En la parte de implementación estaremos ya ligados a un fabricante determinado. Sin embargo, todo lo que estudiamos será perfectamente válido para cualquier tecnología y fabricante.

Los ejemplos que encontraréis a continuación están creados con la herramienta Modelsim XE III de Mentor Graphics. No obstante, si utilizáis cualquier otro compilador de VHDL, debería funcionar sin problemas. El hecho de utilizar

Ved también

En el módulo "Implementación de sistemas digitales sobre dispositivos de tipo FPGA" se tratará en con detalle el tema de la implementación directa de los diseños en FPGA.

esta herramienta es simplemente porque existe una versión gratuita y la posibilidad de implementación directa de los diseños en FPGA, como veremos en el próximo módulo.

Dicho esto, empecemos a ver cómo funciona el lenguaje VHDL. Lo haremos de la manera más simple posible. Crearemos una puerta AND en VHDL, algo que de alguna manera sería como el "Hello world" que se utiliza a menudo para introducir cualquier lenguaje de programación.

2.2. "Hello world": nuestro primer programa VHDL

Empezaremos nuestro aprendizaje de VHDL con un ejemplo muy sencillo que nos permitirá entender la filosofía del lenguaje. Empecemos por describir, mediante código, una puerta AND. El código VHDL sería:

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

-- definición de la entidad
entity puerta_and is
          port(
                   in0,  in1: in bit;
                   out0:  out bit
          );
end entity puerta_and;

-- definición de la arquitectura
architecture arquit_and of puerta_and is
begin
          out0 <= in0  and in1;
end architecture arquit_and;
```

Si leéis el código anterior, veréis que es bastante intuitivo. Fijémonos en que tenemos:

- Una descripción de las bibliotecas que se utilizan: estas bibliotecas nos permiten hacer uso de los tipos, los operadores, las funciones, etc. que podemos utilizar en VHDL.

- La definición de la **entidad** (*entity*), que dice explícitamente qué puertos tiene nuestro circuito y de qué tipo son. En nuestro caso, son sencillamente dos bits de entrada y uno de salida.

- La definición de la **arquitectura** (*architecture*), que indica cómo está constituido internamente el circuito que relaciona entradas y salidas.

Una descripción VHDL de un circuito constará de un conjunto de entidades, para cada una de las cuales habrá que definir una o más arquitecturas que la implementan.

Es importante remarcar que una determinada entidad puede disponer de más de una arquitectura. Es decir, una determinada entidad se podrá hacer de varias maneras. Finalmente, VHDL nos debe permitir ver las diferencias entre crear un circuito con una arquitectura u otra. A modo de ejemplo: un determinado circuito combinacional lo podemos crear con una determinada función sin simplificar o simplificándola. Para una misma entidad tendremos dos posibles arquitecturas y, probablemente, una será óptima y la otra no.

Evidentemente, éste es nuestro primer ejemplo y no hemos visto ni todos los tipos de datos, ni todos los operadores que podemos utilizar, ni las sentencias de control de flujo. Las iremos viendo en los ejemplos sucesivos. Antes, sin embargo, comentemos algunos aspectos genéricos importantes:

a) VHDL no distingue entre mayúsculas y minúsculas. Por lo tanto, *Entidad_a* y *entidad_a* serán equivalentes. Sin embargo, y para evitar errores o interpretaciones incorrectas, trataremos de escribir siempre los diferentes elementos y las variables de la misma manera.

b) Las normas básicas que son válidas para otros lenguajes de programación son válidas también para VHDL. Así, hemos de intentar siempre crear código documentado, claro, sencillo, modular y, por supuesto, eficiente. En relación con la documentación, cualquier línea que empiece con dos guiones (--) se considera un comentario.

c) Los identificadores de señales y variables pueden contener letras, números y _. Siempre deben empezar por una letra y no se permiten dos símbolos _ seguidos.

d) Es una buena costumbre, cuando se trabaja con proyectos grandes, que los ficheros dispongan de un encabezamiento explicativo de su funcionamiento en el que se indique una descripción breve del contenido del fichero.

Sobre los identificadores VHDL

VHDL permite utilizar también identificadores extendidos con cualquier carácter utilizando secuencias de escape (esencialmente \). A pesar de ello, lo intentaremos evitar para ser más claros.

Hemos visto el código de un programa en VHDL. A pesar de que es muy sencillo, nos debe permitir simular nuestro primer circuito.

2.3. Simular la puerta AND

Seguramente, leyendo la descripción anterior, muchos de vosotros debéis de pensar que VHDL es, más bien, un lenguaje puramente teórico. Como veremos a continuación, esto no es cierto y podremos incluso analizar cómo evolucionan las señales dentro de nuestro circuito.

Aunque las diferentes herramientas de trabajo con VHDL tienen una filosofía parecida, los ejemplos que se mostrarán a continuación utilizarán el simulador Modelsim XE. Lo que haremos esencialmente es crear un proyecto nuevo en el que incluiremos el código que ya hemos creado.

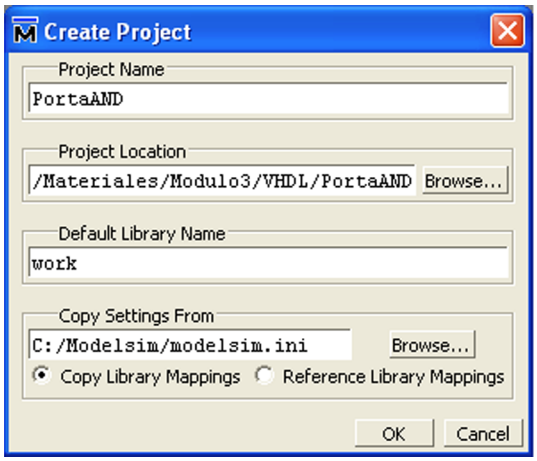

Figura 2. Creación de un nuevo proyecto de simulación

A continuación, sólo deberemos añadir los ficheros VHDL existentes o crearlos de nuevo. En la figura anterior, y aunque por ahora no es crítico, fijémonos en el apartado "Default Library Name", que nos permitirá hacer referencias a objetos, como veremos más adelante. Una vez guardados los ficheros los podemos compilar. Si todo es correcto, obtendremos un mensaje que nos lo indicará así.

De momento, sin embargo, tenemos una descripción del circuito pero deberemos crear el banco de pruebas. Este banco de pruebas nos debe permitir determinar si el funcionamiento del circuito es el esperado.

Este caso es muy sencillo y podríamos aplicar directamente unos determinados valores a las entradas y ver cuál es el valor de las salidas. Sin embargo, optaremos por un enfoque más metodológico que utilizaremos tanto para los casos sencillos, como para otros más complicados. Esencialmente, lo que haremos es separar la entidad que queremos probar de la generación de los vectores de prueba.

Analicémoslo gráficamente. Lo que tenemos es una caja negra con dos entradas, in_0 e in_1, y una salida, out_0. Lo que haremos ahora es aplicarle unas señales de entrada a esta caja negra y ver cuál es el valor de la salida. Esto lo haremos generando un segundo fichero VHDL para nuestro entorno de test.

```
-- fichero banco_pruebas.vhd
-- pruebas para la puerta and

library ieee;
use ieee.std_logic_1164.all;

entity banco_pruebas is
end banco_pruebas;

architecture arq_banco_pruebas of banco_pruebas  is

  signal  a,b,c: bit;

begin
  --  primero de todo instanciamos el circuito
   circuit_test: entity work.puerta_and(arquit_and)
    port  map(in0=>a,in1 => b, out0 => c);

     process
     begin
      a  <= '0';
      b  <= '0';
       wait for 200 ns;
      a  <='1';
      b  <='0';
      wait  for 200 ns;
      a  <='1';
      b  <='1';
       wait for 200 ns;
      a  <='0';
      b  <='1';

       assert false
         report "Fin de simulación"

         severity failure;
    end  process;
end arq_banco_pruebas;
```

Antes de analizar más en detalle el código, expliquemos qué es lo que hacemos conceptualmente. Estamos definiendo una nueva entidad sin entradas ni salidas, que es el banco de pruebas. Esta entidad tiene una arquitectura, que esencialmente lo que hace es referenciar una caja negra con dos entradas y una salida. A esta caja negra le aplica las señales *a* y *b* en las entradas y toma la salida en la señal *b*.

Nota

Aunque en este ejemplo pueda parecer trivial, es importante separar el código del circuito de su banco de pruebas. Aporta más claridad y resulta más sencillo encontrar errores en el diseño.

Gráficamente lo que hacemos es:

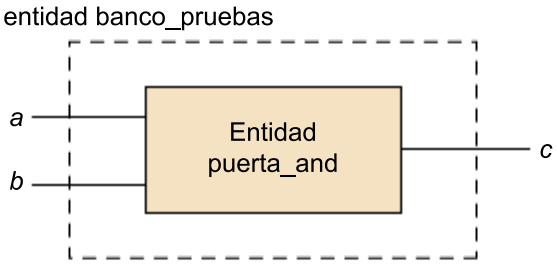

Figura 3. Diagrama de bloques del modelo VHDL

Fijémonos en que lo que hacemos es crear una supraentidad que aísla la entidad que estamos analizando y nos permite centrarnos únicamente en las señales de pruebas. Antes de ver el resultado de la simulación, analicemos algunos aspectos importantes del código anterior:

- La arquitectura del banco de pruebas la hacemos a partir de otro componente: en concreto, de la entidad *puerta_and* (podría contener más de una), que internamente tiene la arquitectura *arq_puerta_and*. Fijémonos en que se instancia como *work.puerta_and(arq_puerta_and)*.

- Las señales que generemos las asociamos a la disposición de pins de la entidad *puerta_and*. En concreto, esto lo hacemos con el código *map(in0=>a,in1 => b, out0 => c)*;. A partir de este punto, la constitución interna de la puerta nos es irrelevante.

- Es fundamental la directiva *process*. Aunque no lo hemos dicho hasta ahora, el código VHDL es de ejecución paralela. La directiva *process* nos permite la secuenciación de operaciones.

Nuestro objetivo es simular el diseño. Desde el propio entorno de Modelsim, disponemos de la opción "Simulate". Elegimos la opción "Simulate all" e indicamos que queremos actuar sobre nuestro banco de pruebas:

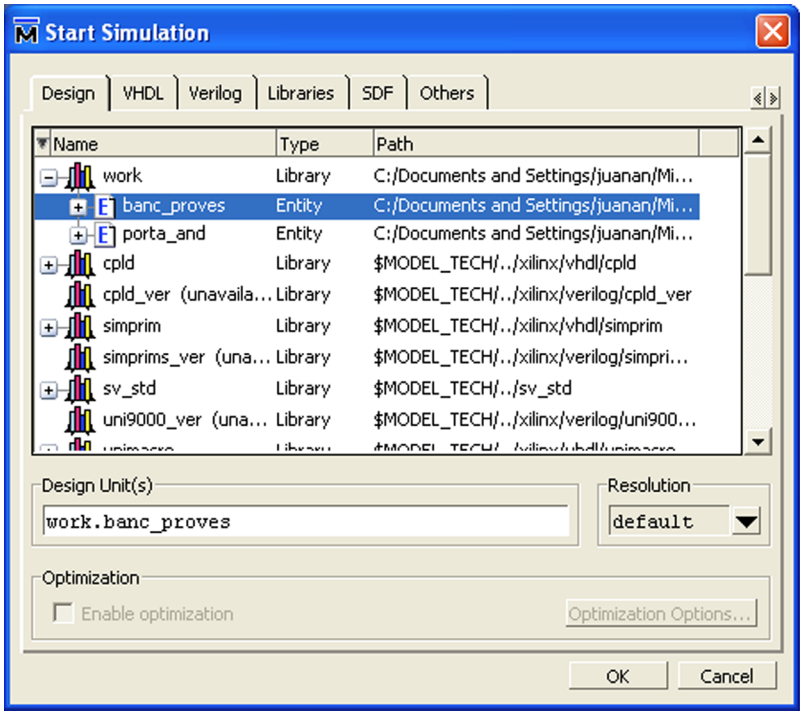

Figura 4. Inicio de la simulación

Si ahora ejecutamos la simulación ("Simulation-Run-All"), obtendremos un gráfico con las variables que tenemos disponibles para visualizar. En nuestro caso, tenemos:

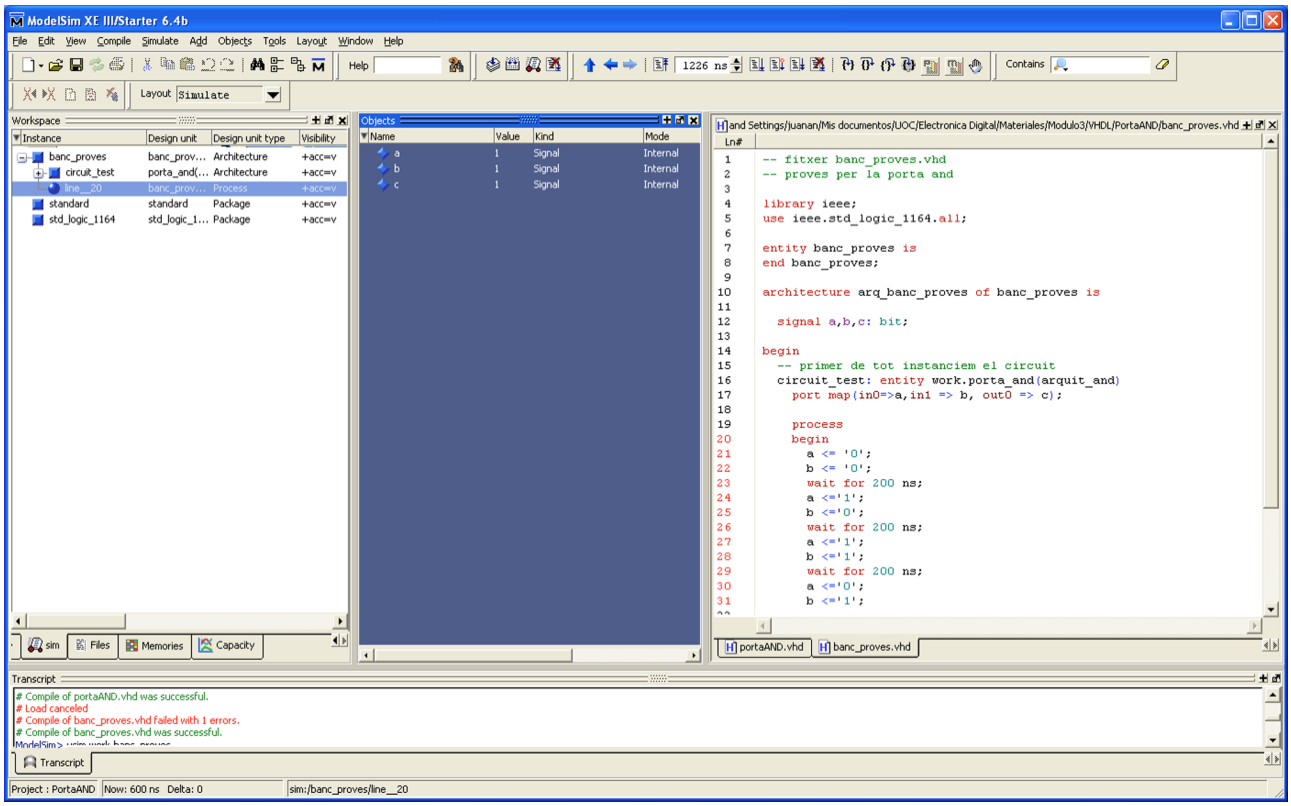

Figura 5. Selección de variables que se han de visualizar

Ahora sólo debemos añadir las variables por graficar. En nuestro caso son *a*, *b* y *c*. Desde el panel de objetos, y con el botón derecho, podemos añadirlas con la opción "Add to wave". Tenemos así el resultado de nuestra simulación, que es:

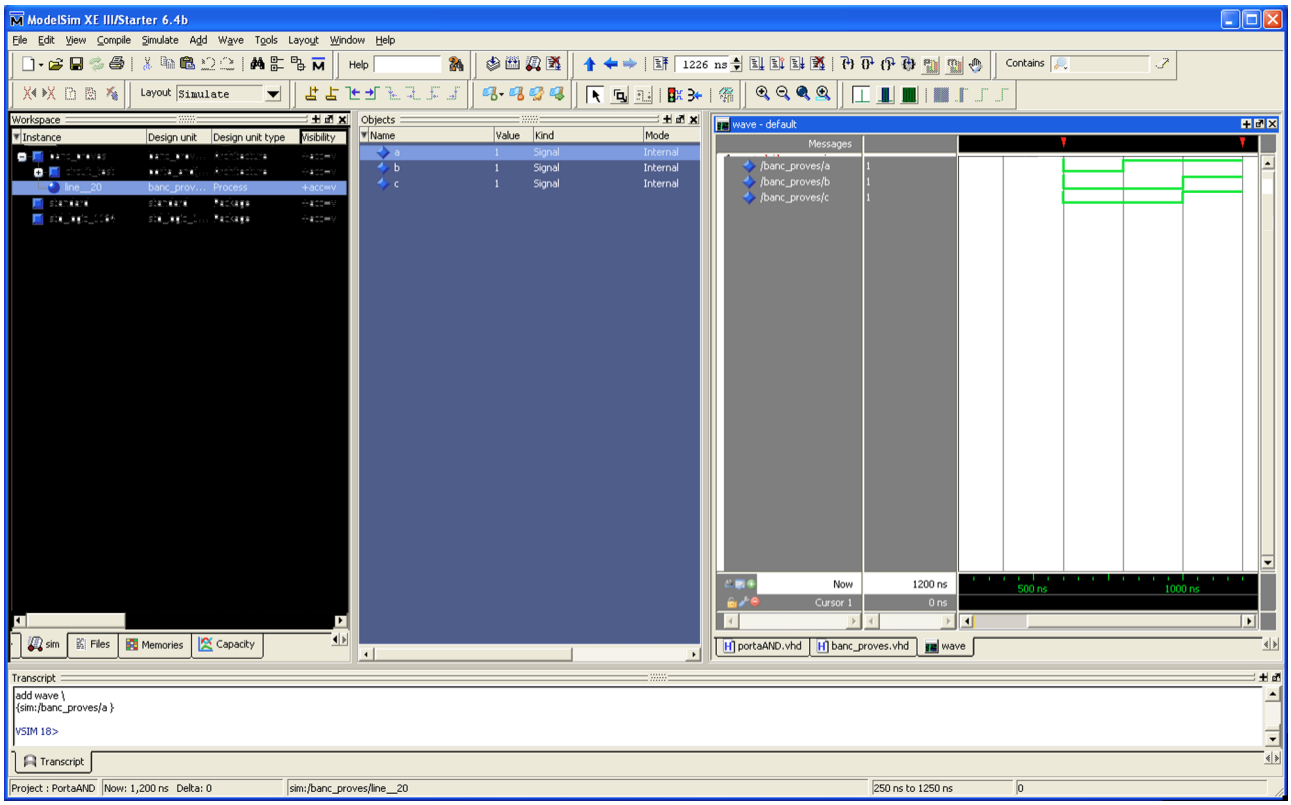

Figura 6. Visualización de los resultados

Como veremos, el resultado es el esperado. Nuestro ejemplo sencillo se comporta realmente como una puerta AND. Sólo cuando las dos entradas están en 1 lógico, la salida también lo está. Y lo que es más importante: ya tenemos un método para empezar a describir circuitos más complejos. A continuación, lo veremos.

3. Simulación de circuitos combinacionales genéricos con VHDL

Tomando como base lo que hemos visto en el apartado anterior, abordaremos el diseño de circuitos combinacionales. De hecho, y para ver un componente ya analizado, tomaremos como modelo el multiplexor.

Como en el caso anterior, generaremos un fichero para describir el circuito y utilizaremos otro como banco de pruebas. Antes, sin embargo, incidiremos sobre algunos aspectos más que nos serán de utilidad a medida que se compliquen los diseños: los tipos de datos que podemos utilizar y las operaciones que se permiten. Empecemos por ver qué tipos de datos nos permite utilizar VHDL.

3.1. Tipos de datos habituales y operaciones permitidas

Los tipos de datos determinan qué valores se pueden asignar y qué operaciones están permitidas sobre una determinada variable.

VHDL permite la definición de tipos personalizados. Sin embargo, hay un conjunto de tipos estándares que utilizaremos habitualmente:

- *bit*: que presentará dos posibles valores, 0 o 1;
- *std_logic*: es un valor que utilizaremos a menudo, ya que, aparte de permitir los valores 0 y 1, permite también los valores 'Z' (alta impedancia), U (no inicializado) y X (valor desconocido);
- *boolean*: que presenta valores *true* o *false*. Como cabe suponer, se utilizará para realizar tareas de control de flujo en un proceso;
- *integer*/*real*: que permiten almacenar valores numéricos, enteros o de coma flotante;
- *time*: para definir unidades de tiempo. Se puede utilizar a la hora de crear bancos de pruebas;
- *caracter*: para almacenar valores alfanuméricos.

Adicionalmente, podemos definir **matrices** de señales del mismo tipo. Por defecto, VHDL incorpora las matrices de bits (*bit_vector*), *string* (matriz de caracteres) y *std_logic_vector*. La definición se efectuará de la manera siguiente:

```
signal bus_ejemplo: bit_vector (7 downto 0);
```

Con esto definiríamos un bus de 8 bits, que numeraríamos del 7 al 0.

Hoy por hoy, estos tipos serán suficientes para nuestras necesidades.

Implementación física de los tipos

Como veremos, no todos los tipos son implementables directamente. Algunos serán de utilidad únicamente para nuestra simulación.

Mucho cuidado con las matrices

Podemos asignar valores –o matrices– a otra matriz siempre que sean del mismo tipo. Hay que tener en cuenta que en la definición de matrices se establece un orden. Así, no es lo mismo definir:

```
signal bus_ejemplo: bit_vector (7 downto 0);
```

que

```
signal bus_ejemplo_2: bit_vector (0 to 7);
```

Pensad –y simulad– qué sucedería con una asignación:

```
bus_ejemplo_2 <= bus_ejemplo;
```

Conocidos los tipos con los que trabajaremos, analizaremos ahora qué operaciones están permitidas:

- *not*: negación. Es la operación más prioritaria. Está definida por variables o señales de tipo *bit*, *bit_vector* o *boolean*.

- *and*, *or*, *nand*, *nor*, *xor*, *xnor*: definidas por el mismo tipo que la operación *not*.

- Asignación: todas las matrices permiten la asignación con el operador <=. Se pueden asignar partes de una matriz mediante la definición de rangos, por ejemplo, *a(0)*, en la que *a* representa una matriz.

- Comparación: todos los tipos estándar permiten la comparación con los operadores clásicos: menor (<), menor o igual (<=), igual (=), mayor (>), mayor o igual (>=) o diferente (/=).

- Para el caso de las matrices, cuando se efectúe una comparación ésta se llevará a cabo de izquierda a derecha. Así, una matriz 110 es mayor que 10111 (fijémonos en que no depende de la longitud de la matriz).

- Enteros, reales y temporales permiten operaciones aritméticas (+ -, * /, ** –potencia–, *mod*, *abs*, *rem* –resto de la división–).

3.2. Simular el multiplexor

Ya conocemos el funcionamiento del multiplexor del segundo módulo de la asignatura. Describamos, a continuación, el comportamiento en VHDL. Según hemos comentado, seguiremos una metodología en la que separemos la arquitectura del banco de pruebas. Para la arquitectura tenemos:

```
-- Simulación del multiplexor
library ieee;
use ieee.std_logic_1164.all;


entity multiplexor is
  port(
   entrada: in std_logic_vector(1 downto 0);
     selector: in std_logic;
     salida: out std_logic
  );
end multiplexor;


architecture arq_1 of multiplexor is
begin
  salida  <= (entrada(0) and (not(selector))) or ((entrada(1)
             and selector));
end arq_1;
```

Y como banco de pruebas, podemos utilizar éste:

```
-- Banco de pruebas para el diseño del  multiplexor
library ieee;
use ieee.std_logic_1164.all;

entity test_arq is
end test_arq;

architecture arq_test_arq of test_arq is

  signal  a: std_logic_vector(1 downto 0);
  signal  b,c: std_logic;

begin
  --  primero de todo instanciamos el circuito
   circuito_test: entity work.multiplexor(arq_1)
    port  map(entrada=>a,selector => b, salida => c);

     process
     begin
      a  <= "00";
      b <= '0';
       wait for 200 ns;
      a  <="01";
      b  <='0';
       wait for 200 ns;
      a  <="10";
      b  <='0';
       wait for 200 ns;
      a  <="10";
      b  <='1';
       wait for 200 ns;
      a  <="11";
      b  <='1';
       wait for 2000 ns;

       assert false
         report "Fin de simulacion"
         severity failure;
    end  process;
end arq_test_arq;
```

Si tratamos de hacer la simulación, obtenemos el resultado siguiente:

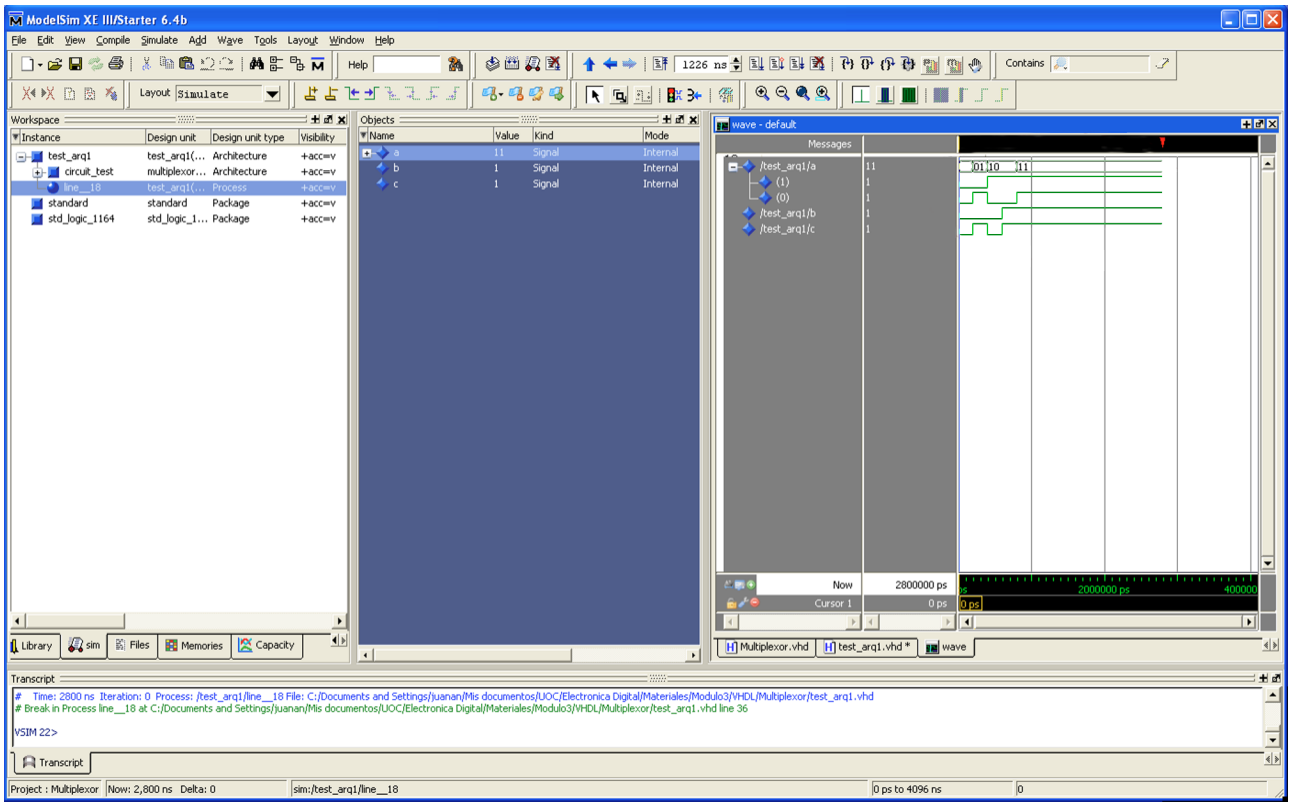

Figura 7. Resultado de la simulación del multiplexor

El resultado de la simulación es el esperado. Comentaremos, sin embargo, algunos aspectos del código:

- Como siempre, fijémonos en la separación entre entidad, arquitectura y banco de pruebas.

- Se ha utilizado un formato de matriz para las entradas al multiplexor. Fijaos en la notación *std_logic_vector(1 downto 0)* para la señal *entrada* y en la referenciación *entrada(0)*.

- Para las entradas de un único bit, se ha utilizado el tipo *std_logic*. Podríamos haber utilizado también el tipo *bit*, ya que en este caso no es relevante.

- Hemos utilizado los operadores lógicos (*and*, *not*, etc.) sobre las variables *std_logic*, bien directamente, bien como parte de las matrices. Es importante, y esclarecedor, marcar el orden de las operaciones con paréntesis.

Para el caso concreto del multiplexor, VHDL nos da una posibilidad añadida, que es la asignación condicional. Esta permite asignar una variable en función de una expresión. En concreto, la sintaxis sería:

```
señal    <=      expresion_1  when (expresion_booleana) else
                 expresion_2  when (segunda_expresion_booleana)
...
                 expresion_n  when (n-esima_expresion_booleana;
```

Actividad

Diseñad una arquitectura alternativa para el multiplexor utilizando la asignación condicional. ¿Qué cambios serían necesarios para utilizar esta arquitectura en el banco de pruebas?

Solución

Para diseñar una arquitectura alternativa, sólo hemos de añadirla al fichero donde ya disponemos de la anterior:

```
architecture arq_2 of multiplexor is
begin
  salida  <= entrada(0) when selector='0' else entrada(1);
end arq_2;
```

Y si ahora queremos que nuestro banco de pruebas utilice esta nueva arquitectura, sólo habrá que cambiar en el fichero del banco de pruebas la referencia a la arquitectura, que será:

```
  circuit_test: entity work.multiplexor(arq_2)
```

Fijaos en que con un enfoque modular podemos describir y comprobar circuitos combinacionales arbitrariamente complejos. Adicionalmente, la posibilidad de definir diferentes arquitecturas nos será muy útil a la hora de probar diferentes diseños para una misma función. Estamos, pues, en condiciones de avanzar hacia definiciones intrínsecamente más complejas: los circuitos secuenciales.

4. Modelado y simulación de circuitos secuenciales

Hasta ahora nos hemos limitado a circuitos combinacionales, es decir, aquéllos en los que las salidas, en un instante dado, son función únicamente de las entradas del circuito en aquel momento. Sin embargo, habitualmente nos encontraremos con sistemas secuenciales: aquéllos en los que la salida depende de la entrada y de la evolución que ha tenido el sistema.

Con el fin de analizarlos, empezaremos por hacer algunas consideraciones sobre estos sistemas. A continuación, modelizaremos, mediante VHDL, un sistema secuencial sencillo. Con esto estaremos en condiciones de abordar el diseño de sistemas secuenciales complejos que, como veremos, tendrán una representación funcional fundamentada en una máquina de estados.

4.1. Consideraciones generales sobre los sistemas secuenciales

> Un sistema secuencial será aquél cuya salida, en un instante determinado, depende de las entradas actuales y pasadas del sistema. Es decir, tiene en cuenta la evolución histórica del sistema.

Esta evolución histórica del sistema se almacena en lo que denominamos *variables de estado* o, simplificando, *estado*. Un estado será un indicador del punto en el que se encuentra nuestro sistema. A modo de ejemplo, podemos tener un sistema que, cuando se ponga en funcionamiento, vaya a un estado que denominaremos *inicialización*, en el que se efectúa un conjunto de tareas e, hipotéticamente, se queda a la espera de que suceda algo.

Por poner un ejemplo más concreto, un ordenador es un sistema secuencial: fijémonos en que, cuando arranca, va a un estado inicial –en el que inicializa todo un conjunto de variables– y a partir de aquel momento va actuando en función de cómo evolucionan las entradas –por ejemplo, el teclado o el ratón–. En este caso, evoluciona al ritmo que marca un reloj.

De manera gráfica, podemos representar un sistema secuencial de la siguiente manera:

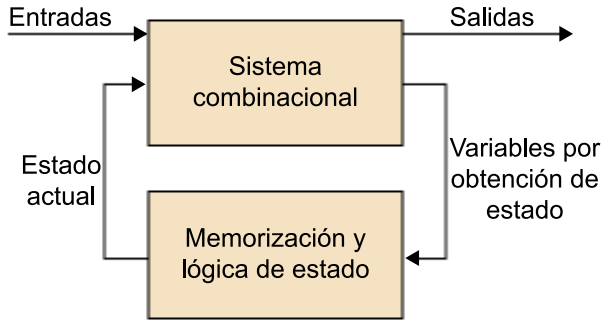

Figura 8. Representación del sistema secuencial

Una de las entradas al sistema nos permitirá clasificar los sistemas secuenciales en dos tipos:

1) Sistemas secuenciales asíncronos: en éstos se lleva a cabo un análisis continuo de las entradas y la salida cambia en el momento en el que lo provocan las entradas.

2) Sistemas secuenciales síncronos: en éstos el análisis de las variables y, por lo tanto, los posibles cambios de salida se efectúan en el momento en el que lo indica una señal específica, el reloj.

Los sistemas asíncronos tienen la ventaja de ser más rápidos de respuesta que los síncronos (habrá que esperar un ciclo de reloj para obtener la salida). Adicionalmente, considerarán cualquier cambio que se produzca en las entradas. En el caso de los sistemas síncronos, una variación muy rápida de alguna entrada que se produzca en un tiempo más pequeño que la frecuencia del reloj puede no ser considerada.

Los síncronos, por su parte, tienen la ventaja de proporcionar más simplicidad en el diseño. Por otra parte, con los relojes actuales –velocidad de MHz o GHz–, el análisis que se efectúa de las entradas es prácticamente continuo.

Con estas consideraciones, nos decantaremos por el uso y diseño de sistemas secuenciales síncronos siempre que sea posible. Hay que remarcar, sin embargo, que puede haber algunos casos en los que una problemática concreta pueda requerir diseños asíncronos.

Igual que hemos hablado del reloj (*clock*) como señal especial, los sistemas secuenciales suelen tener como mínimo otra señal especial. Se trata de la **señal de *reset***, que lleva el sistema a condiciones iniciales. Aunque diseñemos sistemas síncronos, esta señal suele tener una consideración asíncrona y provoca que el sistema vuelva a condiciones iniciales.

4.2. VHDL y diseño de sistemas secuenciales

En el módulo "Introducción al diseño de sistemas digitales", ya vimos una introducción a algunos sistemas secuenciales. De hecho, nos sirvió para analizar que el diseño se hacía más complejo y que, quizá, necesitábamos otras herramientas como el VHDL.

La pregunta que nos hacemos ahora es: ¿dispone el lenguaje VHDL de algún tipo de ayuda a la hora de diseñar sistemas secuenciales? La respuesta, como no podía ser de otra manera, es sí. En concreto, tendremos ayuda para los puntos siguientes:

a) Detección de subida o bajada de una señal: obviamente, si queremos trabajar con un sistema síncrono que funcione al ritmo del reloj, tendremos que poder detectar los flancos. De hecho, los sistemas síncronos pueden operar mediante flanco de subida –las variables se analizan cuando el reloj pasa de 0 a 1– o de bajada –cuando el reloj pasa de 1 a 0–.

VHDL permite detectar el flanco con la palabra reservada *event*. Así, si tenemos una señal denominada *reloj*, para detectar el flanco se hará de la manera siguiente:

```
if reloj'event
```

Y, en concreto, podemos detectar el flanco de subida de la manera siguiente:

```
if (reloj'event and reloj='1')
```

que será el modo habitual de funcionamiento de los sistemas síncronos: por flanco de subida del reloj.

Nota

El *package standard logic* de VHDL proporciona las funciones *rising_edge* (señal) y *falling_edge* (señal), que también tienen la función de detección de flancos.

b) Procesamiento de cambios: si indicamos a la directiva *process* un conjunto de señales, las instrucciones que siguen no se ejecutarán hasta que no haya un cambio de alguna de estas señales. Así, por ejemplo, en la modelización VHDL de sistemas secuenciales será habitual una estructura del tipo:

```
process(reset, reloj)
begin
 --  conjunto de instrucciones
end
```

En este caso, las directivas entre *begin* y *end* se analizan cuando se produce un cambio en *reset* o *reloj*. El caso más habitual es que si la señal de *reset* ha pasado de 0 a 1, llevaremos el sistema a condiciones iniciales y, en caso contrario, analizaremos si el reloj ha tenido un flanco de subida. Esto en VHDL quedaría como:

Nota

Estamos hablando de un ejemplo: podríamos tener otros sistemas en los que el *reset* opere por 0 lógico o el reloj actúe por flanco de bajada. Nada lo impide, y tampoco el VHDL.

```
process(reset, reloj)
begin
          if (reset='1')  then
          -- inicialización del sistema
          elseif  reloj='1' and reloj'event then
          -- lógica del  sistema
          end  if;
end process;
```

c) Uso de variables y señales de memorización: una de las cosas que caracteriza a los sistemas secuenciales es la necesidad de monitorear el estado. Con VHDL podemos hacer uso de variables y señales, y asignarles valores. A modo de ejemplo, si tenemos un sistema con 4 estados, podríamos definir, a la hora de describir la arquitectura de un sistema, la señal siguiente:

```
signal estado: integer range 0 to 3;
```

Esto nos permitiría trabajar con una variable –en este caso *integer*– para memorizar el estado. Si queremos hacer más comprensible el código, podemos definir un nuevo tipo:

```
type posibles_estados is(EstadoInicial,EstadoMedio,EstadoFinal)
variable estado: posibles_estados;
```

Y ahora podríamos asignar a la variable cualquiera de los valores permitidos.

4.3. Descripción de sistemas secuenciales mediante máquinas de estados

Supongamos un sistema de control de una puerta automática (por ejemplo, una puerta de un aparcamiento). Esta puerta se encontrará habitualmente cerrada y se podrá abrir mediante una orden de activación (que, en nuestro ejemplo, puede provenir de un control remoto). En el caso de recibir esta orden, se debe activar una señal para abrir la puerta.

La puerta se debe mantener abierta mientras no recibimos otra orden. Un sensor nos avisará de que la puerta está totalmente abierta. A partir de aquel momento, la puerta se podrá cerrar mediante una nueva orden que, en este caso, indicará el cierre. Es decir:

- Dispondremos de dos sensores que nos identifican, respectivamente, que la puerta está completamente abierta o cerrada.
- Dispondremos de una entrada de mando que indica que la puerta se debe abrir en caso de que esté cerrada o se debe cerrar en caso de que esté abierta.

Nuestro objetivo es generar dos señales que controlarán el cierre y la apertura de la puerta, respectivamente. Estas señales operan sobre un motor y, en caso de que la puerta esté completamente abierta o completamente cerrada, deben estar desactivados. Gráficamente, podemos ver el sistema como la caja negra siguiente:

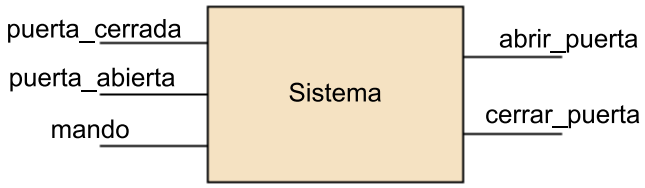

Figura 9. Esquema conceptual del sistema

El ejemplo anterior constituye un ejemplo de sistema secuencial (que, además, supondremos que lo diseñamos de manera síncrona y, por lo tanto, hará uso interno de un reloj). De hecho, su comportamiento lo podríamos definir gráficamente según se muestra en la figura siguiente:

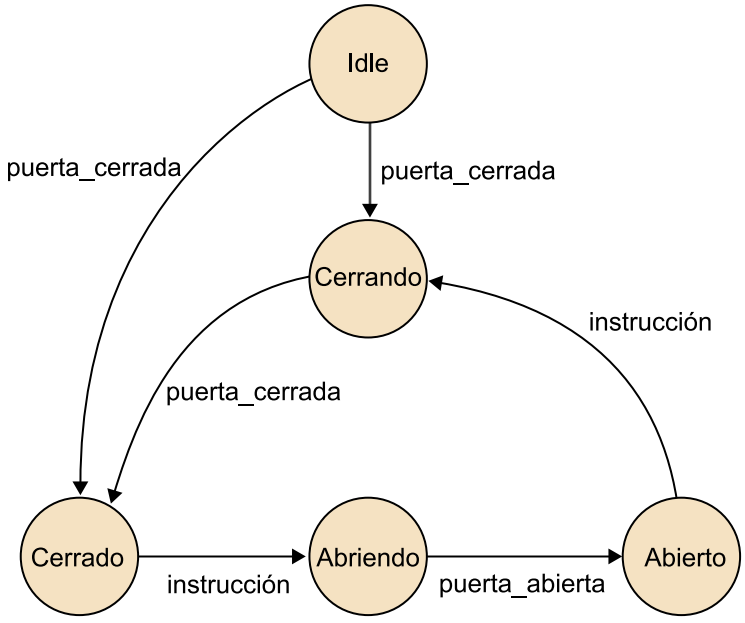

Figura 10. Diagrama de estados

La figura anterior constituye un diagrama de estados de lo que denominaremos *máquina de estados finitos* (MSF[3]). Aprovecharemos este ejemplo concreto para aprender a diseñar y modelizar los sistemas mediante VHDL.

(3)Abreviaremos *máquina de estados finitos* como MSF. A menudo, la encontraremos en la bibliografía como FSM, del inglés *finite state machine*.

Una máquina de estados finitos nos permitirá modelizar un sistema que transiciona entre un número finito de situaciones. El diagrama mostrará los diferentes estados y las transiciones que provocan un cambio de uno a otro.

En el diseño de esta máquina de estados finitos habrá dos partes claramente diferenciadas:

- La lógica de estado: que determina, en función del estado actual y las entradas, cuál ha de ser el estado siguiente.
- La lógica de salida: que determina, a partir del estado y en ciertos casos de las entradas, cuál ha de ser la salida.

Para evitar errores de diseño, separaremos siempre la lógica de estado de la lógica de salida.

Si nos fijamos en las partes de la MSF, vemos que el valor de la salida depende del estado y, en ciertos casos, también directamente de las entradas. Esto da lugar a las dos representaciones habituales de las MSF:

- Representación de Mealy: la salida depende del estado y de las entradas.
- Representación de Moore: la salida depende únicamente del valor del estado.

Gráficamente, las dos representaciones se muestran en la figura siguiente:

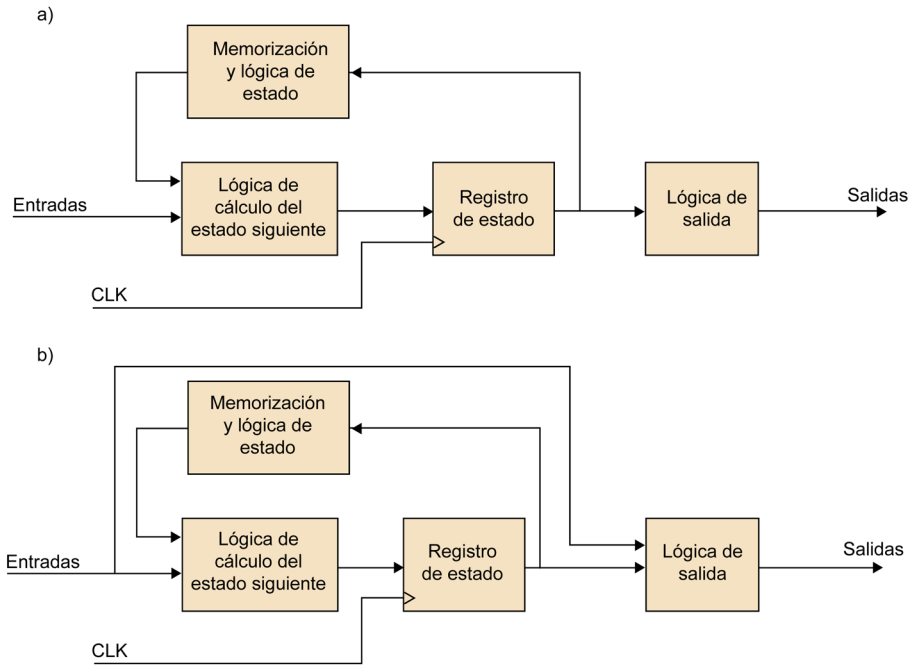

Figura 11. Esquemas del autómata de Mealy (a) y de Moore (b)

No es nuestro objetivo entrar en las diferencias conceptuales entre los dos modelos. De hecho, y sin entrar en detalle, podemos ver que la representación de Mealy es más genérica que la de Moore.

Volviendo a nuestro problema, nuestra MSF define qué hace el sistema, pero no cómo lo hace. Vamos a verlo a continuación. Fijémonos en que en las representaciones de las MSF hemos definido un registro de estado que permite memorizar el estado. Antes de entrar en la codificación, tengamos presente la representación de la figura de Mealy –o de Moore– en la que definiremos las señales siguientes antes y después del registro de estado:

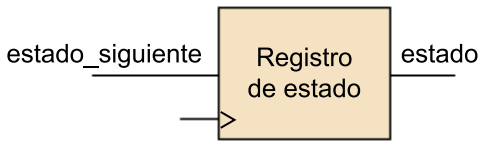

Figura 12. Esquema del registro de estado

Antes de continuar, tengamos en mente lo siguiente:

- La MSF del sistema que queremos diseñar (figura 10).
- El esquema de Mealy/Moore.
- El nombre de las señales que utilizaremos para memorizar el estado.
- Y el hecho de que siempre separaremos la lógica de estado de la lógica de salida.

Empecemos, pues, por diseñar la lógica de estado. Si tenemos claros los puntos que acabamos de mencionar y lo que hemos visto en este módulo, el código siguiente debería ser sencillo de entender. En primer lugar, modelizamos la máquina de estados:

```
process (clk, reset)
begin
     if (reset='1')  then
            estado <=  EstadoInicial;
     elsif  (clk'event and clk='1') then
            case  estado is
                  when EstadoInicial =>
                        if (puerta_cerrada='0') then estado
                           <= Cerrando;
                        else  estado <= Cerrado;
                        end  if;

                  when  Cerrando =>
                        if (puerta_cerrada='1') then estado
                           <=Cerrado;
                        end  if;
                  when  Cerrado =>
                        if (mando='1') then estado
                           <=Abriendo;
                        end  if;
                  when  Abriendo =>
                        if (puerta_abierta='1') then estado
                           <=Abierto;
                        end  if;
                  when  Abierto =>
                        if (mando='1') then estado
                           <=Cerrando;
                        end  if;
             end  case;
     end  if;
end process;
```

Como segunda parte, podemos determinar para cada estado sus salidas asociadas:

```
process (estado)
begin
case estado is
      when EstadoInicial  => abrir_puerta<='0';
                cerrar_puerta<='0';

      when  Cerrando    =>     cerrar_puerta<='1';
      when  Cerrado     =>     cerrar_puerta<='0';
      when  Abriendo    =>     abrir_puerta<='1';
      when  Abierto     =>     abrir_puerta<='0';

end case;
end process;
```

5. Problemas resueltos

Para complementar el conocimiento de VHDL que hemos visto en este módulo, simularemos un sistema combinacional y un sistema secuencial que ya conocemos y comprobaremos su funcionamiento. En concreto, resolveremos dos de los problemas planteados en el módulo "Introducción al diseño de sistemas digitales".

Ved también

Podéis encontrar estos dos problemas en el apartado 5 del módulo "Introducción al diseño de sistemas digitales".

5.1. Problema 1

Enunciado

Diseñad y simulad, utilizando VHDL, un comparador de un bit de manera que se pueda utilizar para implementar comparadores de un número de *n* bits. Con este módulo, construid un comparador de 4 bits.

Solución

Aprovecharemos el diseño de este sistema combinacional para trabajar la estructura modular y hacer llamamientos a módulos ya diseñados.

Recordad que en el módulo "Introducción al diseño de sistemas digitales" hemos resuelto el problema en dos pasos:

a) Creación de un comparador universal de un bit, módulo de 5 entradas y 3 salidas. Las entradas corresponden a los valores de comparación de los módulos previos ($a > b$, $a < b$, $a = b$) y a los dos bits que queremos comparar. Por otra parte, en la salida tendremos la indicación del resultado de la comparación.

b) Creación del comparador de *n* bits (en nuestro caso, $n = 4$) partiendo del comparador de un bit.

Comencemos por la realización del primer punto. En este caso, utilizaremos directamente las funciones lógicas que encontramos en el problema del módulo "Introducción al diseño de sistemas digitales". Podemos implementar, por lo tanto, el comparador con el código VHDL siguiente:

```
-- definición de entidad
entity comparador_un_bit is
       port(
              a_gt_b,a_lt_b,a_eq_b,a,b:  in bit;
              a_gt_b_out,a_lt_b_out,a_eq_b_out:  out bit
       );
end entity comparador_un_bit;

-- definición arquitectura
architecture arquit_comparador_u of  comparador_un_bit is
begin

     a_gt_b_out <=  a_gt_b OR (a_eq_b AND (a AND (NOT b)));
     a_lt_b_out <=  a_lt_b OR (a_eq_b AND (NOT a AND b));
     a_eq_b_out <=  a_eq_b AND (((not a) and (not b)) OR (a AND b));

end architecture arquit_comparador_u;
```

Podemos ver la definición de la entidad y de su arquitectura. A la hora de implementar la arquitectura, hemos utilizado directamente las funciones lógicas que ya habíamos encontrado.

Falta ahora implementar el comparador de 4 bits tomando como base la estructura ya construida. En concreto, lo que hemos de hacer es encadenar los comparadores. Recordemos que la estructura que hay que implementar es la de la figura 35 del módulo "Introducción al diseño de sistemas digitales", que recordamos a continuación:

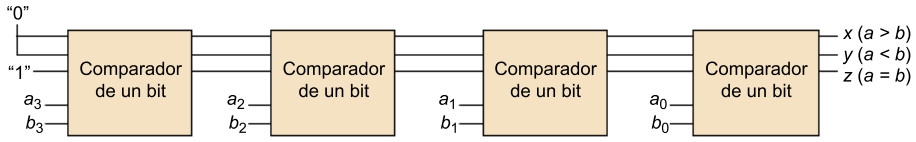

Figura 13. Construcción del comparador de 4 bits a partir de comparadores de un bit

Vamos a ver a continuación cómo VHDL nos permite crear este diseño modular. Antes de nada, y para hacer más comprensible el código, daremos nombres a las señales internas que interconectan los diferentes módulos comparadores de un bit dentro del comparador de 4 bits.

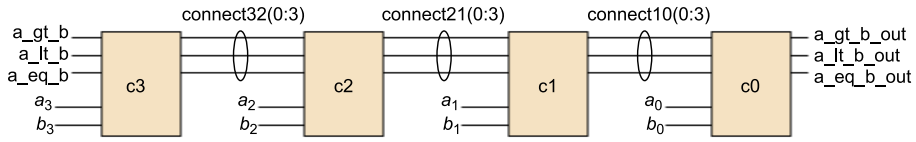

Figura 14. Comparador de 4 bits con marcado de las conexiones internas

En primer lugar definiremos, pues, la entidad:

```
entity comparador_multi_bit is
       port(
             a_gt_b,a_lt_b,a_eq_b:  in bit;
             a,b:  in bit_vector(3 downto 0);
             a_gt_b_out,a_lt_b_out,a_eq_b_out:  out bit
       );
end entity comparador_multi_bit;
```

Este código es parecido al que ya habíamos visto. Lo único que estamos definiendo es un comparador de 4 bits con 3 puertos de entrada que, como hemos visto, sirven para encadenar comparadores (*a_gt_b,a_lt_b,a_eq_b*), los dos valores que queremos comparar, que ahora ya son de 4 bits (*a* y *b* definidos como vectores) y, finalmente, las señales de salida, que nos indicarán cuál de los valores es mayor.

Ahora necesitaremos definir cómo está constituido este comparador internamente, es decir, su arquitectura. A diferencia de los ejemplos vistos hasta ahora, no describiremos a escala de puerta lo que hacemos, sino que aprovecharemos un módulo que ya hemos definido (el del comparador de un bit). Exponemos el código y a continuación lo explicamos:

```
architecture modular of comparador_multi_bit  is

  signal  connect32 ,connect21 ,connect10: bit_vector(2 downto 0); -- señales internas

   component comparador_un_bit -- utilizaremos este bloque como componente
       port (a_gt_b,a_lt_b,a_eq_b,a,b: in bit;
       a_gt_b_out, a_lt_b_out,a_eq_b_out: out bit);
  end  component;

begin

  --  definimos 4 comparadores encadenados
  c3:  comparador_un_bit port map  (a_gt_b,a_lt_b,a_eq_b,a(3),b(3),connect32(2),
connect32(1),connect32(0));
  c2:  comparador_un_bit port map  (connect32(2),connect32(1),connect32(0),a(2),b(2),
connect21(2),connect21(1),connect21(0));
  c1:  comparador_un_bit port map  (connect21(2),connect21(1),connect21(0),a(1),b(1),
connect10(2),connect10(1),connect10(0));
  c0:  comparador_un_bit port map  (connect10(2),connect10(1),connect10(0),a(0),b(0),
a_gt_b_out,a_lt_b_out,a_eq_b_out);



end modular;
```

Expliquemos a continuación el código anterior, ya que nos será muy útil a la hora de crear diseños modulares. Como vemos, hay partes relevantes:

- Definición de señales internas al módulo.
- Definición de los componentes.
- Uso de los componentes del diseño.

Fijémonos en que lo que hacemos es utilizar un conjunto de piezas e interconectarlas. Miremos de nuevo la figura 14. ¿Qué encontramos? En primer lugar, vemos un conjunto de bloques interconectados por un conjunto de señales. Son estas señales las que definimos inicialmente dentro de la arquitectura.

```
signal connect32 ,connect21 ,connect10:  bit_vector(2 downto 0);
-- señales internas
```

Estas señales se utilizan para interconectar un conjunto de "cajas negras". Como regla mnemotécnica, hemos denominado a las señales *connectxy*, donde *x* es el módulo del que provienen e *y* el módulo de destino. Es decir, la señal *connect32* conecta la "caja negra" 3 con la "caja negra" 2.

En nuestro caso, estas cajas negras son los comparadores de un bit. Definirlos es tan simple como invocar la pieza de código que ya hemos programado anteriormente:

```
    component comparador_un_bit -- utilizaremos este bloque como
                                componente
        port (a_gt_b,a_lt_b,a_eq_b,a,b: in bit;
        a_gt_b_out, a_lt_b_out,a_eq_b_out: out bit);
  end  component;
```

Faltará, finalmente, reflejar cómo se conectan. Tomemos a modo de ejemplo el módulo *c1*, que recibe como entradas los bits de la señal *connect* que interconecta el bloque 2 y el 1 (*connect21*), junto con los bits *a(1)* y *b(1*). Obtendremos como salida las señales que conectan el módulo *c1* y el *c0* (que hemos denominado *connect10*):

```
   c1:  comparador_un_bit port map  (connect21(2),connect21(1),
   connect21(0),a(1),b(1),connect10(2),connect10(1),connect10(0));
```

En este punto, y para tener la visión global, podemos ver cómo ha quedado en conjunto nuestro comparador, incluyendo tanto el comparador de un bit como el generado modularmente:

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


-- definición de entidad
entity comparador_un_bit is
       port(
             a_gt_b,a_lt_b,a_eq_b,a,b:  in bit;
             a_gt_b_out,a_lt_b_out,a_eq_b_out:  out bit
       );
end entity comparador_un_bit;


-- definición de arquitectura
architecture arquit_comparador_u of  comparador_un_bit is
begin
```

```
       a_gt_b_out <=  a_gt_b OR (a_eq_b AND (a AND (NOT b)));
       a_lt_b_out  <= a_lt_b OR (a_eq_b AND (NOT a AND b));
   a_eq_b_out <= a_eq_b AND (((not a) and (not b)) OR (a AND b));

end architecture arquit_comparador_u;


entity comparador_multi_bit is
       port(
             a_gt_b,a_lt_b,a_eq_b:  in bit;
             a,b:  in bit_vector(3 downto 0);
             a_gt_b_out,a_lt_b_out,a_eq_b_out:  out bit
       );
end entity comparador_multi_bit;

architecture modular of comparador_multi_bit  is

  signal  connect32 ,connect21 ,connect10: bit_vector(2 downto 0); -- señales internas

  component  comparador_un_bit -- utilizaremos este bloque como componente
       port (a_gt_b,a_lt_b,a_eq_b,a,b: in bit;
       a_gt_b_out, a_lt_b_out,a_eq_b_out: out bit);
  end  component;

begin

  --  definimos 4 comparadores encadenados
  c3:  comparador_un_bit port map (a_gt_b,a_lt_b,a_eq_b,a(3),b(3),connect32(2),
connect32(1),connect32(0));
  c2:  comparador_un_bit port map  (connect32(2),connect32(1),connect32(0),a(2),b(2),
connect21(2),connect21(1),connect21(0));
  c1:  comparador_un_bit port map  (connect21(2),connect21(1),connect21(0),a(1),b(1),
connect10(2),connect10(1),connect10(0));
  c0:  comparador_un_bit port map  (connect10(2),connect10(1),connect10(0),a(0),b(0),
a_gt_b_out,a_lt_b_out,a_eq_b_out);


end modular;
```

Como punto final, podemos efectuar la simulación de nuestro circuito. En este caso, hemos utilizado un banco de pruebas como el que se plantea a continuación:

```
-- fichero banco_pruebas.vhd
-- pruebas por el comparador multibit
```

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

entity banco_pruebas is
end banco_pruebas;



architecture arq_pruebas of banco_pruebas is

  signal  mayor, menor, igual: bit;
  signal  a,b: bit_vector(3 downto 0);
  signal  mayor_out,menor_out, igual_out: bit;

begin
  --  primero de todo instanciamos el circuito
   circuit_test: entity work.comparador_multi_bit(modular)
     port map(a_gt_b=>mayor,a_lt_b=>menor, a_eq_b=>igual,a =>
a,b =>b, a_gt_b_out => mayor_out,a_lt_b_out => menor_out,a_eq_b_out  => igual_out);

     process
     begin
       mayor <= '0';
       menor<='0';
       igual<='1';

       a  <="0100";
       b  <="0100";
       wait for 200 ns;
       a  <="0100";
       b  <="0011";
       wait for 200 ns;
       a  <="0011";
       b  <="1000";
       wait for 200 ns;
       a  <="0111";
       b  <="0111";

       assert  false
         report "Fin de simulacion"
         severity failure;
    end  process;
end arq_pruebas;
```

Este código nos muestra cómo el comparador opera como esperamos:

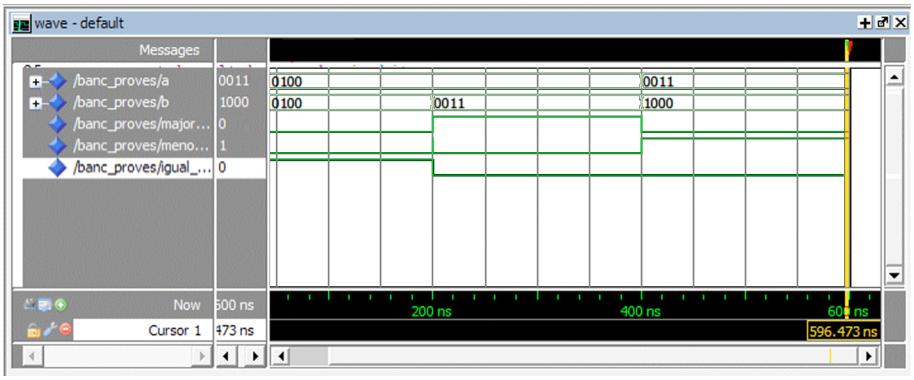

Figura 15. Resultado de la simulación

Con lo que el problema queda resuelto.

5.2. Problema 2

Enunciado

Queremos diseñar un sistema de seguridad que, en caso de alarma, envíe un código a una central. El sistema en cuestión dispone de tres entradas –dos entradas de sensores (I_1, I_2), un pulsador (I_3) y una señal para inicializar el sistema (RST)– y dos salidas, que denominaremos ALARM y LINE.

La salida ALARM es una señal indicadora de estado de alarma. En estado de reposo –cuando el sistema se inicializa– toma el valor 0. Se activa cuando se producen 8 flancos de subida del sensor I_1, seguidos de 4 flancos –bien sea de subida bien de bajada– del sensor I_2.

En caso de que la salida ALARM tome el valor 1, se considera una señal de alarma. En este caso, LINE, que conecta con la central de alarmas, transmite en serie a una velocidad de 100 Kbps un código de manera cíclica (101010...).

La activación del pulsador (I_3, activo para 1) durante 10 µs desactiva la condición de alarma y provoca que el sistema vuelva a su estado de reposo.

Simulad, en VHDL, el sistema planteado y comprobad que funcione correctamente. Considerad que disponéis de un único reloj de 1 MHz.

Nota

Con el fin de incluir el resultado de la simulación en una única gráfica de manera que los resultados sean visibles, hemos considerado que I_3 está activo únicamente 10 µs para provocar un *reset* (en lugar de los 4 s que habíamos propuesto en el problema del módulo "Introducción al diseño de sistemas digitales"). Como veremos, no afecta en absoluto al diseño y es únicamente para mejorar la visualización gráfica del resultado.

Solución

Empezaremos el problema con el diseño modular al que llegamos en su momento (figura 40 del módulo "Introducción al diseño de sistemas digitales"):

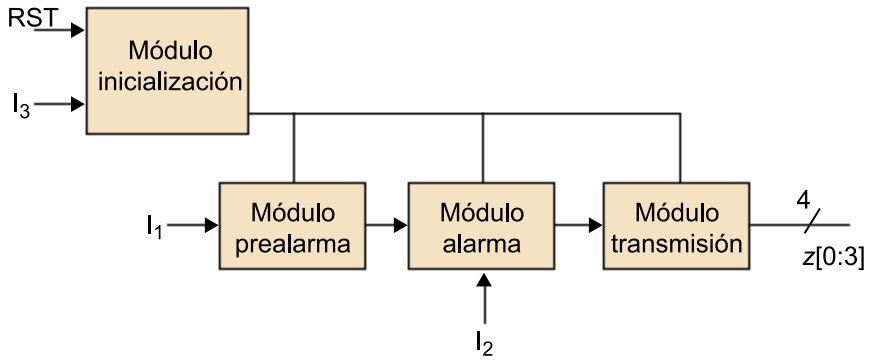

Figura 16. Diseño modular del problema 2

Tal como hemos hecho en el problema anterior, realizaremos una aproximación modular. Sin embargo, simularemos este caso fundamentándonos en el comportamiento de los diferentes módulos en lugar de una translación directa de los componentes electrónicos a los que llegamos en su momento. Es decir, trataremos de definir de nuevo los módulos no desde el punto de vista de componentes, sino de comportamiento.

De entrada, definimos nuestro circuito exponiendo únicamente las entradas y salidas que presenta. Esto en VHDL será:

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

-- definición de entidad

entity AlarmaModulo3 is
       port(
             i1,i2,i3:  in std_logic;
             clk:  in std_logic;
             reset:  in std_logic;
             alarm,line:  out std_logic
       );
end AlarmaModulo3;
```

Definimos así nuestra entidad, que, como vemos, tiene cinco entradas (las líneas i_1, i_2 e i_3, aparte del reloj y la señal de *reset* global) y dos salidas (el indicador de alarma y la línea de transmisión). Nos hace falta ahora una definición modular de nuestra entidad:

```
architecture modular of AlarmaModulo3 is


  signal  pciglobal: std_logic; -- señales internas
  signal  connex_prea: std_logic;
  signal  connex_alarm: std_logic;


   component modulo_init
       port (reset,i3,clk: in std_logic;
       pcisist: out std_logic);
  end  component;
```

```
    component modulo_prealarma
        port (pcisist,i1: in std_logic;
        prea: out std_logic);
   end  component;


    component modulo_alarma
        port (pcisist,prea,i2: in std_logic;
        alarma: out std_logic);
   end  component;


    component modulo_transmit
        port (pcisist,alarma2,clk: in std_logic;
        z:  out std_logic);
   end  component;

begin

   --  definimos 4 módulos
   modulo_1:  modulo_init port map (reset,i3,clk,pciglobal);
   modulo_2:  modulo_prealarma port map (pciglobal,i1,connex_prea);
   modulo_3:  modulo_alarma port map (pciglobal,connex_prea,i2,connex_alarm);
   modulo_4:  modulo_transmit port map (pciglobal,connex_alarm,clk,line);

   --
   --  falta por generar la salida alarm, que la extraemos directamente del modulo_3
   alarm  <=connex_alarm;


end modular;
```

Fijémonos en cómo lo que hacemos es definir un total de 4 componentes, que responden a los bloques de la figura 16. Remarcamos únicamente que la señal de alarma se utiliza tanto como salida, como para ser utilizada como entrada en el cuarto módulo. Por este motivo, hemos generado una señal interna que permitirá esta doble conexión.

Estamos, pues, en condición de abordar cada uno de los módulos del sistema:

a) Módulo de inicialización

Permite inicializar el sistema, bien sea por activación de la señal de *reset*, bien a causa de la presencia de la señal i_3 activa durante un número de ciclos de reloj (en nuestro caso, 10 ciclos). Podemos implementar este comportamiento en VHDL con la estructura siguiente:

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

entity modulo_init is
          port(
   reset,  i3,clk: in std_logic;
   pcisist: out std_logic
);
end modulo_init;



architecture interna of modulo_init is

    begin
     process(clk)

     variable num_ticks: integer;
     variable reseteando: integer;
     variable num_cicles_reset: integer;
     constant max_ticks:integer :=10;

     begin
      if  (clk'event and clk='1') then
         if (i3='1) then
           num_ticks:=num_ticks+1;
         else
           num_ticks:=0;
         end if;
    end  if;

    if  (num_ticks=max_ticks or reset='1') then
       pcisist <='1';
    else
       pcisist <='0';
  end  if;


  end  process;

end interna;
```

La variable *max_ticks* es la encargada de detectar cuántos ciclos de reloj hace que está activa la señal i_3. Si llegamos al número máximo, la señal permite la activación del *reset* del sistema, lo que devolverá tanto la línea como la señal de alarma al estado de reposo.

b) Módulo de prealarma

Es encargado simplemente de detectar los flancos de subida de la señal i_1. Su implementación en VHDL será:

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

entity modulo_prealarma is
  port(
   pcisist, i1: in std_logic;
  prea:  out std_logic
);
end entity modulo_prealarma;


architecture interna of modulo_prealarma is

  begin
     process(pcisist,i1)


     variable num_subidas: integer;
     constant max_subidas:integer :=4;

     begin

      if (pcisist='1') then
         prea<='0';
        num_subidas  :=0;
       elsif (i1'event and i1='1') then
            if (num_subidas < max_subidas) then
              num_subidas:=num_subidas+1;
            end if;
            if (num_subidas = max_subidas) then
               prea<='1';
           end if;
       end if;

end process;
end interna;
```

Fijémonos en cómo lo que hace el sistema es, por una parte, atender al posible *reset* global que indica el primero de los módulos ya analizados y, por otra, contar las subidas de la señal i_1.

c) Módulo de alarma

Si una vez se han producido los flancos de subida de la señal i_1 se producen 4 de la señal i_2 (en este caso, de subida o de bajada), será necesario activar una alarma. Fijémonos, también, en que sólo se activa en caso de que estemos en situación de prealarma (es decir, ya se han producido las fluctuaciones de i_1).

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

entity modulo_alarma is
  port(
     pcisist, prea, i2: in std_logic;
     alarma: out std_logic
);
end entity modulo_alarma;


architecture interna of modulo_alarma is

begin
     process(pcisist,i2)

     variable num_subidas: integer;
     constant max_subidas:integer :=4;

     begin

      if  (pcisist='1') then
         alarma<='0';
        num_subidas  :=0;
       elsif (i2'event and prea='1') then
            if (num_subidas &lt; max_subidas) then
              num_subidas:=num_subidas+1;
            end if;
            if (num_subidas = max_subidas) then
               alarma<='1';
           end if;
       end if;

end process;
end interna;
```

d) Módulo de transmisión

Efectúa una transmisión a una frecuencia de 1/10, la del reloj del sistema, en caso de que haya alarma. Esto lo podemos implementar con el código:

```
entity modulo_transmit is
    port(
       pcisist,alarma2,clk: in std_logic;
      z  : out std_logic
);
end entity modulo_transmit;



architecture interna of modulo_transmit is
    signal salida: std_logic;
begin
     process(clk)

     variable num_cicles: integer;
     variable enviando: integer;
     variable valor_a_linea: integer;
     constant clk_divider:integer :=5;

     begin

      if  (pcisist='1') then
         num_cicles:=0;
        valor_a_linea  :=0;
       end if;


         if (clk'event and clk='1') then
           if (num_cicles=clk_divider) then
             num_cicles:=0;
             if (valor_a_linea =0) then
               valor_a_linea:=1;
             else
               valor_a_linea:=0;
             end if;
           else
             num_ciclos:=num_ciclos+1;
           end if;
         end if;

         if (valor_a_linea =1) then
           z <= alarma2;
         else
           z <='0';
         end if;

end process;
end interna;
```

Nos queda, finalmente, crear la simulación. En nuestro caso, hemos utilizado este banco de pruebas para validar el comportamiento del circuito:

```
-- fichero banco_pruebas.vhd
-- pruebas para el comparador multibit

library ieee;
use ieee.std_logic_1164.all;

entity banco_pruebas is
end banco_pruebas;


architecture arq_pruebas of banco_pruebas is

  signal  i1,i2,i3: std_logic;
  signal  clk: std_logic;
  signal  reset: std_logic;
  signal  alarm,line: std_logic;

begin
  --  primero de todo, instanciamos el circuito
   circuit_test: entity work.AlarmaModulo3(modular)
     port map(i1,i2,i3,clk,reset, alarm, line);

     process
     begin
      i1  <= '0';
      i2  <='0';
      i3  <='0';
      reset <='1';

      wait for 50 us;
      reset  <='0';
      i1  <='1';
      i2  <='0';
      wait for 20 us;
      i1  <='0';
      wait for 50 us;
      i1  <='1';
      wait for 20 us;
      i1  <='0';
      wait for 50 us;
      i1  <='1';
      wait for 20 us;
      i1  <='0';
      wait for 50 us;
      i1  <='1';
      wait  for 20 us;
      i1  <='0';
      wait for 50 us;
      i1  <='1';
      wait  for 20 us;
      i1  <='0';
      wait  for 50 us;
      i1  <='1';
      wait  for 20 us;
      i1  <='0';
      wait for 50 us;
      i1  <='1';
      wait  for 20 us;
      i1  <='0';
      wait for 50 us;
      i1  <='1';
```

```
        wait  for 20 us;
        i2  <='0';
        wait for 50 us;
        i2  <='1';
        wait  for 20 us;
        i2  <='0';
        wait for 50 us;
        i2  <='1';
        wait  for 20 us;
        i2  <='0';
        wait for 50 us;
        i2  <='1';
        wait  for 20 us;
        i2  <='0';
        wait for 50 us;
        i2  <='1';

        wait for 100 us;
        i3  <='1';
        wait for 20 us;
        i3  <= '0';

        assert false
           report "Fin de simulacion"
           severity failure;
     end  process;
end arq_pruebas;
```

Con esto podemos ver el resultado gráfico de la simulación, que es la figura siguiente, que, como vemos, responde a los requisitos del problema.

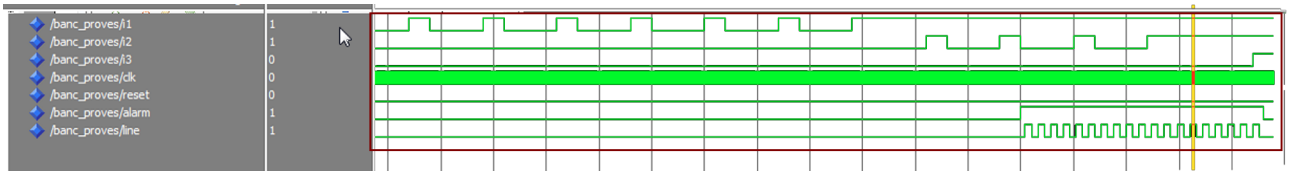

Figura 17. Resultado de la simulación

6. Del diseño VHDL a la síntesis: pasos siguientes

En este módulo, hemos avanzado más en nuestro conocimiento de los circuitos digitales. Hemos aprendido a crear la simulación de nuestros diseños, lo que nos permite validarlos y, en caso de detectar errores, corregirlos antes de enviar nuestro diseño a fabricación.

Si echamos la vista atrás en la asignatura, cada vez nos alejamos más del diseño de circuitos de transistores y capas de silicio. Aunque finalmente serán piezas imprescindibles, nos vamos abstrayendo cada vez más de este "bajo nivel" de la electrónica.

De todos modos, seguro que muchos de vosotros os preguntaréis si no nos estamos alejando demasiado del mundo tangible. Hemos definido el comportamiento de circuitos pero, finalmente, no lo hemos montado. ¿Cómo pasamos este diseño VHDL a un circuito real y tangible que funcione? Pues bien: VHDL nos dará herramientas para sintetizar estos diseños, tal como veremos en el módulo "Implementación de sistemas digitales sobre dispositivos de tipo FPGA".

Ved también

En el módulo "Implementación de sistemas digitales sobre dispositivos de tipo FPGA" encontraréis las herramientas del VHDL para sintetizar los diseños de circuitos.

Antes de pasar de módulo, es necesario que hagamos una reflexión sobre lo que nos permite el lenguaje VHDL: fijémonos en que tenemos una vía para simular sin fabricar. Huelga señalar que simular siempre es más económico, en tiempo y dinero, que el montaje de hardware. No despreciemos, pues, esta potencia y demos por bien utilizado el tiempo que empleamos en simular nuestros circuitos.

¿Qué hemos perdido por el camino? Reflexionaremos sobre esto también en el módulo "Implementación de sistemas digitales sobre dispositivos de tipo FPGA", pero cuanto más nos alejamos del "bajo nivel" –entendiendo como tal el silicio, los transistores o las puertas lógicas– más nos alejaremos de diseños "óptimos", entendiendo como tal aquellos que utilizan el número mínimo como transistores –o de puertas. De todos modos, la electrónica ha ganado tal capacidad de integración, que hace que posiblemente esto no sea un problema o, como mínimo, no tan grave como podemos pensar. Si hacemos un paralelismo con el mundo informático, los lenguajes de alto nivel no son tan óptimos como el ensamblador, pero su flexibilidad y la potencia del hardware provocan que sean hoy día utilizados mayoritariamente.

Finalmente, el mundo de la ingeniería es un compromiso. Tenemos más capacidad de diseño y simulación, una corrección de errores más fácil y más rapidez en el diseño, lo que finalmente significa que el tiempo, desde que nuestro diseño está en la mesa hasta que el prototipo aparece en la calle, es más bajo. Dejamos por el camino un diseño óptimo, que la propia electrónica se encarga

de compensar dándonos millones y millones de transistores en un solo chip. Si además hacemos que las conexiones entre estos transistores sean programables y que las podamos configurar con, por ejemplo, VHDL, daremos el paso definitivo en nuestro diseño electrónico de circuitos.

Resumen

En este módulo hemos avanzado un paso más en nuestros diseños digitales. Este avance es paralelo al que ha experimentado la electrónica digital. Hoy en día, sintetizar grandes diseños a partir de puertas es complicado y es necesario disponer de herramientas que nos permitan un nivel de abstracción más elevado.

Por este motivo, hemos profundizado en el análisis del lenguaje VHDL como lenguaje descriptor del hardware. Hemos visto su estructura y cómo nos ha permitido simular desde simples puertas hasta circuitos combinacionales y secuenciales arbitrarios.

De ahora adelante, siempre que planteemos un diseño electrónico de cierta entidad deberemos tener presente este lenguaje para hacer el modelado. Finalmente, es uno de los lenguajes más soportados por los fabricantes, lo que nos permitirá no quedarnos únicamente con la idea de un código simulador, lo que limitaría su utilidad, sino profundizar y utilizarlo para implementar nuestros diseños, algo que sería imposible sin el apoyo de los fabricantes a este estándar.

Bibliografía

Chu, P. P. (2008). *FPGA Prototyping by VHDL examples: Xilinx Spartan-3 Version*. Wiley Interscience.

García Zubía, J. (2002). *Manual de VHDL: síntesis lógica para PLD's*. Deusto: Universidad de Deusto.

Zwolinski, M. (2000). *Digital System Design with VHDL*. Prentice Hall.