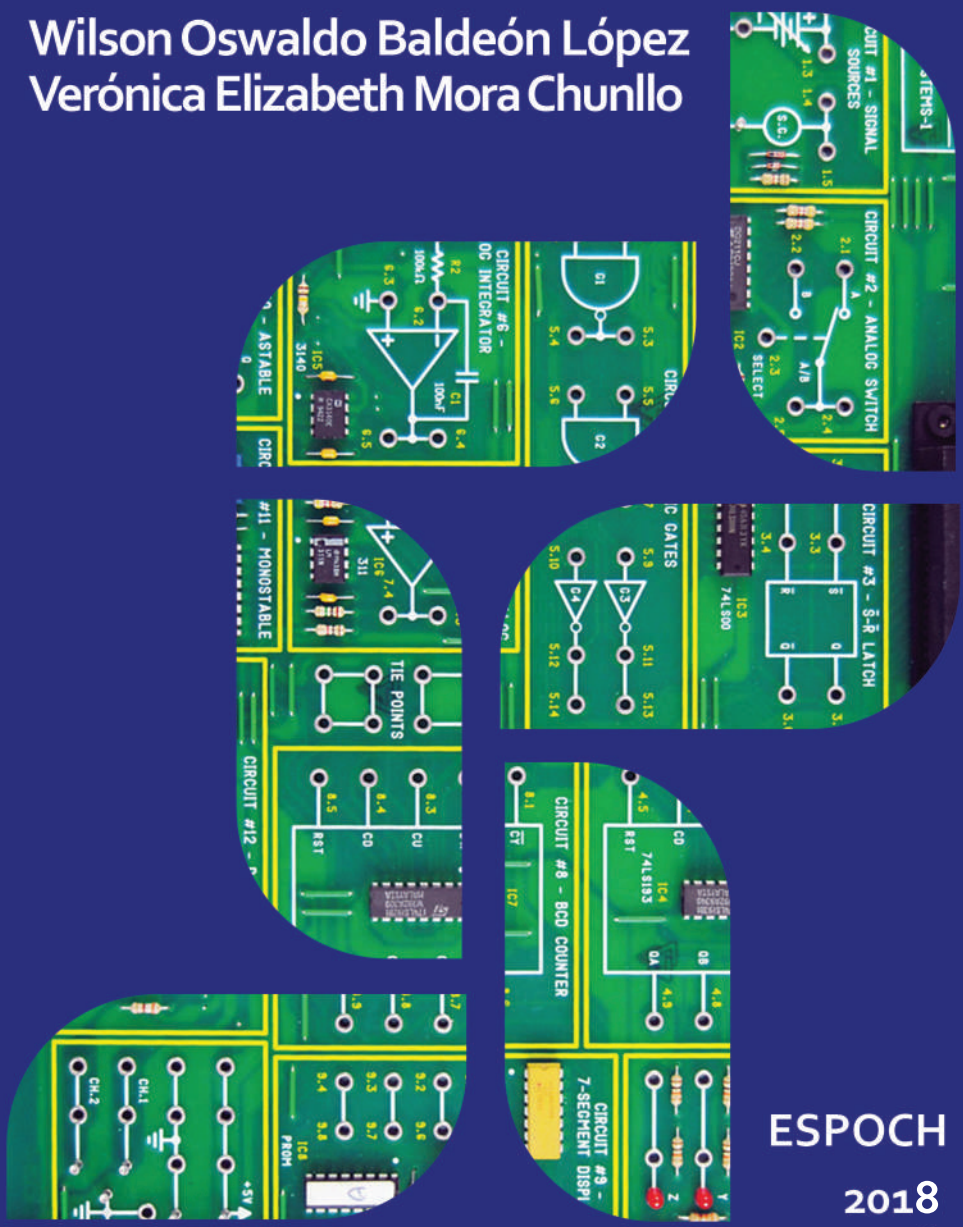
Sistemas digitales sincrónicos y VHDL

Introducción a VHDL

Wilson Oswaldo Baldeón López
Verónica Elizabeth Mora Chunllo



ESPOCH
2018

SISTEMAS DIGITALES SINCRÓNICOS Y VHDL
INTRODUCCIÓN A VHDL

SISTEMAS DIGITALES SINCRÓNICOS Y VHDL
INTRODUCCIÓN A VHDL

WILSON BALDEÓN
VERÓNICA MORA

DIRECCIÓN DE PUBLICACIONES

SISTEMAS DIGITALES SINCRÓNICOS Y VHDL
INTRODUCCIÓN A VHDL

Panamericana Sur, kilómetro 1 ½
Dirección de Publicaciones Científicas
Riobamba, Ecuador
Teléfono: (593 03) 299 8200
Código postal: EC0600155

Aval ESPOCH

Este libro se sometió a arbitraje bajo el sistema de doble ciego

(*peer review*)

Corrección y diseño

La Caracola

Editorial Politécnica ESPOCH

Impreso en Ecuador

CDU: 004.3 + 004.4
SISTEMAS DIGITALES SINCRÓNICOS Y VHDL
INTRODUCCIÓN A VHDL
Riobamba: Escuela Superior Politécnica de Chimborazo
Dirección de Publicaciones, Año 2019
90 pp. vol: 17 x 24 cm
ISBN: 978-9942-35-649-9
1. Ciencia y tecnología de los ordenadores
2. *Hardware*. Componentes físicos del ordenador.
3. *Software*. Equipo lógico, componentes lógicos.

A Solange Baldeón

A Efraín Baldeón

A Sean Risley

ÍNDICE

ACERCA DE LOS AUTORES

Wilson Oswaldo Baldeón López es ingeniero electrónico graduado en la Escuela Superior Politécnica del Litoral; es máster en Informática graduado en Chile, obtuvo su máster en Diseño de Sistemas Electrónicos por la Universidad Tecnológica de La Habana, Cuba; es magíster en Gestión Académica Universitaria, tiene un diplomado superior en Pedagogía Universitaria y es experto en procesos *e-learning*.

Es miembro fundador del grupo de investigación GITCE; sus intereses de investigación son los sistemas digitales, el modelamiento y predicción del índice de radiación ultravioleta (IUV). Ha publicado algunos textos básicos y varios artículos en revistas científicas, fue ingeniero de diseño digital en ELECSA, ha ganado varios premios.

Es docente titular en la Facultad de Informática y Electrónica de la Escuela Superior Politécnica de Chimborazo; fue autoridad académica y profesor titular en la Facultad de Ingeniería de la Universidad Nacional de Chimborazo; es miembro de la Asociación Mundial de Tutores virtuales (ATM).

Verónica Elizabeth Mora Chunllo es ingeniera en electrónica y computación graduada en la Escuela Superior Politécnica de Chimborazo; magíster en Ingeniería de *Software* graduada en la Escuela Superior Politécnica del Ejército; es máster en Diseño de Sistemas Electrónicos por la Universidad Tecnológica de La Habana, Cuba; es magíster en Educación a Distancia, tiene un diplomado superior en las Nuevas Tecnologías de Información y Comunicación Aplicadas a la Práctica Docente, es experta en procesos *e-learning*.

Es docente titular en la Facultad de Informática y Electrónica de la Escuela Superior Politécnica de Chimborazo, es Directora y fundadora del grupo de investigación GITCE; sus intereses de investigación son los sistemas digitales. Es Miembro de la Asociación Mundial de Tutores virtuales (ATM), fue miembro de la Comisión Editorial de la revisa científica *Perspectivas* FIE-Espoch, fue Subdirectora de Posgrado en la Espoch, ha publicado algunos textos básicos y varios artículos en revistas científicas nacionales.

PREFACIO

Este libro está basado en las experiencias académicas que los autores adquirieron en dos importantes universidades. Este es el tercero de cuatro libros que conforman el estudio de las máquinas secuenciales sincrónicas.

Esta obra está diseñada para un segundo curso de sistemas digitales, que es un curso fundamental, en la mayoría de las carreras de Ingeniería Electrónica. Un diseñador de sistemas digitales debe dominar las técnicas de diseño de sistemas digitales combinacionales y secuenciales, así como también, las aplicaciones informáticas que permiten el diseño asistido por computadora (CAD).

La ciencia del diseño de sistemas digitales, en lo relativo a sus conceptos y fundamentos, no ha sufrido un cambio radical todavía; sin embargo, la aparición de las herramientas EDA y, en la década de los ochenta, del lenguaje de descripción de *hardware* para circuitos integrados de muy alta velocidad (VHDL), ha cambiado sustancialmente la forma cómo se diseñan sistemas digitales.

Este libro muestra los conceptos fundamentales VHDL. Existen infinidad de libros sobre sistemas digitales, sin embargo, este libro se adapta a las necesidades académicas y de laboratorios de las carreras de Ingeniería Electrónica de la Espoch, de ahí que uno de los objetivos, es resolver estas necesidades.

Este texto presenta VHDL de tal forma que el lector pueda desarrollar habilidad intuitiva para entender y aplicar los conceptos fundamentales de VHDL en el diseño de máquinas secuenciales sincrónicas.

Esta obra presenta los conceptos fundamentales de VHDL. Tomando en cuenta las complicaciones que presenta este lenguaje para estudiantes novatos, se estudian los constructores que son suficientes para que el estudiante pueda desarrollar proyectos sin dificultad. En forma gradual, se estudia el diseño de circuitos combinacionales y secuenciales, se presentan y resuelven ejemplos que permiten simultáneamente aprender y aplicar VHDL.

CAPÍTULO 1 LENGUAJE DE DESCRIPCIÓN DE *HARDWARE*

Introducción

Antes de la aparición de los lenguajes de descripción de *hardware* y en general de las herramientas de diseño asistido por computadora, el análisis, diseño e implementación de sistemas digitales era una tarea bastante compleja que ,en general, consistía en diseñar un circuito a papel y lápiz; luego, el circuito se construía con dispositivos reales sobre un *protoboard*, y una vez que el circuito funcionaba adecuadamente se pasaba a un circuito impreso y allí se soldaban los dispositivos.

Esa metodología de diseño tenía varios inconvenientes. Por ejemplo, el hecho de tener una gran cantidad de cables conectando los diferentes dispositivos sobre el *protoboard* incrementaba la capacitancia, inductancia y resistencia parásita; peor aún, si el circuito implementado sobre el *protoboard* no funcionaba adecuadamente, había que desarmar el circuito, volver a diseñarlo y rearmarlo, este proceso dependiendo de la complejidad del circuito podría durar muchísimos días provocando una gran pérdida de tiempo y recursos.

En la actualidad, los circuitos se analizan, diseñan e implementan utilizando alguna herramienta para diseño electrónico automático (EDA). Para el caso de los circuitos digitales, los lenguajes de descripción de *hardware* (HDL) y los arreglos de compuertas programables en campo (FPGA) o los dispositivos lógicos programables complejos (CPLD), en conjunto, son las herramientas utilizadas.

1.2 VHDL

VHDL es un lenguaje de descripción de *hardware* y es un estándar del Institue of Electrical and Electronic Engineers IEEE. Las siglas VHDL provienen de la unión de V y HDL; la V proviene a su vez de las siglas de *very high speed integrated circuits*, y HDL son las siglas para *hardware description language*, por lo que la VHDL se traduce como lenguaje de descripción de *hardware* para circuitos integrados de muy alta velocidad.

VHDL nace como una iniciativa del departamento de defensa de los Estados Unidos en 1980. La primera versión se denominó VHDL 87 y la segunda VHDL 93. VHDL fue el primer y original lenguaje de descripción de *hardware* estandarizado por el (IEEE). El estándar fue el IEEE 1076 *standard*, posteriormente se añadió el estándar IEEE 1164 que incluye multivalores.

Por ser VHDL un lenguaje estándar, es reusable y portable. Dos son los campos de aplicación típicos de VHDL. El uno es en el campo de los dispositivos lógicos programables como los FPGA y los CPLD, y el otro es en el campo de los circuitos integrados para aplicaciones específicas *application specific integrated circuits* –(ASIC). A partir de un programa o código escrito con VHDL, se construye el respectivo dispositivo físico (el *hardware*). Este *hardware*, no es más que un dispositivo programable o un ASIC. Entre los dispositivos programables están los FPGA y los CPLD. Los circuitos integrados ASIC se construyen por pedido especial a un fabricante de circuitos integrados.

Los dispositivos programables, a diferencia de los ASIC, se pueden comprar en el mercado electrónico mundial. Un ASIC lógicamente no está disponible en una tienda electrónica, hay que mandarlo a fabricar. Hay una amplia variedad de fabricantes de FPGA y CPLD entre estos están Altera y Xilinx.

Muchos ASIC, como por ejemplo los microcontroladores o microprocesadores, son diseñados y construidos mediante algún lenguaje de descripción de *hardware*.

VHDL a diferencia de un lenguaje de programación común, que ejecuta las sentencias en forma secuencial, ejecuta todas las sentencias al mismo tiempo, en forma paralela, es decir, la primera línea de código se ejecuta al mismo tiempo que la última línea de código, por esta razón este lenguaje es concurrente y en lugar de programa se suele llamar código. Sin embargo, este lenguaje puede ejecutar sentencias en forma secuencial mediante un process, procedure o una function, esto será presentado más adelante.

1.3 Descripción de circuitos con VHDL

Un lenguaje de descripción de *hardware*, como su nombre lo indica, es un lenguaje que describe el funcionamiento de un circuito (el *hardware*). Para entender su significado, se hace referencia, primero, a la metodología para diseñar un circuito digital y luego se asocia esta metodología con VHDL.

Cuando se diseña un circuito digital o algo más complejo un sistema digital muy grande, generalmente la metodología de diseño sigue el procedimiento que se describe a continuación.

Se estudia las especificaciones de la manera cómo debe funcionar el circuito o sistema y, a partir de estas, se identifican las entradas y las salidas del circuito o sistema.

Luego se dibuja un diagrama de bloque (una abstracción) que represente al circuito o sistema. En el bloque, se dibujan las entradas y las salidas (incluyendo el número de bits o tipo de datos). A continuación, se establece la relación que hay entre las salidas y las entradas mediante una tabla de verdad en el caso de los circuitos combinacionales o mediante un algoritmo dibujado con símbolos especiales para el caso de los circuitos secuenciales. Finalmente la partida de la tabla de verdad o del algoritmo, se construye el circuito físico o *hardware*.

El diagrama de bloque tiene un nombre que hace referencia a lo que el circuito hace, por ejemplo: sumador, multiplexor, etc. Las señales de entrada se dibujan entrando al bloque y cada señal tiene su respectivo nombre. Las señales de salida se dibujan saliendo del bloque y también tienen sus respectivos nombres. Un ejemplo de un diagrama de bloque muestra la figura 1.1. Las entradas van desde E1 hasta En (el tipo de datos que representan son bits, un bit por señal, para este ejemplo) y las salidas van desde S1 hasta Sm.

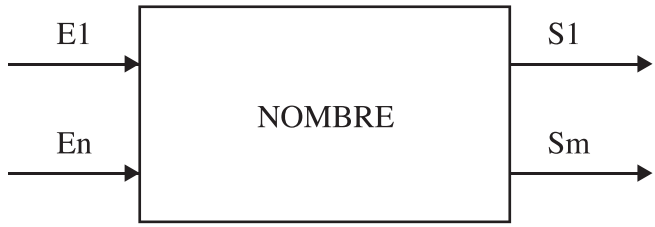


Figura 1.1 Ejemplo de un diagrama de bloques.

La relación entre esta metodología de diseño y VHDL es la siguiente: en VHDL, al diagrama de bloque (el circuito), se le denomina ENTIDAD. A la manera como debe funcionar el circuito o sistema se le llama ARQUITECTURA.

Por lo tanto, en VHDL, el diagrama de bloque es la ENTIDAD el funcionamiento del circuito es la ARQUITECTURA.

De ahí que un programa en VHDL debe estar asociado a un diagrama de boques y al funcionamiento del mismo.

El funcionamiento de un circuito puede ser descrito de diferentes maneras, por lo tanto, se pueden tener varias arquitecturas asociadas a una misma entidad. Esto se hará evidente más adelante.

1.4 ENTIDAD

Como se indicó, la entidad está asociada con el diagrama de bloques. Por lo tanto, debe haber, en VHDL, alguna manera de indicar las partes que tiene ese diagrama de bloques, entendiéndose por partes a:

- Su nombre.
- Los nombres de las entradas.
- Los nombres de las salidas.
- El tipo de datos que tienen las entradas y salidas.

En VHDL, al bloque se hace referencia con la palabra reservada: *ENTITY,* el nombre del bloque, por lo tanto, es el nombre de la entidad.

Las señales de entrada y salida en VHDL se conocen como puertos y se hace referencia a ellas mediante la palabra reservada *PORT*. Si son entradas son puertos de entrada y, si son salidas, son puertos de salida, y se utilizan las palabras reservadas *In* para las entradas y *out* para las salidas.

VHDL a diferencia de algunos lenguajes de programación, no hace distinción entre las letras mayúsculas y minúsculas, por lo tanto, las palabras: entrada, ENTRADA, EnTrADA son equivalentes.

A continuación, se presenta un ejemplo en donde se declara una entidad. La figura 1.2 muestra el diagrama de bloques del circuito que se quiere diseñar. Como se ve, tiene dos entradas E1 y E2, ambas de un solo bit, y dos salidas S1 y S2 también de un solo bit; el bloque se llama ejemplo.

El programa o código de la declaración de la *entidad* es el siguiente:

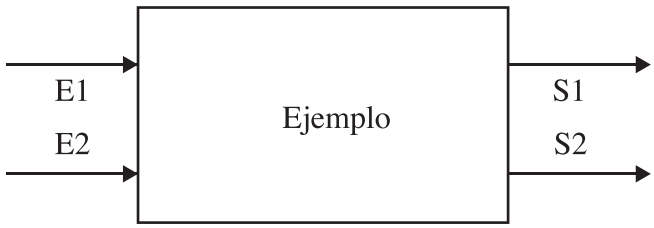


Figura 1.2. Bloque para declarar una entidad

- dos guiones medios seguidos en VHDL se utilizan para escribir comentarios.

- Este es un ejemplo de declaración de entidad

```
--
--
    ENTITY ejemplo IS
    PORT( E1, E2: IN BIT;
          S1, S2: OUT BIT);
    END ejemplo;
```

Las dos primeras líneas de este programa empiezan con dos guiones medios (--), estos guiones indican que lo que se escribe a continuación son comentarios y el compilador ignora esa línea.

La tercera línea empieza con la palabra reservada ENTITY, esta palabra hace referencia al bloque e indica el comienzo de la declaración de la entidad. A continuación se escribe el nombre que se le asignó al bloque que, por supuesto, es el nombre de la entidad (ejemplo) y, la línea termina con la palabra reservada IS.

En la cuarta línea, se hace referencia a las entradas y salidas del bloque mediante la palabra reservada PORT, como si se dijera "el bloque está compuesto por las siguientes entradas y salidas". A continuación, se especifican los nombres de las entradas al bloque para el ejemplo, E1 y E2 las entradas se escriben una a continuación de otra separadas por comas y terminadas por dos puntos (:), estos dos puntos asignan todo lo que se escribe a su lado derecho a los nombres que se escriben a su lado izquierdo, siguiendo con el ejemplo, a continuación de los dos puntos están escritas las palabras reservadas IN y BIT, indican que E1 y E2 son entradas y de un solo bit, las sentencias en VHDL se terminan con un punto y coma (;).

En la quinta línea, siguiendo con el ejemplo, se escriben separados por comas los nombres de las salidas colocamos dos puntos al final y a continuación, se escriben las palabras reservadas OUT y BIT indicando que se asignan OUT y BIT a lo que está escrito a la izquierda de los dos puntos, es decir a S1 y S2. De esta manera, S1 y S2 quedan identificadas como salidas de un solo bit.

En resumen, en PORT, se deben escribir los nombres de las entradas, las salidas y el tipo de datos que representan.

La declaración de la entidad se termina con la palabra reservada END, el nombre de la entidad y seguido de un punto y coma.

1.5 Identificadores

Un Identificador en VHDL, es un nombre que va a identificar a algo (una variable, una señal, una entidad, una arquitectura, etc). Un nombre para ser escrito correctamente, debe seguir las siguientes reglas:

- Puede tener letras mayúsculas y/o minúsculas. Ejemplo: sUmaDor.
- No puede iniciar con un número. Ejemplo: 2sumador.
- El segundo carácter no puede ser un guion bajo. Ejemplo: s_umador.
- No puede tener dos guiones bajos seguidos. Ejemplo: suma__dor.
- No puede tener símbolos especiales. Ejemplo: suma&%$dor.
- No se pueden utilizar palabras reservadas. Ejemplo: entity, end, out, in.

1.6 Puertos de entrada y salida

Los puertos de entrada y salida deben ser identificados de la siguiente manera:

- Con un nombre. Un identificador que lo caracteriza.
- Un modo. Hace referencia a la dirección en que los datos son transferidos por el puerto.
- El tipo de datos. Es la clase (tipo) de información que contendrán las entradas y salidas escritas en el puerto.

El modo de un puerto puede ser:

- Modo IN: indica que el puerto es una entrada.
- Modo OUT: indica que el puerto es una salida.
- Modo INOUT: indica que el puerto es bidireccional, es decir, puede actuar como entrada y salida y puede ser realimentado al interior de la entidad o hacia el exterior.
- Modo BUFFER: indica que el puerto puede ser realimentado internamente hacia la entidad pero no puede realizar realimentaciones al exterior de la entidad. En esencia, se comporta como una salida. Un ejemplo concreto es la salida Q de un *latch* básico: sus salidas Q y /Q son rea-

limentadas hacia el interior del circuito, pero solo se comportan como salidas. Q se debe declarar como modo *buffer*.

VHDL, soporta una gran variedad de tipos de datos. En esta sección, solo se mencionan algunos de ellos.

- Bit. Representa a: un uno lógico o a un cero lógico.
- Bit_vector. Representa a un conjunto de bits, a es un vector de bits.
- Boolean. Representa un valor verdadero o falso.
- Integer. Representa a un número entero.

Un programa en VHDL tiene cinco unidades básicas de diseño y son: declaración de entidad, declaración de arquitectura, configuración, declaración de paquete y cuerpo de paquete. De estas cinco unidades, dos siempre deben estar presentes y son la declaración de entidad y de arquitectura, el resto de las unidades pueden o no estar. A continuación, se presentan algunos ejemplos.

Ejemplo 1.1

Declare la entidad de un sumador medio para datos de un solo bit.

Un sumador medio tiene dos señales de entrada que son los números que se van a sumar y dos salidas el uno contiene el resultado de las suma y el otro el acarreo, el diagrama de bloque del sumador medio se muestra en la figura 1.3.

Figura 1.3. Diagrama de bloques del sumador medio

La declaración de la entidad es:

```
ENTITY sumador_medio IS
PORT( D1, D2: IN BIT; S, A: OUT BIT);
END sumador_medio;
```

Ejemplo 1.2

Declare la entidad de un multiplexor de 2 a 1 para datos de un bit.

Un multiplexor (mux) de dos a uno tiene tres líneas de entrada: la línea de selección (S), dos líneas de datos de entrada (D1, D2) y una línea de salida (Y), La figura 1.4 muestra el diagrama de bloque del multiplexor de 2 a 1 para datos de un bit.

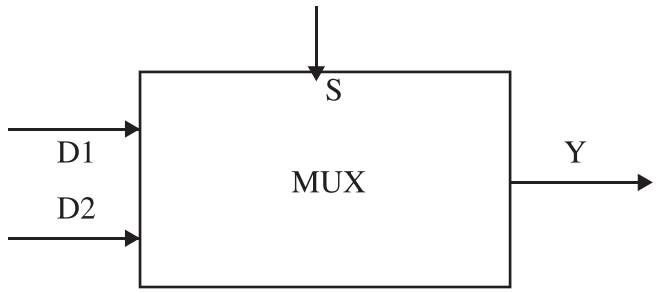


Figura 1.4. Diagrama de bloques de un mux de dos a uno.

La declaración de la entidad es:

```
ENTITY mux IS
PORT(S, D1, D2: IN BIT; Y:  OUT BIT);
END MUX;
```

Ejemplo 1.3

Declare la entidad de un sumador completo, que sume números de cuatro bits de información.

Un sumador completo de cuatro bits tiene una entrada de un bit que es el acarreo de entrada, dos líneas de cuatro bits cada una (vectores de cuatro bits de datos cada uno) una salida de cuatro bits (vector de cuatro bits) que representa a la suma y una línea de salida de un bit que representa al acarreo de salida, cómo se muestra en la fig. 1.5.

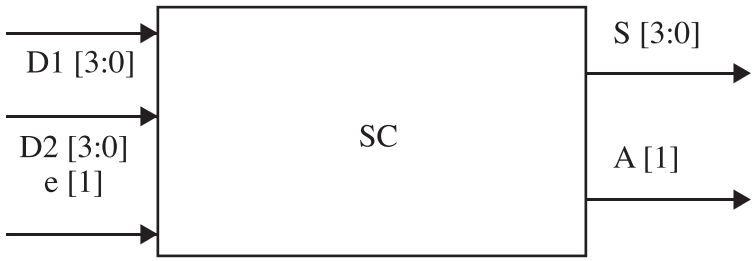


Figura 1.5. Diagrama de bloque el sumador

La declaración de entidad es:

```
ENTITY SC IS
PORT(e: IN BIT;
        D1, D2: IN BIT_VECTOR (3 downto 0);
        A: OUT BIT;
        S: OUT BIT_VECTOR (3 downto 0));
END SC;
```

Cuando un puerto se declara del tipo BIT_VECTOR es necesario especificar el orden de las celdas del vector.

Por ejemplo si se utiliza esta representación: (3 downto 0) significa que el bit más significativo se encuentra en la posición 3 y el menos significativo se encuentra en la posición 0, como se indica en la tabla 1.1.

Valores más significativos y menos significativos			
3	2	1	0
Bit más significativo			Bit menos significativo

Tabla 1.1 Uso de *downto*.

Si se especifica el vector como (0 to 3), significa que el bit más significativo se encuentra en la posición 3. Como se indica en la tabla 1.2.

Valores más significativos y menos significativos			
0	1	2	3
Bit menos significativo			Bit más significativo

Tabla 1.2 *Uso de To.*

1.7 Librerías y paquetes

Una librería en VHDL contiene paquetes. Los paquetes, a su vez, contienen funciones, procedimientos, componentes, constantes y tipos de datos. Si algún tipo de información (código) se utiliza continuamente, se debe poner dentro de una librería.

Las funciones, procedimientos, componentes, constantes y tipos de datos, contienen un programa en VHDL que realiza alguna función específica.Por ejemplo, puede ser un programa que realiza la multiplicación de números en forma muy eficiente. Para que estas funciones, procedimientos, componentes, constantes y tipos de datos puedan ser usados y reusados por un programa o por otros programas son puestos dentro de paquetes y estos paquetes se compilan dentro de una librería destino quedando así la información que contiene la librería lista para ser utilizada las veces que se requiera y en los programas que se necesiten.

Por ejemplo si se tiene una función que tiene el código que multiplica números y esta función está dentro de un paquete y este paquete se ha compilado en una librería entonces la información (el código que multiplica números) que está en la librería puede ser utilizada por cualquier programa las veces que sean necesarias.

Si, en un programa, se necesita multiplicar números, ya no será necesario escribir el programa que multiplique los números, sino que más bien habrá que llamar a la librería y al paquete que contiene esa multiplicación.

Para usar la información de una librería hay que llamarla, indicar el paquete de esa librería y la parte del paquete que contiene la información requerida, este proceso es necesario porque una librería contiene varios paquetes y cada paquete tiene funciones, procedimientos, etc.

La ventaja de tener librerías está en que el código que contienen puede ser reutilizado en un mismo diseño o compartido en otros diseños, como ya se indicó antes.

1.8 Declaración de librería

Cuando se quiere utilizar una librería hay que declararla (hacerla visible) dentro del programa escribiendo dos líneas de código y son:

```
LIBRARY nombre_libreria;

USE nombre_libreria.nombre_paquete.parte_del_paquete;
```

LIBRARY es la palabra reservada para librería, nombre_libreria es el nombre de la librería que se va a utilizar.

USE es una palabra reservada y a su derecha se debe escribir el nombre de la librería, el nombre del paquete y el nombre de la parte del paquete que se va a utilizar. Si se va a utilizar todo el paquete se escribe la palabra reservada ALL.

En un programa en VHDL normalmente se utilizan las librerías y paquetes que se indican en la tabla 1.3.

Librería	**Paquete**	**contenido**	**Visible por *default***
IEEE	STD_LOGIC_1164	Contiene todos los tipos de datos	**sí**, hay que declararla
STD	STANDARD	Contiene tipos de datos básicos.	**No**, no hay que declararla.

WORK	-	Programas del usuario.	**No**, no hay que declararla.

Tabla 1.3 Librerías y paquetes de uso frecuente

Las librerías STD y WORK son visibles por default y no hay que declararlas, otras librerías hay que declararlas o hacerlas visibles antes de utilizar su información.

Ejemplo 1.4

Analice la siguiente declaración:

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;
```

Esta dos líneas hacen visible la librería IEEE; además, la segunda línea indica que el paquete de esta librería que se va a utilizar es el: std_logic_1164 y se va a usar no una parte del paquete sino todo su contenido (all).

Ejemplo 1.5

A pesar de que no es necesario declarar las librerías STD y WORK, declárelas e indique la librería, el paquete y la parte del paquete que se utiliza.

La declaración es:

```
LIBRARY std;
USE std.standard.all;
LIBRARY work;
USE work.all;
```

En el primer caso, la librería se llama: std, el paquete que está dentro de esa librería es el: standard y la parte del paquete que se va a utilizar es todo el paquete (all).

Para el segundo caso el nombre de la librería es work, no tiene paquetes y se va a utilizar toda la librería, work contiene toda la información que vaya generando el usuario.

1.9 Tipos de datos predefinidos

VHDL tiene varios tipos de datos predefinidos, que el usuario puede utilizar. En la tabla 1.4, se muestran dos librerías con sus respectivos paquetes.

LIBRERÍA	PAQUETE	EL CONTENIDO DEL PAQUETE DEFINE LOS SIGUIENTES TIPOS DE DATOS	
STD	**standard**	**BIT**	1 y 0. Dos niveles lógicos
		BOOLEAN	True, False
		INTEGER 32-bit	Enteros de -2 147 483 647 a +2 147 483 647.
		NATURAL	Enteros de 0 a +2 147 483 647
		REAL	de -1.0E38 to +1.0E38. No sintetizable
IEEE	**std_logic_1164**	**STD_LOGIC**	1, 0, H, L, Z, X, -, W (ocho niveles lógicos)
		STD_ULOGIC	U, 1, 0, H, L, Z, X, -, W (nueve niveles lógicos)
	std_logic_arith Para operaciones aritméticas y de comparación.	**SIGNED**	+ 3 0 011 -3 1 101 Ca2
		UNSIGNED: números mayores a cero ej: 3	
	std_logic_signed	Transforma tipos de datos std_logic_vector a Signed (número con signo): ejemplo: transforma el vector 011 (bits) al número 3 con signo +3 (0011). Permite a los vectores ser tratados como números con signo.	
	std_logic_unsigned	Transforma un grupo de bits (vector) a un número sin signo tipo UNSIGNED. Ejemplo: el vector 011 (Bits) se convierte en el número 3 (011) sin signo, UNSIGNED. Permite a los vectores ser tratados como números sin signo.	

Tabla 1.4 Tipos de datos predefinidos

La tabla 1.5 muestra los símbolos de los niveles lógicos predefinidos que están en el paquete: std_logic_1164, de la librería IEEE.

Símbolo	**Valor lógico al que representa**
X	Valor desconocido (sintetizable desconocido)
0	Nivel bajo (‘0’ lógico es sintetizable)
1	Nivel alto (‘1’ lógico es sintetizable)
Z	Alta impedancia (*buffer tri-state* sintetizable)
W	Valor débil desconocido
L	Nivel débil bajo
H	Nivel débil alto
–	No importa
U	Valor lógico no resuelto.

Tabla 1.5 Significado de los símbolos lógicos

1.10 Tipos de datos definidos por el usuario

El usuario puede definir dos tipos de datos y son los enteros y los enumerados,

Los tipos de datos enteros definidos por el usuario deben declararse como se indica a continuación:

TYPE mi_nombre IS RANGE valor inicial TO valor final;

La declaración se inicia con la palabra reservada TYPE, un nombre que el usuario debe asignar (mi_nombre) a sus tipos de datos y además debe definir el rango de esos datos (valor inicial, valor final) mediante las palabras reservadas: IS RANGE y TO.

Ejemplo 1.6

Analice la siguiente declaración:

TYPE mis_enteros IS RANGE -100 TO 100;

Esta declaración indica que el tipo de datos entero, mis_enteros (nombre), es un subconjunto de los enteros; está en el rango que va desde el -100 hasta 100.

Ejemplo 1.7

Analice la siguiente declaración:

TYPE número_estudiantes_vhdl IS RANGE 1 TO 52;

Esta declaración indica que el tipo de datos entero: número_estudiantes_vhdl (nombre) está en el rango que va desde el 1 hasta 52.

Para el caso de los tipos de datos enumerados, definidos por el usuario, deben declarase siguiendo el siguiente formato:

TYPE nombre IS (elemento1, elemento 2, ..);

TYPE, IS son palabras reservadas. Nombre es el nombre que el usuario le asigna a su tipo de datos enumerados. Entre paréntesis y separados por comas se enumeran los elementos que conforman el conjunto de datos. Los elementos se escriben entre comillas simples (apóstrofes) si el tipo es un subconjunto de un tipo predefinido.

Automáticamente estos elementos son guardados en el orden que se escribieron (salvo que se indique otra cosa) y se les asigna un código binario, este código tendrá los bits que sean necesarios, por ejemplo si son 8 elementos serán necesarios tres bits cuyo código ira desde el 000 hasta el 111.

Ejemplo 1.8

Analice la siguiente declaración:

TYPE tamaño IS (bajo, medio, alto);

El tipo de datos enumerado se llama: tamaño y tiene los tres siguientes elementos: bajo, medio y alto.

Ejemplo 1.9

Analice la siguiente declaración:

TYPE mis_niveles IS (‘0’, ‘1’, ‘z’);

El tipo de datos enumerado se llama: mis_niveles y tiene los tres siguientes elementos: 0, 1, z. los elementos están entre comillas simples porque sus elementos son parte de un tipo de datos predefinidos (std_logic).

Ejemplo 1.10

Analice la siguiente declaración:

TYPE mis_estados IS (a, b, c, d, e, f)

El tipo de datos enumerado se llama: mis_estados y tiene los seis siguientes elementos (estados): a, b, c, d, e, f. Este tipo de datos se utiliza en las máquinas secuenciales.

1.11 Subtipos

Un subtipo es un subconjunto de un tipo de datos, o es un tipo de datos con alguna restricción, o es un tipo de datos dentro de otro tipo de datos.

Los subtipos se suelen utilizar, por ejemplo, cuando se tiene incompatibilidad entre tipos de datos. Así, si se tiene una señal x declarada como de tipo BIT y otra señal w declarada como de tipo STD_LOGIC, no se pueden realizar operaciones entre estas dos señales porque son de diferente tipo.

Un subtipo de datos se declara siguiendo el siguiente formato:

SUBTYPE nombre_subtipo IS nombre_tipo RANGE (valor inicial TO valor final);

SUBTYPE, IS, RANGE y TO son palabras reservadas.

Ejemplo 1.11

Analice la siguiente declaración:

TYPE mis_estados IS (a, b, c, d, e, f);

SUBTYPE mi_subtipo IS mis_estados RANGE a TO c;

El tipo mis_estados está compuesto por los elementos: a, b, c, d, e, f. El subtipo mi_subtipo está compuesto por una parte de los elementos de mis_estados y va desde el elemento a al c, es decir, a, b, c.

Ejemplo 1.12

Analice la siguiente declaración:

TYPE color IS (rojo, verde, azul, blanco);

SUBTYPE mi_color IS color RANGE rojo TO verde;

El tipo color está compuesto por los elementos: rojo, verde, azul y blanco. El subtipo mi_color está compuesto por una parte de los elementos de color y va desde el elemento rojo al verde.

Ejemplo 1.13

Analice la siguiente declaración e indique entre qué señales se pueden realizar operaciones.

SUBTYPE mi_subtipo IS STD_LOGIC RANGE '0' TO '1';

SIGNAL x: BIT;

SIGNAL w: STD_LOGIC;

SIGNAL z: mi_subtipo;

No se pueden realizar operaciones entre x y w porque son de diferente tipo (bit, STD_LOGIC), sin embargo como z y w son del mismo tipo (STD_LOGIC) se pueden realizar operaciones entre estas.

TYPE mis_enteros IS RANGE -100 TO 100;

TYPE tamaño IS (bajo, medio, alto);

TYPE mis_niveles IS ('0', '1', 'z');

TYPE color IS (rojo, verde, azul, blanco);

SUBTYPE mi_color IS color RANGE rojo TO verde;

1.12 Operadores

Los siguientes operadores están predefinidos en VHDL:

- Asignación
- Lógicos
- Aritméticos
- Relacionales
- De desplazamiento
- De concatenación

1.13 Operadores de asignación

Los operadores de asignación permiten asignar valores a:

- Señales
- Variables
- Constantes

Sus símbolos son:

- <=
- :=
- =>

En la tabla 1.6, se resume la forma cómo se utilizan estos operadores.

Operador	Símbolo	Uso
Asignación	<=	Para asignar valores a señales.
	:=	Para asignar valores a variables, constantes y genéricos. Para establecer valores iniciales.
	=>	Para asignar valores individuales a cada uno de los elementos de un vector. Se utiliza también con la instrucción OTHERS.

Tabla 1.6. El operador de asignación.

Ejemplo 1.14

Se muestra cinco ejemplos de asignaciones a señales y variables.

a <= ‘0’; se asigna un cero a la señal a.

b := “0011”; se asigna 0011 a la variable b.

c <= “0111”; se asigna 0111 a la señal c.

d <= (0 =>’1’, OTHERS =>’0’); a la señal d se asigna 1000. En este caso d es un vector con cuatro celdas; a la celda cero se asigna 1 y al resto 0.

e <= (0 =>’0’, OTHERS =>’1’); a la señal e (que es un vector de cuatro celdas) se asigna 0111.

1.14 Operadores lógicos

Los operadores lógicos permiten realizar operaciones lógicas entre datos del tipo:

- BIT
- BIT_VECTOR
- STD_LOGIC
- STD_LOGIC_VECTOR
- STD_ULOGIC
- STD_ULOGIC_VECTOR

Los operadores lógicas son:

- NOT
- AND
- OR
- NAND
- NOR
- XOR
- XNOR

Ejemplo 1.15

Escriba tres ejemplos de uso de los operadores lógicos.

a <= NOT z AND b; operación lógica and entre z negado y b.

c <= a AND b); operación lógica y (and) entre a y b.

y <= a OR b; operación lógica o (or) entre a y b.

1.15 Operadores aritméticos

Los operadores aritméticos se utilizan para realizar operaciones aritméticas entre datos de tipo:

- Enteros con signo y sin signo
- Reales

Si son utilizados los paquetes:

- std_logic_signed
- std_logic_unsigned

Los datos del tipo std_logic_vector pueden ser utilizados directamente para realizar operaciones de suma y resta.

Los símbolos de los operadores aritméticos son:

- +, para la suma
- -, para la resta
- *, para la multiplicación
- /, para la división
- **, para la exponenciación (la base y el exponente deben ser enteros).
- MOD, para el módulo
- REM, para el residuo
- ABS, para el valor absoluto

1.16 Operadores de comparación

Los operadores de comparación o relacionales son:

- =, para igualdad
- /=, para no igual
- <, para menor que
- >, para mayor que
- <=, para menor o igual a
- >=, para mayor o igual a

1.17 Operadores de desplazamiento

Los operadores de desplazamiento permiten desplazar bits de datos. La sintaxis es la siguiente:

Dato de tipo: BIT_VECTOR	operador de desplazamiento	entero

Los operadores de desplazamiento son:

SLL es el desplazamiento lógico a la izquierda. Los bits son desplazados una posición a la izquierda, la posición del bit que se encuentra más a derecha es reemplazada por un cero. La figura muestra esta operación.

1	1	0	1	1	Dato original
1	0	1	1	0	Dato desplazado

SRL es el desplazamiento lógico a la derecha. Los bits se desplazan una posición a la derecha. La posición del bit que se encuentra más a la izquierda es reemplazado por un cero.

1	1	0	1	1	Dato original
0	1	1	0	1	Dato desplazado

1.18 Operadores de concatenación

Los operadores de concatenación son:

- &
- (, , , ,)

Los tipos de datos que soportan son:

- BIT, BIT_VECTOR
- STD_LOGIC, STD_LOGIC_VECTOR
- STD_ULOGIC, STD_ULOGIC_VECTOR
- SIGNED, UNSIGNED

1.19 Atributos de las señales

El atributo de una señal da o retorna algún tipo información de la señal. Por ejemplo, si una señal no ha cambiado, o ha cambiado, o se ha mantenido estable por un tiempo. La tabla 1.7 muestra la información que retorna cada atributo de una señal.

En este punto, con la finalidad entender el concepto de señal en un sentido muy simple (más adelante se formalizará este concepto) y poder identificarla con facilidad en un circuito se da la siguiente orientación. Una señal puede ser relacionada con los cables que conectan a los diferentes dispositivos en un circuito; es decir, todo cable asocia a una señal o donde hay un cable, hay una señal.

La tabla 1.7 muestra la señal denominada s y sus posibles atributos.

Atributo	**Retorna** **V= verdadero**	**Comentario**
s'EVENT:	v	Si un evento, un cambio, en la señal se ha producido.
s'STABLE:	v	Si un evento, un cambio, en la señal NO se ha producido.
s'ACTIVE:	V	Si s=1,
s'QUIET <tiempo>	V	Si un evento no se ha producido en la señal s en el tiempo especificado.
s'LAST_EVENT:	tiempo	Retorna el tiempo transcurrido desde el último evento en la señal s.

s'LAST_ACTIVE:	tiempo	Retorna el tiempo transcurrido desde el último evento donde s fue igual 1.
s'LAST_VALUE:	valor	Retorna el valor que tuvo s antes de su último evento.

Tabla 1.7. Los atributos de una señal

El atributo que más se suele utilizar en circuitos secuenciales es: s'EVENT, donde s es la señal del reloj.

Ejemplo 1.16

Si clk es una señal del reloj que resultado retorna: clk'EVENT,

Retorna un valor verdadero cuando la señal clk cambie de nivel de alto a bajo o de bajo a alto, es decir, si se ha producido un flanco de subida o bajada en la señal clk.

CAPÍTULO 2 DISEÑO DE CIRCUITOS COMBINACIONALES CON VHDL

2.1 Introducción

En esta sección, se profundiza VHDL estudiando nuevas particularidades.

Un circuito combinacional tiene las siguientes características:

- Sus salidas son función o dependen exclusivamente del valor de sus entradas.
- No tiene caminos de retroalimentación.
- No tiene elementos de memoria.

Para implementar circuitos combinaciones mediante VHDL, hay dos caminos: el uno es la utilización de código concurrente y el otro es mediante código secuencial. Al usuario, a medida que va adquiriendo habilidad con VHDL, le será fácil elegir el código más adecuado.

Para escribir código concurrente (paralelo) se pueden utilizar operadores y declaraciones como se indica a continuación.

- Operadores.
- Las declaraciones son:
  - WHEN-ELSE
  - WITH-SELECT-WHEN
  - GENERATE
  - BLOCK

2.1.1 Implementación de circuitos combinacionales con operadores

En VHDL los operadores son utilizados para escribir código concurrente en forma básica. En general, los operadores se utilizan para escribir las ecuaciones booleanas que describen el funcionamiento de un circuito.

El funcionamiento de un circuito está especificado en su arquitectura; por lo tanto, es necesario, primero, saber cómo se la declara.

2.2 Declaración de la arquitectura

Para declarar una arquitectura se debe seguir la siguiente sintaxis:

ARCHITECTURE nombre_ARQUITECTURA OF nombre_ENTIDAD IS

Declaraciones

BEGIN

Código;

END nombre_ARQUITECTURA;

Las palabras en mayúsculas y con negrillas son palabras reservadas. ARCHITECTURE identifica el inicio de la declaración de una arquitectura y es la primera palabra que se debe escribir al declararla, nombre_ARQUITECTURA es el nombre asignado. nombre_ENTIDAD es el nombre de la entidad con la cual está relacionada la arquitectura. La sección Declaraciones, es opcional, y son las declaraciones, de por ejemplo, señales, variables y constantes que sean requeridas por el código de la arquitectura. BEGIN indica el inicio del código (el programa). Código es el programa que describe el funcionamiento del circuito. END indica el final de la arquitectura.

Es importante indicar que el nombre de la arquitectura puede ser el mismo nombre de la entidad con la cual está relacionada. Como hay diferentes formas de describir cómo trabaja, como funciona un mismo circuito, es posible también tener más de una arquitectura relacionada a una misma entidad.

Ejemplo 2.1

Utilizando operadores, diseñe una puerta AND de dos entradas. Utilice Quartus II y el entrenador DE2 de Altera para implementar este circuito.

El diagrama de una puerta AND de dos entradas se muestra en la figura 2.1. Las entradas están identificadas con los nombres a y b y la salida con el nombre f.

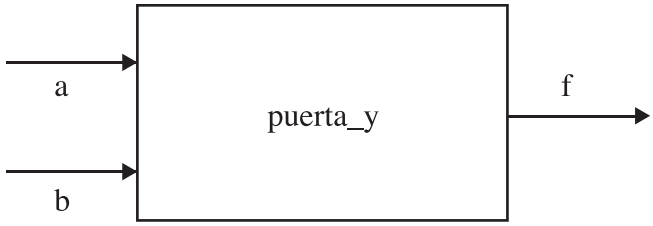


Figura 2.1. Diagrama de una puerta AND.

La declaración de entidad y arquitectura es la siguiente:

```
-- Declaración de la ENTIDAD
ENTITY puerta_y IS
PORT( a, b:in bit;
        f: out bit);
END puerta_y;
-- Declaración de la arquitectura
ARCHITECTURE APY OF puerta_y IS
  BEGIN
     f<= a AND b;
END APY;
```

Nótese que, en la arquitectura la ecuación booleana: f<= a and b, describe el funcionamiento de la puerta AND, en este caso el operador lógico AND se ha utilizado para escribir el código de la arquitectura.

La ecuación booleana indica que la operación lógica AND entre las señales a y b es asignada (<=) a la señal de salida f.

En la figura 2.2 se muestra el programa con la declaración de la entidad y la arquitectura escrito en Quartus II.

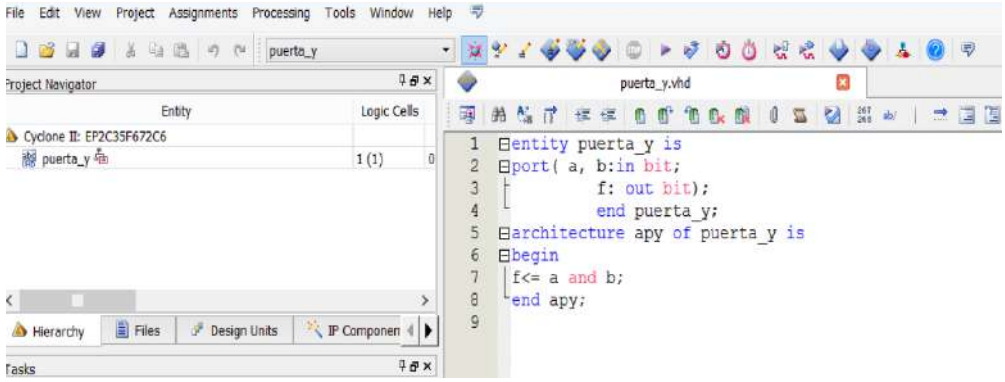


Figura 2.2. Programa para la puerta AND en Quartus II

Para implementar esta puerta con Quartus II, se crea el proyecto denominado: puerta_y siguiendo los pasos indicados en el capítulo: introducción a Quartus II. El nombre de la entidad de más alto nivel que debe escribirse en el proyecto debe ser el mismo nombre de la primera entidad que se escriba en el programa (para este caso, el proyecto y la entidad se llaman puerta_y). Por esto, es buena práctica escribir con el mismo nombre el proyecto y la entidad de más alto nivel cuando se está creando el proyecto.

Como se va a implementar la puerta AND en el entrenador DE2 de Altera, en la ventana de selección de la familia del FPGA se debe seleccionar el ciclón II (DE2 trabaja con el ciclón II), y el dispositivo de esa familia que se debe seleccionar es el que tiene la serie: EP2C35F672C6 (este número esta sobre el chip ciclón II del entrenador DE2 o en las hojas técnicas del DE2).

La simulación se indica en la figura 2.3. Como se puede ver cuando a y b juntas valen 1 lógico, la salida vale también 1 lógico; para el *reset* 0 de los caso la salida es 0 lógico.

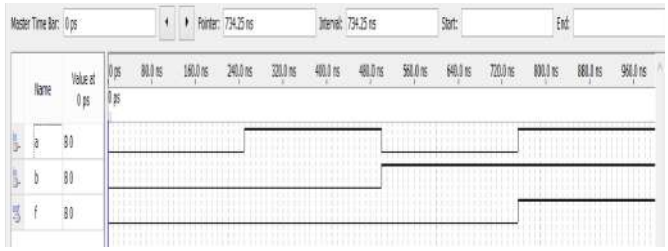


Figura 2.3. Simulación de la puerta AND

Para programar el FPGA disponible en el entrenador DE2 de Altera, se procede de la siguiente manera:

1. Con la información de la hoja técnica de distribución de los pines, *switchs* y *leds* del DE2 (una parte se muestra en la tabla 1.1), se seleccionan los *switchs* que se van a conectar a las entradas a y b de la puerta, así como, el *led* que se va a conectar a la salida f de la puerta AND.

2. En la barra del menú de la ventana principal de Quartus II se selecciona Assignments/ pin planner y aparece la ventana que se indica en la figura 2.4.

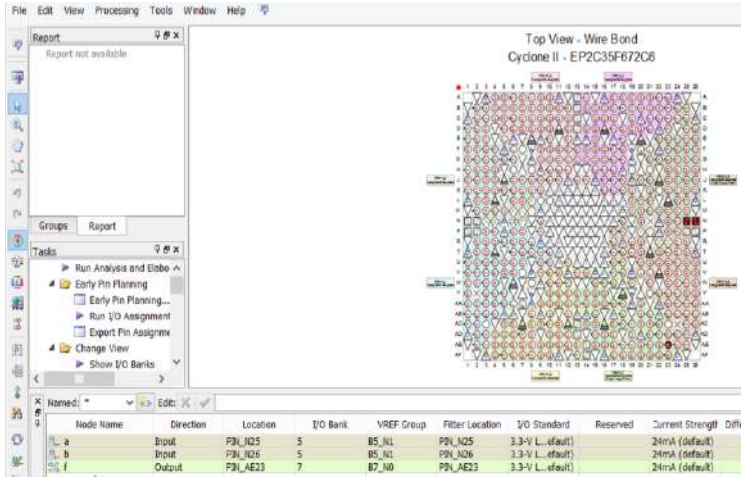


Figura 2.4. Ventana de pin planner

Se hace clic en la columna *location* a la altura de una de las señales (el rectángulo rojo indica este punto) que se encuentran en la parte inferior de esta ventana y aparece la ventana que se muestra en la figura 2.5. En esta, se seleccionan los pines del FPGA.

To	Location
SW[0]	PIN_N25
SW[1]	PIN_N26
SW[2]	PIN_P25
SW[3]	PIN_AE14
SW[4]	PIN_AF14
SW[5]	PIN_AD13
SW[6]	PIN_AC13
KEY[3]	PIN_W26
LEDR[0]	PIN_AE23
LEDR[1]	PIN_AF23
LEDR[2]	PIN_AB21
LEDR[3]	PIN_AC22
LEDR[4]	PIN_AD22
LEDR[5]	PIN_AD23
LEDR[6]	PIN_AD21

Tabla 1.1. Distribución de los pines en el DE2

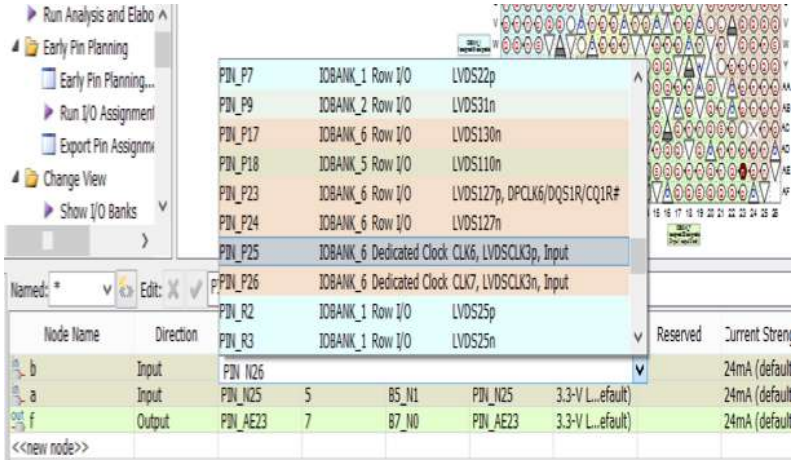


Figura 2.5. Asignación de pines en el FPGA

Por ejemplo de la información de la tabla 1.1, al PIN_N26 del FPGA se asigna la entrada b y el *switch* (sw[1]) conectado a ese pin en el entrenador DE2. El PIN_N25 se asigna a la entrada a y el *switch* conectado a ese pin en el entrenador DE2 es el sw[2]. Finalmente al PIN_AE23 del FPGA se asigna a la salida f; este pin está conectado el led rojo denominado LEDR[0] en el entrenador DE2. Una vez concluida la asignación de los *switch* y *leds* se compila otra vez y se procede a programar el FPGA. Para llevar a cabo esto, en el menú de la ventana principal de Quartus II, se selecciona Tool/Programmer y aparece la ventana que se muestra en la figura 2.6.

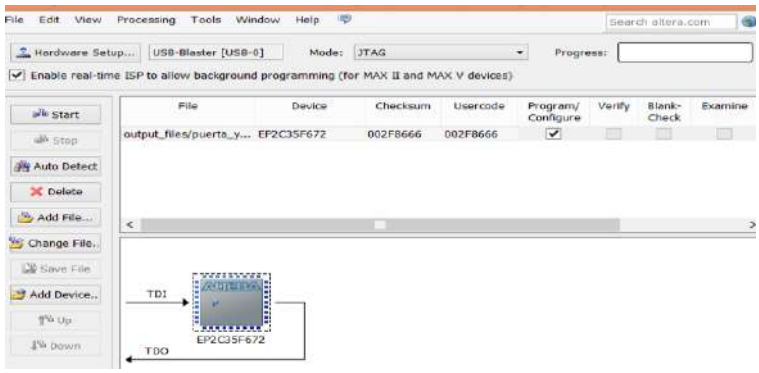


Figura 2.6. Ventana de programación de los pines

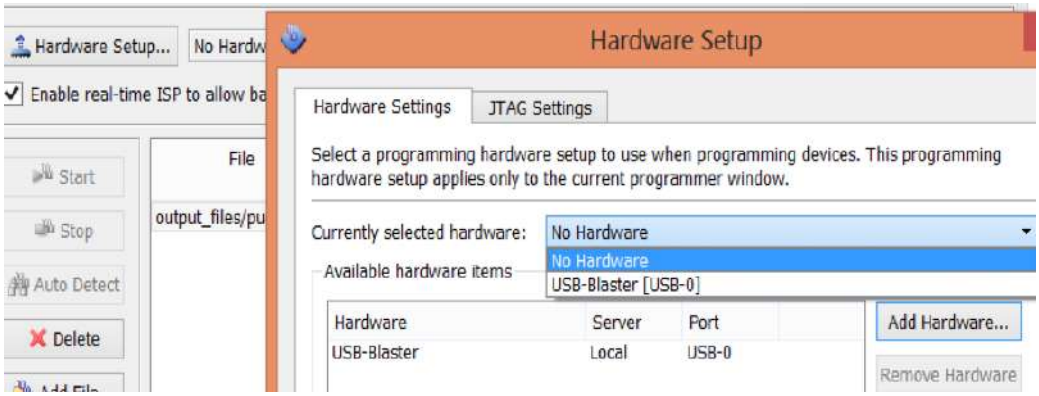


Figura 2.7. Ventana Setup

Si en lugar de la figura 2.7 aparece *no hardware* significa que no está conectado el DE2 a la computadora o que el USB- Blaster no ha sido reconocido. En cualquier caso, se procede de la siguiente manera: se hace clic sobre la pestaña:

hardware setup y aparece la ventana: *hardware setup*, que se indica en la figura 2.8. Allí se hace clic sobre la pestaña *no hardware*, se selecciona USB-Blaster.

Se hace clic sobre el botón: start que se encuentra en la figura 2.7 y el proceso de programación del FPGA se inicia como muestra la figura 2.8. Cuando en la pestaña *progress*, se completa al 100 %, el FPGA está programado y listo para ser utilizado.

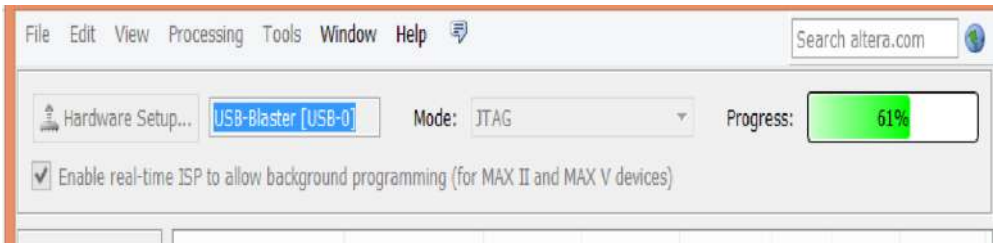


Figura 2.8. FPGA programándose

1. 2.1 Operadores lógicos

Los operadores lógicos son:

- AND
- OR
- NAND
- NOR
- XOR
- XNOR
- NOT

La prioridad en la que se realizan las operaciones de los operadores lógicos es la siguiente:

- Expresiones entre paréntesis
- Complementos
- Función AND
- Función OR

Los complementos o negación tienen la más amplia prioridad y el código que se encuentra entre paréntesis es el primero que se ejecuta, si hay varios paréntesis se ejecutan los paréntesis de izquierda a derecha y desde los paréntesis más internos. Las operaciones de igual prioridad se ejecutan de izquierda a derecha.

Ejemplo 2.2.

Diseñe un multiplexor de cuatro a uno mediante operadores lógicos e impleméntelo en el entrenador DE2 de Altera.

El diagrama de bloques de un multiplexor (mux) de cuatro a uno se muestra en la figura 2.9. El dispositivo tiene dos líneas de selección S1 y S0. S1 es la variable más significativa y S0 es la menos significativa, las cuatro entradas de datos son: A, B, C, D. A es la variable menos significativa y D la más significativa, la única salida es y. La tabla 2.2 muestra el funcionamiento del multiplexor.

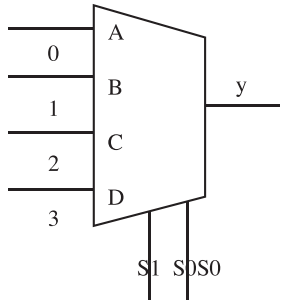


Figura 2.9. Diagrama de bloques de un multiplexor de cuatro a uno

La ecuación booleana de la salida es: y= (/s0 /s1)A + (/s0 s1)C + (s0 /s1)B + (s0 s1)D, que escrita con los operadores lógicos de VHDL queda: y <= (A AND NOT s1 AND NOT s0) OR (B AND NOT s1 AND s0) OR (C AND s1 AND NOT s0) OR (D AND s1 AND s0).

S1	S0	y
0	0	A
0	1	B
1	0	C
1	1	D

Tabla 2.2. Tabla de verdad de un multiplexor de cuatro a uno

Las declaraciones de librería, entidad y arquitectura son las siguientes.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------------------------------------------
-----------
ENTITY mux IS
PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
                    y: OUT STD_LOGIC);
END mux;
 -------------------------------------------------------------------------------------------
------------
 ARCHITECTURE amux_booleana OF mux IS
 BEGIN
    y <= (A AND NOT s1 AND NOT s0) OR  (B AND NOT s1 AND s0) OR
    (C AND s1 AND NOT s0) OR (D AND s1 AND s0);
 END amux_booleana;
```

Para implementar este mux con el FPGA del entrenador DE2 de Altera se procede igual que en el ejemplo anterior. En Quartus II, se crea un proyecto nuevo mediante el asistente para nuevo proyecto, teniendo la precaución de poner el nombre de la entidad de más alto nivel con el nombre de mux, que es el nombre de la entidad en la codificación de VHDL. Se selecciona el ciclon II y dentro de este se selecciona el FPGA: EP2C35F672C6 disponible en el DE2.

Luego, en el menú principal, se selecciona New y en la ventana que aparece se selecciona File VHDL y aparece la ventana en donde se escribe el código del multiplexor, se compila y, una vez compilado, se simula como se indicó en el capítulo denominado introducción Quartus II.

Las formas de onda de la simulación con Quartus II se muestran en la figura 2.10. Como se puede ver, el multiplexor trabaja en forma correcta, selecciona

cada una de las entradas en función del código de las líneas de selección. Por ejemplo, cuando las líneas de selección tienen las dos el valor 00 lógico la línea de entrada que se selecciona es la entrada a.

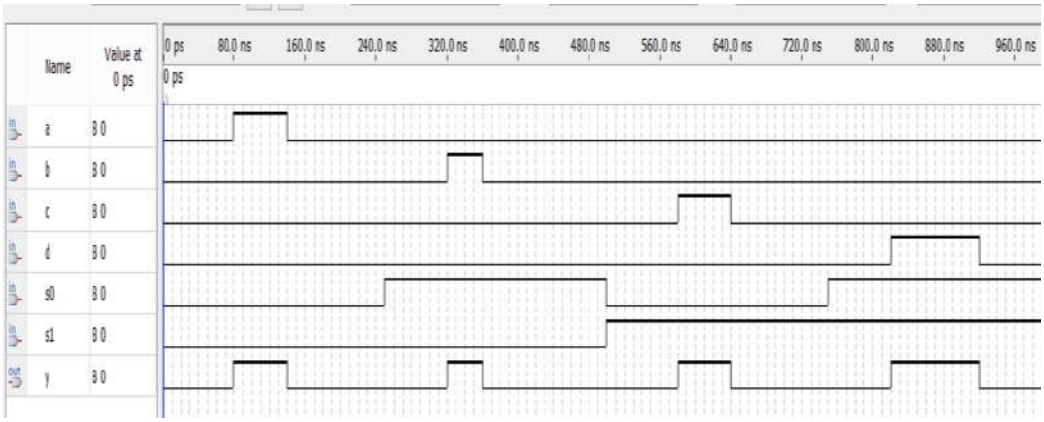


Figura 2.10. Simulación del mux

Una vez comprobado que el mux trabaja a la perfección, se procede a programar el FPGA del DE2. Se seleccionan los *switchs* que van a estar conectados a las entradas: a, b, c, d del mux y a los pines del FPGA, la tabla 2.3 muestra esta asignación. Estos *switch* se encuentran localizados en el extremo derecho del DE2.

***Switch* DE2**	***Location* FPGA**	**Señal de entrada**
SW[0]	PIN_N25	a
SW[1]	PIN_N26	B
SW[2]	PIN_P25	C
SW[3]	PIN_AE14	d

Tabla 2.3. Asignación de *switchs* y pines del DE2

Con la finalidad de distinguir de mejor manera los *switch* asignados a las entradas y a líneas de selección del mux, se seleccionan los *switch* del extremo izquierdo del DE2, para conectar a las líneas de selección del mux como se indica en la tabla 2.4.

Switch DE2	*Location* FPGA	Señal de selección
SW[16]	PIN_V1	s0
SW[17]	PIN_V2	s1

Tabla 2.4. Asignación de *switchs* a las líneas de selección del mux

El *led* seleccionado para conectar a la salida y del mux se muestra en la tabla 2.5.

Led del DE2	Location FPGA	Señal de salida
LEDR[1]	PIN_AF23	y

Tabla 2.5. Asignación del *Led* a la salida y del mux.

Se procede a planear los pines en el FPGA. Para esto, en el menú principal de la ventana de Quartus II, se selecciona: Assignments/ pin planner y aparece la ventana que se indica en la figura 2.11, en la columna Location se procedió a seleccionar los *switchs* y pines como se indican en las tablas anteriores. Para ver el circuito lógico del multiplexor, se selecciona en el menú de la ventana principal de Quartus II: Tool/Netlist Viewers/RTL Viewer y aparece la ventana RTL Viewer que contiene el circuito, la figura 2.12 muestra el circuito.

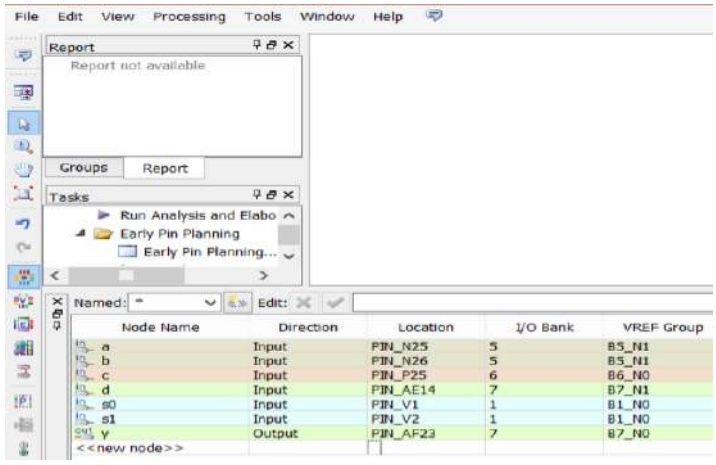


Figura 2.11. Asignación de los pines del FPGA

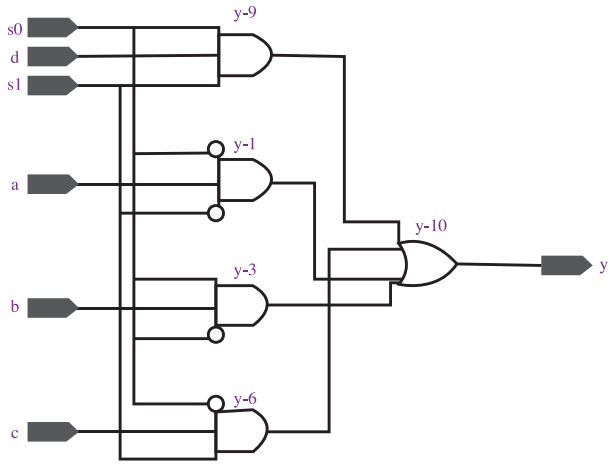


Figura 2.12. Circuito del multiplexor

2.3 Implementación de circuitos combinacionales con declaraciones condicionales

Los circuitos combinacionales pueden ser diseñados con declaraciones del tipo WHEN ELSE o WITH SELECT WHEN.

2.3.1 La declaración WHEN-ELSE

Para escribir código concurrente con las declaraciones WHEN-ELSE, se debe seguir la siguiente sintaxis:

Asignación **WHEN** condición **ELSE**

Asignación **WHEN** condición **ELSE**

….**;**

Asignación, hace referencia al valor que se va a asignar a alguna señal cuando (WHEN) se cumpla o se cumplan las condiciones indicadas en condición.

Ejemplo 2.3.

Utilice la declaración WHEN-ELSE para asignarle un valor 1 lógico a una señal S, cuando las señales a y b son verdaderas. Para cualquier otro valor de a y

b asigne a S un 0 lógico.

```
S<= '1' WHEN (a='1' and b='1') else
        '0';
```

Ejemplo 2.4.

Utilice la declaración WHEN-ELSE para asignarle el vector 1001 a una señal S, cuando las señales a y b tienen valores de 1 o 0 respectivamente. Si a y b tienen un valor de 0 asigne a S el vector 111, para cualquier otro valor de a y b asigne a S el vector 000.

```
S <= "1001" WHEN (a='1' OR b='0') ELSE
        "111" WHEN (a='0' and  b='0') ELSE;     "000";
```

2.3.2 La declaración WITH-SELECT-WHEN

Para escribir código concurrente con las declaraciones WITH - SELECT - WHEN, se debe seguir la siguiente sintaxis:

```
WITH identificador SELECT
Asignación WHEN valor,
Asignación WHEN valor,
...;
```

Asignación tiene el mismo significado que en WHEN-ELSE, valor es el valor o valores que tiene el identificador.

Valor puede ser un solo valor, un rango de valores en cuyo caso se indicara como: valor1 to valor2, esto es aplicable solo en el caso de datos tipo enumerados. O puede ser: valor1 o valor2 o valor3 o……valorn. Entonces WHEN puede tener alguna de las tres formas siguientes:

WHEN valor – un solo valor.

WHEN valor1 to valor2 – para un rango de valores (solo tipo enumerados).

WHEN valor1 | valor2 |...| valorn – para valor1 o valor2 o……valorn.

Ejemplo 2.5.

Utilice la declaración WITH-SELECT-WHEN para asignarle un valor 1 lógico a una señal S, cuando las señales a y b son verdaderas. Para cualquier otro valor de a y b asigne a S un 0 lógico.

A las señales a y b se les asigna el vector V, es decir, V(1)V(0)=ab, con la finalidad de tener una sola señal, el identificador.

```
WITH V SELECT
S <= '1' WHEN "11", -- a S se asigna 1 cuando V=11.
     '0' WHEN OTHERS,- -- a S se asigna 0 cuando V no es igual a 11.
```

Ejemplo 2.6.

Utilice la declaración WITH-SELECT-WHEN para asignarle el vector 1001 a una señal S, cuando las señales a y b tienen valores de 1 o 0 respectivamente. Si a y b tienen un valor de 0 asigne a S el vector 111, para cualquier otro valor de a y b asigne a S el vector 000.

A las señales a y b se les asigna el vector V, es decir, V(1)V(0)=ab, con la finalidad de tener una sola señal, el identificador

```
WITH V SELECT
S<= "1001" WHEN "10",
    "111" WHEN "00",
   "000" WHEN OTHERS;
```

Ejemplo 2.7.

Diseñe el multiplexor de cuatro a uno, con la declaración WHEN-ELSE.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------------------------------------------
-----------
```

```
ENTITY mux IS
PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
                    y: OUT STD_LOGIC);
END mux;
-------------------------------------------------------------------------------
------------
ARCHITECTURE amux_booleana OF mux IS
BEGIN

        y <= a WHEN (s1=' 0' AND s0=' 0' ELSE
             b WHEN (s1=' 0' AND s0=' 0' ELSE
             c WHEN (s1=' 0' AND s0=' 0' ELSE
             d;
END amux_booleana;
```

Ejemplo 2.8.

Diseñe el multiplexor de cuatro a uno, con la declaración WHEN-ELSE, pero las líneas de selección S1 y S0 asuma que es el vector *select*.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------------------------------
-----------
ENTITY mux IS
PORT ( a, b, c, d: IN STD_LOGIC;
                    selec: IN STD_LOGIC_VECTOR ( 1 DOWNTO 0);
                    y: OUT STD_LOGIC);
END mux;
```

```
-------------------------------------------------------------------------------------------
------------
ARCHITECTURE amux_booleana OF mux IS
BEGIN

                y <= a WHEN (selec="00") ELSE
                     b WHEN (selec="01") ELSE
                     c WHEN (selec="10") ELSE
                     d;
END amux_booleana;
```

Ejemplo 2.9.

Diseñe un multiplexor de cuatro a uno, con la declaración WHEN-ELSE, pero las líneas de selección S1 y S0 asuma que es un número entero *selec*.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------------------------------------------
-----------
ENTITY mux IS
PORT ( a, b, c, d: IN STD_LOGIC;
                   selec: IN INTEGER RANGE ( 0 DOWNTO 3);
                   y: OUT STD_LOGIC);
END mux;
-------------------------------------------------------------------------------------------
------------
ARCHITECTURE amux_booleana OF mux IS
BEGIN
```

```
            y <=  a WHEN selec= 0 ELSE
                  b WHEN selec= 1 ELSE
                  c WHEN selec= 2 ELSE
                  d ;
    END amux_booleana;
```

Ejemplo 2.10.

Diseñe el multiplexor del ejemplo 1.24, con la declaración WITH-SELECT-WHEN.

A las líneas de selección S1 y S0 se les denomina el vector selec.

```
    LIBRARY ieee;
    USE ieee.std_logic_1164.all;
    ---------------------------------------------------------------------------------------------
-----------
    ENTITY mux IS
    PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
                        selec: IN STD_LOGIC_VECTOR ( 1 DOWNTO 0);
                        y: OUT STD_LOGIC);
    END mux;
     ---------------------------------------------------------------------------------------------
------------
     ARCHITECTURE amux_booleana OF mux IS
     BEGIN
        WITH selec SELECT
                    y <= a WHEN selec= “00”,
                         b WHEN selec= “01” ,
                         c WHEN selec= “10”,
```

```
                    d WHEN OTHERS; -- es obligatorio utilizar OTHERS, no
se puede usar : when      selec="11".
    END amux_booleana;
```

Ejemplo 2.11.

Diseñe el multiplexor del ejemplo 1.24 con la declaración WITH-SELECT-WHEN.

A las líneas de selección S1 y S0, tratelas como números enteros.

```
    LIBRARY ieee;
    USE ieee.std_logic_1164.all;
    ---------------------------------------------------------------------------------------------
-----------
    ENTITY mux IS
    PORT ( a, b, c, d: IN STD_LOGIC;
                        selec: IN INTEGER RANGE ( 0 TO 3);
                        y: OUT STD_LOGIC);
    END mux;
     ---------------------------------------------------------------------------------------------
------------
     ARCHITECTURE amux_booleana OF mux IS
     BEGIN
        WITH selec SELECT
                    y <= a WHEN selec= 0,
                         b WHEN selec= 1 ,
                         c WHEN selec= 2,
                         d when OTHERS;
    END amux_booleana;
```

CAPÍTULO 3 DISEÑO DE CIRCUITOS SECUENCIALES CON VHDL

3.1 Introducción

Los circuitos secuenciales se caracterizan por realizar actividades en forma repetitiva (secuencial) al pasar por un número de estados finitos y tienen las siguientes características:

- Tienen caminos de retroalimentación.
- Tienen elementos de memoria.
- Sus salidas dependen no solo del valor de las entradas, sino también de los valores que estuvieron presentes en las entradas.

A pesar de que VHDL es un código esencialmente concurrente, es posible escribir código secuencial siempre y cuando se encuentre dentro de un proceso, procedimiento o función.

El código secuencial en VHDL se escribe utilizando las siguientes sentencias: IF, WAIT, CASE, LOOP.

Para circuitos secuenciales interesa más escribir código secuencial dentro de un proceso dejando las funciones y procedimientos para sistemas digitales mucho más grandes.

3.2 Proceso

Un proceso denominado PROCESS contiene código secuencial, que se ejecuta línea por línea secuencialmente durante cada período del reloj, en caso de que lo tenga. La sintaxis es la que se muestra a continuación:

Nombre: **PROCESS** (lista sensitiva)

    Declaración de variables

BEGIN

    Código secuencial;

END PROCESS nombre;

Nombre es el nombre que se le asigne al proceso. No es obligatorio, pero, si un programa tiene varios subprocesos, el nombre permite identificar cada uno.

La lista sensitiva o sensible contiene una lista de nombres de señales, su importancia está en que cada vez que alguna de estas señales cambie de valor lógico el proceso se ejecuta. Dicho de otra manera, si las señales de la lista sensitiva no cambian el proceso no se ejecuta.

El proceso no tiene una lista sensitiva cuando se utiliza WAIT. Entre BEGIN y END PROCESS se encuentra el cuerpo o código del proceso incluyendo la instrucción *wait*.

Las señales, las variables así como las declaraciones IF, WAIT, CASE, o LOOP son utilizadas continuamente, cuando se escribe código secuencial en VHDL. Por lo tanto es necesario tener por lo menos un conocimiento básico sobre estos temas.

3.3 Señales y variables

Una señal se declara en la sección declarativa de una entidad, una arquitectura o un paquete: es global y por lo tanto puede ser invocada desde cualquier parte del programa. En otras palabras, es vista desde cualquier lugar de un programa en VHDL.

Por el contrario, una variable es local y debe ser escrita en la sección de declaración de un código secuencial como por ejemplo dentro de un proceso, un procedimiento o una función. Una variable tiene validez solo dentro de un proceso o procedimiento o función, fuera de ellos simplemente no existe.

Si es necesario sacar el valor de una variable fuera de un proceso, por ejemplo, hay que asignarle su valor a una señal.

En un process, una variable se actualiza inmediatamente. En la siguiente línea de código, tiene su nuevo valor, una señal no se actualiza de inmediato sino solo al final de haber terminado su ciclo el proceso.

3.4 Declaración IF

La declaración IF solo puede estar dentro de un PROCESS, una FUNCTION o un PROCEDURE. La sintaxis es la siguiente.

IF condiciones THEN asignaciones;

ELSIF condiciones THEN asignaciones;

...

ELSE asignaciones;

END IF;

Ejemplo 3.1.

Describa línea a línea el siguiente código.

1. IF (a=b) THEN z:=”10110000”;
2. ELSIF (a<b AND d =’0’) THEN z:=”01010101”;
3. ELSE z:=(OTHERS =>’1’);

En la primera línea, se pregunta si a=b; si esto es cierto, entonces se asigna el valor 10110000 a la variable z. La segunda línea se ejecuta siempre que: a < b y d=’0’ y a z se asigna: 01010101. La tercera línea se ejecuta si las dos líneas de código anteriores no se ejecutaron, es decir al evaluar la condición, esta, fue falsa, y se asigna a z el valor 11111111.

3.5 Declaración CASE

Igual que IF, CASE se utilizada exclusivamente para código secuencial, la sintaxis es la siguiente:

CASE identificador IS

WHEN valor1 del identificador => asignaciones;

WHEN valor2 del identificador => asignaciones;

...

END CASE;

Ejemplo 3.2.

Para el código que se indica a continuación explique cuál es su función.

```
CASE seleccion IS
WHEN “000” => z<=a; w<=b;
WHEN “001” => z<=b; w<=c;
WHEN “111” => z<=b; w<=k;
WHEN OTHERS => z<=”0000”; w<=”0000”;
END CASE;
```

CASE seleccion IS inicia la declaración de CASE, el identificador es seleccion.

WHEN “000” => z<=a; w<=b; indica que cuando seleccion=000 a z se le asigna a y w=b.

WHEN “001” => z<=b; w<=c; indica que cuando seleccion=001, a z se le asina b y a w se asigna c.

WHEN “111” => z<=b; w<=k; indica que cuando seleccion= 111, a z se asigna b y a w se asigna k.

WHEN OTHERS => z<=”00000000”; w<=”0000”; indica que, para los otras combinaciones de selección (cuando se utiliza CASE, se debe obligatoriamente probar todas las combinaciones de selección) a z se le asigna el valor 00000000 y a W el valor 0000.

END CASE; indica la terminación de la declaración de CASE.

Ejemplo 3.3.

Diseñe un multiplexor de cuatro a1 utilizando la declaración IF.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------------------------------------------------------
-----------
```

```
ENTITY mux IS
PORT ( a, b, c, d: IN STD_LOGIC;
                   select: IN STD_LOGIC_VECTOR( 1 DOWNTO 0);
                   y: OUT STD_LOGIC);
END mux;
-------------------------------------------------------------------------------
------------
 ARCHITECTURE amux_booleana OF mux IS
 BEGIN
PROCESS(select)
BEGIN
IF (select="00") THEN y<= a;
ELSIF (select="01") THEN y<= b;
ELSIF (select="10") THEN y<= c;
ELSE
    y<= d;
END IF;
END PROCESS;
END amux_booleana;
```

Ejemplo 3.4.

Diseñe el multiplexor de cuatro a uno utilizando la declaración IF. Las líneas de selección del mux deben ser declaradas como enteros.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------------------------------
-----------
ENTITY mux IS
```

```
PORT ( a, b, c, d: IN STD_LOGIC;
                   selec: IN INTEGER RANGE 0 TO 3;
                   y: OUT STD_LOGIC);
END mux;
-------------------------------------------------------------------------------
------------
 ARCHITECTURE amux_booleana OF mux IS
 BEGIN
PROCESS(select)
BEGIN
IF (select= 0) THEN y<= a;
ELSIF (select=1) THEN y<= b;
ELSIF (select=2) THEN y<= c;
ELSE
    y<= d;
END IF;
END PROCESS;
END amux_booleana;
```

La simulación con Quartus II del multiplexor de este ejemplo se muestra en la figura 3.1, y el a la figura 3.2 se muestra el circuito.

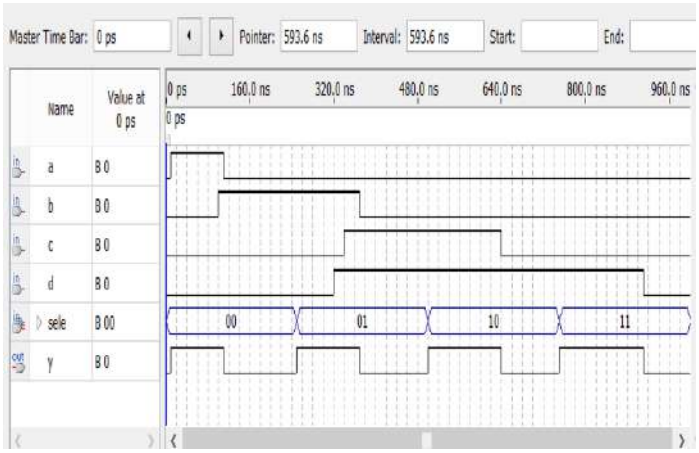


Figura 3.1. Simulación del multiplexor con Quartus II

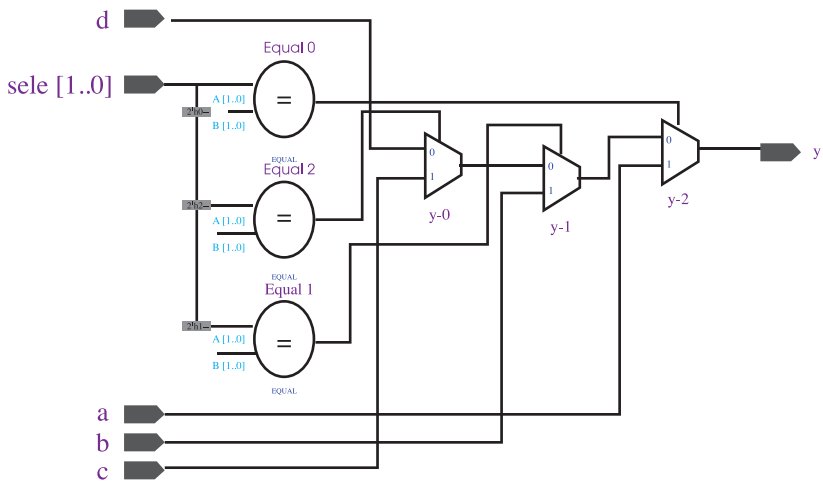


Figura 3.2. Circuito del multiplexor generado por Quartus II

Ejemplo 3.5.

Diseñe el multiplexor de 4 a 1 utilizando la declaración CASE y las líneas de selección del mux deben ser declaradas como enteros.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
-------------------------------------------------------------------------------------------------
-----------
ENTITY mux IS
PORT ( a, b, c, d: IN STD_LOGIC;
                    selec: IN INTEGER RANGE( 0 TO 3);
                    y: OUT STD_LOGIC);
END mux;
 -------------------------------------------------------------------------------------------------
------------
 ARCHITECTURE amux_booleana OF mux IS
 BEGIN
PROCESS (select)
BEGIN
CASE select IS
WHEN 0 => y<= a;
WHEN 1 => y<= b;
WHEN 2 => y<= c;
WHEN OTHERS => y<= d;
END CASE;
END PROCESS;
END amux_booleana;
```

Ejemplo 3.6.

Diseñe un *flip-flop* tipo D, que tiene una sola entrada D aparte del reloj (CLK); simule utilizando Quartus II e indique el circuito obtenido.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------------------------------------------------
-----------
```

```
ENTITY ff IS
PORT (D, CLK: IN STD_LOGIC;
            Q: OUT STD_LOGIC);
END ff;
-------------------------------------------------------------------------------
------------
 ARCHITECTURE aff OF ff IS
 BEGIN
PROCESS(CLK)
BEGIN
IF (CLK'event AND CLK='1') THEN Q<= D;
END IF;
END PROCESS;
END aff;
```

La figura 3.3 muestra la simulación del *flip-flop* tipo D en Quartus II.

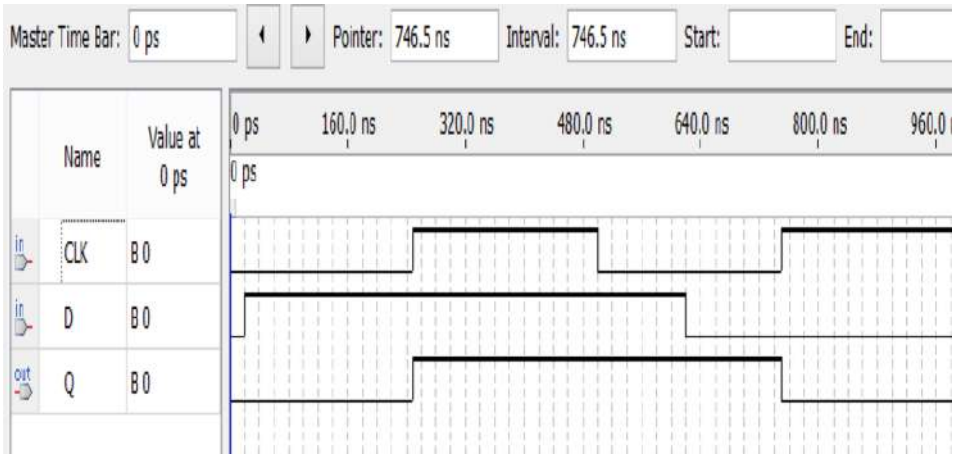


Figura 3.3. Simulación del flip-flop tipo D

La figura 3.4 muestra el circuito del *flip-flop* D generado por Quartus II

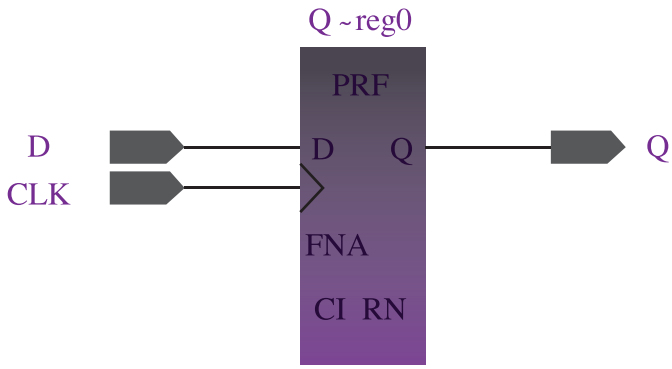


Figura 3.4. Circuito del *flip-flop*

Ejemplo 3.7

Utilizando CASE diseñe un *flip-flop* tipo D, que tiene las entradas D, CLK y una señal de RESET, cuando RESET=0 Q=D y cuando RESET=1 la salida Q=0. Simule utilizando Quartus II e indique el circuito obtenido.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------------------------------------------
-----------
ENTITY ff IS
PORT (D, CLK, RESET: IN STD_LOGIC;
                Q: OUT STD_LOGIC);
END ff;
 -------------------------------------------------------------------------------------------
------------
 ARCHITECTURE aff OF ff IS
 BEGIN
PROCESS(CLK, RESET)
BEGIN
CASE RESET IS
WHEN   ‘0’     => IF (CLK’event AND CLK= ‘1’)THEN Q<=D;
```

```
        END IF;
          WHEN OTHERS => NULL;
    END PROCESS;
    END aff;
```

La figura 3.5 muestra la simulación obtenida con Quartus II y la figura 3.6 muestra el circuito generado con la misma herramienta.

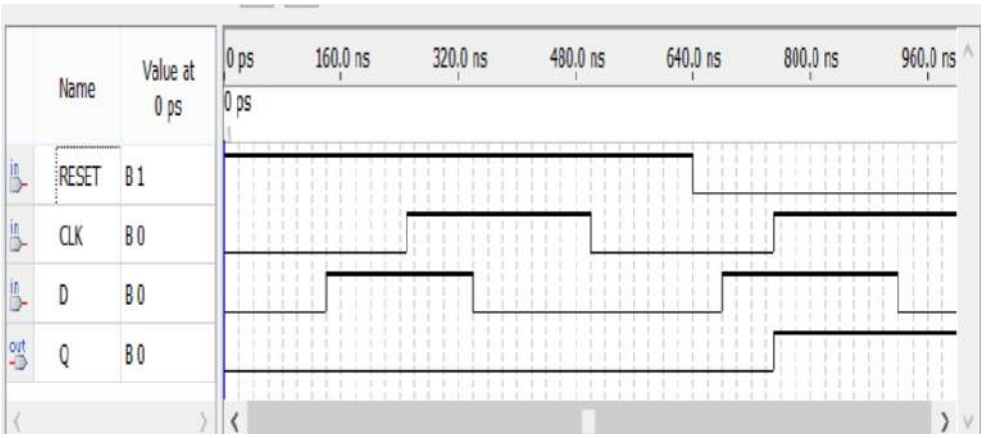


Figura 3.5. Simulación del *fli-flop* D.

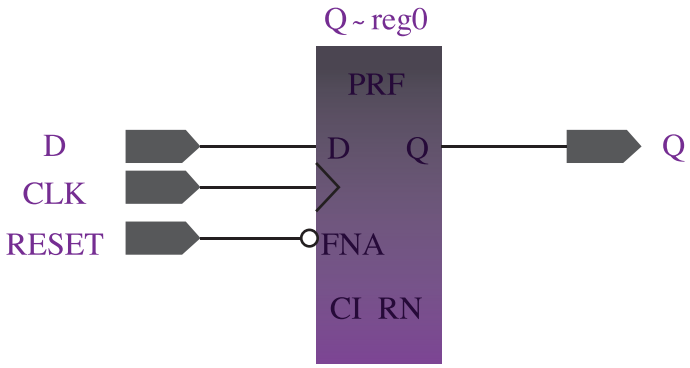


Figura 3.6. Circuito obtenido con Quartus II

3.6 Declaración WAIT

A diferencia de IF, CASE, LOOP, WAIT no requiere de un proceso con lista sensitiva. Su funcionamiento es similar a IF; existen tres formas de WAIT y su sintaxis es la siguiente.

WAIT UNTIL condición que una señal debe cumplir;

WAIT ON señal1, señal2, ….., señaln;

WAIT FOR tiempo;

WAIT UNTIL, espera hasta que se cumpla la condición en la señal que está a continuación de UNTIL para ejecutar el proceso.

Ejemplo 3.8.

En cada línea del código siguiente escriba un comentario que aclare lo que hace el programa.

PROCESS – inicia el proceso y no tiene una lista sensitiva

BEGIN – inicio del cuerpo del proceso.

WAIT UNTIL (clk'EVENT AND clk='1'); -- espera hasta que se produzca el flanco de subida de la señal de reloj (CLK).

IF (reset='0') THEN – pregunta si la señal reset=0.

Q <= '0; -- si reset=0, asigna 0 a Q.

ELSIF (clk'EVENT AND clk='1') THEN -- pregunta por el flanco de subida de la señal de reloj (CLK).

Q <= D; -- asigna a Q el valor de D.

END IF; -- termina el IF.

END PROCESS; -- termina el proceso.

Ejemplo 3.9.

Analice el siguiente código y explique.

```
LIBRARY IEEE;
    USE IEEE.std_logic_1164.all;
    USE IEEE.std_logic_unsigned.all;
ENTITY sumador_mal IS
    PORT
    (
            s1, s2   : in  std_logic_vector(3 downto 0);
            r        : out std_logic_vector (3 downto 0)
    );
END sumador_mal;
ARCHITECTURe a_sumador_mal of sumador_mal IS
BEGIN
    PROCESS( s1)
    BEGIN
    r<= s1+s2;
    END PROCESS;
END a_sumador_mal;
```

El programa (código) realiza la suma de dos números (s1, s2) de cuatro bits, sin acarreo de entrada. En la sección de declaración de librerías, se utiliza el paquete std_logic_unsigned que permite tener a los vectores s1 y s2 como números sin signo; si no se utiliza este paquete no se pueden sumar s1 y s2.

En la declaración de arquitectura, se utiliza un único proceso, la lista sensitiva está formada solo por s1. De acuerdo a como trabaja un *process*, este se ejecuta cada vez que cualquiera de los elementos de la lista sensitiva cambia, de acuerdo a esto, el proceso debe ejecutarse solo cuando s1 cambia; si s1 no cambia y solo s2 cambia, el proceso no debe ejecutarse.

Se utiliza Quartus II para simular este circuito y el resultado se muestra en a la figura 3.7.

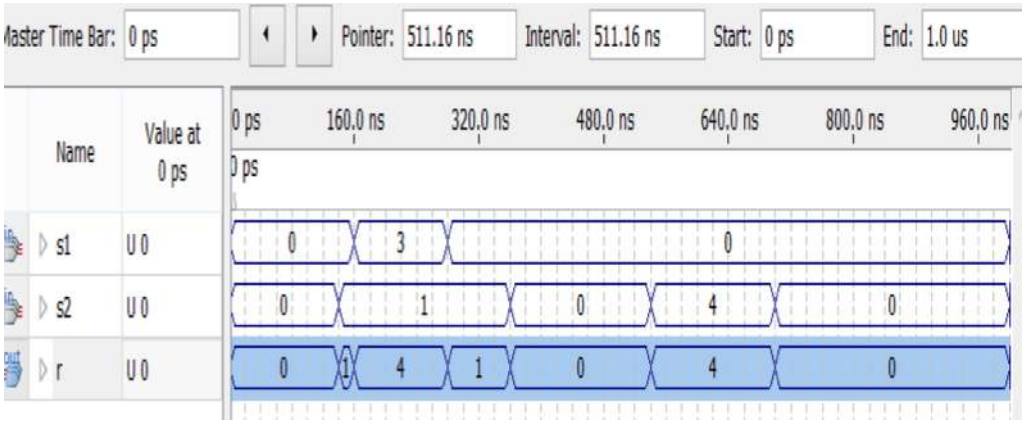


Figura 3.7. Resultado de la simulación del sumador en Quartus II

En la figura 3.7, se observa que en el intervalo en donde s1 cambia, el proceso se ejecuta, realizando la suma; sin embargo, hay un resultado extraño en el tercer intervalo en donde s1 no cambia, permanece en cero, y el proceso en ese intervalo no debería haberse ejecutado. El resultado de la suma debió haberse mantenido con el ultimo valor justo hasta cuando s1 cambió, es decir en uno, pero el proceso se ejecuta y suma los valores de s1 y s2.

Las advertencias que envía Quartus II son:

Warning (10492): VHDL Process Statement warning at sumador_mal.vhd(22): signal "s2" is read inside the Process Statement but isn't in the Process Statement's sensitivity list

En esta advertencia claramente se indica que s2 está dentro del proceso y que debería estar en la lista sensitiva.

Se realiza una nueva simulación, esta vez, se mantiene s1 en cero, sin ningún cambio, pero el proceso igual se ejecuta como muestra la figura 3.8.

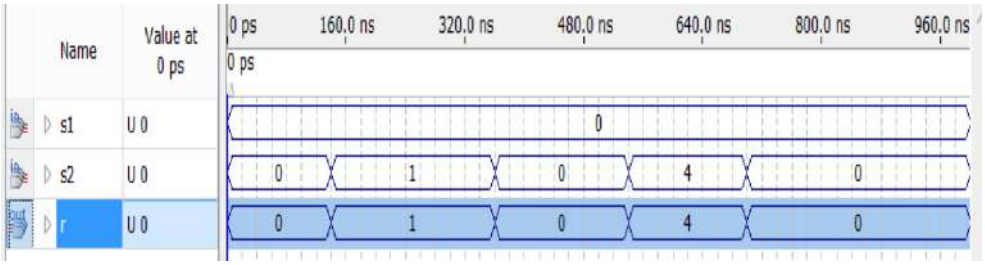


Figura 3.8. Resultado de la simulación con s1 sin cambiar

En la figura 3.9 se muestra el diagrama del sumador, tomado de Quartus a través de tool/ Netlist Viewers/ RTL Viewer. Como era de esperarse, el diagrama es de un sumador de cuatro bits.

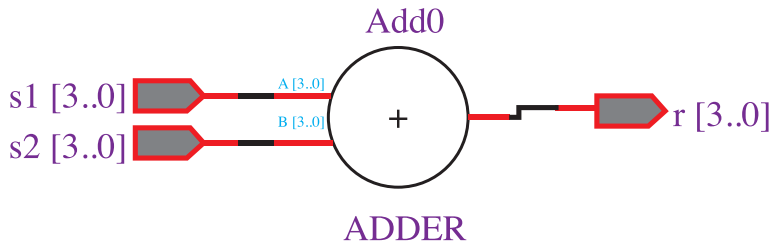


Figura 3.9. Diagrama del sumador

Ejemplo 3.10.

En el código del ejercicio 3.9, añada s2 a la lista sensitiva del *process* y explique.

El programa queda como se indica a continuación.

```
LIBRARY IEEE;
    USE IEEE.std_logic_1164.all;
    USE IEEE.std_logic_unsigned.all;
ENTITY sumador_mal is
    PORT
```

```
    (
            s1, s2   : in  std_logic_vector(3 downto 0);
            r        : out std_logic_vector (3 downto 0)
    );
END sumador_mal;
ARCHITECTURE a_sumador_mal of sumador_mal IS

BEGIN
    PROCESS( s1, s2)
    BEGIN
    r<= s1+s2;
    END PROCESS;
END a_sumador_mal;
```

El resultado de la simulación se muestra en la figura 3.10.

En la figura 3.11 se muestra el diagrama de este circuito tomado de Quartus II.

Figura 3.10. Respuesta de la simulación del sumador

Estos ejemplos muestran el resultado extraño de la simulación en donde aparentemente el proceso se ejecuta, cuando no debería haberse ejecutado, si la lista sensitiva no cambia.

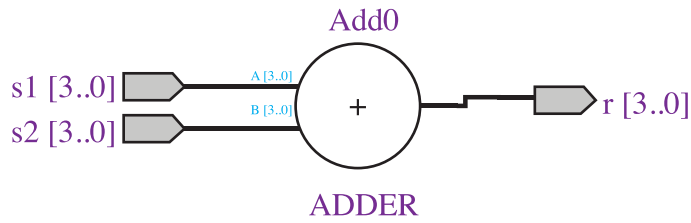


Figura 3.11. Diagrama del sumador

Estos dos ejemplos se simulan con ISE y los resultados son diferentes. El proceso no se ejecuta cuando el *process* tiene solo s1 en su lista sensitiva y s1 no cambia. Cuando se incluye a s2 en la lista sensitiva del *process*, el sumador suma cuando s1 y/o s2 cambian.

Ejemplo 3.11.

El código que se indica a continuación es el de un comparador de cuatro bits; tiene dos entradas s1 y s2 y una salida r. Simule con Quartus II y analice el resultado.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
ENTITY compara_mal is
    PORT
    ( s1, s2 : in  std_logic_vector(3 downto 0);
              r          : out std_logic);
END compara_mal;
ARCHITECTURE a_ compara_mal of compara_mal IS
BEGIN
    PROCESS (s1, s2)
    BEGIN
              IF s1=s2 then  r<='1';
```

```
        END IF;
    END PROCESS;
END a_ compara_mal;
```

El resultado de la simulación muestra la figura 3.12. Como se puede ver la salida del comparador es siempre uno independientemente de los valores de las entradas s1 y s2.

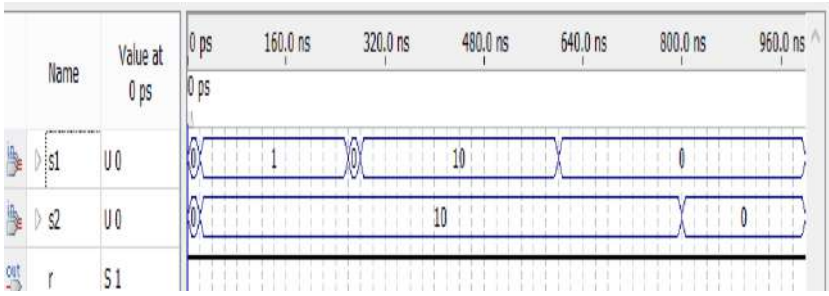


Figura 3.12. Resultado de la simulación con Quartus II

Para tener más clara la razón de este resultado, se revisa el gráfico del circuito proporcionado por Quartus II y es el que se indica en la figura 3.13.

Del análisis de la figura 3.13 se concluye que la salida efectivamente siempre está en uno y es una clara falla en el diseño del comparador.

Se revisa la codificación y se nota que, en la sentencia IF, se indica claramente cuál debe ser el resultado cuando s1 es igual a s2, pero no se indica qué valor debe tener r cuando s1 es diferente de s2, lo que es un claro error de diseño.

Figura 3.13. Grafico del circuito del comparador

La arquitectura debe tener además el valor de la salida r cuando s1 no es igual a s2, es decir:

```
IF s1= s2 THEN
r<= '1';
ELSE
r<='0';
END IF;
```

El código del comparador corregido se muestra a continuación:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
ENTITY compara_mal is
    PORT
    (
            s1, s2   : in  std_logic_vector(3 downto 0);
            r        : out std_logic
    );
END compara_mal;
ARCHITECTURE a_ compara_mal of compara_mal is
BEGIN
    PROCESS (s1, s2)
    BEGIN
            IF s1=s2 THEN
                    r<='1';
            ELSE
                    r<='0';
```

```
        END if;
    END PROCESs;
END a_ compara_mal;
```

El resultado de la simulación se muestra en la figura 3.14.

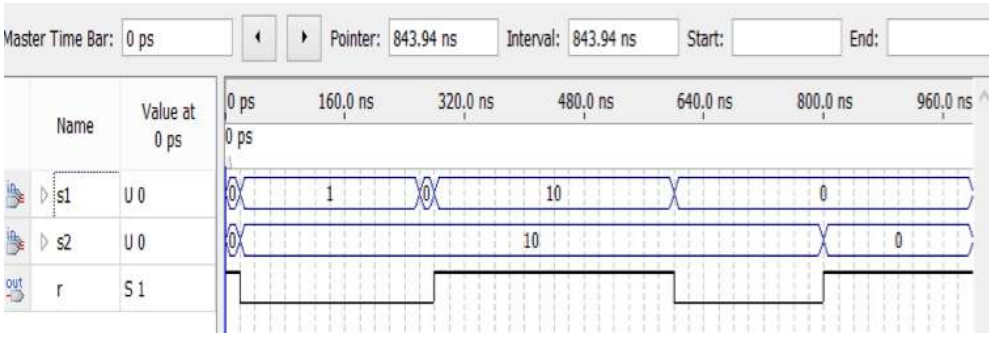


Figura 3.14. Resultado de la simulación del comparador

Como muestra esta figura 3.14 el gráfico del circuito comparador muestra el resultado para la salida r del comparador es la correcta, en conclusión, no se debe dejar el IF sin considerar el ELSE.

El IF debe tener el lazo completo.

El diagrama del circuito del comparador se muestra en la figura 3.16.

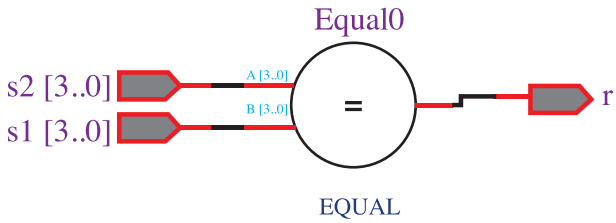


Figura 3.16. Gráfico del circuito comparador

Ejemplo 3.12.

El comparador del ejercicio anterior se ha modificado para que tenga tres salidas denominadas: IGUAL. MENOR, MAYOR. MENOR significa que s1 es menor que s2, MAYOR significa que s1 es mayor que s2 e IGUAL significa que s1 es igual a S2.

El código que se indica a continuación es el propuesto para el comparador. Analice y corrija si es necesario. Utilice Quartus II.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
ENTITY compara_mal IS
    PORT
    (
            s1, s2   : in  std_logic_vector(3 downto 0);
            IGUAL, MENOR, MAYOR   : out std_logic0);
END compara_mal;
ARCHITECTURE a_ compara_mal of compara_mal is
BEGIN
    PROCESS (s1, s2)
    BEGIN
    IF s1=s2 THEN
            IGUAL<='1';
    ELSIF s1<s2 THEN
            MENOR<='1';
    ELSE
            MAYOR<='1';
    END IF;
```

```
END PROCESS;
END a_ compara_mal;
```

Se simula el código y el resultado se muestra en la figura 3.17. Como se ve, la respuesta es que todas las salidas son iguales a uno independientemente de los valores que tengan las entradas. En el proceso de compilación no se observó ningún problema de sintaxis siendo la compilación exitosa.

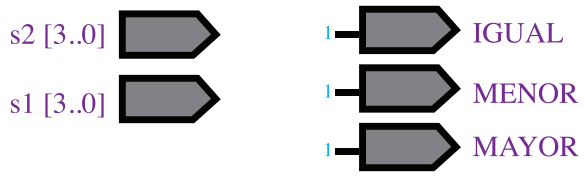


Figura 3.17. Resultado de la simulación del comparador

Como es buena estrategia cuando se tienen resultados extraños se recure al gráfico del circuito proporcionado por Quartus II. La figura 3.18 muestra este gráfico.

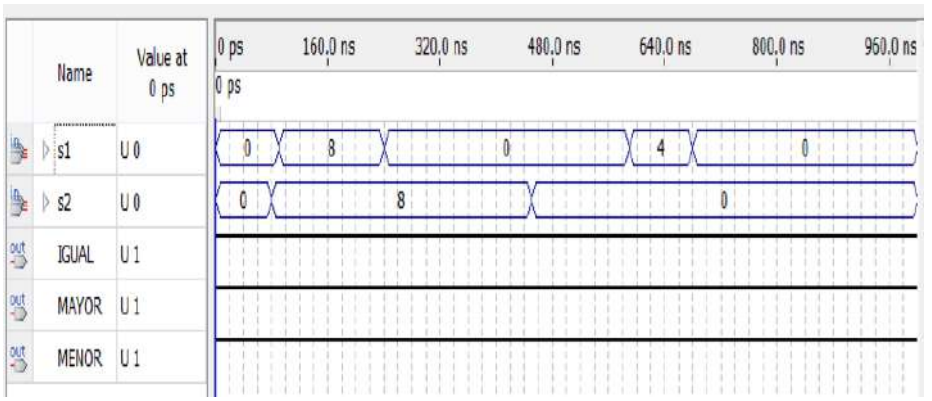


Figura 3.18. Gráfico del circuito comparador

Según el diagrama del circuito todas las salidas deben estar a un uno lógico y claramente se ve que hay una falla en el diseño.

El problema en el diseño de este comparador no es la simulación del lazo IF, que ya está cerrado, por lo tanto no es el mismo error que en el caso anterior. Sin embargo analizando las salidas se observa que: solo una salida a la vez se está encendiendo y no se dice que acción debe realizar el comparador con las otras salidas, por ejemplo se indica que si s1=s2 entonces la salida IGUAL debe ser igual a uno, pero con respecto a MAYOR Y MENOR no se indica que valores deben tener.

Así que el primer paso es modificar el código aumentando las líneas que son necesarias. Así, si s1=s2, entonces IGUAL=1 y además se debe indicar que MAYOR=0 y MENOR =0, se procede de igual manera para los otros casos. El código modificado se muestra a continuación.

```
LIBRARY IEEE;
    USE IEEE.std_logic_1164.all;

ENTITY SS IS

    PORT
    (
            s1, s2   : in  std_logic_vector(3 downto 0);
            IGUAL, MENOR, MAYOR   : out std_logic
    );
END SS;
ARCHITECTURE a_sumador_mal of SS IS
BEGIN
    PROCESS (s1, s2)
    BEGIN
    IF s1=s2 THEN
            igual<='1'; MAYOR<='0';
            MENOR<='0';
```

```
    ELSIF s1<s2 THEN
            MENOR<='1'; igual<='0';
            MAYOR<='0';
    ELSE
            MAYOR<='1'; igual<='0';
            MENOR<='0';
    END IF;
    END PROCESS;
END a_sumador_mal;
```

La figura 3.19 muestra el diagrama del este circuito obtenido de Quartus II. Debe notarse que cuando se declara exactamente cuál es el valor de las salidas el circuito sabe cómo actuar.

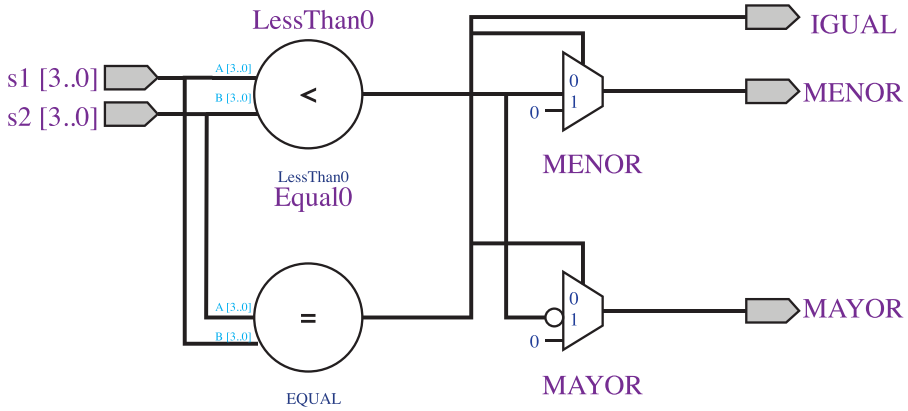


Figura 3.19. Circuito del comparador

La figura 3.20 muestra la simulación de este circuito. Como puede verse las señales de salida se encienden y apagan según como sean sus entradas, si son iguales o diferentes.

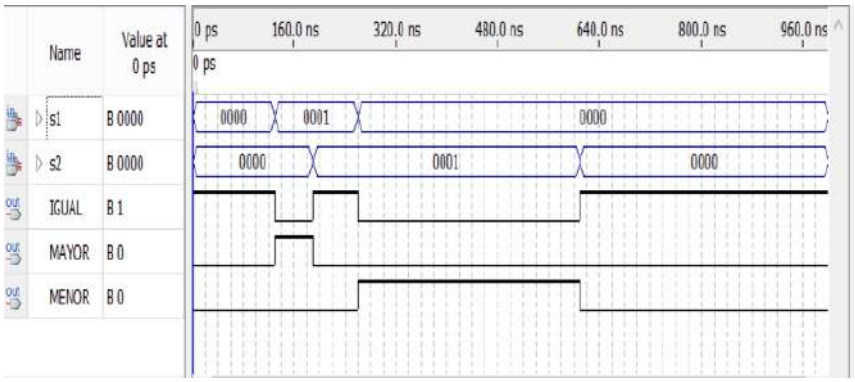


Figura 3.20. Resultado de la simulación del circuito

Ejemplo 3.13

Analice el siguiente programa, indique qué hace, simule con Quartus II y obtenga el circuito.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
--------------------------------------------------------------------------------------------
ENTITY mux IS
PORT ( a, b, c, d: IN STD_LOGIC;
        sele: IN INTEGER range (0 to 3);
                        y: OUT STD_LOGIC);
END mux;
 ------------------------------------------------------------------------------------------
 ARCHITECTURE amux_booleana OF mux IS
 BEGIN
PROCESS(sele)
BEGIN
```

```
IF (sele= 0) THEN y<= a;
ELSIF (sele=1) THEN y<= b;
ELSIF (sele=2) THEN y<= c;
ELSE
    y<= d;
END IF;
END PROCESS;
END amux_booleana;
```

Cuando se compila, el programa envía el siguiente aviso Quartus II: Error (10500): VHDL syntax error at mux.vhd(6) near text “)”; expecting “!”, or “=>”.

El programa es el mismo que el del ejercicio anterior, solo que, en la declaración de entidad se ha puesto entre paréntesis el rango. Quitados los paréntesis, la compilación se ejecuta sin problemas. La figura 3.21 muestra el gráfico del circuito.

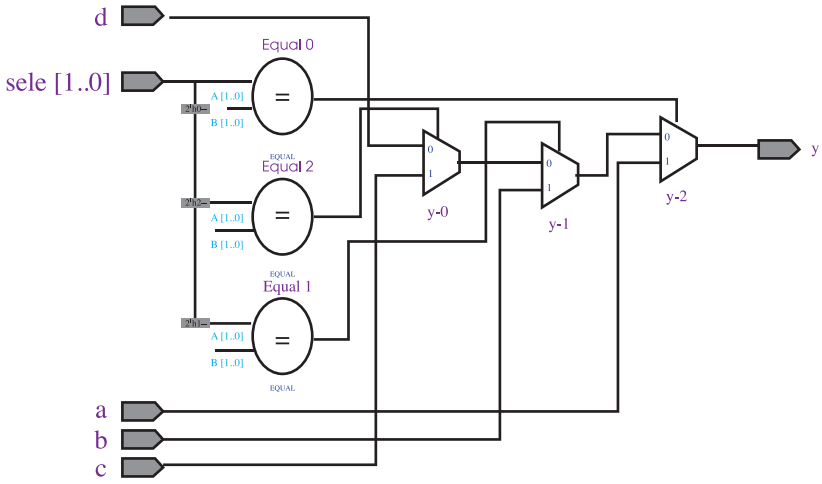


Figura 3.21. Diagrama del circuito

Ejemplo 3.14

Escriba y enumere las palabras reservadas de VHDL.

La siguiente es una lista de las palabras reservadas que utiliza VHDL para

construir sus unidades de diseño.

- Abs
- Access
- After
- Alias
- all
- and
- architecture
- array
- assert
- attribute
- begin
- block
- body
- buffer
- bus
- case
- component
- configuration
- constant
- disconnect
- downto
- else
- elsif
- end
- entit y
- exit
- file
- for
- function
- generate
- generic
- guarded
- if
- impure
- in

- inertial
- inout
- is
- label
- library
- linkage
- literal
- loop
- map
- mod
- nand
- new
- next
- nor
- not
- null
- of
- on
- open
- or
- others
- out
- package
- port
- postponed
- procedure
- process
- pure
- range
- record
- register
- reject
- rem
- report
- return
- rol

- ror
- select
- severity
- shared
- signal
- sla
- sll.
- sra..
- srl.
- subtype.
- then.
- to.
- transport..
- type.
- unaffected.
- units.
- until .
- use.
- variable.
- wait.
- when.
- while.
- with.
- xnor.
- xor.

Ejemplo 3.15

Con ejemplos escriba cómo en VHDL se escriben números, caracteres y tiras de caracteres (*strings*).

En VHDL los números pueden ser enteros, reales, naturales y se pueden escribir en diferentes bases, por ejemplo:

1. 2016 es un entero.

2. 2_016 es el mismo número entero anterior.

3. 56E7 es un entero.

4. 1.2 es un real.
5. 2.3E6 es un real.
6. 2#110011# es un número en base dos, porque inicia con: 2#, 2 identifica a la base dos y el símbolo # indica el inicio y fin del número.
7. 2#11_00_11# es el mismo número en base dos escrito en la línea anterior.
8. 16#4C# es un nuero expresado en base 16.
9. 'a' es el carácter a; los caracteres van encerrados entre apóstrofes.
10. 'B' es el carácter B.
11. '0' es el carácter 0.
12. 1 es el número 1.
13. 'o' es el carácter o.
14. 0 es el número cero.
15. "2015" es un *string*, los *string* van encerrados entre comillas.
16. "2#11_00_11#" es un *string*.
17. "VHDL" es un *string*.
18. "VHD_L" es un *string* diferente de: "VHDL".
19. "11110000" es un *string*.
20. "1111_0000" es un *string* diferente de: "11110000".

Ejemplo 3.16.

¿Qué es un objeto en VHDL y cuántos tipos de objetos hay y cuál de ellos no puede ser sintetizado?

Un objeto es el nombre de un elemento con que mantiene el valor de un tipo de dato específico.

Hay cuatro tipos de objetos: señal (*signal*), variable (*variable*), constante (*constant*), archivo (*file*).

El objeto que no puede ser sintetizado es el *file*.

Ejemplo 3.17

Indique tres diferencias entre una señal y una variable.

Una señal se le puede ver como un cable con memoria, algo parecido a un *latch*. El símbolo de asignación a una señal es <=.

Una variable es un valor y a ese valor por lo tanto no se le puede asociar una forma de onda. Su símbolo de asignaciones:= y puede ser vista como una localidad de memoria en donde se almacena temporalmente un valor. Una señal dentro de un proceso no se actualiza inmediatamente; en cambio, una señal fuera de un proceso se actualiza de inmediato.

Ejemplo 3.18

Explique que es un alias en VHDL. Indique un ejemplo.

Un alias, es un nombre alternativo para un objeto. El propósito es dar mayor claridad y facilidad de lectura del código.

Por ejemplo, se tiene un vector llamado trama con varios campos y se desea que cada campo quede plenamente identificado de acuerdo a la función que cumple dentro de la trama.Para conseguir este objetivo, a cada uno de los campos se puede hacer referencia con un alias, de tal manera que las partes del vector queden claramente reconocibles. En la tabla 3.4, se muestra la trama, cada campo de la trama y el número de bits que integran cada campo.

Trama							
Campo cabecera		Campo datos				Campo fin trama	
7	6	5	4	3	2	1	0

Tabla 3.4. El vector trama y sus campos

```
signal trama:  std_logi_vector( 7 downto 0);
alias campo_cabecera: std_logi_vector( 1downto 0) is trama (7downto 6);
alias campo_datos: std_logi_vector( 3 downto 0) is trama (5 downto 2);
alias campo_fin_trama: std_logi_vector( 1downto 0) is trama (1 downto 0);
```

Como se puede notar, los alias permiten identificar con claridad cada campo y el código es fácilmente entendible.

Ejemplo 3.19

Escriba la asignación W<=”10101010” de cuatro formas diferentes:

Las siguientes asignaciones para W son equivalentes:

```
W<=(‘1’, ‘0’, ‘1’, ‘0’,1’, ‘0’, 1’, ‘0’)
W<=(7 =>‘1’, 6=> ‘0’, 5=> ’1’,  4 =>’ 0’, 3 => ‘1’, 2=> ‘0’, 1=>1’, 0=> ‘0’);
W<= ( 7  5  3 =>’1’, 6 4 2 0 => ‘0’);
W<= ( 7  5  3 =>’1’, others => ‘0’);
```

Ejemplo 3.20

El siguiente dato de cuatro bits, 0100, explique de cuántas formas puede ser interpretado.

Podrían ser interpretados como un grupo de cuatro bits independientes, es decir, podrían ser los valores de cuatro señales.

Otra interpretación es que podría representar al número 12 en caso de que sea declarado como dato de tipo *unsigned* (sin signo).

Otra posible interpretación es que represente al +4 suponiendo que fue declarado como un dato de tipo *signed*, es decir, es un número que está en complemento a dos y el bit más a la izquierda es el bit de signo.

Ejemplo 3.21

¿Cuál es la diferencia y similitud entre los paquetes: numeric_ std y std_logic_arith.

Diferencias: std_logic_arith no es un estándar de lEEE, pero, numeric_ std si es un estándar del IEEE.

Similitudes, los dos paquetes tienen fines similares, se utilizan para realizar operaciones aritméticas.

std_logic_arith, al igual que los paquetes std_logic_unsigned y std_logic_unsigned son incluidos en algunas herramientas EDA de algunos vendedores y puestos en la librería IEEE.

Ejemplo 3.22

¿Es conveniente representar a una señal física como de tipo bit?.

No, porque solo tendría dos valores posibles cero o uno, y una señal puede tener otros valores (nueve valores) como alta impedancia.

Ejemplo 3.23

Para una señal de cuatro bits con dos operaciones simples, divida el valor de para cuatro (la división no debe tener residuo).

La señal puede ser considerada como un vector de cuatro bits, con cuatro bits, el número máximo que se puede representar es el 15, como la división debe ser exacta, entonces, las divisiones permitidas son entre los siguientes números: 12/4=3, 8/4=2 y 4/4=1. En binario, se tiene:

12 /4= 3; 1100/0100= 0011

8/4=2; 1000/0100= 0010

4/4=1; 0100/0100=0001

Como se puede ver, el resultado de la división siempre tiene los 2 bits más significativos del numerador que se convierten en los bits menos significativos

de la respuesta. Por lo tanto, hay que tomar los dos bits más significativos del numerador y añadir dos bit 00 en la posición más significativa de la respuesta.

Sea z el número y r la respuesta entonces: r<= “00” & z(3 downto 2) realiza la división exacta de un número de cuatro bits para cuatro.

3.7 EJERCICIOS PROPUESTOS

Formular las preguntas: ¿Qué?, ¿Por Qué? …Etc, utilizando signos de interrogación.

1. ¿Qué es un lenguaje de descripción de hardware?
2. ¿Cuál es el significado de las siglas VHDL?
3. ¿Por qué VHDL es un estándar?
4. ¿Cuál fue el origen de VHDL?
5. ¿Cuáles son las versiones de VHDL?
6. Defina una entidad.
7. Escriba la sintaxis para declarar una entidad.
8. Declare la entidad para una puerta AND.
9. Las declaraciones de entidad para una puerta AND y OR podrían ser exactamente las mismas. Explique.
10. Declare la entidad para un sumador medio de ocho bits.
11. Declare la entidad para un generador de paridad par de cuatro bits.
12. Declare la entidad para un decodificador de cuatro a 16.
13. ¿Qué es un identificador?
14. ¿Qué caracteres no puede tener el nombre de un identificador?
15. ¿Qué es un puerto?
16. ¿Qué es el modo y el tipo de un puerto?

17. Qué significado tiene la expresión: BIT_VECTOR(8 DOWNTO 0).
18. ¿Cuál es el significado de la siguiente expresión BIT_VECTOR(0 TO 8)?
19. ¿Cuál es la diferencia entre las expresiones: BIT_VECTOR(8 DOWNTO 0) y BIT_VECTOR(0 TO 8)?
20. Explique la utilidad de una librería en VHDL.
21. ¿Qué contiene una librería en su interior?
22. Escriba las dos líneas de código que son necesarias para hacer visible una librería.
23. Indique las librerías que no es necesarios declarar.
24. Indique las librerías que deben declararse obligatoriamente para poder utilizarlas.
25. Indique los paquetes que están dentro de la librería IEEE.
26. ¿Cuál es la diferencia entre los paquetes std_logic y std_ulogic?
27. ¿Para qué se puede utilizar el paquete std_logic_signed?
28. ¿Qué tipos de datos pueden ser definidos por el usuario?
29. ¿Qué son los subtipos de datos, escriba un ejemplo?
30. ¿Cómo puede identificar una señal dentro de un circuito?
31. ¿Cuál es la diferencia entre una señal y una variable?
32. ¿Qué es un operador?
33. Enumere los operadores de asignación.
34. Enumere los operadores lógicos.

35. Enumere los operadores aritméticos.

36. ¿Para qué se utilizan los atributos de una señal?

37. Explique el atributo ‘EVENT.

38. Explique el atributo ‘QUIET.

39. ¿Cuáles son las características de un circuito combinacional?

40. Explique la sintaxis de declaración de una arquitectura.

41. Declare la arquitectura de un sumador medio.

42. Cree una tabla de verdad de cuatro variables de entrada y dos salidas, e impleméntela con operadores.

43. La tabla del ejercicio anterior, impleméntela con la sentencia: WHEN – ELSE.

44. Diseñe un decodificador de cuatro a uno con la sentencia WHEN-ELSE.

45. Diseñe un multiplexor de 2 a 1 para datos de cuatro bits con la sentencia WHEN-ELSE

46. Simule los cuatro ejercicios anteriores con Quartus II e implemente en el entrenador DE2 de Altera.

47. Indique para que se utiliza la ventana *pin planner* de Quartus II.

48. Explique la sintaxis de la declaración: WITH-SELECT-WHEN.

49. Explique las características de un circuito secuencial.

50. Explique que es un proceso (PROCESS).

51. ¿Cuándo un PROCESS no requiere una lista sensitiva=?

52. Explique la sintaxis de IF-ELSE.

53. Explique la sintaxis de CASE.

54. Diseñe un *fli-flop* JK mediante IF-ELSE.

55. Dideñe un *flip-flop* tipo T con IF-ELSE.

56. Diseñe un multiplexor de cuatro a uno utilizando IF-ELSE.

57. Diseñe un contador de cuatro bits. El contador debe tener una señal de habilitación.

58. Diseñe un *buffer* de tres estados.

59. Diseñe un circuito secuencial que obtenga el complemento a dos de un número de cuatro bits. Los números ingresan uno a uno y con cada ciclo del reloj.

60. Diseñe un circuito secuencial sincrónico que controle dos motores. Los motores no deben estar ni encendidos ni apagados al mismo tiempo. Cada motor debe estar encendido el doble de tiempo que esté apagado.

61. Diseñe un generador de onda cuadrada. La frecuencia debe ser de 100 Khz y la amplitud de cinco voltios.

62. Diseñe un circuito secuencial sincrónico que genere el siguiente código: 0000-0011-0000-1100, en forma indefinida.

63. Diseñe un circuito secuencial sincrónico que detecte el código 111011, sin traslapamiento.

64. Diseñe un circuito secuencial sincrónico que detecte el código 111011, con traslapamiento.

BIBLIOGRAFÍA

Volnei, A. (2004). *Circuit Design and Simulation with VHDL*. Cambridg, Massachusetts*:* MIT Press.

Maxinez, J. Alcalá (2012) *VHDL el arte de programar sistemas digitales*. Mexico*:* CECSA.

Chu, P., (2006). *RTL hardware design using VHDL*. Nueva Jersey: John Wiley & Sons.

Brown, S., Vranesic, Z. (2006). *Fundamentos de Lógica Digital con Diseño VHDL*. México: Mc Graw Hill.

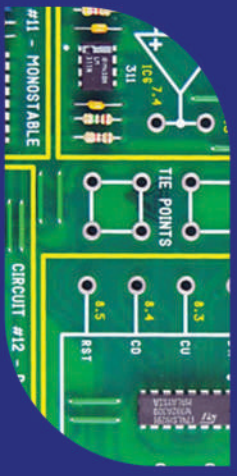
Este libro muestra los conceptos fundamentales VHDL. Existen infinidad de volúmenes sobre sistemas digitales, sin embargo, este libro se adapta a las necesidades académicas y de laboratorios de las carreras de Ingeniería Eléctrica de la Espoch, de ahí que uno de los objetivos es resolver estas necesidades.

Este libro presenta VHDL de tal forma que el lector pueda desarrollar habilidad intuitiva para entender y plicar los conceptos fundamentales de VHDL en el diseño de máquinas secuenciales sincrónicas.

Esta obra presenta los conceptos fundamentales de VHDL. Tomando en cuenta las complicaciones que presenta este lenguaje para estudiantes novatos, se estudian los constructores que son suficientes para que el estudiante pueda desarrollar proyectos sin dificultad. En forma gradual, se estudia el diseño de circuitos combinacionales y secuenciales, se presentan y resuelven ejemplos que permiten simultáneamente aprender y aplicar VHDL.

Wilson Oswaldo Baldeón López es ingeniero electrónico graduado en la Escuela Superior Politécnica del Litoral; es máster en Informática graduado en Chile; es máster en Diseño de Sistemas Electrónicos por la Universidad Tecnológica de La Habana, Cuba; es magíster en Gestión Académica Universitaria, tiene un diplomado superior en Pedagogía Universitaria y es experto en procesos *e-learning*.

Verónica Elizabeth Mora Chunllo es ingeniera en electrónica y computación graduada en la Escuela Superior Politécnica de Chimborazo; magíster en Ingeniería de Software graduada en la Escuela Superior Politécnica del Ejército; es máster en Diseño de Sistemas Electrónicos por la Universidad Tecnológica de La Habana, Cuba; es magíster en Educación a Distancia, tiene un diplomado superior en las Nuevas Tecnologías de Información y Comunicación Aplicadas a la Práctica Docente, es experta en procesos *e-learning*.



ISBN: 978-9942-35-649-9