UNIVERSIDAD POLITÉCNICA DE MADRID

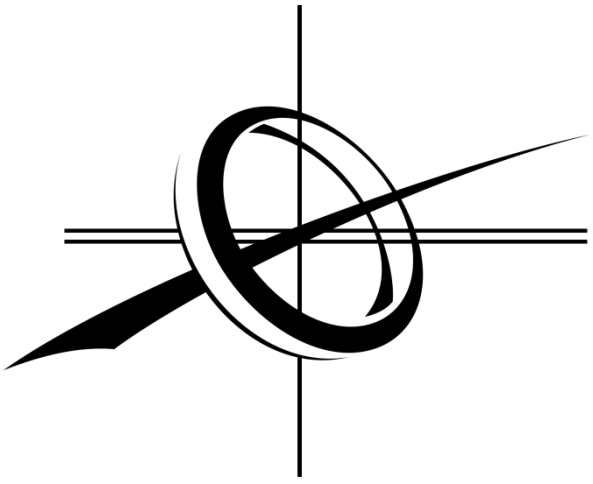
Escuela Técnica Superior
de Ingeniería de Sistemas Informáticos

Madrid

PROYECTO FIN DE GRADO EN INGENIERÍA DE COMPUTADORES

Curso académico 2016/17

Lenguajes de descripción hardware para la síntesis de circuitos: VHDL y Verilog. Analogías y diferencias. Aplicación a un caso práctico



Alumno:

Blanca Díaz Fernández

Tutor:

Virginia Peinado Bolós

Fecha: julio 2017

RESUMEN

Este proyecto tiene como finalidad comparar los lenguajes de descripción hardware (HDL) más populares. Estos lenguajes son VHDL y Verilog.

Para poder realizar una comparación lo más completa posible es necesario describir los diferentes lenguajes de los que se va a hablar. Antes de explicar cada lenguaje de forma particular, se explicará el tipo de lenguaje al que pertenecen.

Verilog y VHDL están englobados dentro de Lenguajes de Descripción Hardware (HDL). Un HDL es un lenguaje de programación especializado que se utiliza para definir la estructura, diseño y operación de circuitos electrónicos y electrónicos digitales. Así, estos lenguajes hacen posible una descripción formal de un circuito electrónico, y posibilitan su análisis automático y su simulación.

Posteriormente se describen los lenguajes a comparar. Ambos lenguajes son descritos en su totalidad, y es expuesta su metodología de diseño así como su metodología de programación, con el fin de que el lector pueda entender cualquier diseño descrito en estos lenguajes además de poder crearlos él mismo.

El lenguaje VHDL es un lenguaje de descripción de circuitos electrónicos digitales cuyo fin es la descripción de circuitos digitales o el modelado de fenómenos científicos. Al no ser un lenguaje de programación, para poder diseñar circuitos con él es necesario conocer su sintaxis y tener en cuenta una serie de cuestiones. Este lenguaje VHDL es un estándar de dominio público llamado IEEE 1076-1993. Al ser un estándar no depende de ningún fabricante o dispositivo, es independiente; esto también provoca que se puedan reutilizar los diseños; y por último, al ser un estándar tiene la ventaja de que es un diseño jerárquico, por lo que se mantiene un orden y se siguen ciertas reglas jerárquicas.

El lenguaje Verilog también es un lenguaje de descripción de circuitos digitales. Soporta el diseño, la prueba y la implementación de circuitos analógicos, digitales y de señales mixtas a diferentes niveles de abstracción. Al igual que VHDL, este lenguaje es un estándar de dominio público. Este lenguaje se diseñó basandose en el lenguaje de programación C, con el fin de que resultara familiar para los diseñadores y así fuese rápidamente aceptado. Por ello, Verilog tiene un preprocesador como C y la mayoría de sus palabras reservadas son similares a las de C.

El ultimo capitulo consta de la comparación de los lenguajes descritos anteriormente. La comparación se divide en dos aspectos:

- Comparación de las características de los lenguajes: se comparan las diferentes características de cada lenguaje mostrando las deficiencias y virtudes de los lenguajes.
- Comparación de los elementos del lenguaje: un mismo diseño será descrito en los dos lenguajes, permitiendo comparar la forma en la que se programan los lenguajes y sus diferentes elementos.

Después de todo ello, se llegará a la conclusión de que lenguaje es mejor, en caso de que lo haya.

ABSTRACT

The aim of this project is to compare the most popular hardware description languages (HDL). These languages are VHDL and Verilog.

In order to be able to realize the most complete comparison possible it is necessary to describe the different languages that are going to be introduced. Before explaining each language individually, it will be explained the kind of language they belong to.

Verilog and VHDL are grouped under the hardware description languages. An HDL is a specialized programming language that is used to define the structure, design and operation of electronic circuits and digital electronics. This way, these languages enable to a formal description of an electronic circuit and allow for their automatic analysis and simulation.

Afterwards the languages to compare are described. both languages are described in its entirety, being exposed their methodology of design and their methodology of programming for the purpose of the reader being able to understand any described design in these languages and being able to create them by himself.

The VHDL language is a description language of electronic digital circuits whose aim is the description of electronic circuits or modelling of scientific phenomena. Since it is not a programming language, in order to be able to design circuits using it, it is necessary to know its syntax and consider a series of issues. This language VHDL is a standard of public domain called IEEE 1076-1993. Being a standard means that it does not depend on any manufacturer or device, it is independent; this also results in the possibility of reuse these designs. Finally, being a standard has the advantage of being a hierarchic design. Consequently, an order is maintained and some hierarchic rules are followed.

The Verilog language is also a description language of electronic digital circuits. It supports the design, test and implementation of analogy, digital and mixed signals circuits at different levels of abstraction. As well as VHDL, this language is a standard of public domain. It was designed based on the programming language C in order to being familiar to designers and this way it was promptly accepted. For these reasons, Verilog has a similar pre-processor to c and the mainly of its reserved words are similar to C's reserved words.

The last chapter consist of the comparison between the languages previously described. This comparison is divided in two aspects:

- Comparison of characteristics of the languages: the different characteristics of each language are compared showing their weaknesses and strengths.
- Comparison of language elements: the same design will be described in both languages, allowing the comparison between the ways in which each language is programmes and their differences.

Finally and taking into consideration all the previous information, it will be concluded which language is better, in case any of them is.

Objetivo

El primer objetivo de este proyecto es la creación de un manual que contenga el estudio de los Lenguajes de Descripción Hardware más importantes, los cuales son: VHDL y Verilog.

En este manual se incluirán los diferentes elementos que componen los lenguajes, para así, poder implementar cualquier diseño en un dispositivo hardware.

El objetivo principal de este proyecto es, después del estudio de los lenguajes previamente mencionados, la comparación de los lenguajes con el fin de encontrar las diferencias y analogías entre los mismos. Esta comparación se realizará de forma teoría y de forma práctica.

Palabras clave

Leguajes de programación

HDL

Verilog

VHDL

Síntesis

Comparación

Diseño

Abstracción

Ejecución

Abreviaturas utilizadas

HDL: Hardware Description Language

ANSI: American National Standards Institute

IEEE: institute of Electrical and Electronics Engineers

VHDL: VHSIC Hardware Description Language

VHSIC: Very High Speed Integrated Circuit Hardware Description Language

STD: standard

VAL: Valor de

SUCC: valor siguiente a

PRED: valor anterior a

LEFTOF: valor a la izquierda de

RIGHTTOF: valor a la derecha de

ABS: valor Absoluto de un número

MOD: Módulo de dos números

REM: Resto de una división

ROL: Rotate Left

ROR: Rotate Right

SLA: Shift Left Arithmetic

SLL: Shift Left Logic

SRA: Shift Right Arithmetic

SRL: Shift Right Logic

RST: Reset

CLK: Clock

OVI: Open Verilog International

MSB: Most Significant Bit

LSB: Least Significant Bit

RTL: Register Transfer Language

DUT: Design Under Test

Índice

Capítulo I: HDL

1. Introducción

Un Lenguaje de Descripción Hardware (HDL) es un lenguaje de programación software utilizado para modelar el funcionamiento previsto de un circuito hardware. Existen dos aspectos de los HDL que facilitan las descripciones hardware:

- El modelado de comportamiento abstracto: un lenguaje es declarativo con el fin de facilitar la descripción del comportamiento hardware del circuito implementado. El diseño o los aspectos estructurales no perjudican al comportamiento hardware.
- Modelado de la estructura hardware: La estructura hardware puede ser modelada en un HDL independiente al HDL usado para modelar el comportamiento del diseño.

Este comportamiento puede ser modelado y representado en varios niveles de abstracción durante el proceso de diseño. Los modelos con altos niveles de abstracción describen el comportamiento del diseño de forma abstracta, mientras que los modelos con niveles bajos de abstracción incluyen más detalle.

Los lenguajes de descripción hardware son lenguajes especializados en la descripción de estructuras, el diseño de las mismas y el comportamiento del hardware.

Con estos lenguajes se pueden representar diagramas lógicos, circuitos con diferentes grados de complejidad, desde expresiones booleanas hasta circuitos más complejos. Estos lenguajes sirven para representar sistemas digitales de manera legible para las máquinas y para las personas.

2. Historia

[1] En los inicios de la electrónica integrada, allá por los años cincuenta y sesenta, cuando los circuitos eran analógicos, cuando el número de componentes que los formaban no era mayor de la centena. Ya a mediados de los sesenta Gordon E. Moore ya vaticinó el gran desarrollo de la tecnología en el que cada año la escala de integración se doblaría, de igual manera aumentó la capacidad de integrar funciones complejas además de mejorar la velocidad de proseado de esas funciones.

Los primeros circuitos integrados eran diseñados a partir del trazado directo de las máscaras y sin apenas ninguna comprobación antes de ser fabricados. Con la aparición de los primeros ordenadores se incorporaron complejos programas de resolución de ecuaciones diferenciales que permitían, alimentándolos con un modelo matemático del circuito, probar la funcionalidad de los circuitos antes de su fabricación. Este esquema estaba constituido por circuitos digitales. Paralelamente al desarrollo de esta tecnología se desarrollaron los editores de esquemas, creados para probar la funcionalidad de los circuitos sin necesidad de crearlos físicamente.

Sin embargo, en la década de 1970, la complejidad de los circuitos digitales aumentaba y las prestaciones de los editores de esquemas no eran suficientes para resolver una capacidad de diseño tan elevada. Esto provocó la necesidad de un lenguaje que permitiese a los diseñadores de circuitos un alto nivel de abstracción, debido a la creciente complejidad de los circuitos. Además, estos lenguajes permitieron incluir en los circuitos características propias de los circuitos eléctricos.

3. Metodología del diseño

[3] Es la manera en la que los procesos, respecto a la complejidad y la abstracción del diseño, se ordenan para conseguir que el costo y el tiempo de desarrollo sea el menor posible. Esto garantiza la confiabilidad del producto final y sus prestaciones además de lograr la mayor independencia de las herramientas y tecnologías.

La metodología que por lo general se usa para el diseño es:

- Definir el nivel de abstracción inicial.
- Realizar una descomposición jerárquica.
- Definir la estructuración de los nuevos niveles jerárquicos.
- Desarrollar la arquitectura necesaria.
- Seleccionar la tecnología que vamos a utilizar.

Existen dos diseños diferentes para orientar el orden de las acciones denominado flujo de diseño:

- Diseño Bottom -Up: Este diseño no es recomendado debido a que es ineficiente en diseños complejos y a que depende de la tecnología.
  Se empiezan describiendo los componentes más pequeños del sistema, estos componentes se agrupan en bloques de mayor complejidad y así sucesivamente hasta conseguir un único bloque.
- Diseño Top-Down: Este diseño es el más utilizado debido a que sus descripciones son independientes de la tecnología lo que aumenta la reutilización del diseño en diferentes casos.
  Se comienza con un sistema a nivel funcional, incluyendo la descripción y simulación de especificaciones. Se descomponen los diferentes sistemas en subsistemas y bloques hasta llegar a una descripción en componentes sintetizables.

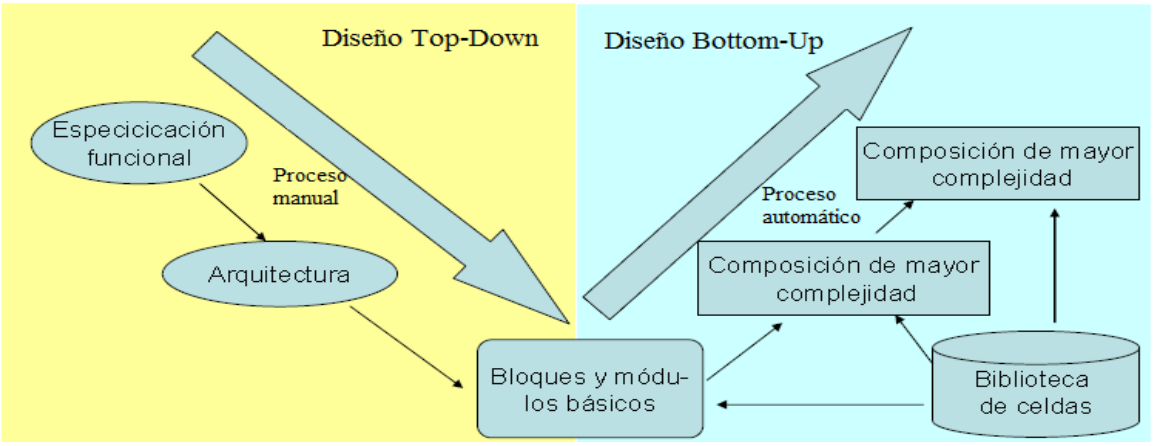


Ilustración 1.1 - Diseño Top-Down y Bottom-up

4. Fundamentos

[2] Estos lenguajes tienen una serie de fundamentos necesarios para entender la tecnología que vamos a explicar a continuación:

4.1 Niveles de abstracción

Los niveles de abstracción se refieren al grado de detalle de una descripción HDL respecto a su implementación física. Un diseño se puede representar bajo distintos punto de vista.

- Funcional: Relación entre entradas y salidas sin referencia a la implementación. Dada la posibilidad de que varios procesos puedan necesitar un mismo recurso, o la necesidad de sincronizar procesos independientes, se utilizan instrucciones similares a las de los lenguajes de programación concurrente.
  La precisión temporal son relaciones causales sin planificación temporal.

- Arquitectural: División en bloques funcionales, con planificación en el tiempo de las acciones a desarrollar. Se agrupa diferentes acciones en diferentes estados y se sincronizan por un reloj. En él se describe el sistema mediante diagramas de transferencia, tablas de verdad o ecuaciones lógicas

- Lógico: Poca precisión debido a retrasos de componentes y conexiones. Está formado por componentes de circuito expresados en ecuaciones o puertas lógicas y elementos de una biblioteca.

4.2 Estilos de descriptores

En los HDL se pueden diferenciar tres estilos de descripciones:

- Descripción estructural: Especifica los elementos que se pueden usar y sus interconexiones. Estos elementos pueden ser: procesadores, memorias, puertas lógicas, bloques funcionales o transistores.
- Descripción algorítmica: Describe el funcionamiento del circuito, componente o modulo. Es similar a la descripción de los programas

software. Estos descriptores pueden ser un sistema, algoritmos de comportamiento, transferencias de registros o ecuaciones lógicas.

- Descripción flujo de datos: Genera los datos necesarios para la realización física del módulo, estos datos reflejan el flujo de información y dependencia entre operaciones y datos. Este tipo de descriptor puede ser un grupo de bloques, particiones físicas, un trazado o un plano base.

5. Estructurados y no estructurados

Dentro de los lenguajes de descripción hardware encontrados dos clases diferentes, los HDL estructurados y los no estructurados.

- Los no estructurados son lenguajes sencillos orientados a la realización de un único circuito, entendiendo como circuito un único modulo o componente con una única función.
- Los lenguajes estructurados permiten la realización de circuitos complejos o de varios circuitos. Este tipo de lenguaje se caracteriza por:
  - Su capacidad multinivel, permitiendo así diferentes niveles de jerarquía
  - Su capacidad de combinar sus diferentes estilos de descripciones
  - Sus instrucciones, que definen una sintaxis independiente del nivel de descripción
  - Su independencia tecnológica, que le hace no depender de los fabricantes de los componentes físicos
  - Su universalidad, que garantiza la posibilidad de utilizar un gran número de herramientas de diseño
  - Su facilidad de comprensión y lectura que simplificar la documentación del circuito.

6. Herramientas asociadas

Estos lenguajes tienen asociadas dos herramientas:

- Síntesis: convierte una descripción de un sistema digital en una lista de componentes y sus interconexiones.
- Simulación: Predice la forma en la que se comportará el hardware antes de implementar el circuito. Permite detectar errores funcionales de diseño.

7. Ventajas

[11] Estos lenguajes son estándares ANSI e IEEE además de ser de dominio público, por lo que los diseños implementados en ellos son totalmente portables. Esto quiere decir que son independientes de la tecnología empleada y del fabricante. Gracias a ello, el intercambio entre los fabricantes y los usuarios es más fácil permitiendo a los usuarios usar una descripción de un componente en la simulación de sus diseños. Además estos modelos son reutilizables permitiendo ser sintetizados por diferentes librerías de distintos fabricantes.

También te permite verificar la funcionalidad del diseño muy temprano en el proceso de diseño sin necesidad de implementar el circuito. Esto se debe a que la simulación a un alto nivel permite probar la arquitectura y rectificar el diseño cuando el proyecto está en su fase inicial. Esto se traduce en un gran ahorro de tiempo cuando el diseño falla o no es el deseado, este ahorro de tiempo también se debe a la existencia de herramientas comerciales automáticas que permiten crear descriptores.

Estos lenguajes tienen diferentes formas de modelar el lenguaje. Como hemos visto anteriormente, soporta tres estilos de descripción: comportamental, flujo de datos y estructural, además de una combinación de estilos; también tiene diferentes métodos de diseño como hemos visto, top-down, bottom-up y una mezcla; y por último, cabe destacar que estos lenguajes soportan modelos síncronos y asíncronos. También permite implementar distintas técnicas de modelado digital

Las especificaciones del sistema a diseñar vienen definidas en la propia descripción del diseño y sirven como documentación del diseño. Estos modelos pueden ser verificados de forma fácil y precisa por simuladores. Además, estas especificaciones no son ambiguas, lo que es un requisito muy importante.

Este lenguaje se rige por jerarquías. Esto permite que el sistema pueda ser modelado como un conjunto de componentes interconectados y que a su vez, cada componente pueda ser modelado como un conjunto de subcomponentes.

En cuanto a las personas es más sencillo entender qué función realiza el diseño que un esquema de interconexión de puertas gracias a que la descripción de los HDL se centra en la funcionalidad.

En cuanto al testeo del diseño, se pueden escribir las diferentes pruebas a ejecutar en el mismo lenguaje que el de los diseños. Esta función permite un mejor manejo del modelo ya que permite asociarlo a sus estímulos de simulación; también permite que estos estímulos sean reutilizables en otros diseños y simuladores; además permite describir los retardos de programación y limitaciones temporales en el mismo lenguaje.

8. Inconvenientes

Un inconveniente cuando se quiere empezar a utilizar estos lenguajes es el aprendizaje que requiere ya que supone aprender una nueva metodología.

Para el uso de estos lenguajes es necesaria la adquisición de nuevas herramientas de simulación y de sintetización de HDL, teniendo que mantener el resto de herramientas para otras fases del diseño.

Usando este tipo de lenguajes se pierde un poco el control sobre el aspecto físico del diseño, dándole mayor importancia a la funcionalidad del diseño.

Como podemos comprobar, si comparamos la cantidad de ventajas y de inconvenientes que poseen estos lenguajes podemos observar que los inconvenientes son insignificantes comparado con la cantidad de ventajas que estos lenguajes ofrecen.

9. Aplicaciones

- Modelización: Procedimiento para la creación del diseño. Antes de expresar el diseño en algún HDL se crea un diseño en flujo de datos que nos ayuda a hacernos una idea de los componentes que necesitamos en el diseño.

- Simulación: Usan los HDL como entradas en las herramientas de simulación. La simulación se utiliza para verificar el correcto funcionamiento de los modelos en cualquier etapa del diseño, además la simulación permite corregir fallos de forma teórica, sin necesidad de que el circuito este implementado. A altos niveles de abstracción, representan la funcionalidad y son rápidos; a niveles bajos de abstracción descienden a mayor detalle y son lentos.

  El proceso de simulación comienza describiendo el circuito en algún HDL además de definir un conjunto de estímulos a los que se les aplicará la simulación. Posteriormente se procede a simular y para finalizar se estudia la respuesta obtenida por el simulador.

- Síntesis: Se usan los HDL como entrada en las herramientas de síntesis, obteniendo como salida el diseño del circuito, estas herramientas también transforman la descripción de un circuito de un formato a otro. En cada etapa del diseño se puede obtener una versión del circuito diferente, dependiendo del nivel de descripción utilizado.

- Reusabilidad: La capacidad que tiene el código que hemos creado en ser usado en otros programas diferentes.

10. Lenguajes

Existen gran cantidad de Lenguajes de Descripción Hardware, pero en la actualidad se utilizan mayoritariamente VHDL y Verilog. Los siguientes lenguajes más usados son SystemC, y UDI/L que se utiliza exclusivamente en Japón.

En este apartado vamos a explicar brevemente alguno de estos lenguajes:

- VHDL (Very High Speed Integrated Circuit Hardware Description Language). Desarrollado por el Departamento de Defensa de EEUU para modelar y simular sistemas, además de para poseer una herramienta estándar e independiente.

  El Instituto para la Ingeniería Eléctrica y Electrónica (IEEE) lo adopta y formula una norma que lo estandariza: norma IEEE 1076-1987.
- Verilog: Es un lenguaje de la firma Gateway y posteriormente propiedad de CANDENCE Design Systems. Este lenguaje fue estándar industrial hasta la llegada De VHDL como estándar IEEE. En 1990 CANDANCE lo hace público y en 1995 el IEEE lo estandariza.
- Nuevos lenguajes de alto nivel: A partir de la entrada en el nuevo milenio han nacido nuevos lenguajes muy similares a los lenguajes informáticos más usuales en ingeniería. Estos lenguajes tienen como objetivo facilitar la traslación de los lenguajes de diseño y pruebas a la descripción hardware, haciendo casi idénticos los lenguajes tanto a nivel de síntesis como el método de codificación. Los lenguajes más conocidos son SystemC y Handel-C.

En los siguientes apartados se hablará los LHD más utilizados hasta el momento: VHDL y Verilog. Se explicarán los diferentes lenguajes por separado con el fin de comprarlos al final del documento.

Capítulo II: VHDL

1. Introducción

DESCRIPCIÓN

[6] VHDL cuyo significado es VHSIC (Very High Speed Integrate Circuits) Hardware Description Language.

VHDL se trata de un lenguaje de descripción de circuitos electrónicos digitales, esta significa que mediante él se pueden describir los comportamientos de los circuitos. Este comportamiento puede transferirse a algún dispositivo con sus componentes propios, con ellos logrará el comportamiento deseado. El comportamiento el dispositivo es independiente al hardware del dispositivo.

VHDL no es un lenguaje de programación, por ello, para saber diseñar circuito con él, no solo se necesita conocer su sintaxis, sino que también es necesario tener una serie de cuestiones en cuenta:

Es necesario pensar con puertas lógicas y biestables, no en variables y funciones como en los lenguajes de programación. También es importante evitar los bucles combinacionales y los relojes que estén condicionados. Y la última cuestión a tener es cuenta es que es necesario saber diferenciar las partes del circuito en combinacionales y secuenciales.

Este leguaje permite una descripción de la estructura del circuito a partir de subcircuitos más sencillos, y también permite especificar la funcionalidad de un circuito usando un formato similar al de los lenguajes de programación, permitiendo a los usuarios adaptarse más fácilmente a este tipo de lenguaje.

El VHDL es un estándar de dominio público llamado IEEE 1076-1993. Al ser un estándar no depende de ningún fabricante o dispositivo, es independiente; esto también provoca que se puedan reutilizar los diseños; y por último, al ser un estándar tiene la ventaja de que es un diseño jerárquico, por lo que se mantiene un orden y se siguen ciertas reglas jerárquicas.

HISTORIA

[8] Debido al incremento de la complejidad de los circuitos digitales surgió la necesidad de los Lenguajes de Descripción Hardware. Por ello, en 1981 el Departamento de Defensa de los Estados Unidos desarrolló un proyecto llamado VHSIC para hacer el mantenimiento del hardware más sencillo y así, rentabilizar las inversiones en él. Con este proyecto pretendían hacer posible la reutilización de los diseño hardware, lo cual anteriormente no era posible porque no existía una herramienta adecuada para ello.

En 1983, las empresas IBM, Intermetrics y Texas Instruments fueron contratadas para desarrollar un lenguaje de diseño que permitiera la estandarización, facilitando la depuración de los algoritmos y el mantenimiento de los diseños. Por ello, el IEEE propuso su estandarización en 1984.

En 1987, tras varias versiones del lenguaje llevadas a cabo con la colaboración de las universidades y de la industria, se publicó el estándar IEEE std 1076-1987 que estableció el punto de partida decisivo para lo que después se conocería como VHDL. Las diferentes influencias, la de las universidades y la de las empresas contratadas provocó que el estándar fuera semejante a los lenguajes de descripción hardware que ya habían desarrollado previamente los fabricantes, de manera que éste quedó como ensamblado y por consiguiente un tanto limitado en su facilidad de utilización haciendo más difícil su total comprensión.

En 1993, el estándar fue revisado para mejorar sus deficiencias. Una deficiencia que no pudo ser mejorada fue la dificultad del uso del lenguaje y su compresión. Pero esta deficiencia se ve recompensada por la disponibilidad pública, y la seguridad que le otorga ser revisada y sometida a mantenimiento por el IEEE.

Gracias a la independencia en la metodología del diseño de la que goza el lenguaje, su versatilidad para la descripción de sistemas complejos, la posibilidad de reutilizar los diseños, su capacidad descriptiva y la independencia de los fabricantes han hecho que VHDL se haya convertido en uno de los lenguajes de descripción hardware más usados.

TIPOS DE DESCRIPCIÓN

[8] Como se ha mencionado previamente, este lenguaje sirve para describir circuitos electrónicos. Se puede describir un mismo circuito de diferentes formas, las diferentes opciones para describirlo son:

- Descripción de comportamiento

  Especifica cómo se comportan las salidas con respecto a las entradas. No proporciona información de cómo será el circuito, dejando al sintetizador determinarlo.

  Para entenderlo mejor veremos un ejemplo:

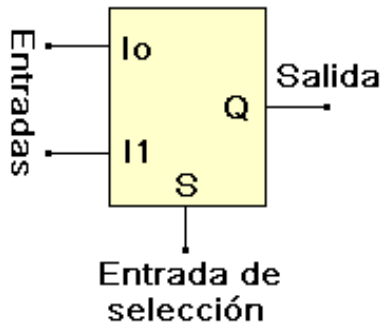


Ilustración 2-1. Multiplexor dos entradas

La descripción de comportamiento de un multiplexor de dos entradas y una salida y una entrada de selección sería: la salida será igual a la primera entrada (I0) si la entrada de selección esta desactivada; y la salida será la segunda entrada (I1) si la entrada de control esta activada.

La estructura básica de esta descripción es:

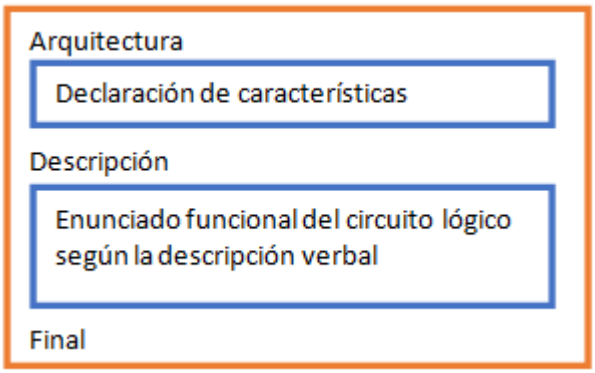


Ilustración 2-2. Estructura descripción de comportamiento

- Descripción de flujo de datos

  Especifica el circuito como un conjunto de expresiones booleanas concurrentes que describen como circula la información.

  La descripción de flujo de datos del mismo multiplexor que en el apartado anterior (dos entradas, una salida y una entrada de selección) tendría como salida una ecuación lógica tal que: Q = (I0 AND NOT S) OR (I1 AND S)

  Lo que quiere decir esta ecuación es que la salida será el resultado del valor de la primera entrada (I0) y el valor contrario de la entrada de selección, o por lo contrario, la salida será el resultado del valor de la segunda entrada (I1) y el valor de la entrada de selección.

  La estructura básica de la descripción de flujo de datos es:

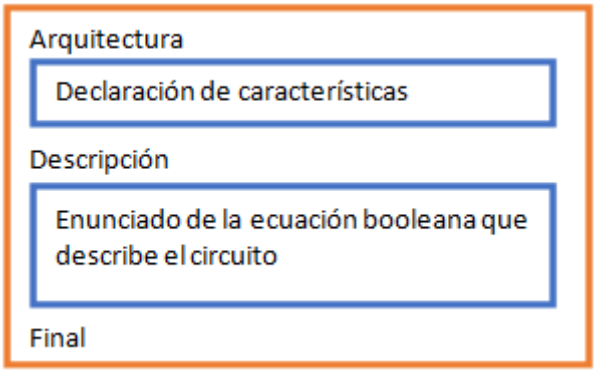


Ilustración 2-3. Estructura descripción de flujo de datos

- Descripción estructural

  Especifica el circuito describiendo las conexiones entre los distintos módulos y la descripción de estos módulos previamente en el apartado de arquitectura. Los módulos previamente descritos son puertas lógicas.

  Por lo tanto, si usamos el ejemplo del multiplexor anteriormente descrito, (dos entradas, una salida y una entrada de selección), la descripción estructural consistirá en explicar que el multiplexor tiene una puerta inversora (NOT), dos puertas AND de dos entradas y una puerta OR de dos entradas. También consiste en explicar sus conexiones:

- La entrada de la puerta inversora es S, y su salida se conecta con la segunda entrada de la segunda puerta AND.
- Las entradas de la primera puerta AND son la S y I0 y su salida se conecta con una entrada de la puerta OR.
- Las entradas de la segunda puerta AND son la salida de la puerta inversora (NOT) y la entrada I1, y su salida se conecta con la segunda entrada de la puerta OR.
- Las entradas de la puerta OR son las salidas de las dos puertas AND y su salida es la salida del multiplexor, es la salida final.

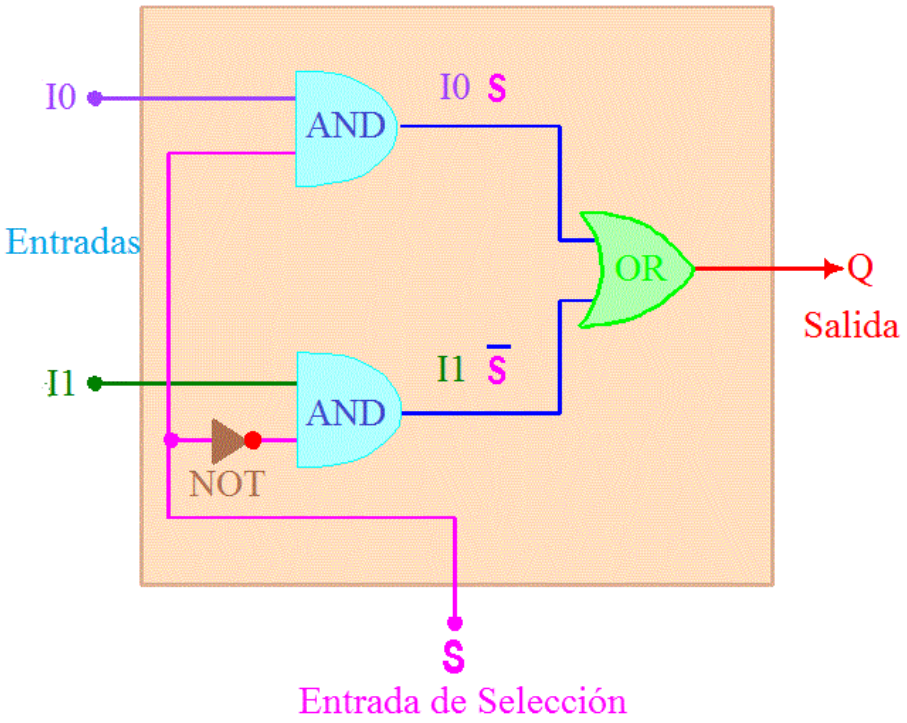


Ilustración 2-4. Esquema multiplexor de dos entradas con puertas AND

La estructura básica de la descripción estructural es:

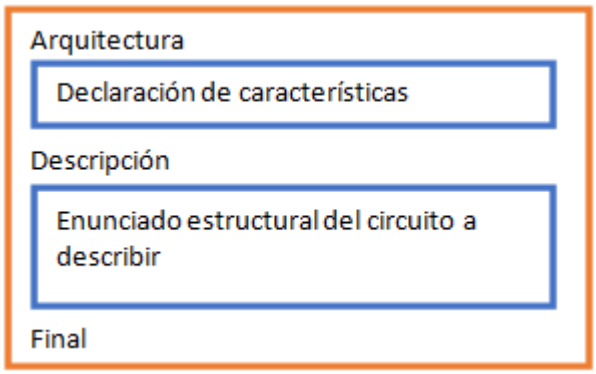


Ilustración 2-5. Estructura descripción estructural

2. Elementos básicos del lenguaje

[12] Antes de explicar la estructura que se debe seguir es necesario conocer algunos elementos básicos del lenguaje:

2.1 Los comentarios

Los cometarios se identifican porque van encabezados con dos guiones. Los guiones solo tienen efecto en su línea. Estas líneas de comentarios son ignoradas.

```
--  Esto es un comentario de ejemplo
```

2.2 Base numérica

Los números suelen representarse en base 10, aunque está permitido que se representen en otras bases numéricas utilizando diferentes símbolos para que el sintetizador los utilice correctamente.

Los números se pueden escribir en binario, octal, hexadecimal y decimal.

- Para números enteros y reales se escriben de la siguiente manera:
  - Binario: 2#1010#
  - Decimal: 11
  - Hexadecimal: 16#E#
- Para vectores de bits:
  - Binario: “010101”

- Octal: O#20#
- Hexadecimal: 16#FF#

2.3 Identificadores

Los identificadores sirven para identificar las variables, las señales, los procesos o cualquier dato que aparezca en el script. Los identificadores pueden ser cualquier cadena de caracteres con cualquier nombre compuesto por letras o números y letras, estos nombres pueden incluir el símbolo "_". Las minúsculas y las mayúsculas no se consideran diferentes, por lo que el nombre "EJEMPLO" y el nombre "ejemplo" representan el mismo elemento. No está permitido crear un identificador con una palabra reservada del lenguaje.

Existen tres clases de objetos definidos por defecto:

- *Constant:* estos objetos tienen un valor asignado previo a la simulación que no puede ser modificado durante la misma. La asignación de este valor es:

```
constant identificador: tipo:=valor;
```

- *Variable*: Estos objetos son un único valor que puede ser modificado durante la simulación. Este tipo de objeto se suele utilizar como índices, principalmente en bucles, o también se suelen usar para tomar valores que admitan modelar los componentes. Pueden ser determinadas en la parte de declaración de características, aunque posteriormente se modifiquen. Las variables **no** representan estados de memoria o conexiones.

```
variable identificador: tipo [:=valor]
```

La asignación de una variable se realiza con ":="

```
nombre_variable := valor o expresión;
```

- *Signal:* Estos objetos representan señales, que a su vez representan conexiones y elementos en memoria. A todos los elementos signal del código les corresponde un elemento de memoria o un cable en el circuito. Por lo que su comportamiento en la simulación será el mismo que el elemento físico asociado aunque no esté descrito en el código explícitamente. Deben ser declarados como una descripción de características.

```
signal identificador: tipo;
```

La asignación de una señal a un valor se realiza con "<="

```
Nombre_señal <= valor o expresión;
```

2.4 Tipos de datos

[10] En este lenguaje se pueden diferenciar dos tipos de datos: compuestos y escalares:

- Tipos de datos escalares:

  Son datos simples que contienen algún tipo de magnitud:

  - Enteros: Son valores numéricos enteros, se definen con la palabra RANGE. Esta definición lo que hace es acotar el rango limitando el valor del mismo.

    ```
    TYPE valor IS RANGE 0 TO 100;
    ```

  - Reales o con coma flotante: Se definen del mismo modo que los Enteros, con la única diferencia que los limitadores son números reales.

    ```
    TYPE valor IS RANGE 0.0 TO 100.9;
    ```

  - Enumerados: Son diferentes datos pertenecientes a una lista o conjunto finito. Este grupo de valores se define por el usuario.

    ```
    TYPE valor IS (perro, gato, pájaro);
    ```

  - Físicos: Son datos con un valor y unas unidades. Estos datos corresponden con magnitudes físicas (distancia, peso, temperatura, etc).

    ```
    TYPE valor IS RANGE 0 TO 10.0e7;
       UNITS
          unidad1;
          unidad2 = 60 unidad1;
          unidad3 = 5 unidad2;
          in=56 unidad2;
       END UNITS;
    ```

- Tipos compuestos:

Son tipos de datos compuestos por los elementos de otros tipos. Existen dos tipos compuestos: los arrays y los records.

- Arrays: Se trata de un conjunto de elementos del mismo tipo, accesible mediante un índice.

```
TYPE indice IS ARRAY (rango) OF tipo;
```

De este modo, también se pueden crear matrices, utilizando este tipo de dato:

```
TYPE indice IS ARRAY (rango) OF tipo;
TYPE indice1 IS ARRAY (rango1) OF tipo1;
```

- Record o Registro: Es un objeto de datos que consiste en una colección de datos de distintos tipos.

```
TYPE nombre IS RECORD
      element1: tipo1;
      element2: tipo2;
END RECORD
```

Para asignar un valor externo a un dato interno de una señal de tipo record se uso un punto. Una vez este definido el tipo y tenga asignado un nombre, se podrá definir cualquier señal. La asignación a una señal de tipo compuesto o escalar se hará utilizando el operador para señales "<=".

```
SIGNAL dato: nombre
dato.element1 <= "hola";
dato.element2 <= 23;
```

En este punto es necesario explicar algunas palabras reservadas del lenguaje que hemos utilizado o que podemos utilizar en este apartado:

- *TYPE:* es una palabra reservada para definir ese dato como un tipo de algo. Se acompaña con otras palabras reservadas a lo largo de la asignación para definir el tipo específico.

-*IS RANGE TO:* es una palabra reservada que se utiliza para determinar el rango. El valor que precede a la palabra TO es el primer elemento del rango, y el valor que le sucede es el último.

- *UNITS:* es una palabra reservada que da paso a la creación de diferentes unidades. Una vez esta unidades han sido creadas se cierra la sección con END UNITS.

- *IS ARRAY:* Esta palabra reservada junto con la palabra reservada TYPE crea un array con un índice, un rango de un tipo.

- El rango puede recorrerse de forma ascendente o descendente utilizando las palabras reservadas TO o DOWNTO, por lo que el rango (20 DOWNTO 0) hace lo mismo que el rango (0 TO 20).

- OTHERS: palabra reservada utilizada para realizar una acción determinada a un conjunto de datos determinados. Por ejemplo, asignar en todas las posiciones de un array el valor 0.
V <= (OTHERS => '0')

2.5 Subtipos de datos

Los subtipos de datos corresponden a tipos de datos ya existentes. Generalmente, se usan subtipos para que cada elemento tenga su propio tipo y no se sintetice erróneamente. Por ejemplo, el tipo *integer* es un bus de 32 líneas, si el elemento necesita muy pocas líneas se le crea un subtipo para acotar su longitud. Existen dos tipos dependiendo de las restricciones:

- Restricciones de un tipo escalar a un rango:

```
SUBTYPE indice IS character RANGE 'hola' TO 'luego';
```

- Restricciones del rango de un array:

```
SUBTYPE indice1 IS STRING (0 TO 7);
```

2.6 Conversión de datos

La conversión de datos, conocida como *casting*, resulta ser necearía en ocasiones para convertir unos tipos a otros.

Algunas conversiones son automáticas "**7 => 7.0**", otras conversiones necesitan realizarse de manera explícita, indicando el nombre del tipo de dato al que se quiere convertir el valor, seguido del dato a convertir entre paréntesis.

```
integer (7.6);
```

Otra forma de conversión es la transformación entre bits y números enteros, este tipo de conversión es muy utilizada en los diseños. La metodología de conversión es la misma explicada anteriormente, con la diferencia de que al pasar un entero a bits es necesario designar el número de bits de tamaño:

```
conv_integer (std_logic_vector); -- conversión vector a entero
conv_std_logic_vector (integer, n); -- de entero a vector
```

2.7 Atributos

Un atributo es una característica que puede ser atribuida a ciertos elementos del modelado VHDL.

Existen dos posibles atributos:

- Atributos predefinidos: Existen diferentes tipos de atributos predefinidos:
  - Atributos de tipos: permiten acceder a elementos de un tipo.

```
TYPE q IS ('2', '4', 'T', 'Y', '6');
SUBTYPE t IS q RANGE ('T' TO '6');
```

Sus atributos más comunes son:

Atributos	Descripción
(subtipo)'BASE	Base del tipo
(subtipo) 'LEFT	Límite izquierdo
(subtipo) 'RIGHT	Límite derecho
(subtipo) 'HIGH	Límite superior
(subtipo) 'LOW	Límite inferior
(subtipo) 'POS(v)	Posición de v
(subtipo) 'VAL(p)	Valor en la posición p
(subtipo) 'SUCC(v)	Valor siguiente a v
(subtipo) PRED(v)	Valor anterior a v
(subtipo) 'LEFTOF(v)	Valor a la izquierda de v
(subtipo) 'RIGHTOF(v)	Valor a la derecha de v

Tabla 2-1.Atributos de tipo más comunes VHDL

  - Atributos de array: permiten acceder al rango, longitud o extremos de un array.

```
SIGNAL d: STD_LOGIC_VECTOR (0 TO 50);
```

Sus atributos más comunes son:

Atributos	Descripción
(array) ‘LEFT	Límite izquierdo
(array) ‘RIGHT	Límite derecho
(array) ‘HIGH	Límite superior
(array) ‘LOW	Límite inferior
(array) ‘RANGE	rango
(array) ‘REVERSE_RANGE	Rango inverso
(array) ‘LENGTH	longitud

Tabla 2-2. Atributos de array más comunes VHDL

- Atributos de señales: permiten modelar características o propiedades de las señales. Sus atributos más comunes son:

```
señal’atributo
```

Atributo	Tipo	Descripción
S‘DELAYED(t)	Señal	Genera una señal exactamente igual, pero retrasada t
S‘STABLE(t)	Señal BOOLEAN	Vale TRUE si la señal no ha cambiado de valor durante t
S‘EVENT	Valor BOOLEAN	Vale TRUE si la señal ha tenido un evento
S‘LAST_EVENT	Valor TIME	Tiempo transcurrido desde el ultimo evento de la señal
S‘LAST_VALUE	valor	Valor de la señal antes del último evento
S‘QUIET(t)	Señal BOOLEAN	TRUE si la señal no ha recibido ningún cambio en el tiempo t
S‘ACTIVE	Valor BOOLEAN	TRUE si la señal a tenido algún cambio
S‘LAST_ACTIVE	Valor TIME	Tiempo desde el último cambio
S‘TRANSATION	Señal BIT	Cambiar de valor cada vez que cambia la señal
S’DRIVING	BOOLEAN	False, si el driver de S está desconectad
S’DRIVING_VALUE	señal	Valor actual de S

Tabla 2-3. Atributos de señal más comunes VHDL

2.8 Operadores

Un operador nos otorga la posibilidad de construir expresiones de diferentes tipos con las que se puede calcular datos utilizando diferentes señales. Existen diferentes tipos de operadores proporcionados por el lenguaje.

A continuación se mostrarán los tipos de operadores:

- Aritméticos:
  - Suma/ signo positivo (+): Se utiliza para indicar la suma entre un par de dígitos. Este operador también actúa como símbolo si está situado delante de una expresión.
  - Resta/ signo negativo (**-**): Al igual que el símbolo de suma, sirve para indicar la resta entre dos números. O si se posiciona delante de una expresión, modificará el signo de esa expresión.
  - Multiplicación (*): Multiplica dos dígitos de cualquier tipo (enteros, reales,…)
  - División (**/**): Divide dos números de cualquier tipo.
  - Exponencial (**): Eleva un número a una potencia. El número que se desea elevar puede ser de cualquier tipo, pero la potencia solo puede ser un número entero.
  - Valor absoluto (ABS()): Devuelve como resultado el valor absoluto del número introducido como su argumento. **`ABS(24)`**
  - Modulo (MOD): Calcula el modulo de dos números. **`n1 MOD n2`**
  - Resto (REM): Calcula el resto de una división entre dos números. **`n1 REM n2`**

- Desplazamientos: Estos operadores fueron incluidos en la última revisión del lenguaje en 1993.Solo son validos en los tipos bits. El uso de estos operadores es el siguiente:

  `dato_bits operador numero_de_veces`

  - ROL (ROtate Left): Rotación a la izquierda. Los huecos se van rellenando con los bits que van quedando fuera.
  - ROR (ROtate Right): Rotación a la derecha. Los huecos se van rellenando con los bits que van quedando fuera.

- SLA (Shift Left Arithmetic): Desplazamiento aritmético a la izquierda.
- SLL (Shift Left Logic): Desplazamiento lógico a la izquierda, rellenando el desplazamiento con ceros.
- SRA (Shift Right Arithmetic): Desplazamiento aritmético a la derecha.
- SRL (Shift Right Logic): Desplazamiento lógico a la derecha, rellenando el desplazamiento con ceros.

o Lógicos: Solo actúan con los tipos bit, bit_vector y boolean. Cuando se utilizan este tipo de operadores en vectores, la operación se realiza bit a bit. Los operadores lógicos son:

- NOT: Negación de un dato.

IN	NOT
0	1
1	0

Tabla 2-4. Tabla de verdad operador NOT VHDL

- AND: Es una multiplicación lógica.
- NAND: Es la operación AND invertida. NAND: AND y NOT

IN A	IN B	AND	NAND
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

Tabla 2-5. Tabla de verdad operadores AND y NAND VHDL

- OR: Es una suma lógica.
- NOR: Al igual que el operador NAND, este operador invierte la salida del operador OR.

IN A	IN B	OR	NOR
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

Tabla 2-6. Tabla de verdad de los operadores OR y NOR VHDL

- XOR: También llamada OR exclusiva. La salida de este operador es "1" cuando el numero de "1" 's en la entrada son impares, en el resto de las ocasiones, la salida es "0".
- NXOR: Como los demás operadores negados, este operador niega la salida del operador XOR.

IN A	IN B	XOR	NXOR
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Tabla 2-7. Tabla de verdad de los operadores XOR y NXOR VHDL

- Relaciones: Estos operadores solo devuelven valores booleanos:
  - Igualdad (**==, /=**): El operador == devuelve *true* si los valores son iguales y *false en* caso de que no lo sean. El operador /= indica desigualdad, funcionado con la lógica cambiada respecto al anterior.
  - Menos (**>, >=**): El primero es menor estricto, y el segundo es menor o igual. Los tipos de datos que pueden manejar son de tipo escalares o arrays.
  - Mayor (**<, <=**): El primero, al igual que los menores, es un símbolo de mayor estricto, y el segundo es un símbolo de mayor o igual. Al igual que los símbolos de menor, los datos que pueden manejar son de tipo escalares o arrays.

- Otros operadores importantes:
    - Concatenación (&): Concatena vectores, modificando la dimensión del array, siendo la dimensión resultante la suma de las dimensiones de los vectores con lo que se ha operado.

- Orden de preferencia: clasificados de mayor preferencia a menor:
    1. ** , ABS , NOT
    2. * , / , MOD , REM
    3. Signos + , -
    4. Operaciones + , - , &
    5. = , /= , < , <= , > , >=
    6. AND, NAND, OR, NOR, XOR, XNOR

3. Entidad

Una entidad es la abstracción de un circuito. Desde un sistema electrónico complejo compuesto de subcircuitos más simples, los cuales están compuestos por circuitos aun más simples hasta llegar a la simpleza de las puertas lógicas.

La entidad define únicamente la forma externa del circuito; en ella se describen las entradas y salidas del circuito. En esta descripción se incluyen los nombres de los diferentes puertos, su cantidad y también se especifican los tipos de datos de entrada y salida de cada uno.

Esta definición, del mismo modo, debe incluir toda la información necesaria para conectar el circuito descrito a otros posibles.

3.1 ENTITY

Una entidad es una función descriptiva donde se definen las entradas y salidas de un circuito en concreto.

Para definir una entidad se utiliza la palabra reservada ENTITY.

Debemos entender el concepto hardware de entidad como una caja cerrada a la que no se tiene acceso, en la cual solo se pueden conectar cables. La ventaja de ver la entidad de esta manera es que es más fácil entender la ejecución concurrente que ocurrirá en el hardware.

La descripción de cómo funciona la caja por dentro es la arquitectura. A continuación se muestra la sintaxis de la entidad:

```
ENTITY nombre_entidad IS
    [GENERIC ( parámetros en lista); ]
    [PORT ( puertos en lista); ]
END [ENTITY] nombre_entidad;
```

Para entender el correcto uso de la sintaxis, a continuación se muestra un ejemplo:

```
ENTITY nombre_entidad IS
    GENERIC (cte1: tipo:= valor1; cte2: tipo:= valor2;
                cte3: tipo := valor3; …);
  PORT (entrada1, entrada2, … : in tipo;
        salida1, salida2, …: out tipo;
        puertoi : modo tipo);
END nombre_entidad;
```

La palabra reservada **GENERIC** permite declarar las propiedades del modulo. Estas propiedades o constantes son semejantes a los parámetros en las funciones de otros lenguajes de programación. Estas constantes pueden ser inicializadas, en caso de que no se les introdujera un valor inicial tomarían valores por defecto. Para declarar una propiedad se debe indicar el nombre seguido de dos puntos y el tipo de dato del que se trata, opcionalmente se le asignará el valor inicial precedido por el operador de asignación **:=**. Las constantes se separarán con punto y coma.

```
Nombre : tipo := valor_inicial;
Const1 : integer := 2; const2 : integer := 44
```

La palabra reservada **PORT** define las entradas y salidas del modulo en definición.

La declaración de los puertos consiste en indicar el nombre de la señal seguido por dos puntos y posteriormente se indica la dirección del puerto, seguido de un espacio y del tipo de dato del que se trata el puerto. Los puertos se separarán de la misma manera que las constantes, mediante puntos y comas.

```
Nombre : dirección_puerto tipo;
Port1 : IN bit; port2 : OUT bit
```

Dentro de una entidad los puertos se consideran como señales, representando la función que harían los cables en un diseño hardware tradicional, es decir, transportando información y estableciendo conexiones. Existen varios tipos de puertos:

- Puerto **IN:** son señales de entrada, no se pueden modificar, solo pueden ser leídas. Por lo tanto, su funcionalidad es similar a las constantes.
- Puerto **OUT:** son señales de salida, estas señales no pueden ser utilizadas como constantes porque no pueden ser leídas, pero su valor puede ser modificado.
- Puerto **INOUT:** este tipo es una mezcla de los dos tipos anteriores. Pueden ser utilizados para lectura y para escritura, es decir que, les llega un valor ya determinado que puede ser utilizado para tomar decisiones, y a su vez, puede ser modificado.
- Puerto **BUFFER:** este tipo es idéntico al anterior, pero solo puede ser modificado por una fuente.

A continuación se muestra un ejemplo real de una entidad:

```
ENTITY mux IS
     GENERIC (cte1: integer:= 33;
              cte2: integer:= 2);
     PORT (entrada1 : IN bit;
           salida1 : out bit;
```

```
                puertoi : INOUT bit);
    END mux
```

4. Arquitectura

[8] La arquitectura es lo que define el modo en el que se comporta un circuito. Todas las arquitecturas van asociadas a una entidad. Existe la posibilidad de que una entidad tenga varias arquitecturas asociadas. En la primera instrucción de la descripción de una arquitectura aparecerá la palabra reservada **ARQUITECTURE** seguida del nombre que se le va a dar a esa arquitectura, seguida de la palabra OF y la entidad a la que se está asociando, finalizando con la palabra IS.

```
ARQUITECTURE comportamiento OF entidad IS
```

Después de esta primera línea de la definición pueden aparecer instrucciones declarando constantes, componentes, funciones, señales, etc. Las señales que se definen son internas, por lo que no se puede acceder a ellas desde la entidad, por lo tanto, los niveles superiores no podrán acceder a ellas.

Se observa que en estas declaraciones no se especifica si las señales son de entrada o salida, esto se debe a que son señales internas y pueden ser escritas y leídas sin restricciones.

Una vez se han definido todas las variables aparece la palabra reservada **BEGIN**, lo que da paso a la descripción del funcionamiento del circuito.

La estructura de una arquitectura es la siguiente:

```
ARQUITECTURE N_arq OF N_entidad IS
   SIGNAL int1, int2 : BIT; [declaraciones]
BEGIN
   Int1 <= cont; int2 <= e1 AND int1; [sentencias]
END ARQUITECTURE N_arq;
```

Las arquitecturas pueden ser descritas de tres maneras diferentes:

4.1 Descripción de comportamiento

Este tipo de descripción se caracteriza porque incorpora la programación en serie, la cual se define en bloques que se indican con la palabra reservada PROCESS. En una misma arquitectura puede haber varias descripciones de comportamiento, cada bloque equivalente a esta descripción se corresponderá con una instrucción concurrente. En otras palabras, la ejecución de las instrucciones internas de los bloques es en serie, pero entre ellos es concurrente.

Por lo tanto, una estructura secuencial va en el interior de los bloques.
La estructura de estos bloques es:

```
PROCESS [lista de sensibilidad]
  [declaración de variables]
BEGIN
   [sentencias secuenciales]
END PROCESS;
```

4.2 Descripción estructural

Permite realizar diseños jerárquicos. Este tipo de descripción se puede realizar mediante diferentes metodologías:

- Definición de componentes:
  Consiste en definir componentes dentro de un diseño, esto es posible con la palabra reservada COMPONENT. Un componente se corresponde con una entidad que ha sido declarada en otro modulo o en una biblioteca. La declaración de este componente se realiza en la parte declarativa de la arquitectura que se está desarrollando.
  La forma de declarar un componente es la siguiente:

```
COMPONENT NOMBRE [IS]
      [GENERIC (lista_params);]
      [PORT (lista_ports);]
 END COMPONENT NOMBRE;
```

- Referencia de componentes:

  Reside en copiar la arquitectura del componente que se quiere utilizar, tantas veces como sea necesario para crear el diseño.

  La sintaxis de referenciación de un componente es la siguiente:

```
REF_ID:
[COMPONENT] id_component | ENTITY id_entidad
  [(id_arquit)] | CONFIGURATION ID_config
[GENERIC MAP (params)]
[PORT MAP (ports)];
```

4.3 Descripción de flujo de datos

Muestra con detalle la transferencia de información entre las entradas y salidas de la entidad, es decir, se describe como la información es transmitida de señal a señal y de la entrada a la salida sin el uso de asignaciones secuenciales.

Esta descripción es muy útil para diseños lógicos sencillos, ya que las tablas de verdad y las ecuaciones lógicas son parte fundamental en la descripción de un circuito lógico.

No se debe olvidar que hay que describir un componente hardware, por lo que necesitamos tener en cuenta que un circuito electrónico puede tener muchos elementos ejecutándose al mismo tiempo.

Por lo que es necesario tener en cuenta que varios elementos pueden ejecutarse de forma paralela. Esto es lo que se denomina concurrentes, por lo que es necesario utilizar sentencias concurrentes para definir los diseños lógicos. En estas sentencias se le asigna a la señal de salida un determinado valor dependiendo de las entradas obtenidas siguiendo la tabla de verdad del diseño.

Otra manera de implementar un diseño en flujo de datos es utilizando la ecuación lógica del diseño, con la cual se obtienen todos los datos de la tabla de verdad.

A continuación se muestra el comportamiento de una puerta lógica AND. Esta puerta lógica solo obtiene como salida ‘1’ si ambas entradas son ‘1’. Se puede observar en su tabla de verdad:

IN A	IN B	AND
0	0	0
0	1	0
1	0	0
1	1	1

Tabla 8. Tabla de verdad puerta AND

Este comportamiento también se puede obtener de su ecuación lógica:

$$S = A*B$$

Por lo tanto, para hacer una descripción de flujo de datos de esta puerta lógica se pueden utilizar cualquiera de los dos métodos presentados. A continuación se muestra el código utilizando la tabla de verdad:

```
. . .
ARCHITECTURE FLUJO_TABLA OF TABLA IS
   BEGIN
      S <= '1' when (A= '1') and (B= '1') else
           '0' when (A= '0') or (B = '0') else
           '0';
   END ARCHITECTURE FLUJO_TABLA;
```

Posteriormente vamos a realizar la misma operación pero en esta ocasión utilizando su ecuación lógica:

```
. . .
ARCHITECTURE FLUJO_TABLA OF TABLA IS
   BEGIN
      S <= (A AND B);
   END ARCHITECTURE FLUJO_TABLA;
```

5. Estructura básica

[9] Como hemos visto en los apartados anteriores los diseños en VHDL tienen dos partes: la entidad y la arquitectura; en esta última se describe el comportamiento del circuito, a este modelo de programación se le denomina *behavioral*.

Dentro de la arquitectura se pueden encontrar:

- Sentencias de asignación y sentencias concurrentes: deben realizarse obligatoriamente.
- Señales y tipos intermedios: necesarios para la descripción del comportamiento del circuito o subcircuito.
- Uno o varios PROCESS: contienen sentencias condicionales y/o asignaciones a señales, dando lugar a hardware secuencial.

5.1 Sentencias concurrentes

Las sentencias concurrentes son sentencias condicionales que tienen al menos un valor para cuando no se cumple ninguna condición establecida. Las sentencias particulares que los desarrolladores escogieron para este caso fueron:

- WHEN – ELSE: Se le asignan ciertos valores a una señal dependiendo de la condición y se finaliza la sentencia con ELSE "valor", para el caso de que el dato introducido en la condición no haya cumplido ningún requisito.

  Las condiciones pueden actuar sobre distintas señales, además, la colocación de las condiciones indica la preferencia sobre las otras, la primera condición tiene preferencia sobre el resto, y así sucesivamente.

  Su sintaxis:

```
Señal <= valor1 WHEN condiccion1 ELSE
         Valor2 WHEN condiccion2 ELSE
         ...
         Valor_por_defecto;
```

En caso de que no se dé ninguna condición y se quiera dejar el valor introducido intacto se usará la palabra reservada UNAFFECTED:

```
Señal <= valor1 WHEN condiccion1
       ELSE UNAFFECTED;
```

- WITH – SELECT – WHEN: Esta sentencia es menos general que la anterior, aunque se utiliza más en síntesis.
  La asignación se hace según el contenido de una señal, modificando el valor de una señal dependiendo de los valores de una señal condición, dando un número limitado de posibilidades, los valores posibles de la señal condición.
  Su sintaxis es:

```
WHITH señal_cond SELECT
Señal <= valor1 WHEN condiccion_valor1,
          Valor2 WHEN condiccion_valor2,
          ...
          Valor_por_defecto WHEN OTHERS;
```

- BLOCK: En ocasiones es conveniente agrupar un conjunto de sentencias en bloques. Esto da la posibilidad de dividir el sistema en módulos, para finalmente, usar estos módulos como componentes para otros módulos.
  El nombre del bloque y la expresión de guardia no son obligatorios.
  La estructura de esta sentencia es la siguiente:

```
blockID: BLOCK (expresión de guardia)
    declaraciones
BEGIN
      Sentencias concurrentes
END BLOCK blockID
```

5.2 Sentencias PROCESS

Un PROCESS es una sentencia concurrente, porque a la hora de su ejecución, todos los PROCESS y las demás sentencias se ejecutarán sin un orden establecido. Sin embargo las sentencias que pertenecen a los PROCESS se ejecutan de forma ordenada, son sentencias secuenciales.
Esta estructura define los limites de un dominio que se ejecutará, si y solo si alguna señal de su lista de sensibilidad se ha modificado en el paso previo de simulación. Por lo tanto, se podría decir que los PROCESS solo se ejecutan si su lista de sensibilidad cambia.

Esta sentencia es una de las más utilizadas en este lenguaje ya que las sentencias condicionales y la descripción secuencial del hardware se realiza dentro de él.

La estructura de PROCESS es la siguiente:

```
PROCESS [lista de sensibilidad]
   [declaración de variables]
PORT
  [sentencias secuenciales]
END PROCESS;
```

5.3 Sentencia secuenciales

Las sentencias secuenciales deben ir obligatoriamente dentro de un PROCESS. Las sentencias más comunes son:

- IF – THEN –ELSE: Esta sentencia admite cualquier tipo de combinación y encadenamiento. Permite la ejecución de un bloque dependiendo de una o varias condiciones. Su sintaxis es:

```
IF <condición1> THEN
        [sentencias 1]
ELSIF <condición2> THEN
        [sentencias 2]
```

```
ELSE
        [sentencias N]
END IF;
```

- LOOP: Es la forma de hacer bucles en VHDL Y admite cualquier tipo de sentencias en su interior. Su estructura es:

```
[etiqueta:] LOOP
    [sentencias]
END LOOP [etiqueta];
```

- WHILE – LOOP: Es una variación de la sentencia LOOP, pero con algún tipo de condición. Sigue la misma mecánica que en otros lenguajes, mientras se cumple una condición especificada, se realiza el bucle. Su estructura es la siguiente:

```
[etiqueta:] WHILE Condición LOOP
    [sentencias]
END LOOP [etiqueta];
```

- FOR – LOOP: Es otra variación de la sentencia LOOP. Se realiza una acción la cantidad de veces que el rango ha establecido. Admite cualquier tipo de sentencia en su interior. Su estructura es:

```
[etiqueta:] FOR Valor IN [RANGE] LOOP
    [sentencias]
END LOOP [etiqueta];
```

- CASE: Es similar a la sentencia IF…THEN…ELSE pero en este caso se evalúa una expresión en vez de una condición. Es necesario tener en cuenta todos los casos posibles, es decir, incluir la opción WHEN OTHERS para las posibilidades no contempladas. Su estructura es:

```
CASE <expresión> IS
    WHEN  < Valor1 > => [sentencias 1]
    WHEN  < Valor2 > => [sentencias 2]
```

```
        WHEN < Rango_valores > => [sentencias N]
        WHEN OTHERS  => [sentencias M]
    END CASE;
```

- ASSERT: Se usa para dar avisos si se cumple la condición establecida. Su sintaxis es:

```
ASSERT <Condición>
      [REPORT  <Expresión>];
      [SEVERITY   <Expresión>];
```

- NEXT: Permite detener la ejecución actual y continuar con la siguiente, su estructura es:

```
[id_next:]
NEXT  [ID_bucle] [WHEN   Condición];
```

- EXIT: Provoca que se salga del bucle superior al que se ejecuta, su estructura es:

```
[id_next:]
EXIT  [ID_bucle] [WHEN   Condición];
```

- WAIT: Produce una parada en la ejecución hasta que se cumpla la condición, la lista sensible o el tiempo. Solo se usa en los bloques PROCESS. Si se utiliza una lista de sensibilidad no se puede usar la sentencia WAIT, pero se pueden usar varias sentencias WAIT en el mismo bloque. Su estructura es:

```
WAIT  Lista_sensible UNTIL Condición FOR Time_out;
```

La lista sensible es una o más señales separas por comas; la condición activará la ejecución después de la parada; y el time_out es el tiempo durante el cual la ejecución esta parada.

No es necesario utilizar las tres opciones a la vez, se pueden usar por separado. En el caso de que se use más de una opción la ejecución se reanudará cuando se cumpla una de ellas.

6. Organización

Debido a la complejidad de algunos sistemas se necesita organizar el código de tal manera que se pueda trabajar con mayor comodidad. Existen varias formas de organizar el código, una de ellas se ha visto anteriormente, los bloques; otra de las maneras de organización son los subprogramas. Estos subprogramas harán más legibles las descripciones de los sistemas. Además pueden ser agrupados junto a diferentes tipos de definiciones en estructuras, lo que formarían los paquetes que definiremos en este punto. A su vez, si se juntas estos paquetes con elementos de configuración se formarían las denominadas librerías, que se describirán en este punto.

6.1 Subprogramas

Al igual que en otros lenguajes de programación, es posible estructurar el programa mediante subprogramas. Los subprogramas son, en realidad, funciones o procedimientos que realizan alguna tarea en concreto.

6.1.1 Funciones y procedimientos

Es posible realizar las declaraciones de estos elementos en las parte declarativas de cualquier estructura expuesta anteriormente. A continuación se muestra la estructura de las funciones:

```
[PURE | IMPURE]
FUNCTION NOMBRE [(parámetros)] RETURN tipo IS
      [declaraciones]
BEGIN
      [sentencias]
```

```
END [FUNCTION] [NOMBRE];
```

Por otro lado la estructura de los procedimientos es:

```
PROCEDURE NOMBRE [(parámetros)] IS
      [declaraciones]
BEGIN
      [sentencias]
END [PROCEDURE] [NOMBRE];
```

Dependiendo del elemento utilizado los parámetros tendrán un significado u otro y unas restricciones u otras. En el caso de las funciones solo es posible utilizar el tipo de puerto IN (de entrada); además estos parámetros pueden ser CONSTANT o SIGNAL, es decir, constantes o señales, por defecto los parámetros son CONSTANT, para que sean señales se debe indicar explícitamente. Por otro lado, los procedimientos pueden tener varios tipos de puertos, IN, OUT e INOUT, de entrada, salida o entrada-salida; por lo tanto, los parámetros de entrada son CONSTANT por defecto y el resto de parámetros son de tipo VARIABLE, aunque también pueden ser de tipo SIGNAL si se declara explícitamente.

Las funciones pueden ser PURES o puras, devolviendo el mismo valor para unos parámetros determinados, e IMPURES o impuras si para esos mismos valores de entrada se devuelve un valor diferente. Realmente, estas palabras reservadas hacen la función de comentario, puesto que una función no se hace pura o impura solo por indicarlo.

Existen varias diferencias entre estos dos elementos:

- Los procedimientos pueden tener una sentencia WAIT, en cambio, las funciones no pueden tenerla nunca.
- Las funciones devuelven un valor mientras que los procedimientos devuelven los valores a través de sus parámetros de salida. Por ello, las funciones deben contener la palabra reservada RETURN que les

permitirá devolver el resultado pero los procedimientos no necesitan disponer de esta sentencia, en caso de contener esta palabra se interrumpirá la ejecución del procedimiento en esa sentencia.

- Por ello, y como se ha mencionado anteriormente, las funciones solo tienen parámetros de entrada mientras que los procedimientos pueden tener parámetros de entrada, salida y de entrada-salida.
- Los procedimientos se llaman con sentencias secuenciales o concurrentes mientras que las funciones se usan en expresiones.

6.1.2 Llamadas a subprogramas

Las llamadas a los subprogramas se realizan indicando el nombre de dicho subprograma seguido de los argumentos necesarios en el mismo, los cuales irán entre paréntesis.
Un ejemplo de un subprograma:

```
PROCEDURE ejemplo (din: std_logic_vector(22 downto 0), Dout : std_logic, long : IN integer := 200) IS
BEGIN
      IF din = X"00000001" THEN
            Dout:= '1';
       ELSE
            Dout := '0';
      END IF;
END ejemplo;
```

En este lenguaje existen varias maneras de pasar los argumentos al subprograma:

- Asociación implícita: los parámetros se introducen en la llamada en el mismo orden que en el que se declaran en el subprograma. Un ejemplo seria:
  ejemplo(ctj(44 downto 22), valorout, longin);

- Asociación explicita: Los argumentos se colocan en cualquier orden, indicando que valor corresponde a cada uno de los declarados en el subprograma. Un ejemplo es:

```
ejemplo(Dout=>valor, log=>valorlog, din=>ctj(44 downto 22));
```

- Asociación libre: Existe la posibilidad de dejar parámetros por especificar en las llamadas a los subprogramas, de forma que tengan unos valores predeterminados para que en el caso de que no se introduzca ningún valor se tome ese. Algunos ejemplos podrían ser:

```
ejemplo(ctj(44 downto 22), valorout);
ejemplo(ctj(44 downto 22), valorout, 400);
```

6.1.3 Sobrecarga de operadores

La sobrecarga de métodos es provocada por la existencia de más de una función con el mismo nombre pero con diferentes parámetros. Al igual que en otros lenguajes de programación, en VHDL también se puede dar esta sobrecarga, pero, a diferencia de los otros, en este se necesita tener más cuidado a la hora de declarar procedimientos con parámetros opcionales. Debido a ello, es muy probable que se pueda dar el caso de crear varias funciones con argumentos libres y por lo tanto, las llamadas a estas funciones sean las mismas y no se pueda saber a qué función se refiere.

6.2 Paquetes

Un paquete consta de un conjunto de subprogramas, declaraciones, funciones, constantes, etc., con la intención de implementar algún servicio. De este modo se pueden ocultar las descripciones de los subprogramas y hacer visibles las interfaces.

Estos paquetes están formados por dos tipos de subprogramas, que ayudan a mejorar la estructuración, la descripción y la legibilidad de los diseños.

6.2.1 Declaración de paquetes

Estos subprogramas se denominan *declaración* y *cuerpo*.

- *Declaración:*
  - En este subprograma se definen todas las funciones que formaran el paquete.
  - Siempre debe estar presente y contener todas las funciones definidas en el cuerpo del paquete.
  - El nombre del subprograma debe coincidir con el nombre del cuerpo.

- *Cuerpo:*
  - Contiene las funciones y procedimientos que describen los métodos definidos en la parte declarativa del paquete.
  - Este subprograma es prescindible si no se describen funciones o procedimientos.
  - Los nombres de los métodos y del subprograma deben ser iguales a los establecidos en la parte declarativa.

La estructura de los paquetes es la siguiente:

```
--Declaracion de paquete
PACKAGE NOMBRE IS
   [Declaraciones]
   END [PACKAGE] [NOMBRE];

--Declaracion del cuerpo
PACKAGE BODY NOMBRE IS
   Declaraciones
    Subprograma
       . . .
   END [PACKAGE BODY] [NOMBRE];
```

6.2.2 Llamadas a los paquetes

Para acceder a los tipos creados en un paquete existen dos formas de realizarlo:

- Cuando el paquete no está visible es necesario indicar el nombre del paquete seguido de un puto, seguido del elemento que se desea utilizar. Un ejemplo seria:

```
VARIABLE dir : MiPaquete.mi_paquete.dir_type;
dir := MiPaquete.mi_paquete.inicio;
```

- Otra posibilidad es hacer visible el paquete, para de ese modo, no tener que indicar el nombre del paquete cada vez que se quiere utilizar uno de sus elementos. Para ello se usa la sentencia USE seguido del paquete que se quiere hacer visible, seguido por un punto y el método que se va a utilizar, es necesario utilizar esta sentencia como tantos métodos se quieran usar, o en el caso de querer incluir todos, se usa la palabra reservada ALL. Un ejemplo seria:

```
USE MiPaquete.mi_paquete.ALL
VARIABLE dir : dir_type;
dir := inicio;
```

6.3 Librerías

Hasta ahora se han visto varios elementos del lenguaje VHDL, como las arquitecturas, los paquetes, las entidades, los subprogramas, etc.

Cuando se realiza una descripción se utilizan estas unidades en uno o más ficheros, estos ficheros son denominados *ficheros de diseño.*

Una vez se han creados todos estos ficheros y la descripción esta completa se compilan los ficheros para obtener una *librería* o *biblioteca de diseño,* de forma que contienen los elementos que componen el circuito. La biblioteca donde se guardan estos ficheros se denomina *work*.

Una librería se compone de dos partes diferencias, esta diferenciación se realiza dependiendo de las unidades que la formen. Por un lado, *las unidades primarias*, que corresponden a entidades, paquetes y archivos de configuración. Y por otro lado, están *las unidades secundarias,* que corresponden por arquitecturas y cuerpos de paquetes. Por lo tanto, se puede deducir que por cada unidad secundaria deberá haber una unidad primaria asociada a ella.

Al realizar una compilación se analizan las unidades según aparecen en el texto, por ello, es muy importante establecer un orden lógico de las unidades, para que se puedan cumplir las dependencias que existen entre ellas.
La forma en la que se crea la librería una vez compilada no sigue un estándar, por ello, la forma que toma es muy diversa; dependiendo de la herramienta usada para compilar así será la librería.

Para incluir una librería en un diseño es necesario usar la sentencia LIBRARY seguida del nombre de la librería. Además, como en el caso de los paquetes, también es posible hacer visibles los elementos internos de las bibliotecas. Utilizando la sentencia USE seguido de la librería, seguido del paquete de la librería y seguido por el método que se quiere hacer visible, todo ello separado por puntos. Al igual que en los paquetes, se pueden hacer visibles todos los elementos de un paquete utilizado la palabra ALL.

En este lenguaje existen dos librerías que no necesitan ser importadas. La primera de ellas es la librería *work*, que es la librería que se crea al compilar los ficheros de diseño. Y la segunda es la librería *std* que contiene los paquetes *standard* y *textio*, los cuales contienen definiciones de tipos y funciones para el acceso a ficheros de texto.

6.3.1 Librería ieee

Esta librería es una de las que más se utiliza en el mundo de la industria.
Contiene diferentes tipos y funciones que complementan a las que vienen por defecto en el propio lenguaje.

7. Ficheros

En ocasiones es necesario o conveniente utilizar ficheros para leer información o almacenarla. El uso de ficheros únicamente es válido en la simulación y no pueden almacenar matrices multidimensionales, punteros u otros ficheros.

Para trabajar con ficheros de texto se usa el paquete textio, que permite la conversión de tipos, y en este caso, la conversión de los datos codificados por VHDL a un lenguaje entendible por el sistema y por los humanos.

7.1 Apertura y cierre

En primer lugar es necesario declarar el tipo de datos que contendrá el archivo, para ello se usan las palabras reservadas TYPE y FILE.

Posteriormente se accede al fichero. Existen dos maneras dependiendo del tipo de compilador:

- Compilador VHDL'87: Se indica el tipo del fichero seguido del modo, que puede ser IN si va a leer y OUT si va a escribir, seguido del nombre del fichero a usar. El modo predeterminado es el de lectura.
- Compilador VHDL'93: Se indica el tipo del fichero, seguido opcionalmente de la apertura del fichero y del modo en el que se va a usar el fichero; write_mode para indicar escritura, read_mode para indicar lectura (por defecto) y append_mode para indicar concatenación. Seguido del nombre del fichero. No tiene la obligación de abrir el fichero en el momento en el que es declarado, existe la posibilidad de abrirlo posteriormente. Para realizar la apertura y cierre posterior a la declaración del fichero se usan las palabras reservadas file_open y file_close.

A continuación se muestra la estructura de lo explicado previamente:

```
TYPE fichero_tipo IS FILE OF tipo;
```

```
FILE nombre:
        fichero_tipo IS [modo] "fichero":        -- VHDL'87
        [fichero_tipo [OPEN modo] IS "fichero"] -- VHDL'93
```

También se muestra un ejemplo práctico:

```
TYPE arch_integer IS FILE OF integer;
-- Declaración y apertura a la vez
FILE datos: arch_integer OPEN read_mode IS "input.txt";
-- Declaración y apertura posterior
FILE datos: arch_integer;
        .....
file_open(datos, "input.txt",read_mode);
-- Cerrar
file_close(datos);
```

7.2 Lectura y escritura

Aparte de concretar el modo de uso de un fichero, como ya hemos visto en el apartado anterior, existe la posibilidad de realizar diferentes operaciones con ellos. Es posible realizar la lectura, escritura y comprobación de un fichero. Estas operaciones se pueden realizar a través de los siguientes métodos:

```
PROCEDURE read(FILE f: tipo_archivo; value: OUT tipo);
PROCEDURE write(FILE f: tipo_archivo; value: IN tipo);
PROCEDURE endfile(FILE f: tipo_archivo) RETURN boolean;
```

Con la función read se da la posibilidad de leer un archivo; con la función write es posible escribir en el archivo; y con la función endfile se comprueba si el fichero se ha cerrado.

8. Simulación

VHDL realiza su simulación utilizando la técnica "simulación por eventos discretos". Esta técnica permite avanzar la simulación a intervalos variables, en función de la planificación de la ocurrencia, en este caso, en función del cambio de señales. Esto significa que el comportamiento del circuito se simula desde que ocurre un evento hasta el siguiente, pudiendo pasar uno o varios pico-segundos. Durante el intervalo de tiempo en el que no se produce ningún evento, se mantiene el valor de las señales.

8.1 Fases de la simulación

La simulación consta de tres fases:

- Fase 0: también conocida como fase de inicialización, en ella, la simulación asigna unos valores iniciales a las señales y se pone el tiempo a cero. La asignación se realiza rellenando una lista de eventos para el instante t = 0.
- Fase 1: Se ejecutan todas las transiciones planificadas para ese instante de tiempo. Es decir, el código es ejecutado ordenadamente, teniendo en cuenta las señales modificadas y cumpliendo las normas de ejecución.
- Fase 2: Las señales modificadas a causa de las transiciones planificadas en el instante t se escriben en la lista de eventos para el instante $t + \delta$. Donde δ es un instante infinitesimal.

Las fases 1 y 2 se repiten hasta que no existen más transiciones. Además, en los instantes entre eventos se mantienen los valores de las señales.

A continuación, se muestran 2 ejemplos de simulación. En el primer ejemplo se simularan asignaciones fuera del PROCESS, y en el segundo asignaciones dentro del mismo.

A tiene valor 0 en el instante 0 ns y 1 en el instante 5 ns.

- Asignaciones concurrentes:

```
B <=A;
C<=B;
```

La tabla de asignaciones es la siguiente:

t(ns)	0	0 + δ		5	5 + δ	
A	0	0	No cambios	1	1	No cambio
B	Unassigned	0		0	1	
C	Unassigned	0		0	1	

Tabla 2-9. Ejemplo de asignaciones en simulación fuera del PROCESS VHDL

En el instante 0 se asigna el valor de A; se asigna el valor de A a B y asimismo, de B a C, se escriben los cambios para el siguiente paso de simulación. En el siguiente paso de simulación, 0 + δ, no ha cambiado A, por lo que no hay cambios.

En el instante 5, se asigna 1 a A; se repite el procedimiento anteriormente definido llegando al instante 5 + δ sin cambios.

En la siguiente figura se muestra gráficamente lo expresado en la tabla de asignaciones:

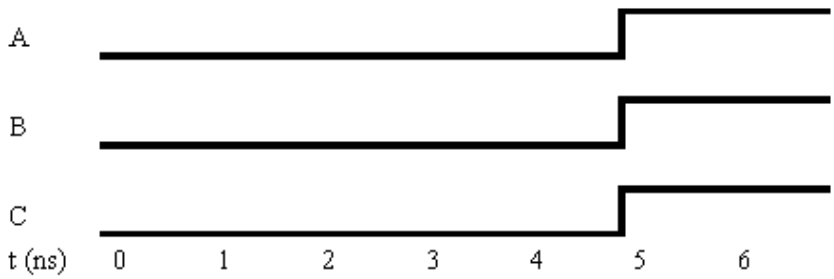


Ilustración 2-6. Simulación asignaciones fuera del PROCESS VHDL

- Asignaciones dentro de un PROCESS:

```
PROCESS (A)
BEGIN
        B <=A;
        C<=B;
END PROCESS;
```

La tabla de asignaciones es la siguiente:

t(ns)	0	0 + δ		5	5 + δ	
A	0	0	No cambios	1	1	No cambios
B	Unassigned	0		0	1	
C	Unassigned	Unassigned		Unassigned	0	

Tabla 2-9. Ejemplo 2. Asignaciones dentro del PROCESS VHDL

En el instante 0 se entra dentro del PROCESS y se asigna el valor de A a B y el de B (valor cuando se entró al PROCESS) a C y se escriben los cambios para el siguiente paso de simulación. En el siguiente instante no ha cambiado A, por lo tanto se mantienen los valores anteriores.
En el instante 5, cambia A y se entra en el PROCESS, por lo tanto, los valores se asignan de la misma manera que en instante 0. En el siguiente instante no se realizan cambios en A, por lo que se mantienen los valores.

En la siguiente figura se muestra lo expresado en la tabla de asignaciones:

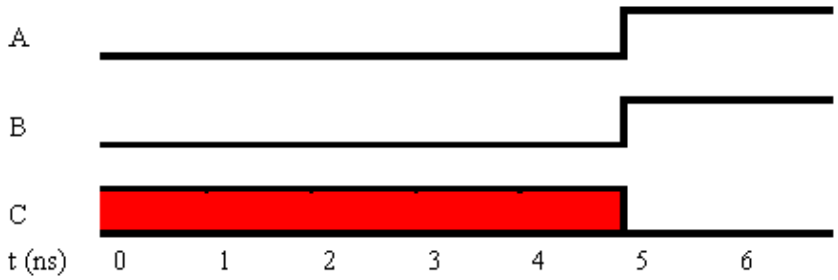


Ilustración 2-7. Simulación de las asignaciones dentro del PROCESS VHDL

8.2 Sentencias de simulación

A continuación se describen los diferentes tipos de sentencias que generan restricciones relacionadas con el momento de ejecución de otras sentencias.

8.2.1 Retrasos: AFTER

Los retrasos son uno de los elementos más importantes en la simulación. Esto se debe a que el comportamiento del circuito puede variar dependiendo del cambio de las señales.
Las asignaciones se realizan de forma inmediata, si no hay nada que indique lo contrario. Para retrasar alguna asignación y así variar el comportamiento de un circuito se utiliza la opción AFTER cuando se asigna un valor a una señal.

La sintaxis es la siguiente:

```
señal <= valor AFTER tiempo;
```

Donde *tiempo* es el valor que indica el tiempo de espera. También se debe indicar la unidad de tiempo.

Pero esta sentencia puede ser más compleja de lo que aparenta inicialmente. Por ejemplo, se puede asignar un valor inicial y modificarlo pasado el tiempo establecido. También existe la posibilidad de modificar el valor de la señal cada cierto tiempo.

```
rst <= '1', '0' AFTER 15 ns; -- Inicialmente rst=1, después de 15 ns rst=0
clk <= not clk AFTER 5 ns; -- Cada 5 ns clk cambia de valor
```

8.2.2 Retrasos: WAIT

Otra forma de introducir retrasos entre dos sentencias es mediante la palabra reservad WAIT. La sintaxis es la siguiente:

```
WAIT [ON lista | UNTIL condición | FOR tiempo];
```

Esta sentencia puede ser combinada de diferentes maneras. Se puede usar un solo elemento o ninguno, también se pueden combinar dos de ellos o se pueden usar todos los elementos a la vez.

El elemento ‘lista’ se corresponde con una lista de sensibilidad de las señales, lo que provocará que permanezca en espera hasta que se produzca un cambio en alguna señal de la lista.

El elemento ‘condición’ se trata de una espera indeterminada hasta que la condición se cumpla.

Por último, el elemento ‘tiempo’ es un valor definido en una unidad de tiempo la cual indica el tiempo que tendrá que trascurrir hasta que se continúe con la ejecución.

A continuación se muestran algunos ejemplos:

```
stop <= '1';
WAIT FOR 15 ns;
stop <= '0';
BIN_COMP : PROCESS
BEGIN
  WAIT ON A, B UNTIL CLK = '1';
  . . .
END PROCESS;
```

8.2.3 Niveles lógicos

Existen tres niveles lógicos: 1, 0 y X. Donde X se refiere a un valor desconocido.

También existen las llamadas fuerzas:

- S: Salida obtenida de una conexión a alimentación o tierra a través de un transistor.
- R: Salida obtenida a través de una conexión a una resistencia conectada a alimentación o tierra.
- Z: Obtiene la salida mediante una conexión a alimentación de alta impedancia.
- I: fuerza desconocida en el bus.
- U: Fuerza que solo aplica el tipo std_logic. Indica que una señal no se ha inicializado.
- - : Fuerza que solo puede ser aplicada por el tipo std_logic. Indica que no importa el valor que se le ponga a la señal.

8.2.4 Notificaciones

En las simulaciones de circuitos puede ser interesante el uso de las notificaciones, con ellas se puede saber que señales has sido activadas o

comprobar si una señal a tomado un valor determinado. La sintaxis es la siguiente:

```
ASSERT <condición>
[REPORT <expresión>]
[SEVERITY <expresión>];
```

Se usa la palabra ASSERT seguida de una condición como elemento de activación. Puede estar seguida de un mensaje que se especifique y del nivel de gravedad. Los niveles de gravedad pueden ser (de menor a mayor): *note*, *warning*, *error* (por defecto) y *failure*.

Si la condición **NO** se cumple aparece en la pantalla del simulador el mensaje especificado y el nivel de gravedad. Si no se indica ningún mensaje aparecerá "*Assertion Violation"* por defecto.

8.3 Descripción de un banco de pruebas

Una de las partes más importantes en el diseño de cualquier sistema son las pruebas para verificar el funcionamiento del sistema.

Tradicionalmente la verificación solo era posible tras implementar el sistema físicamente, lo que se traducía en un alto riesgo y coste adicional.

En la actualidad, existe la posibilidad de probar el sistema antes de implementarlo. Para hacer las verificaciones lo más sencillo es cambiar las entradas para ver como son las salidas, en una herramienta de simulación, cuando el sistema a probar sea sencillo, en caso contrario lo ideal sería crear un banco de pruebas.

Un banco de pruebas es una entidad sin puertos, cuya estructura está formada por un componente que corresponde al circuito que se quiere simular y las alteraciones de las señales de entrada del componente, para poder realizar más casos de pruebas.

Se recomienda realizar la descripción del banco de pruebas a la vez que se describe el diseño del sistema.

A continuación se muestran las diferentes metodologías que se pueden llevar a cabo para crear los bancos de pruebas:

8.3.1 Método tabular

Para verificar el correcto funcionamiento de un sistema se debe crear una tabla con las posibles entradas al sistema y las salidas esperadas a dichas entradas.

Todo ello debe ser relacionado mediante código VHDL.

8.3.2 Uso de ficheros

Al igual que el caso anterior, es necesario realizar una tabla con las entradas y las salidas esperadas, pero en ese caso se separan los casos de prueba y el código de simulación. Esto es posible ya que VHDL dispone de paquetes de entrada/salida para la lectura/escritura en ficheros de texto.

Por lo tanto, los casos de prueba irían escritos en un fichero en forma de vectores y por otro lado el código relacionado con la simulación, en el que se incluye el acceso al fichero.

8.3.3 Metodología algorítmica

Este tipo de test se basa en realizar algoritmos para cubrir el mayor número de casos posibles.

9. Síntesis

La síntesis de un circuito consiste en abstraer la descripción del mismo hasta conseguir un diseño puramente estructural.

Cualquier descripción en VHDL será sintetizable, sin depender del nivel de abstracción del circuito. Por ello, en algunas ocasiones, el diseño no será el más apropiado puesto que la velocidad que se requiere no se corresponderá a la descripción.

A continuación se verá como son interpretadas algunas instrucciones por las herramientas de síntesis. Además, se expondrán algunos consejos y restricciones para crear un diseño que se obtenga lo mismo en la simulación y en la síntesis.

- **Evitar clausulas temporales y esperas.** El uso de retrasos está prohibido, al igual que el comando *WAIT*, debido a que serán ignorados por el propio sintetizador.
- **Identificadores de puertas claros.**
- **Uso de funciones y módulos.** La división del código ayuda a la síntesis del mismo.
- **Inicialización de variables y señales.** La síntesis no tiene en cuenta ninguna inicialización, a menos que se disponga de un *reset* o algo similar.
- **Asignaciones únicas.** Las asignaciones deber ser únicas para que el sintetizador no las confunda.
- **Niveles lógicos.** No se admiten todos los posibles valores de una señal.
- **Evitar IF's anidados.** Si es necesario, se recomienda usar como máximo tres niveles de IF's anidados, pero la mejor opción es utilizar bloques CASE.
- **Permitir discrepancia.**
- **Sentencias no sintetizables.** Desde la creación de este lenguaje existen sentencias no sintetizables, por lo que no pueden ser utilizadas si se desea sintetizar el diseño.

  Existe un conjunto de sentencias que no son sintetizables, pero existe otro grupo de sentencias que, dependiendo de la herramienta seleccionada, si son sintetizables. Por este motivo, los desarrolladores de estas herramientas compiten entre sí para conseguir reducir el número de sentencias no sintetizables en su herramienta y así ser la mejor herramienta.

Capítulo III: Verilog

1. Introducción

DESCRIPCIÓN

[3] Verilog es uno de los lenguajes de descripción hardware más utilizados.

Con este lenguaje se pueden describir sistemas digitales, tales como procesadores, memorias o elementos más simples. Esto significa que este lenguaje, como los de su mismo tipo, puede utilizarse para describir cualquier hardware a cualquier nivel.

Este lenguaje es de dominio público, al igual que VHDL, porque es un estándar IEEE.

Verilog permite a los diseñadores de hardware expresar sus diseños con construcciones de comportamiento, difiriendo los detalles de la implementación a una etapa posterior en el diseño final.

Este lenguaje se diseñó basando en el lenguaje de programación C, con el fin de que resultara familiar para los diseñadores y así fuese rápidamente aceptado. Por ello, Verilog tiene un preprocesador como C y la mayoría de sus palabras reservadas son similares a las de C, así como el mecanismos de formateo en las rutinas de impresión y en los operadores del lenguaje.
Las diferencias entre estos lenguajes más importantes son:

- Verilog carece de estructuras, punteros y funciones recursivas.
- En el lenguaje de programación C no se encuentra el concepto de tiempo, tan importante en los lenguajes HDL.

Un diseño en Verilog consiste en una jerarquía de módulos. Estos módulos son definidos con conjuntos de puertos de entrada, salida y de entrada-salida; internamente contienen una lista de cables y registros. Las sentencias concurrentes y secuenciales definen el comportamiento del modulo, definiendo las conexiones entre los puertos, los calves y los registros. Las sentencias

secuenciales son colocadas dentro de un bloque y ejecutadas en orden secuencial. Por otro lado, todas las sentencias concurrentes y los bloques donde se meten las sentencias secuenciales se ejecutan en paralelo en el diseño. Un modulo puede contener una o más instancias de otro modulo para definir un subcomportamiento.

HISTORIA

[14] Verilog fue inventado por Phil Moorby mientras trabajaba en la empresa Gateway Design Automation Inc. en 1985. Por lo tanto, este lenguaje nació inicialmente como un lenguaje propietario de modelado de hardware. La empresa Gateway Design Automation Inc. fue comprada por Cadence Design Systems en 1990 que adquirió los derechos sobre Verilog.

Ese mismo año, debido al incremento del éxito y a la presión del mercado, Cadence decidió hacer el leguaje abierto y disponible para su estandarización. Por ello, en 1991 organizó el Open Verilog International (OVI) y, en ese mismo evento entregó la documentación para convertir el lenguaje en un lenguaje abierto denominado Verilog Hardware Description Language.

Verilog fue después enviado a la IEEE que lo convirtió en el estándar IEEE 1364-1995, habitualmente referido como Verilog 95.
Varias extensiones de Verilog 95 fueron enviadas a la IEEE para cubrir las deficiencias que los usuarios habían encontrado en el estándar original de Verilog. Estas extensiones se volvieron el estándar IEEE 1364-2001 conocido como Verilog 2001.
En 2002 se publicó otro estándar conocido como IEEE 1364-200. Esta revisión tuvo muchos errores los cuales fueron corregidos en la revisión de 2003, conocida en la IEEE 1364-2001 revisión C.

TIPOS DE DESCRIPCIÓN

Como se ha mencionad anteriormente, este lenguaje permite describir circuitos electrónicos. Estos sistemas se pueden describir de las siguientes maneras:

- **Nivel estructural**: Se utilizan elementos previamente creados, del desarrollador o de terceros, donde los diferentes elementos se interconectan entre sí. Es decir, se interconectan bloques con una funcionalidad pequeña para que comunicándolos entre sí hagan una tarea mayor.
- **Nivel de comportamiento**: El desarrollador describe la transferencia de información entre registros.

Estos tipos de descripción pueden utilizarse conjuntamente, dando diseños mixtos.

NIVELES DE ABSTRACCION

Verilog es capaz de soportar el diseño de un circuito a diferentes niveles de abstracción, de los cuales destacan:

- **Nivel de comportamiento:** El diseñador define el comportamiento del circuito mediante algoritmos concurrentes. A su vez, cada subprograma se ejecuta de forma secuencial, es decir, un conjunto de instrucciones que se ejecutan una tras otra.

  La principal característica de este nivel es su total independencia de la estructura del diseño.

  A continuación se muestra un ejemplo de este nivel:

```
module (clk, A, B, C, D);
input   clk, A, B;
output  C, D;
reg     C;
always@(C)
begin
D <= C + 10;
end
always@(posedge clk)
begin
C <= A + B;
end
endmodule
```

- **Nivel RTL:** también conocido como nivel de transferencia de registros. Los circuitos diseñados en este nivel tienen la característica de ser sintetizables, por lo tanto, todo código sintetizable es código a nivel RTL. Los diseños en este nivel especifican las características de un circuito mediante operaciones y transferencia de datos entre los registros.

  En este nivel se utiliza un reloj explicito, además, contiene limites de sincronización exactos, por lo que las operaciones están programadas para ocurrir en determinados momentos.

  A continuación se muestra un ejemplo:

```
module flipflop(d, clk, q, q_bar);
input   d, clk;
output  q, q_bar;
reg     q, q_bar;

always@(posedge clk)
begin
  q     <= #1 d;
  q_bar <= #1 !d;
end
endmodule
```

- **Nivel de puerta:** También denominado modelo estructural, además se corresponde con una descripción a bajo nivel. El diseño se describe mediante operaciones primitivas lógicas predefinidas (AND, OR, XOR), conexiones y añadiendo distintas propiedades de tiempo. Estas señales pueden tomar los valores ‘0’, ‘1’, ‘X’ (indefinido) y ‘Z’ (alta impedancia).

  Utilizar este nivel puede no ser buena idea para cualquier nivel de diseño lógico. El código de este nivel es generado por diferentes herramientas, como herramientas de síntesis, y la lista de conexiones que genera se utiliza para la simulación de nivel de puerta.

```
module mux(f, a, b, sel);
input   a, b, sel;
output  f;
```

```
and   #5  g1(f1, a, nsel);
          g2(f2, b, sel);
or    #5  g3(f, f1, f2);
not       g4(nsel, sel);
endmodule
```

2. Elementos básicos del lenguaje

[7] Antes de explicar la estructura que se debe seguir es necesario conocer algunos elementos básicos del lenguaje:

2.1 Comentarios

Los comentarios en este lenguaje se pueden comentar de dos maneras diferentes:

- Los comentarios de una sola línea: van precedidos por los caracteres "//", ignorando todo la información hasta el final de la línea. Un ejemplo es:

```
// Esto es un comentario de una linea
```

- Los comentarios de un conjunto grande de líneas de código: van delimitados por los caracteres "/*" y "*/".
  Ejemplo:

```
/* Esto es un comentario
   de varias lineas */
```

2.2 Base numérica

Los números ene este lenguaje pueden ser decimales, hexadecimales, octales o binarios. También pueden expresarse números negativos, pero estos se representan en complemento a2. El carácter "_" puede ser usado como ayuda para representar los números con más claridad. La sintaxis a seguir para representar un número es:

<TAMAÑO> <BASE> <VALOR>

- *Tamaño*: Representa el numero de bits (en decimal) de la cantidad expresada a continuación. Este dato es opcional, en caso de no estar expresado, el valor por defecto es de 32 bits.
- *Base*: este dato indica en que base se va a expresar el valor. Es un dato opcional, si no se indica el valor es decimal. La manera en la que se indican las distintas bases es la siguiente:

  'b Base binaria

  'd Base decimal

  'h Base hexadecimal

  'o Base octal

- *Valor*: es la cantidad. Además de las cifras permitidas en cada base (0-9 y A-F), se pueden utilizar dos valores más para el caso de base binaria:

  X para indicar que el valor de un bit desconocido.

  Z para indicar que el valor de un bit es alta impedancia.

A continuación se muestran algunos ejemplos:

```
187          // Numero decimal (Utilizado 32 bits)
8'h0a              // Numero hexadecimal (Almacenado 00001010)
3'b1            // Numero binario 3 bits (Almacenado 001)
-4'b10        // Numero binario de 4 bits complemento a2 de 10 (Almacenado 1110)
```

2.3 Identificadores

Un identificador está formado por una letra o el carácter "_" seguido de letras, números y los caracteres "$" o "_". Este lenguaje distingue entre mayúsculas y minúsculas. Algunos ejemplos son:

```
reg A;      //      reg A0;      //      reg data_i;
```

2.4 Tipos de datos

Para definir las variables es necesario especificar el tipo de dato que va a ser esta variable. La sintaxis para declarar estas variables es la siguiente:

```
<TIPO> [<MSB> : <LSB>] <NOMBRE>;
```

- **Tipo:** a continuación se muestran los tipos de variables existentes, aunque los más destacados son *reg* y *wire.*
    - ***reg****:* Este tipo de dato representa variables con la capacidad de almacenar información en ellas.
    - ***wire:*** Este tipo de dato no tiene capacidad de almacenamiento. Representan conexiones estructurales entre componentes.
    - ***real:*** Registro con la capacidad de almacenar números en coma flotante.
    - ***time:*** Registro con la capacidad de almacenar hasta 64 bits sin signo.
    - ***integer:*** Registro con la capacidad de almacenar 32 bits.
- **MSB** y **LSB:** Estos datos representan el tamaño del valor, indicando el valor más pequeño y el más grande. Por defecto, este tamaño es de 1 bit.
- **Nombre:** Indica el nombre de la variable creada.

  A continuación se muestran algunos ejemplos de variables:

```
reg[5:0] data;   // Registro de 6 bits, donde data[0]
es el bit menos significativo
wire outA;       // Net de un bit
```

```
integer numA;    // Registro de 32 bits
reg[31:0] numB;  // Registro de 32 bits, donde
numB[0] es el bit menos significativo
```

2.5 Operadores

Los diferentes operadores que usa el lenguaje son:

2.5.1 Tipos de operadores

2.5.1.1 Binarios aritméticos

El operador aparece entre dos operandos. Teniendo en cuenta que si algún bit es 'X' el resultado de la operación, independientemente del operador, será 'X'. Los operadores son:

- **+** (Suma): Indica una suma entre dos números. También puede actuar como signo positivo si se sitúa delante de una expresión. Algunos ejemplos son:

```
A = B + C;
D = +8'h01;
```

- **–** (Resta): Indica la resta de dos operandos. Si va delante de una expresión modifica su signo. Algunos ejemplos son:

```
A = B - C;
D = -8'h01;
```

- ***** (multiplicación): Multiplica dos números de cualquier tipo. Un ejemplo es:

```
A = B * C;
```

- **/** (división): Divide dos números de cualquier tipo. Un ejemplo es:

```
A = B / C;
```

- **%** (Resto): Obtiene el resto de la división de dos números de cualquier tipo. Un ejemplo es:

```
A = B % C;
```

2.5.1.2 Relacionales

Permiten comparar dos operandos, retornando 1 si es verdadero y 0 si es falso. Si alguno de los valores a comparar es 'X' el resultado será 'X'.

- **==** (Igualdad): Devuelve verdadero si los operandos son iguales, en caso contrario devuelve falso.

  != (Desigualdad): Realiza la función contraria a la igualdad. Si los operandos no son iguales devuelve verdadero y en caso contrario falso. Algunos ejemplos son:

```
// A: 5   B: 6
Desigualdad: (B != A)   Igualdad: (B == A)
// Resultado: 1                        0
```

- **===** (Igualdad): Su funcionalidad es igual a la igualdad descrita anteriormente, pero también compara si los valores son 'X' (indefinidos) o 'Z' (alta impedancia).
  !== (Desigualdad): Su funcionalidad es idéntica a la anterior, pero, también difiere en que compara los valores 'X' y 'Z'. Algunos ejemplos son:

```
// A: 5   B: 6
Desigualdad: (B !== A)  Igualdad: (B === A)
// Resultado: 1                        0
```

- **>** (mayor), **>=** (mayor o igual): Devuelve verdadero si, en el primer caso, el primer operando es estrictamente mayor que el segundo; y en el segundo caso, si es mayor o igual.

- **<** (menor), **<=** (menor o igual): Realiza la función contraria al anterior operador. Devuelve verdadero si, en el primer caso, el primer operando es estrictamente menor que el segundo; y en el segundo caso, si es menor o igual. Algunos ejemplos:

```
// A: 5   B: 6        C: 5
```

Menor: `(B < A)` `(A < B)` Menor o igual: `(C <= A)`

```
//Resultados: 0   //  1      //      1
```

Mayor: `(B > A)` Mayor o igual: `(C >= A)` `(C >= B)` `//Resultados: 1 // 1 //0`

2.5.1.3 Lógicos

Aparece ente dos operandos lógicos y proporciona un valor lógico como resultado: verdadero o falso.

- **!** (Negación): En esta caso, se coloca delante de un operando, y tiene como función cambiar el valor lógico del mismo. Un ejemplo es: `// A: true`

```
(!A)    // Resultado: false
```

- **&&** (AND lógica): El resultado de esta operación es la combinación de los dos operandos. Es decir, para que sea verdadero, ambos operandos tienen que serlo, en otro caso el resultado es falso. Un ejemplo: `// A: true  B: false`

```
(A && B)      // Resultado: false
```

- **||** (OR lógica): Al igual que la AND el resultado es la combinación de los operandos. En este caso, para que el resultado sea verdadero solo es necesario que uno de los operandos lo sea. Ejemplo: `// A: true  B: false`

```
(A || B)      // Resultado: true
```

2.5.1.4 Lógica de bit

Estos operadores se parecen a los anteriores, pero en este caso permite efectuar operaciones lógicas con los bits de los operandos.

Estas operaciones se hacen bit a bit.

- **~** (negación): Tiene como función cambiar el valor lógico del operando bit a bit. Ejemplo: `// B: 4'b1110`

  ```
  (A = ~B)        // Resultado: A: 4'b0001
  ```

- **&** (AND): Como en el anterior operador (AND lógico), el resultado de esta operación es la combinación de los dos operandos, siendo verdadero (1) solo si ambos bit son 1. Ejemplo: `// A: 4'b0110  B: 4'b0101`

  ```
  (C = A & B)  // Resultado: C: 4'b0100
  ```

  Su operador opuesto es: **~&** (NAND) con el cual obtendrías el resultado contrario al del operador AND. Ejemplo:

  ```
  (C = A ~& B)// C: 4'b1011
  ```

- **|** (OR): Como en el anterior operador (OR lógico), el resultado es la combinación de los operandos, siendo verdadero (1) en todos los casos excepto si los dos bit son 0. Ejemplo: `// A: 4'b0110  B: 4'b0101`

  ```
  (C = A | B)  // Resultado: C: 4'b0111
  ```

  El operador contrario a este es: **~|** (NOR) con el cual obtienes el resultado contrario al que obtendrías con el operador NOR. Ejemplo: `(C = A ~| B)// C: 4'b1000`

- **^** (XOR): el resultado de la combinación de los operandos es verdadero (1) solo si los operandos son diferentes, en otro caso (los operandos son iguales) el resultado es 0. Ejemplo:

  ```
  // A: 4'b0110      B: 4'b0101
  ```

  ```
  (C = A ^ B)  // Resultado: C: 4'b0011
  ```

También existe el operador **~^** o **^ ~**(NOT XOR) con el cual obtienes el resultado contrario al que obtendrías con el operador XOR. Ejemplo: `(C = A ~^ B)// C: 4'b1100`

2.5.1.5 Lógica de reducción

Tiene un único argumento, y consiste en aplicar cada uno de los operadores en el operando, de tal manera que el resultado es un único bit.

- **&** (AND): Se realiza un AND de todos los bits, el resultado será siempre 0 excepto en el caso de que todos los bits del argumento sean 1. Ejemplos: `// A: 4'1111 B: 4'b0101`

  ```
  (C = &B) (D = &A) // Resultado: C: 0  D: 1
  ```

  Su operador opuesto es: **~&** (NAND) con el resultado contrario al del operador anterior. Ejemplos:

  ```
  (C = ~&B) (D = ~&A)  // Resultado: C: 1 D: 0
  ```

- **|** (OR): Se realiza un OR para cada bit del argumento, siendo el resultado 1 en todos los casos excepto cuando todos los bits del operando son 0. Ejemplos:

  ```
  // A: 4'b0000  B: 4'b0101

  (C = |B) (D = |A) // Resultado: C: 1  D: 0
  ```

  El operador contrario a este es: **~|** (NOR) obteniendo el resultado contrario. Ejemplo: `(C = ~| B)// C: 0`

- **^** (XOR): Se realiza un XOR a todos los bits de operando. El resultado va a ser un único bit que dependerá de si la cantidad de 1 es par, siendo el resultado 0, o de si es impar, dando como resultado 1. Esto se debe a la forma de realizar la operación que tiene el operador, si hay dos operandos iguales el resultado es 0 y al contrario. Ejemplo:

```
// A: 4'b0110      B: 4'b1101

(C = ^A) (D = ^B) // Resultado: C: 0  D: 1
```

También existe el operador **~^** o **^ ~**(NOT XOR) con el cual el resultado del operador XOR se niega y obtiene el resultado contrario. Ejemplo: `(D = ~^B)// D: 0`

2.5.1.6 Otros

Otros operadores que no se han podido clasificar en los anteriores son:

- **{,}** (Concatenación): Se utiliza para concatenar dos operandos de cualquier tipo. Ejemplos: `// A: 4'0110 B: 4'b0010`

  ```
  C = {A, B} D = {2{B}}        //Resultado:
  C: 8'0110_0010   D: 0110_0110
  ```

- **<<** (Desplazamiento a la izquierda): Desplaza bits a la izquierda. Es necesario indicar el número de desplazamientos y los bits que se quedan "vacios" se rellenan con 0. Ejemplos:

  ```
  // A: 8'b1111_1001

  (A << 2) // Resultado: A: 8'b1110_0100
  ```

- **>>** (Desplazamiento a la derecha): Realiza la misma función que el desplazamiento a la izquierda, pero lo realiza a la derecha. En este caso también es necesario indicar el número de desplazamientos y los bits que se quedan "vacios" se rellenan con 0. Ejemplos:

  ```
  // A: 8'b1111_1001

  (A >> 2) // Resultado: A: 8'0011_1110
  ```

- **?:** (Condicional):Dependiendo del resultado lógico (falso o verdadero)el resultado será uno u otro. Ejemplos:

  ```
  // A: 1'b1  B: 3'b111  C: 3'b001
  A == 1? B : C // Resultado: A: 3'b111
  ```

```
//Si se cumple la igualdad, B cambia su
valor C
```

2.5.2 Precedencia de operadores

El orden de prioridad de los diferentes operadores es el siguiente:

1. Unarios (cambios de signo de un valor), multiplicación, división y resto. +, -, *, /, %
2. Suma, resta, y desplazamientos. +, -, >>, <<
3. Relacionales. <, >, =, <=, >=, ==, !=, ===, !==
4. Reducción. &, !&, ^, ~^, |, ~|
5. Lógicos. &&, ||
6. Condicional. ?:

3. Módulos

3.1 Características

- Cada modulo dispone de entradas y salidas, por las cuales se interconectan otros módulos, aunque existe la posibilidad de que no tengan ningún tipo de entrada o salida, como es el caso de los bancos de pruebas.
- Estos módulos carecen de variables globales.
- Aunque existe la posibilidad de realizar varias simulaciones de forma concurrente, de forma general se suele usar un único modulo el cual emplea el resto de módulos previamente definidos.
- Cada modulo se puede describir de diferente forma (estructural o de comportamiento).
- Fuera de los módulos solo hay directivas del compilador, que afectarán a partir del punto en donde aparece.

3.2 Estructura

[4] En este lenguaje los sistemas digitales se componen de interconexiones de un conjunto de módulos.

La estructura de estos módulos es la siguiente:

```
module <nombre> (<señales>);
<declaración de señales>
<funcionalidad del módulo>
endmodule
```

3.2.1 Interfaz del modulo

Los argumentos del modulo son los que comunican el interior o la funcionalidad del mismo con otros elementos del sistema. Estos argumentos pueden ser de tres tipos:

- *Input:* Este tipo de argumento indica las entradas al modulo, el tipo de dato de estas es *wire.*
- *Output:* Esta palabra reservada indica las salidas del modulo. Dependiendo del tipo de asignación que las genere serán de tipo *wire,* si producen de una asignación continua, y *reg,* si se producen de una asignación procedural.
- *Inout:* Estos argumentos son de entras y salidas a la vez. Son de tipo *wire.*

3.2.2 Procesos

El concepto de procesos que se ejecutan en paralelo es una de las características principales de este lenguaje, siendo ese uno de los aspectos diferenciales con respecto otros lenguajes.

Los procesos comienzan con el inicio de la simulación y secuencialmente van procesando cada una de las líneas que aparecen.

La descripción del comportamiento en este lenguaje debe declararse dentro de un proceso. Se distinguen dos tipos de procesos:

- *Initial:* Este tipo de procesos se ejecutan una sola vez, empezando su ejecución al inicio, por lo tanto, no existen retardos. Este proceso no es sintetizable, por lo tanto no puede descrito a nivel RTL. Se suele usar para la verificación de los distintos diseños, su estructura es la siguiente:

```
initial
begin
   . . . . // Sentencias
end
```

- *Always:* Este tipo de procesos esta en continuamente ejecutándose a modo de bucle. Este proceso es sintetizable y se puede controlar por la temporización o por los eventos.

  Si un proceso se ejecuta para más de un evento, el conjunto de estos eventos se denomina *lista sensible.* La estructura de este proceso es la siguiente:

```
always [<temporización>| <@(lista sensible)>]
begin
   . . . . // Sentencias
end
```

  Si el proceso engloba más de una asignación o de una estructura de control, estas deber estar en un bloque delimitado entre las palabras *begin* y *end.* Por lo tanto no es necesario incluir una lista de sensibilidad o una temporización.

3.2.3 Estructura de control

Este lenguaje dispone de muchas estructuras de control, a continuación se enumeran las estructuras más utilizadas:

1. **If-else**

   Esta sentencia es del tipo condicional y tiene la capacidad de controlar la ejecución de otras sentencias y/o asignaciones. En el caso de haber más de una sentencia o asignación, es necesario usar el bloque **begin-end**. La estructura de este bloque es la siguiente:

```
if (expresión)
  begin
       . . . . // Sentencias 1
  end
else
  begin
       . . . . // Sentencias 2
  end
```

   Si la expresión es cierta se ejecutan las sentencias del bloque *if* (Sentencias 1), en caso de que la expresión sea falsa se ejecutan las sentencias del bloque *else* (Sentencias 2).

2. **Case**

   Esta sentencia también es de tipo condicional. La función de esta sentencia es evaluar la expresión y dependiendo de su valor se ejecutará una sentencia o un bloque de sentencias del caso que coincida. Si no se cubren todos los posibles casos explícitamente, es necesario crear un caso por defecto (default) que se ejecutará cuando la expresión no coincida con ningún caso.

   Como en la sentencia anterior, si existen varias sentencias en un caso se deben agrupar en un bloque **begin-end**. Su estructura es la siguiente:

```
case (expresión)
caso 1:
  begin
       . . . . // Sentencias 1
```

```
      end
   caso 2:
      begin
            . . . . // Sentencias 2
      end
   . . . .
   default:
      begin
            . . . . // Sentencias D
      end
   endcase
```

2.1.Casez y Casex

Estas estructuras corresponden a versiones del *case*, con la particularidad de que los valores lógicos ‘X’ y ‘Z’ se tratan como valores indiferentes.

- Casez: Usa el valor ‘Z’ como indiferente.
- Casex: Usa los valores ‘X’ y ‘Z’ como indiferentes.

Para entender mejor el uso de estas estructuras se expone un ejemplo:

```
  casez (sel)  // sel: 3'b101
    3'b1zz:  A = 4'b0011; // siempre si 1º bit
sea 1
    3'b01?:  A = 4'b1100; // ? significa
indiferente
    default: A = 4'b1111;
   endcase  // Resultado A: 4'b11
casex (sel)  // sel: 3'bz01
    3'b10X:  A = 4'b0011; // X = cualquier valor
    3'bX0?:  A = 4'b1100; // ? significa
indiferente
    default: A = 4'b1111;
```

```
endcase  // Resultado A: 4'b1100
```

3. **For**

   En este caso, el bucle *for* es idéntico a lo utilizados en otros lenguajes.

   Al igual que en las anteriores bucles, si existen varias sentencias se deben incluir en un bloque **begin-end**.

   Su estructura es la siguiente:

```
for (<valor inicial>, expresión, <incremento>)
  begin
       . . . . // Sentencias
  end
```

4. **While**

   Este bucle se ejecuta de forma continuada mientras se cumpla la condición que evalúa. Al igual que en los casos anteriores, si existe más de una sentencia es necesario agruparlas en un bloque **begin-end.** Su estructura es la siguiente:

```
while (<expresión>)
  begin
       . . . . // Sentencias
  end
```

5. **Repeat**

   Este bucle se ejecuta un número determinado de veces, siendo este número la condición de salida del bucle. Si es necesario se usa el bloque **begin-end**.

   Su estructura es la siguiente:

```
repeat (<numero>)
```

```
begin
      . . . . // Sentencias
end
```

6. **Forever**

   Este bloque se ejecuta de forma continua y sin condición de finalización, por lo tanto, este bucle se ejecuta infinitamente. Si es necesario se utilizará el bloque **begin-end.**

   Su estructura es la siguiente:

```
forever
  begin
      . . . . // Sentencias
  end
```

7. **Wait**

   Debido a que la ejecución de un bucle se realiza de forma continua, se ejecutan todas las sentencias y se repite, sería de gran utilidad poder parar la ejecución. Existen dos formas diferentes de parar una ejecución, una es la lista de sensibilidad, y la otra forma es mediante la sentencia *wait*.

   Esta sentencia se compone de la palabra reservada *wait* y de una expresión que iniciará la condición para parar. Esta sentencia puede estar incluida dentro de las sentencias anteriormente mencionadas. Su sintaxis es:

```
wait (expresión)
```

3.3 Asignaciones

En verilog existen dos maneras de asignar valores:

- **Asignación continua**

  Este tipo de asignación se utiliza únicamente para modelar lógica combinacional, sin la necesidad de usar una lista de sensibilidad para la asignación. La sintaxis es la siguiente:

```
assign variable <#delay> = asignación
```

La variable solo puede ser declarada de tipo *wire*. Y la asignación solo puede ser declarada fuera de cualquier proceso o bloque *initial* o *always*.

- **Asignación procedural**

  Este tipo de asignación es la más usual hasta el momento. Su sintaxis es la siguiente:

  ```
  <#delay> variable = asignación
  ```

  Estas asignaciones se usan en el interior de los procesos y el tipo de variable a la que se asigna el dato puede ser de cualquier tipo.

3.4 Temporizaciones

En algunas ocasiones es necesario retrasar la ejecución de alguna asignación prodecural, para ello se usan las temporizaciones. El retardo requerido se especifica mediante el símbolo # seguido de las unidades del retardo.

En las temporizaciones existen dos tipos diferentes de asignaciones procedurales:

- Con bloqueo (blocking procedure): La asignación se realiza antes de proceder con la siguiente, por lo tanto, la asignación se realiza en el orden secuencial establecido. A continuación se muestra un ejemplo:

```
initial
begin
   a = 5;
   #1 a = a + 1;
   b = a + 1;
end
   // Resultado: a = 6 y b = 7
```

- Sin bloqueo (non-blocking procedure): El termino se evalua en el intante inicial, pero hasta que no ha finalizado el tiempo de retardo no se asigna. A continuación se muestra un ejemplo:

```
initial
begin
   a = 5;
   #1 a <= a + 1;
   b = a + 1;
end
   // Resultado: a = 6 y b = 6
```

3.5 Eventos

Las asignaciones procedurales pueden ser controladas por el cambio de una variable en un proceso *always*, esto se denomina control por evento. Para realizar este control se utiliza el carácter @ seguido del evento que permitirá la ejecución de la asignación.

Es posible distinguir dos tipos de eventos:

- Eventos de nivel: El cambio de valor de una o de un conjunto de variables controla la asignación. A continuación se muestra un ejemplo:

```
always @(A or B)
   C = A + B;
```

  Si se alguna de las dos variables (A o B) cambia, se realiza la asignación descrita.

- Evento de flanco: La combinación de flancos de subida (de 0 a 1) o de bajada (de 1 a 0). Para determinar que se están usando la combinación de flancos de subida se usa la palabra “posedge”, y para el caso de los flancos de bajada se usa la palabra “negedge”. Un ejemplo de este tipo de eventos es el siguiente:

```
always @(posedge clk)
   C <= A + 1;
```

3.6 Parámetros

Los parámetros equivalen a las denominadas contantes de otros lenguajes. Su definición solo puede realizarse en el interior de un módulo. Su estructura es la siguiente:

```
parameter nombre = valor;
```

Una aplicación frecuente de los parámetros suele ser la definición del periodo de un reloj. También suele usarse para definir el tamaño de un bus de datos.

3.7 Jerarquía

Como se ha expuesto anteriormente, el identificador *module* se utiliza para la definición de diseños, estos pueden utilizarse para construir diseños más complejos combinando los diferentes módulos con funcionalidad sencilla, dando módulos más complejos. Su estructura es la siguiente:

```
<nombre módulo> <nombre instancia> (lista de puertos);
```

Existen dos maneras diferentes para conexionar los módulos:

- Conexionado por orden: La asignación se realiza de forma implícita, donde cada señal se corresponde con el puerto del modulo. Un ejemplo de esta conexión es:

  ```
  module select(i1, i1, o1)

  . . . . .

  mux muxOrden (i1, i2, o1);
  ```

- Conexionado por nombre: La conexión se realiza de forma explícita, relacionando la señal con el puerto del modulo explícitamente. Para conexionar estas señales y puertos es necesario utilizar la siguiente sintaxis: `.(puerto) señal`

  Un ejemplo de esta interconexión es:

  ```
  mux muxNombre (.a(i1), b(i2), c(o1));

  a →i1        b → i2          c →o1
  ```

4. Funciones

[7] Las funciones son similares a las rutinas en cualquier otro lenguaje de programación, donde tienen argumentos de entrada y se puede dar la posibilidad de tener un elemento de salida. En este lenguaje existen algunas restricciones a tener en cuenta:

- No puede contener control de tiempo.
- Solo puede retornar una salida.
- Solo puede modelar lógica combinacional.
- No se puede invocar una tarea dentro de la función, pero si otra función.

La definición de las funciones debe estar incluida en la definición de los módulos de un diseño. Su estructura es la siguiente:

```
function <tamaño_retorno> <nombre>;
   <argumentos>
   <declaraciones>
   <funcionalidad>
endfunction
```

5. Tareas

Las tareas tienen la posibilidad de contener argumentos de todos los tipos (in, out, inout) pero no retorna ningún valor. Su finalidad es reducir el código del diseño y sus llamadas se realizan en tiempo de compilación. Sus características son las siguientes:

- Se define en el modulo en el que se utiliza. Otra posibilidad es definirlo en un fichero a parte e incluirlo con la palabra reservada *include.*
- Puede invocar a otras tareas o funciones.
- Puede contener control de tiempo.
- Carece de límite para sus entradas y salidas.

- Las variables declaradas son locales de dicha tarea.
- Las tareas pueden usar y/o asignar valores a cualquier señal declarada como global.
- Puede modelar lógica combinacional o secuencial.
- La llamada a una tarea no se puede utilizar dentro de una expresión.

Su estructura es la siguiente:

```
task <nombre>;
   <argumentos>
   <declaraciones>
   <funcionalidad>
endtask
```

6. Funciones del sistema

[15] A continuación se muestran algunas de las funciones del sistema más comunes. Estas funciones no son sintetizables y están enfocadas en la construcción de bancos de pruebas. Para declarar estas funciones tienen que comenzar por el carácter '$' además de estar declaradas dentro de un bloque:

1. **$time**

   Esta función devuelve el tiempo actual de la simulación. `$time`

2. **$display**

   Imprime por pantalla un mensaje cuando se ejecuta su sentencia. El mensaje declarado debe estar entre comillas, seguido de la variable o lista de variables a imprimir junto con su formato. Al final del mensaje se introduce un retorno de carro. A continuación se definen los diferentes formatos y se muestra un ejemplo:

%b: binario	%c: carácter	%d: decimal
%h: hexadecimal	%o: octal	%:s: cadena de caracteres

Un ejemplo es:

```
$display (“Valor variable a %d, valor b %d”, A, B);
```

3. **$monitor**

   Esta función tiene la misma estructura que *$display* con la única diferencia de que en este caso, las variables salen por pantalla si cambian su valor.

4. **$monitoroff**

   Esta función detiene la monitorización de las variables que ejecuta la función *$monitor*. No necesita variables.

5. **monitoron**

   Realiza la función opuesta a *$monitoroff*, habilitando la monitorización de las variables que ejecuta *$monitor*. No necesita lista de variables.

6. **$write**

   Esta función es muy similar a *$display,* con la única diferencia de que en esta no existe el retorno de carro.

7. **$fopen**

   Esta función permite arir ficheros, retornando en una variable definida de tipo *integer* el identificador del fichero. Un ejemplo es:

   ```
   id = $fopen (“arc.txt”); // crea el fichero arch.txt
   ```

8. **$fclose**

   Esta función permite cerrar ficheros, el fichero a cerrar se especifica mediante el identificador previamente asignado.

9. **$fdisplay**

   Realiza una función similar a la función *$dislay* con la diferencia del lugar donde se imprime el mensaje, en este caso, en el fichero. Para ello, es necesario especificar el identificador del mismo antes de especificar el mensaje.

   ```
   $fdisplay (id, “Valor a %d”, A)
   ```

10. $fwrite

Esta función es similar a *$fdisplay* con la única diferencia que en esta función no existe retorno de carro.

11. $finish

Esta función indica el final de la simulación.

12. $ramdom

Esta función toma un valor entero de 32 bits de forma aleatoria cada vez que se ejecuta. Se suele suministrar una variable como semilla para la generación del número.

6.1 Directivas para el compilador

Verilog ofrece un conjunto de directivas para el compilador, dando la posibilidad de obtener diseños diferentes a partir de una misma descripción. A continuación se explicaran las más comunes.

La estructura para definir una directiva es la siguiente:

```
‘directiva <nombre> <valor>
```

6.1.1 Define

Esta directiva permite definir un valor.

6.1.2 Include

Como su nombre indica, esta directiva permite incluir un fichero, el cual puede contener, por ejemplo, la definición de otros módulos.

6.1.3 Ifdef

Esta directiva permite compilar un diseño siempre y cuando se haya definido el símbolo al que hace referencia la misma.

Es necesario el uso de la directiva *‘define.* A continuación se muestra un ejemplo:

```
'define SIMULATION
.....
```

```
always @(posedge clk)
    'ifdef SIMULATION
       data <= A;
    'else
       data <= B;
    'endif
```

6.1.4 Timescale

Esta directiva permite definir las unidades de tiempo con las que se va a trabajar. A continuación se muestra la estructura de esta directiva:

```
‘timescale <unidad de tiempo> / <resolucion>
```

Donde la unidad de tiempo es mayor o igual a la resolución, y los valores que pueden tomar los enteros son 1, 10, 100.

Las posibles unidades de medidas son: ‘s’, ‘ms’, ‘us’, ‘ns’, ‘ps’ y ‘fs’.

7. Simulación

[4] La simulación es el proceso de verificar las características funcionales de los modelos en cualquier nivel de abstracción. Utilizamos herramientas para simular los modelos de hardware.

Para probar si el código RTL cumple con los requisitos funcionales de la especificación, debemos ver si todos los bloques RTL son funcionalmente correctos. Para lograr esto necesitamos escribir un banco de probas, que genere el reloj, y los vectores de prueba requeridos.

7.1 Banco de pruebas

La finalidad de los bancos de pruebas es verificar el correcto funcionamiento de un diseño. Su desarrollo es tan complejo como la realización de un diseño a verificar. La ventaja de estos elementos es la posibilidad de no tener que ser sintetizable.

Para describirlo es necesario tener en cuenta las especificaciones del diseño, en la que quedan reflejadas las funciones del diseño, y por lo tanto, las funciones a verificar.

7.1.1 Estructura

La estructura de un banco de pruebas se compone de tres elementos básicos.

- **Módulo DUT:** (Desing Under Test) Diseño a verificar.
- **Módulo test:** Este modulo es el encargado de activar las señales de entrada del modulo DUT y de analizar las salidas que produce.
- **Módulo tb:** Este módulo integra los módulos DUT y test. Se caracteriza por no tener ni entradas ni salidas, es decir, este módulo corresponde con una declaración estructural del conexionado de los módulos.

7.1.2 Generación de estímulos

Para la generación de estímulos se utilizan los procesos *initial* y *always*. La elección de estos procesos dependerá de las características de las señales a generar.

Dependiendo de la naturaleza de las señales, éstas se pueden generar de forma síncrona o asíncrona.

- **La generación síncrona** se basa en el uso de retardos para activar y desactivar señales.
- **La generación asíncrona** se basa en el uso de eventos, disparados por flancos para activar y desactivar señales

.

8. Sintetizabilidad

[14] En Verilog, disponer de un diseño que simule correctamente es relativamente fácil. No obstante, si lo que se desea es que un diseño sea

sintetizable deben respetarse ciertas normas, algunas de ellas se exponen a continuación:

- **No emplear retardos.** El uso de retardos provoca que los diseños no sean portables debido a que la mayoría de las herramientas suelen ignorar estos retrasos.
- **No modificar una variable en dos procesos diferentes.**
- **En asignaciones procedurales son necesarias las listas de sensibilidad.** Todas las variables de las que dependa la asignación deben aparecer en la lista de sensibilidad.
- **Ultimo valor asignado.** Si no se define completamente una variable, el lenguaje supone que conserva el último valor asignado, sintetizándose un biestable que almacene su estado.
- **Los sintetizadores tienen ciertas restricciones.** Los sintetizadores no admiten procesos *initial*. Estas herramientas tampoco suelen admitir los operadores división y resto.
- **Tener en cuenta el problema *rate condition*.** Este problema se produce cuando dos asignaciones se realizan al mismo instante pero una depende de la otra, y por lo tanto, el orden de ejecución es importante.
- **Sentencias no sintetizables.** Desde que este lenguaje fue creado, y con él diferentes herramientas para su síntesis existe un conjunto de sentencias que no se pueden sintetizar en ninguna herramienta.

  Pero con el paso de los años las herramientas de síntesis se han desarrollando de diferentes formas, dando como resultado herramientas con diferentes sentencias no sintetizables. Es decir, cada herramienta tiene su conjunto de sentencias no sintetizables. Este hecho es motivo de competición entre las diferentes herramientas, que luchan por ser la herramienta con menos sentencias sintetizables.

Capítulo IV: Comparación

En este capítulo, después de estudiar los lenguajes HDL, se realizará la comparación de los lenguajes VHDL y Verilog.

1. Introducción

Desde la creación de Verilog, se debate cuál de los dos lenguajes es el más apropiado para diseñar. La comunidad científica ha llegado a la conclusión de que no existe una razón importante para decantarse por un lenguaje u otro.

Por un lado, el lenguaje VHDL es creado como un lenguaje para resolver las necesidades del diseño en su época, y otras futuras, por lo tanto, cuando este lenguaje fue pensado para ser duradero y dar la posibilidad de crear diseños con mayor dificultad. El resultado es un lenguaje más rígido y con una sintetización en el código que lo hace más legible, aunque sus reglas de diseño obligan a diseñar de forma, a veces poco ágil.

Por otro lado, Verilog nace para cubrir la necesidad de mejorar un flujo de diseño y por lo tanto, es un lenguaje que cubre todas las necesidades de diseño y simulación. El resultado es un lenguaje simple y flexible, fácil de aprender aunque tiene ciertas limitaciones.

A continuación se van a enumerar una serie de diferencias entre ambos lenguajes dando la posibilidad a los usuarios de ver las ventajas y desventajas de los lenguajes, y así, poder elegir el más adecuado para cada diseño.

2. Estudio comparativo teórico

En este apartado se enumerarán las distintas características más destacadas de ambos lenguajes.

2.1 Características

2.1.1 VHDL

- Se presta en más niveles abstractos.

- Fuertemente tipado, por lo que los errores de sintaxis se encuentran con mayor facilidad.
- Este lenguaje tiene la capacidad para definir tipos personalizados.
- Extremadamente detallado en codificación.
- Cuenta con lista de sensibilidad. Si falta alguna señal es esta lista puede causar grandes diferencias entre la simulación y la síntesis.
- Para realizar operaciones con señales de diferentes tipos (aunque estén fuertemente relacionados) es necesario convertir las señales al mismo tipo.
- Este lenguaje o distingue entre mayúsculas y minúsculas. Los usuarios pueden cambiar el caso, siempre y cuando los caracteres en el nombre estén en el mismo orden.

2.1.2 Verilog

- Se presta hasta el nivel de puertas.
- Débilmente tipado, por lo que el código es más propenso a errores debido a la combinación accidental de diferentes tipos de señal.
- No tiene soporte para tipos personalizados.
- Descripciones de bajo nivel más cera del hardware real.
- Carece de lista de sensibilidad.
- Es un lenguaje compacto. La conversiones del lenguaje son similares al leguaje C. Existe la posibilidad de mezclar y combinar las señales de tipos parecidos sin convertir los datos.
- Sensibilidad a las mayúsculas. Diferencia entre mayúsculas y minúsculas.

2.2 Comparación a nivel de código

En primer lugar se van a desarrollar las características más destacables del lenguaje VHDL, y seguidamente se expondrán las del lenguaje Verilog.

2.2.1 VHDL

- Este lenguaje es un poco complicado de aprender debido a que está basado en derivaciones de los lenguajes ADA y Pascal.

- Se pueden desarrollar diseños digitales de distintos estilos como el funcional, flujo de datos y de estilo estructural. Este lenguaje le permite al programador crear un diseño de acuerdo a sus necesidades o a la pericia que tenga para llegar a la resolución de cualquier diseño digital. El único inconveniente es que se necesitan más líneas de código para resolver los diferentes diseños digitales.

- Para obtener una resolución de cualquier diseño de manera más fácil y rápida se recomienda desarrollar la programación del diseño digital estudiando su estructura, comportamiento y dividiéndolos en pequeños procesos, para tener un código más claro y mejor estructurado.

- Se necesita declarar la entidad y la arquitectura del diseño digital a desarrollar de manera obligatoria para que se pueda ejecutar el programa.

- Es obligatorio la declaración de la entidad (ENTITY) y de la arquitectura (ARQUITECTURE) del diseño a desarrollar para que se puede ejecutar el programa, de otra manera, no se podrá.

- Presenta una sola forma de asignar valores.

- La declaración de las puertas básicas conocidas (AND; OR; NOT; etc) se realizan llamándolas por su debido nombre.

- Las estructuras de selección (IF, CASE, WITH, etc) en su sintaxis de declaración tienen forma similar a las de un lenguaje de programación como en pascal.

- A través de la palabra reservada del programa "Event" se pueden generar diferentes eventos o pulsos de reloj.

- Se pueden declarar procesos y funciones.

2.2.2 Verilog

- Es de fácil aprendizaje debido a que es una derivación del lenguaje C y todo ingeniero lo conoce.

- Se pueden desarrollar los ejercicios de la misma manera que presenta VHDL, pero con la ventaja que se necesita menos líneas de códigos para la resolución del mismo diseño digital debido a que el lenguaje es más abstracto.

- Al desarrollar la programación de los diseños estudiando su estructura, su comportamiento y dividiéndolos en pequeños procesos se obtendrá la resolución del diseño de manera fácil y rápida.

- No se necesita una entidad externa para declarar los datos ya que vienen incluido dentro de la misma arquitectura en este caso llamado MODULE (modulo) en donde se realiza el desarrollo del diseño.

- Presenta dos formas de asignar valores.

- La declaración de las puertas lógicas básicas conocidas se las realiza llamándolas por su nombre o a través de símbolos.

- Las estructuras de selección (IF; CASE; WITH; etc) en su declaración es muy semejante a la declaración del lenguaje C.

- A través de la palabra reservada "Posedge y Negedge" se pueden generar diferentes eventos o pulsos de reloj.

- Se pueden declarar módulos y sub módulos.

2.3 Comparación a nivel de simulación

A continuación se muestra una comparación de los lenguajes a nivel de simulación y a nivel de tiempo de simulación.

Algunas de las características de este campo son iguales para ambos lenguajes:

- Se puede observar el comportamiento del diseño en distintos instantes de tiempo.
- Fácil de manipular los datos de entrada y salida del diseño.
- Se utiliza el mismo sistema para la simulación en ambos lenguajes.

Además de estas características en común, también tienen algunas diferencias:

2.3.1 VHDL

- Debido a que lleva más código para la solución de un diseño digital así mismo su tiempo de simulación va a aumentar pero en pocos pequeños ms.
- No define ningún tipo de control de simulación o capacidades de monitoreo dentro del lenguaje. Estas capacidades son herramientas dependientes.

2.3.2 Verilog

- Debido a que lleva menos código para la solución de un mismo diseño digital así mismo su tiempo de simulación va a disminuir pero en pocos pequeños ms.
- Define un conjunto de control básico de simulación en el lenguaje.

2.4 Comparación a nivel de librerías

En este apartado se comparan los lenguajes en base a su capacidad de almacenar las distintas entidades compiladas, arquitecturas, paquetes y configuraciones:

2.4.1 VHDL

- Se necesita declarar las librerías de manera obligatoria para que pueda ser ejecutadas las líneas de código.

2.4.2 Verilog

- No necesitan declararse que ya se ejecutan automáticamente. Carece de librerías.

2.5 Comparación a nivel de software

En el mercado existen una gran cantidad de herramientas para la creación de diseños con estos lenguajes. Al ser dos lenguajes estandarizados, estas herramientas coinciden en algunas características:

- Son de fácil instalación, manipulación y aprendizaje.
- La mayoría de herramientas pueden manipular ambos lenguajes.

3. Estudio comparativo con un caso practico

Para este apartado se han realizado varios diseños para ambos lenguajes, con el fin de poder analizar las diferencias y similitudes del lenguaje de forma más práctica.

Los diseños realizados son los módulos básicos que componen una Unidad Aritmético – Lógica.

Los diseños realizados están realizados con entradas de 4 bits. Los diferentes diseños son: puertas lógicas AND, OR, NOT, XOR; también se han diseñado un sumador con acarreo, un restador en complemento A2 y un multiplexor 4 a 1 utilizando módulos de multiplexor 2 a 1.

A continuación se van a describir los diseños y se van a comparar los diferentes códigos de los lenguajes HDL, para posteriormente, comentar sus características.

3.1 Puertas lógicas

Se ha creado un diseño en el que se realizan varias operaciones lógicas con dos datos de entrada (a, b) de 4 bits. En este diseño se calculan las siguientes operaciones:

- a AND b.
- a OR b.
- NOT a.
- a XOR b.

A continuación se muestra el diseño implementado en VHDL:

```
library IEEE;                                   --*0*
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

ENTITY puertas_logicas IS                          -- *1*
  PORT (a : IN std_logic_vector(3 DOWNTO 0);
        b : IN std_logic_vector(3 DOWNTO 0);
   sal_and : OUT std_logic_vector(3 DOWNTO 0);
   sal_or : OUT std_logic_vector(3 DOWNTO 0);
   sal_nota : OUT std_logic_vector(3 DOWNTO 0);
   sal_xor : OUT std_logic_vector(3 DOWNTO 0));
END puertas_logicas;

ARCHITECTURE puertas OF puertas_logicas IS          --*2*
  BEGIN
    PROCESS (a, b) IS
    BEGIN
    sal_and    <=    std_logic_vector(UNSIGNED(a)    and
UNSIGNED(b));
    sal_or     <=     std_logic_vector(UNSIGNED(a)     or
UNSIGNED(b));
    sal_nota <= std_logic_vector(UNSIGNED(not a) );
    sal_xor    <=    std_logic_vector(UNSIGNED(a)    xor
UNSIGNED(b));
    END PROCESS;
END puertas;                                       --*3*
```

Seguidamente se muestra el diseño implementado en Verilog:

```
module puertas_logicas (a, b, sal_and, sal_or, sal_nota,
sal_xor);                                          //*4*
input [3:0]a, b;
output [3:0]sal_and, sal_or, sal_nota, sal_xor;
assign sal_and = a & b;
assign sal_or = a | b;
assign sal_nota = ~a;
assign sal_xor = a ^ b;
endmodule                                          //*5*
```

Primeramente se va comparar la estructura del diseño en ambos lenguajes.

- Como se pude observar, en el lenguaje VHDL es necesario llamar a la librería *IEEE.* En este caso se usan dos paquetes de la librería, para poder definir los tipos de datos 'std_logic_vector' y para poder decir que la operación el valor no tiene signo 'unsigned'. *0*

- Se puede observar que en VHDL es necesario crear una entidad y una arquitectura para implementar esa entidad. En la entidad se declaran todas las entradas y salidas del diseño, indicando su tipo de dato y si son señales de entrada, de salida o ambas. Después de crear la entidad se crea la arquitectura que la usa. En la arquitectura se realizan las diferentes operaciones.*1* *2*

  Mientras que en Verilog no es necesario nada de lo anterior, simplemente se crea un modulo indicando todas las señales que interactúan con el exterior para posteriormente indicar si son señales de entrada, salida o ambos y su rango. También puede indicarse el tipo de dato pero si no se especifica las señales serán de tipo *wire*. *4*

- Para indicar el tamaño de los datos en VHDL es necesario utilizar una palabra reservada (DOWNTO o TO) que indican si se recorre el array de forma ascendente o descendentemente. Mientras que en Verilog se indica el tamaño con dos números separados por ':'.

- Para asignar las señales a las salidas en VHDL simplemente se utiliza el símbolo '<=' o '=>', mientras que en Verilog se utiliza la palabra reservada 'assign' delante de la asignación.

- Otra diferencia es el cierre del bloque. En VHDL se usa la palabra reservada 'END' seguida del nombre que se le ha otorgado a la arquitectura previamente. Mientras que en Verilog se finaliza con la palabra reservada 'ENDMODULE'. *3* *5*

- Por último, se puede observa que el número de líneas del código en VHDL (21 líneas), es mayor que el número de líneas en Verilog (8 líneas) lo que puede provocar una diferencia en el tiempo de ejecución.

Posteriormente se va proceder a explicar la metodología de cada operación para luego comparar el código y la simulación de cada una de ellas en ambos lenguajes.

3.1.1 Puerta lógica AND

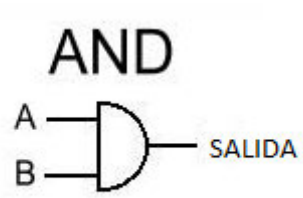


Ilustración 4-1. Esquema puerta AND

La puerta lógica AND implementa la suma lógica, de forma que la salida será '1' o True cuando las dos entradas sean '1' o True.

La sentencia para realizar esta operación en VHDL es:

```
sal_and<= std_logic_vector(UNSIGNED(a) and UNSIGNED(b))
```

Donde se le asigna a la señal sal_and el valor de realizar la operación AND entre las señales a y b. Esta salida será de 4 bits, como las entradas. Como se puede observar, es necesario convertir el

resultado de la operación para poder asignársela a la variable *sal_and,* para que coincida con el tipo de dato con el que se ha definido esta señal. También se puede observar que es los valores están indicados como *'UNSIGNED'* esto quiere decir que no tiene signo, es decir, que todos los valores son positivos. Es recomendable utilizar esta palabra reservada para no tener ningún tipo de inconveniente.

Esta sentencia refleja la operación lógica AND, la cual se realiza bit a bit. Por lo tanto, si los dos operandos tienen un '1' en la misma posición, en el resultado de la operación el bit de esa posición también será '1'. En cualquier otro caso será '0'.

La sentencia donde se realiza esta operación en Verilog es la siguiente:

```
assign sal_and = a & b;
```

En esta sentencia se le asigna a la señal '*sal_and'* el resultado de la operación lógica AND.

Como se puede observar, en Verilog no es necesario indicar el tipo de dato de la operación, ni tampoco es necesario indicar si el dato tiene signo o no.

Una de las grandes diferencias es el símbolo utilizado en cada lenguaje para indicar la operación: en VHDL se utiliza el símbolo '*and'* para indicar la operación, y en Verilog se utiliza el símbolo *'&'*.

También es diferente el modo de asignar el valor a la variable declarada previamente. En VHDL se usa el símbolo '<=' mientras que en Verilog se utiliza la palabra reservada *'assign'* y el símbolo *'='*.

A continuación se muestra un ejemplo de la simulación en ambos lenguajes.

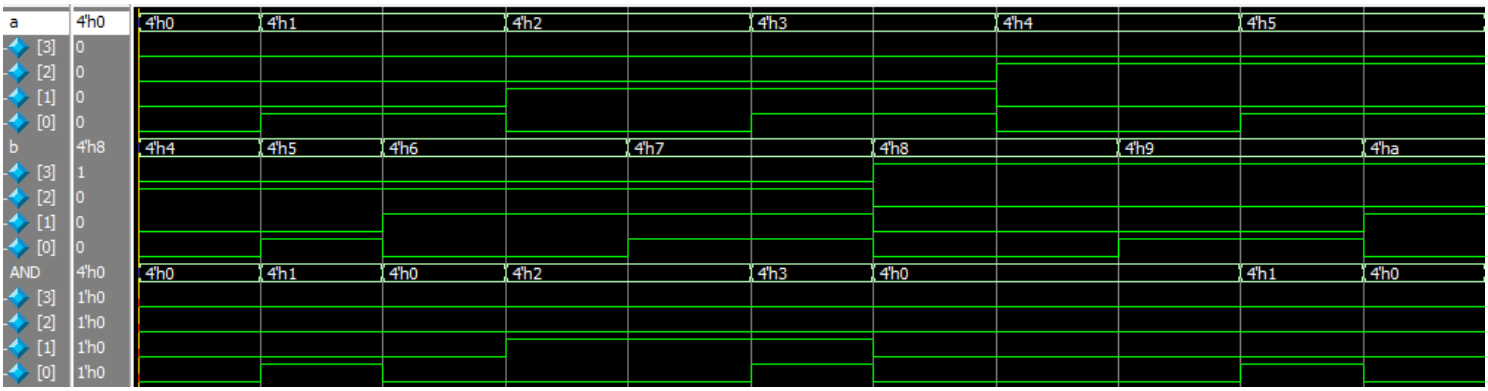


Ilustración 4-2. Ejecución de la puerta AND

En esta imagen se muestran varios ejemplos de la salida de la operación lógica AND. La salida es igual en ambos lenguajes.

3.1.2 Puerta OR

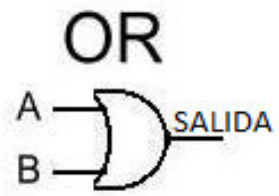


Ilustración 4-3. Esquema puerta OR

La puerta lógica OR implementa la disyunción lógica, de forma que la salida solo será '0' o False en caso de que los dos operandos sean '0' o False, en cualquier otro caso, la salida será '1' o True.

La sentencia para realizar esta operación en VHDL es:

```
sal_or<= std_logic_vector(UNSIGNED(a) or UNSIGNED(b));
```

Donde se le asigna a la señal sal_or el valor de realizar la operación OR entre las señales a y b. Esta salida será de 4 bits, como las entradas. Como se ha comentado anteriormente, es necesario convertir el resultado de la operación.

La sentencia para realizar esta operación en Verilog es:

```
assign sal_or = a | b;
```

Como se ha comentado anteriormente, en Verilog no es necesario indicar ningún tipo de dato ni hacer ninguna conversión.

Se puede observar que el operando es diferente, siendo en VHDL *'or'* y en Verilog *'|'*.

A continuación se muestra un ejemplo de la simulación en ambos lenguajes, al realizar la misma operación, el resultado es el mismo.

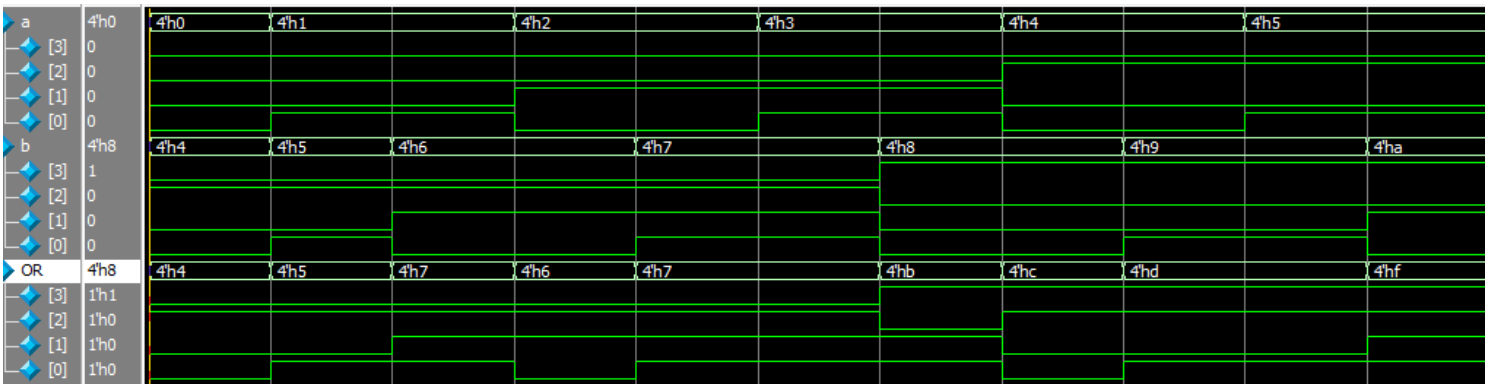


Ilustración 4-4. Ejecución de ejemplos de la puerta OR

3.1.3 Puerta lógica NOT

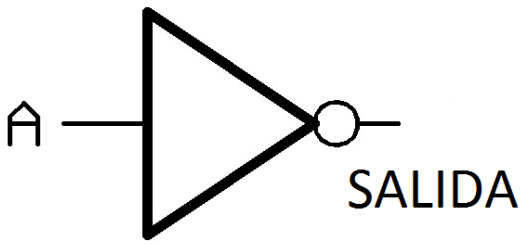


Ilustración 4-5. Esquema de la puerta NOT

La puerta lógica NOT implementa la negación lógica. De forma que cuando la entrada sea '1' o True, la salida será '0' o False, ocurriendo lo mismo para el caso de que la entrada sea '0'.

La sentencia para realizar esta operación en VHDL es:

```
sal_nota<= std_logic_vector(UNSIGNED(not a))
```

Donde se le asigna a la señal sal_nota el valor de realizar la operación NOT en la señal a. Esta salida será de 4 bits, como su entrada.

Al igual que en los otros casos, se recomienda decir que el resultado de la operación no tiene signo; además de ser necesario indicar el tipo de la misma.

La sentencia que realiza esta operación en Verilog es:

```
assign sal_nota = ~a;
```

Como se ha indicado anteriormente, las diferencias son la ausencia de indicar el tipo de dato, otra diferencia es la forma de asignar los valores.

Como se puede ver, los símbolos para realizar la operación son diferentes. En VHDL se usa la palabra reservada '*NOT*' mientras que en Verilog se utiliza el símbolo *'~'*.

A continuación se muestra un ejemplo de la simulación en ambos lenguajes, al realizar la misma operación, el resultado es el mismo.

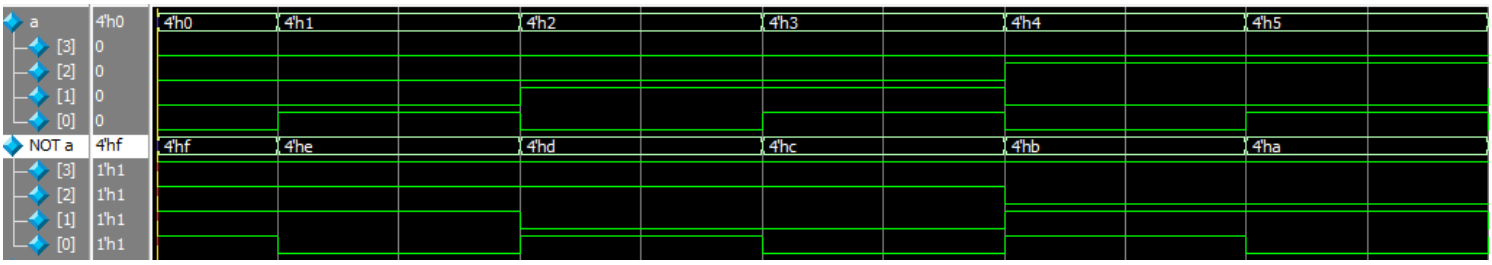


Ilustración 3-6. Ejecución de ejemplos usando la puerta NOT

3.1.4 Puerta lógica XOR

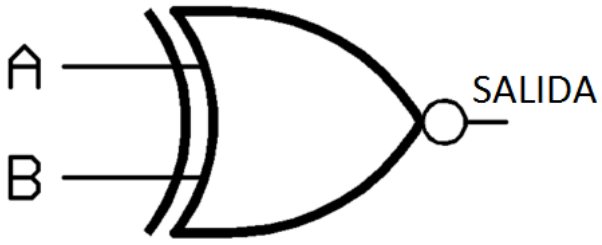


Ilustración 4-7. Esquema de la puerta XOR

La puerta lógica XOR implementa la disyunción lógica exclusiva, de forma que la salida solo será '1' o True en caso de que solo uno de

los operandos sea ‘1’ o True, en cualquier otro caso, la salida será ‘0’ o False.

La sentencia para realizar esta operación en VHDL es:

```
sal_xor<=std_logic_vector(UNSIGNED(a) xor UNSIGNED(b));
```

Esta sentencia sigue la misma estructura que las otras sentencias de operaciones con puertas lógicas en este diseño.

La sentencia en Verilog es:

```
assign sal_xor = a ^ b;
```

A parte de las diferencias mencionadas anteriormente se puede ver que el operando, al igual que en las otras puertas lógicas, es diferente en los lenguajes. En VHDL se utiliza la palabra reservada *‘XOR’* mientras que en Verilog se utiliza el símbolo *‘^’*.

A continuación se muestra un ejemplo de la simulación en ambos lenguajes, al realizar la misma operación, el resultado es el mismo.

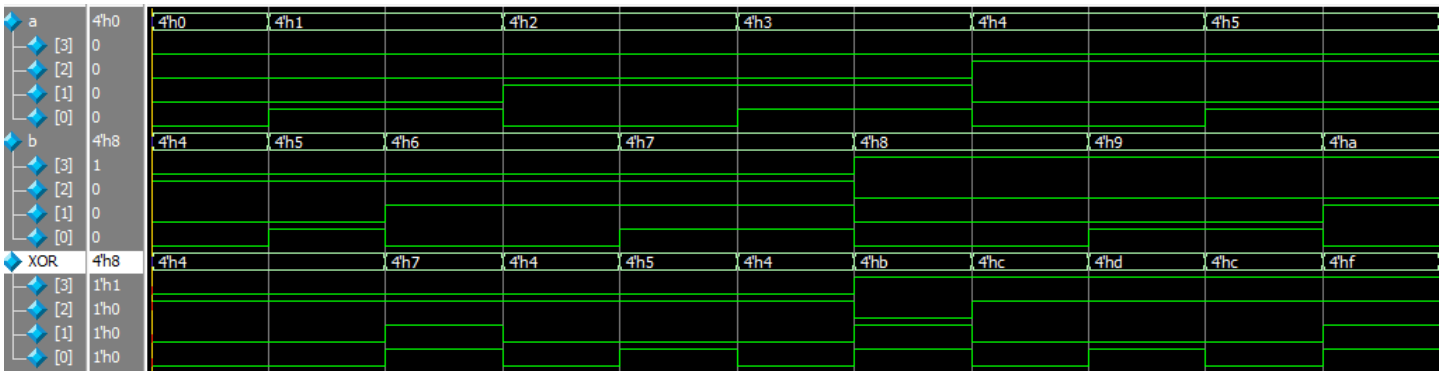


Ilustración 4-8. Ejecución de ejemplos usando la puerta XOR

3.2 Sumador

Se ha creado un diseño en el que se suman dos valores y se obtiene su resultado. Los valores de entrada deben ser de 4 bits, y la salida se requiere que sea del mismo tamaño

.

El código en VHDL es:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

ENTITY sum IS
  PORT (a : IN std_logic_vector(3 DOWNTO 0);
        b : IN std_logic_vector(3 DOWNTO 0);
      acarreo : OUT std_logic;
       salida : OUT std_logic_vector(3 DOWNTO 0));
  END sum;
ARCHITECTURE synth OF sum IS
  BEGIN
    PROCESS (a, b) IS
    VARIABLE suma: std_logic_vector(4 DOWNTO 0);
    BEGIN
      suma  :=  std_logic_vector(UNSIGNED('0'  &  a)  +
UNSIGNED('0' & b));
  salida <=suma(3 DOWNTO 0);
  acarreo <= suma(4);
    END PROCESS;
END synth;
```

El código en Verilog es:

```
module sumador (a, b, y, acarreo);
input [3:0]a, b;
input acarreo;
output [4:0]y;
wire [4:0] suma;
assign y = a + b;
assign y = suma[3:0];
assign acarreo = suma[4];
endmodule;
```

A continuación se van a comparar ambos códigos para ver las diferencias de ambos lenguajes:

- Primeramente se observa que en VHDL es necesario usar la librería *'IEEE'*, mientras que en Verilog no es necesario.

- También se puede ver que en VHDL es necesario crear una Entidad para poder, posteriormente, instanciarlo en la arquitectura a crear. En la entidad se definen las señales de entrada, salida y de entrada-

salida, también se le otorga un tipo de dato a cada una de ellas, si fuera necesario se indica el tamaño de la misma. Mientras que en Verilog solo en necesario nombrar las señales al crear el modulo, y posteriormente indicar el tipo de señal y su tamaño.

- La mayor diferencia en la implementación de este diseño es la forma en la que se realiza la operación: En VHDL, para poder realizar la suma correctamente sin perder ningún bit, además de mantener la salida de 4 bits y de esa manera tener el resultado correcto, es necesario crear otra señal más de tamaño 1 bit, en la cual se almacenará el quinto bit, en caso de que el resultado de la suma no quepa en 4 bits. Para ello es necesario crear una variable de tamaño 5, también es necesario expandir las señales de entrada con un bit más a la izquierda con valor '0', convirtiendo el dato de tamaño 4 bits a tamaño 5 bits y de esta manera, poder realizar la operación. Una vez realizada la operación es necesario asignar a la señal de resultado la operación, indicando el rango que se le va a asignar, este rango debe coincidir con el tamaño de la señal. Posteriormente se le asigna el bit sobrante a la variable auxiliar, también indicando el bit de la variable.

  Por otro lado, en Verilog también es necesario crear una variable intermedia de mayor tamaño, de tal manera que no se pierda ningún dato de la operación. Además, también es necesario crear una señal de salida en la que se contendrá el bit extra cuando sea necesario. A diferencia de VHDL se puede realizar la suma y asignarla a la señal del resultado sin necesidad de expandir ningún dato, ya que Verilog admite este tipo de asignaciones. Posteriormente, se le asignará a las señales de salida los datos, llamando a la variable intermedia y especificando el rango a designar.

- Otra de las grandes diferencias es la extensión del cogido en cada lenguaje, ocupando en VHDL 19 líneas y en Verilog 9 líneas. Lo que puede provocar una diferencia en el tiempo de simulación.

El resultado en ambos casos es el mismo. A continuación se muestra un ejemplo de la ejecución del diseño:

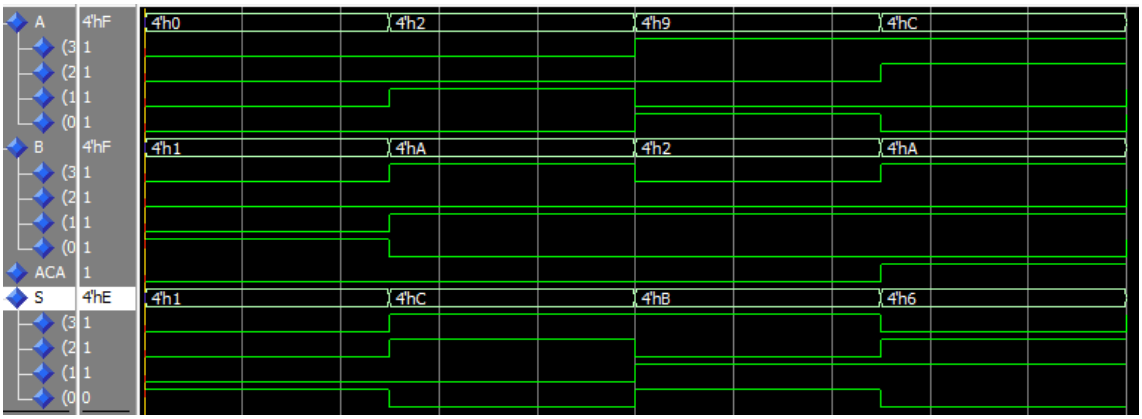


Ilustración 4-9. Ejecución de ejemplos usando el sumador

3.3 Restador en complemento A2

Se ha creado un diseño para implementar la resta de dos valores. En binario la resta como tan no existe, por lo que para realizar una resta de dos número se debe convertir el valor de menor tamaño numero a complemento a 2. Para ello se debe hacer el inverso del número y posteriormente sumarle un bit. Una vez se ha obtenido el complemento a 2 del dato se realiza la suma de ambos valores, y se obtiene la resta.

El código en VHDL ES:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
ENTITY res IS
  PORT (a : IN std_logic_vector(3 DOWNTO 0);
        b : IN std_logic_vector(3 DOWNTO 0);
       signo_salida : OUT std_logic;
        salida : OUT std_logic_vector(3 DOWNTO 0));
  END res;
ARCHITECTURE res_a2 OF res IS
  BEGIN
    PROCESS (a, b) IS
    VARIABLE suma: std_logic_vector(4 DOWNTO 0);
    VARIABLE dato: std_logic_vector(4 DOWNTO 0);
    BEGIN
   if (a >= b) then
```

```
        dato := std_logic_vector(UNSIGNED('0'&(not b))+
"00001");
        suma := std_logic_vector(UNSIGNED('0' & a) +
UNSIGNED(dato));
        salida <=suma(3 DOWNTO 0);
        signo_salida <= '0';
   else
        dato := std_logic_vector(UNSIGNED('0'&(not a))+
"00001");
        suma := std_logic_vector(UNSIGNED(dato) +
UNSIGNED('0' & b));
        salida <=suma(3 DOWNTO 0);
        signo_salida <= suma(4);
   END IF;
    END PROCESS;
END res_a2;
```

El código en Verilog es:

```
module restador_a2 (a, b, y, acarreo);
input [3:0]a, b;
output reg [3:0]y;
output reg acarreo;
reg [4:0] suma;

always@(a or b)
  begin
  if (a >= b)
  begin
    assign suma = ((~b)+"0001") + a;
    assign y = suma[3:0];
    assign acarreo = "0";
  end
  else
  begin
    assign suma = ((~a)+"0001") + b;
    assign y = suma[3:0];
    assign acarreo = suma[4];
  end
  end
endmodule
```

A continuación se van a comparar ambos códigos para ver las diferencias de ambos lenguajes:

- Como ya se ha mencionado anteriormente, en VHDL es necesario usar la librería *'IEEE'*, mientras que en Verilog no es necesario.

- En VHDL es necesario crear una entidad para indicar las señales de entrada, salida, o entrada-salida, además de indicar el tipo de dato de cada una y su rango. Mientras que en Verilog, al crear el modulo se indican las señales y posteriormente, en el código se indica el tipo de dato de cada una y su rango.

- Las operaciones requeridas por el modulo en VHDL se realizan en el '*BEGIN*' de la arquitectura, mientras que en Verilog, estas misma operaciones se realizan en el mismo modulo anteriormente mencionado.

- Para realizar el diseño especificado en VHDL han sido necesarias dos variables internas. En una de ellas se almacena el dato en complemento a dos y en la otra se realiza la suma del dato en complemento a dos y del otro dato de entrada, habiendo realizado previamente la expansión de los mismos para que el resultado coincida con el rango que se le ha asignado a esa variable.

  Mientras que en Verilog, solo es necesaria la creación de una variable interna, en la cual se realiza la suma del dato en complemento a dos y del dato restante.

  Después de la operación, en ambos casos es igual, se realiza la asignación de los 4 bits menos significativos a la salida, y a la salida auxiliar (signo_salida) se le asigna '0' en caso de que el primer dato sea mayor que el segundo y por lo tanto, la resta deba dar signo negativo. Y en el caso contrario se le asigna el bit más significativo ('1'), con el significado de que el signo del valor resultante es negativo.

- Otra de las diferencias es la creación de un proceso para controlar la ejecución de las sentencias que pertenecen a ese bloque. En VHDL se utiliza la palabra reservada *'PROCESS'* seguido la lista de

sensibilidad. Mientras que en Verilog se utiliza la palabra *'ALWAYS'* seguido del signo '@' y de la lista de sensibilidad, también existe la posibilidad de utilizar la palabra reservada *'INITIAL'* pero este tipo de proceso solo se ejecuta una vez al principio de la ejecución.

- Otra de las grandes diferencias es la extensión del cogido en cada lenguaje, ocupando en VHDL 28 líneas y en Verilog 21 líneas. Lo que puede provocar una diferencia en el tiempo de simulación.

  El resultado en ambos casos es el mismo. A continuación se muestra un ejemplo de la ejecución del diseño:

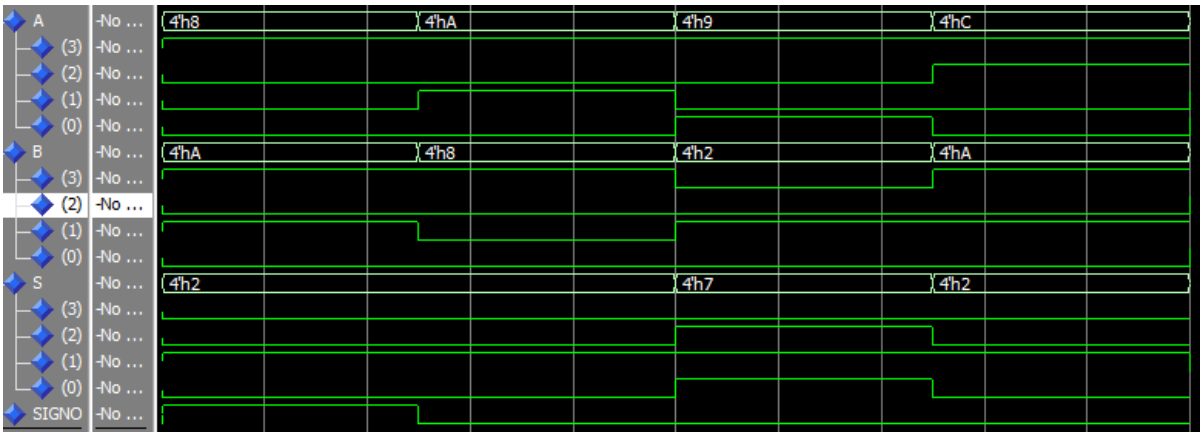


Ilustración 4-10. Ejecución de ejemplos usando el restador con complemento A2

3.4 Multiplexor 4 a 1 usando bloques de multiplexores 2 a 1

Un multiplexor 4 a 1 es la elección de un dato de entrada por encima de otro dependiendo del estado de una señal de selección. En este diseño se ha implementado un multiplexor 4 a 1 con bloques de multiplexores 2 a 1. El diseño sigue el siguiente esquema.

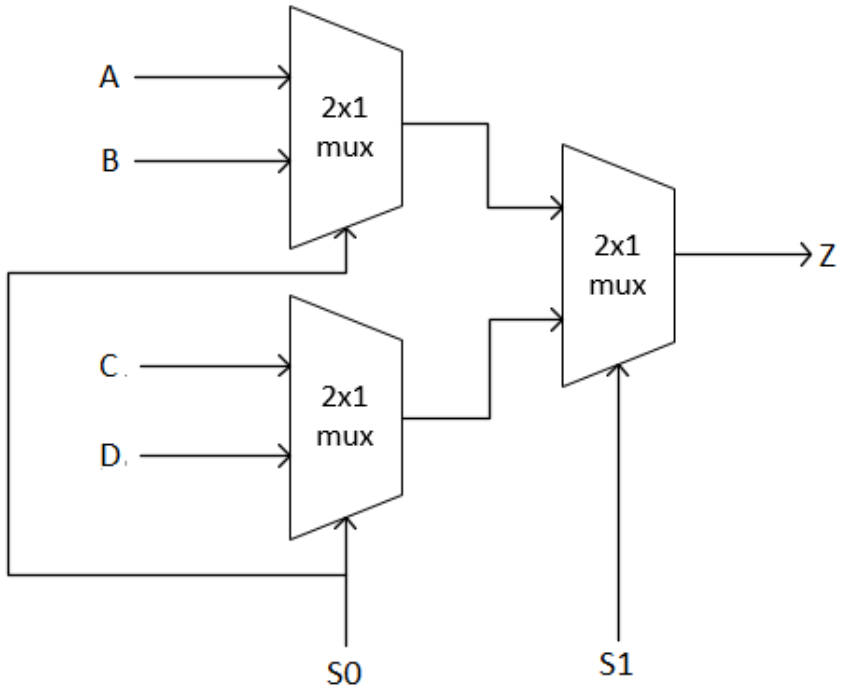


Ilustración 4-11. Esquema del multiplexor 4 a 1 usando bloques de multiplexor 2 a 1

Por lo tanto, el primer multiplexor tiene como salida *A* si *S0* vale '0', en caso contrario su salida va a ser *B*. Para el segundo multiplexor ocurre lo mismo con *C* y *D*. Y en el último multiplexor, dependiendo de si *S1* vale '1' o '0' se escogerá un valor u otro.

Primero se va a comparar el código del diseño del multiplexor 2 a 1, que será usado posteriormente en el multiplexor 4 a 1.

Código en VHDL de bloque mux 2 a 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity mux_2_1 is
  port(x,y : in STD_LOGIC_VECTOR(3 DOWNTO 0);
  s: in STD_LOGIC;
  z: out STD_LOGIC_VECTOR(3 DOWNTO 0));
  end mux_2_1 ;
architecture mux_2a1 of mux_2_1 is
  begin
  process (x,y,s) is
    begin
    if (s ='0') then
      z <= x;
    else
```

```
      z <= y;
    end if;
  end process;
end mux_2a1;
```

Codigo en verilog mux 2 a 1:

```
module mux_2_1 (sel, a, b, y);
input [3:0]a, b;
input sel;
output reg [3:0]y;
always @(a or b or sel)
  if (sel == 0)
    assign y = a;
  else
    assign y = b;
endmodule
```

A continuación se van a comparar lo códigos del multiplexor 2 a 1:

- Como se ha mencionado anteriormente, en VHDL es necesario invocar la librería *'IEEE'*.
- También se ha mencionado anteriormente, en VHDL es necesario crear la entidad, donde se definen las señales y se les indica su tipo y tamaño. Mientras que en Verilog esta acción se realiza en el mismo modulo donde posteriormente se realizaran las operaciones requeridas.
- Una de las diferencias de este código es el uso de la sentencia condicional 'IF'. En VHDL se utiliza la sentencia 'IF … THEN' mientras que en Verilog se utiliza únicamente 'IF' y si se deben realizar más de una sentencias en la condición se deben contener entre las palabras 'BEGIN' y 'END'.
- Otra de las diferencias entre ambos lenguajes es que en Verilog no se tiene que utilizar la palabra reservada 'END' para finalizar la estructura de control 'IF-ELSE' mientras que en VHDL es obligatorio finalizarlo con la sentencia 'END IF'.

- La diferencia más evidente a simple vista es la longitud de cada código. VHDL cuenta con 18 líneas de código mientras que Verilog cuenta con 10 líneas.

Ahora se van a comparar los códigos del multiplexor 4 a 1 usando el bloque de multiplexor 2 a 1.

Código en VHDL de mux 4 a 1:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity mux_4_1 is
  port(A,B,C,D : in STD_LOGIC_VECTOR(3 DOWNTO 0);
       S0,S1: in STD_LOGIC;
       Z: out STD_LOGIC_VECTOR(3 DOWNTO 0));
end mux_4_1 ;
architecture mux_4a1 of mux_4_1 is
 component mux_2_1
  port(x,y : in STD_LOGIC_VECTOR(3 DOWNTO 0);
     s: in STD_LOGIC;
     z: out STD_LOGIC_VECTOR(3 DOWNTO 0));
 end component;
 signal m1, m2: std_logic_vector(3 DOWNTO 0);
 begin
   c1: mux_2_1 port map(A,B,S0,m1);
   c2: mux_2_1 port map(C,D,S0,m2);
   c3: mux_2_1 port map(m1,m2,S1,Z);
end mux_4a1;
```

Codigo en Verilog mux 4 a 1:

```
module mux_4_1(sel0,sel1,a,b,c,d,out);
   input [3:0] a,b,c,d;
   input sel0, sel1;
   output [3:0]out;

   wire [3:0]mux1,mux2;
   mux_2_1 mux_1(sel0,a,b,mux1);
   mux_2_1 mux_2(sel0,c,d,mux2);
   mux_2_1 mux_3(sel1,mux1,mux2,out);
endmodule
```

Las diferencias entre los códigos son:

- Algunas diferencias se han mencionado anteriormente como la de la librería y la entidad.
- Una de las grandes diferencias de este diseño es que en VHDL es necesario instanciar los componentes que va a utilizar más adelante, indicando el nombre de las señales, su tipo y su tamaño. Mientras que en Verilog no es necesario instanciar previamente ningún componente para posteriormente usarlo.
- Otra de las diferencias se puede ver en el momento de llamar a los componentes: en Verilog se escribe directamente el nombre del bloque que se quiere utilizar seguido del nombre asignado a esa operación seguido de la lista de señales en el mismo orden que fueron designadas cuando se creó el bloque que se está poniendo en uso. Mientras que en VHDL primeramente se indica el nombre, seguido del bloque que se desea utilizar, seguido por las palabras reservadas 'PORT MAP' y por último, seguido por la lista de señales en el mismo orden del bloque.
- La última diferencia evidente es la longitud del código. Verilog cuenta con 10 líneas de código mientras que VHDL cuenta con 19 líneas.

El resultado en ambos códigos es el mismo. A continuación se muestra un ejemplo de la ejecución del diseño:

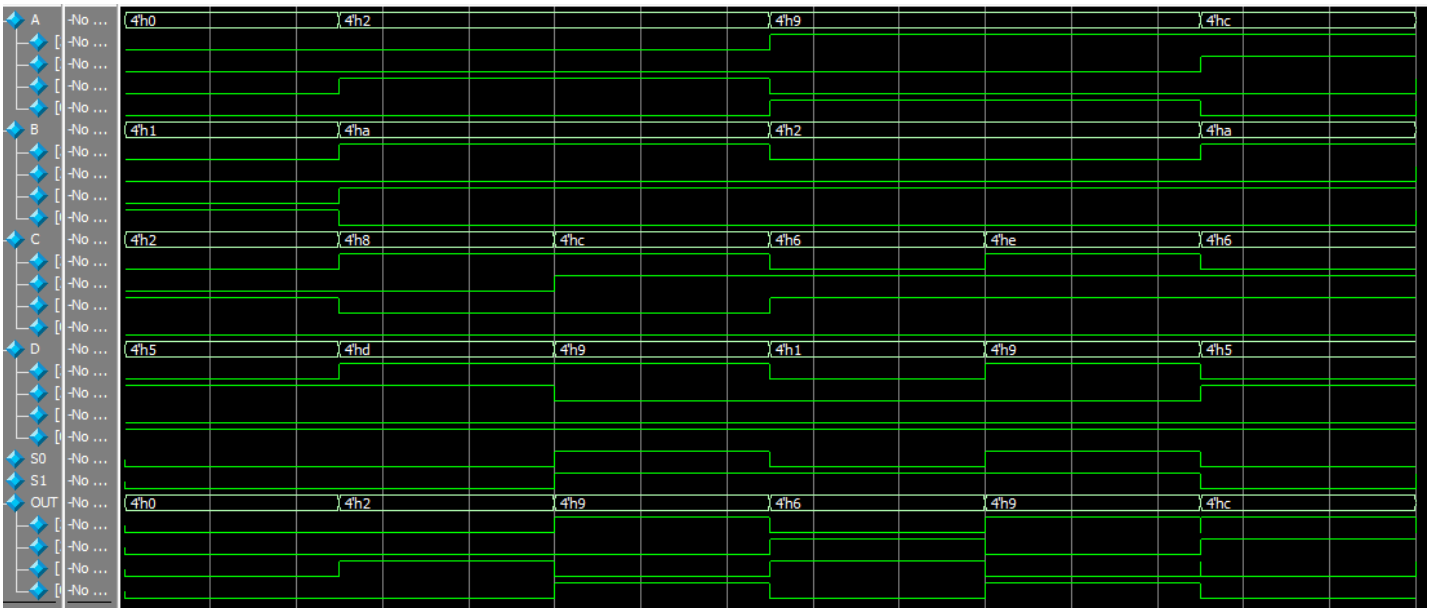


Ilustración 4-12. Ejecución de ejemplos usando el multiplexor 4 a 1

3.5 Archivo de test

En este apartado se ha cogido un fichero de test de alguno de los diseños anteriores para compararlos. Solo se va a comparar un archivo de test porque todos siguen la misma estructura.

Es necesario dejar claro que cada diseño necesita un archivo de test para probar el correcto funcionamiento del código. En este archivo es obligatorio introducir las mismas señales con los mismos tipos y rango que el archivo de diseño. También es obligatorio llamar a este archivo de una manera u otra.

En este caso, se van a comparar los archivos de test del diseño del sumador.

Código en VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
ENTITY prueba_sumador IS
END prueba_sumador;
ARCHITECTURE test3 OF prueba_sumador IS
  COMPONENT sum
  PORT( a, b : in std_logic_vector(3 downto 0);
        acarreo : out std_logic;
        salida : out std_logic_vector(3 downto 0));
```

```
END COMPONENT;
SIGNAL a, b, salida :std_logic_vector(3 downto 0);
SIGNAL acarreo : std_logic;
BEGIN
   i: sum PORT MAP(a, b, acarreo, salida);
   a<="0000" AFTER 0 ns, " 0010" AFTER 5 ns, "1001" AFTER
10 ns, "1100" AFTER 15 ns, "1111" AFTER 20 ns;
   b<="0001" AFTER 0 ns," 1010" AFTER 5 ns, "0010" AFTER
10 ns, "1010" AFTER 15 ns, "1111" AFTER 20 ns;
END test3;
```

Código en Verilog:

```
module sumador_tb;
wire [3:0]t_y;
wire t_acarreo;
reg [3:0]t_a, t_b;
sumador my_gate( .a(t_a), .b(t_b), .y(t_y),
.acarreo(t_acarreo) );
initial
begin
t_a = 4'b0000;
t_b = 4'b0001;
#5
t_a = 4'b0010;
t_b = 4'b1010;
#5
t_a = 4'b1001;
t_b = 4'b0010;
#5
t_a = 4'b1100;
t_b = 4'b1010;
#5
t_a = 4'b1111;
t_b = 4'b1111;
end
endmodule
```

Las diferencias entre estos códigos son:

- Como en todos los diseños de VHDL es necesario llamar a la librería ‘IEEE’ y crear la entidad. En este caso, la entidad esta vacía, al ser un archivo de test.

- En VHDL también es necesario instanciar el componente que se va a utilizar (el diseño del sumador). Mientras que en Verilog no es necesario.

- En ambos lenguajes es necesario crear señales internas que harán referencia a las señales de entrada, salida o entrada-salida del diseño a probar.

- También es necesario referenciar el modulo que se va a usar y sus señales. En VHDL se le asigna un nombre a la referencia, seguido del modulo a referenciar y la palabra reservada 'PORT MAP' y por último la lista de las variables. Por otro lado, en Verilog se usa el nombre del bloque a instanciar después se le otorga un nombre a esta instancia y se sigue de la asignación de las diferentes señales del modulo a probar en las variables internas definidas en el archivo.

- Otra de las diferencias es que en VHDL se la asignación de tiempos se hace en el 'BEGIN' de la arquitectura mientras que el Verilog se realiza en un proceso 'INITIAL'.

- La asignación de valores y la forma de indicar el tiempo en el que esa variable se debe asignar es muy diferente en estos lenguajes. En VHDL se hace una sola asignación, indicando el valor seguido de la palabra 'AFTER', que indica después de que instante debe aparecer el valor asignado, seguido por el instante con unidad de tiempo. Mientras que en Verilog se debe asignar el valor cada vez que cambia, además, no se indica cada instante, sino que usando el símbolo '#' seguido de un numero se está indicando las unidades de tiempo que debe esperar para asignar ese valor desde la última asignación.

- La última diferencia es el número de líneas empleadas para cada código: en VHDL se emplean 18 líneas, mientras que en Verilog se emplean 24 líneas. Es necesario tener en cuenta que cuantas pruebas se quieran realizar en Verilog, van a ser necesarias muchas más líneas.

Conclusiones

Después de casi una década debatiendo sobre cuál de los dos lenguajes es el más apropiado para diseñar no se ha llegado a ninguna solución.

Por un lado, Verilog nace ante la necesidad de mejorar un flujo de diseño y por tanto es un lenguaje que cubre todas las necesidades de diseño y simulación. El resultado es un lenguaje simple, flexible y poco exigente, siendo fácil de aprender aunque tiene ciertas limitaciones.

Por otro lado, VHDL nació como un lenguaje que resuelve las necesidades, pasadas, actuales de diseño y otras futuras, es decir con mayor amplitud de miras. El resultado es un lenguaje más rígido y con una sistematización en el código que la hace mas legible, aunque las reglas de diseño obligan a proceder de forma poco ágil en algunas ocasiones.

Por lo tanto, dependiendo del tipo de uso que se necesite se recomienda más un lenguaje que otro.

Para los nuevos usuarios o los usuarios acostumbrados a manejar lenguajes similares a C se recomienda el aprendizaje de Verilog, ya que su aprendizaje es más rápido. Mientras que si lo que se desea es realizar diseños muy complejos o se tienen conocimientos de ADA o Pascal, se recomienda el aprendizaje de VHDL, ya que es un lenguaje más rígido, y con más proyección.

Después de estudiar y manejar cada lenguaje de forma independiente para posteriormente compararlos las conclusiones que he obtenido han sido:

Ningún lenguaje es mejor que el otro, como se acaba de mencionar, dependiendo del uso que se le quiera dar se recomienda uno u otro, pero lo ideal sería aprender los dos.

Aunque Verilog tiene limitaciones, a la hora de realizar diseños sencillos es mucho más rápido, y al ser tan permisivo no es necesario estar pendiente, por ejemplo, del tamaño de un valor.

Pero a la hora de implementar diseños más complicados que requieren un “plan de abordaje” sobre el problema a resolver, el lenguaje ideal es VHDL, porque al ser más rígido consigue evitar problemas futuros en el diseño.

Como conclusión final, no hay un lenguaje mejor que el otro sino uno más recomendable que otro dependiendo de para que se desee usar. Ambos lenguajes están muy bien terminados y son muy completos.

Bibliografía

[1] Aguirre Echanove, M., Noel Tombs, J., Muñoz Chavero, F., Guzmán Miranda, H. and Nápoles Luengo, J. (2005). *DISEÑO DE SISTEMAS DIGITALES MEDIANTE LENGUAJES DE DESCRIPCIÓN DE HARDWARE.*. [ebook] Sevilla, España. Available at: http://www.dinel.us.es/util/bajar.php?file=eJwrC8wxq5hTrKNxVttTt4__c8HHi9oenty-HtrsDH6e7BlcMMqeCwk,&x=18&y=8&r=0 [Accessed Mar. 2017].

[2] Aparicio Olmedo, A. and Ponluisa Marcalla, F. (2014). *Estudio[comparativo de los lenguajes HDL y su aplicación en la implementación del laboratorio de sistemas digitales avanzados*. [ebook] Riobamba, Ecuador. Available at: http://dspace.espoch.edu.ec/handle/123456789/3342 [Accessed Feb. 2017].

[3] Asic-world.com. (2014). *Welcome To Verilog Page*. [online] Available at: http://www.asic-world.com/verilog/index.html[Accessed Mar. 2017].

[4] Chávez, J. (1999). *Manual de verilog*. 4th ed. [ebook] Sevilla, España. Available at: http://www.gte.us.es/~chavez/doc/verilog.pdf [Accessed Apr. 2017].

[5] Douglas, J. (1997). *HDL Chip Design*. 3rd ed. Done Publications.

[6] Entrena Arrontes, L., Lopez, C., Garcia, M., San Millan, E., Portela, M. and Lindoso, A. (2015). *El lenguaje VHDL, conceptos básicos*. [ebook] Madrid, España. Available at: http://ocw.uc3m.es/tecnologia-electronica/circuitos-integrados-y-microelectronica/teoria_vhdl/vhdl-2-conceptos-basicos-1/view [Accessed Apr. 2017].

[7] Es.wikibooks.org. (2015). *Programación en Verilog - Wikilibros*. [online] Available at: https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_Verilog [Accessed Mar. 2017].

[8] Es.wikibooks.org. (2017). *Programación en VHDL - Wikilibros*. [online] Available at: https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_VHDL [Accessed Mar. 2017].

[9] Grediaga, A. (2014). VHDL. tipos de datos. [Blog] *Researchgate*. Available at: https://www.researchgate.net/publication/39436653_VHDL_Tipos_de_datos [Accessed Apr. 2017].

[10] Lenguajes de descripción de hardware: VHDL. (2008). [ebook] Mendoza, Argentina. Available at: http://www1.frm.utn.edu.ar/tecnicad1/_private/Apuntes/VHDL.pdf [Accessed Apr. 2017].

[11] Lenguajes de descripcion de los sistemas digitales (parte 1). (n.d.). [ebook] Vigo, España: Marcombo, S.A. Available at: http://www.marcombo.com/Descargas/9788426714305_SISTEMAS%20ELECTRONICOS%20DIGITALES/Temas/Tema%2016%20Lenguajes%20de%20descripcion%20%28Parte%201%29.pdf [Accessed Mar. 2017].

[12] López, JM. and Lopez, JC. (1994). *El VHDL en la enseñanza de la electrónica*. Madrid, España, pp.213-222.

[13] Silva Bijit, L. (2010). *Uso de verilog*. [ebook] Valparaíso, Chile, p.apéndice 5. Available at: http://www2.elo.utfsm.cl/~lsb/elo211/clases/ap5.pdf [Accessed Apr. 2017].

[14] Sutherland, S. (2000). *Verilog HDL quick reference guide*. Portland, OR: Sutherland HDL, Inc.

[15] VHDL presentación 2.0. (2017). [ebook] Mendoza, Argentina. Available at: http://www1.frm.utn.edu.ar/tecnicad1/_private/Apuntes/VHDLPresentación2.0[1].pdf [Accessed Mar. 2017].

[16] Martínez Iniesta, M. (2017). *introducción al CAD-EDA*. [ebook] Albacete, España. Available at: http://edii.uclm.es/~miniesta/Introduccion%20EDA.pdf [Accessed Mar. 2017].

ANEXO. 1
Código utilizado en VHDL

1. Diseño puertas lógicas

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

ENTITY puertas_logicas IS
  PORT (a : IN std_logic_vector(3 DOWNTO 0);
        b : IN std_logic_vector(3 DOWNTO 0);
   sal_and : OUT std_logic_vector(3 DOWNTO 0);
   sal_or : OUT std_logic_vector(3 DOWNTO 0);
   sal_nota : OUT std_logic_vector(3 DOWNTO 0);
   sal_xor : OUT std_logic_vector(3 DOWNTO 0));
END puertas_logicas;

ARCHITECTURE puertas OF puertas_logicas IS
  BEGIN
    PROCESS (a, b) IS
    BEGIN
    sal_and    <=    std_logic_vector(UNSIGNED(a)    and
UNSIGNED(b));
    sal_or     <=     std_logic_vector(UNSIGNED(a)     or
UNSIGNED(b));
    sal_nota <= std_logic_vector(UNSIGNED(not a) );
    sal_xor    <=    std_logic_vector(UNSIGNED(a)    xor
UNSIGNED(b));
    END PROCESS;
END puertas;
```

1.1 Test bench puertas lógicas

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

ENTITY prueba_puertas IS
END prueba_puertas;
ARCHITECTURE test_puertas OF prueba_puertas IS
COMPONENT puertas_logicas
PORT( a, b : in std_logic_vector(3 downto 0);
  sal_and, sal_or, sal_nota, sal_xor : out
std_logic_vector(3 downto 0));
END COMPONENT;
SIGNAL a, b, sal_and, sal_or, sal_nota, sal_xor
:std_logic_vector(3 downto 0);
```

```
BEGIN
   i: puertas_logicas PORT MAP(a, b, sal_and, sal_or,
sal_nota, sal_xor);

   a<="0000" AFTER 0 ns, "0001" AFTER 5 ns, "0010" AFTER
15 ns, "0011" AFTER 25 ns,"0100" AFTER 35 ns, "0101"
AFTER 45 ns,"0110" AFTER 55 ns, "1110" AFTER 65 ns;
   b<="0100" AFTER 0 ns,"0101" AFTER 5 ns, "0110" AFTER
10 ns, "0111" AFTER 20 ns,"1000" AFTER 30 ns, "1001"
AFTER 40 ns,"1010" AFTER 50 ns, "1011" AFTER 60 ns;
END test_puertas;
```

2. Diseño sumador

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

ENTITY sum IS
  PORT (a : IN std_logic_vector(3 DOWNTO 0);
        b : IN std_logic_vector(3 DOWNTO 0);
      acarreo : OUT std_logic;
        salida : OUT std_logic_vector(3 DOWNTO 0));
  END sum;
ARCHITECTURE synth OF sum IS
  BEGIN
    PROCESS (a, b) IS
    VARIABLE suma: std_logic_vector(4 DOWNTO 0);
    BEGIN
      suma  :=  std_logic_vector(UNSIGNED('0'  &  a)  +
UNSIGNED('0' & b));
  salida <=suma(3 DOWNTO 0);
  acarreo <= suma(4);
    END PROCESS;
END synth;
```

2.1 Test bench sumador

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
ENTITY prueba_sumador IS
END prueba_sumador;
ARCHITECTURE test3 OF prueba_sumador IS
  COMPONENT sum
  PORT( a, b : in std_logic_vector(3 downto 0);
```

```
       acarreo : out std_logic;
       salida : out std_logic_vector(3 downto 0));
END COMPONENT;
SIGNAL a, b, salida :std_logic_vector(3 downto 0);
SIGNAL acarreo : std_logic;
BEGIN
   i: sum PORT MAP(a, b, acarreo, salida);
   a<="0000" AFTER 0 ns, "0010" AFTER 5 ns, "1001" AFTER
10 ns, "1100" AFTER 15 ns,"1111" AFTER 20 ns;
   b<="0001" AFTER 0 ns,"1010" AFTER 5 ns, "0010" AFTER
10 ns, "1010" AFTER 15 ns,"1111" AFTER 20 ns;
END test3;
```

3. Restador en complemento A2

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
ENTITY res IS
  PORT (a : IN std_logic_vector(3 DOWNTO 0);
        b : IN std_logic_vector(3 DOWNTO 0);
       signo_salida : OUT std_logic;
        salida : OUT std_logic_vector(3 DOWNTO 0));
  END res;
ARCHITECTURE res_a2 OF res IS
  BEGIN
    PROCESS (a, b) IS
    VARIABLE suma: std_logic_vector(4 DOWNTO 0);
    VARIABLE dato: std_logic_vector(4 DOWNTO 0);
    BEGIN
   if (a >= b) then
        dato := std_logic_vector(UNSIGNED('0'&(not b))+
"00001");
        suma := std_logic_vector(UNSIGNED('0' & a) +
UNSIGNED(dato));
        salida <=suma(3 DOWNTO 0);
        signo_salida <= '0';
   else
        dato := std_logic_vector(UNSIGNED('0'&(not a))+
"00001");
        suma := std_logic_vector(UNSIGNED(dato) +
UNSIGNED('0' & b));
        salida <=suma(3 DOWNTO 0);
        signo_salida <= suma(4);
   END IF;
    END PROCESS;
```

```
END res_a2;
```

3.1 Test bench del restador complemento A2

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

ENTITY prueba_rest_a2 IS
END prueba_rest_a2;
ARCHITECTURE test4 OF prueba_rest_a2 IS
COMPONENT res
PORT( a, b : in std_logic_vector(3 downto 0);
  signo_salida : out std_logic;
  salida : out std_logic_vector(3 downto 0));
END COMPONENT;
SIGNAL a, b,salida :std_logic_vector(3 downto 0);
SIGNAL signo_salida : std_logic;
BEGIN
   i: res PORT MAP(a, b,signo_salida, salida);

   a<="1000" AFTER 0 ns, "1010" AFTER 5 ns, "1001" AFTER
10 ns, "1100" AFTER 15 ns,"1111" AFTER 20 ns;
   b<="1010" AFTER 0 ns,"1000" AFTER 5 ns, "0010" AFTER
10 ns, "1010" AFTER 15 ns,"1111" AFTER 20 ns;
END test4;
```

4. Multiplexor 2 a 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity mux_2_1 is
  port(x,y : in STD_LOGIC_VECTOR(3 DOWNTO 0);
  s: in STD_LOGIC;
  z: out STD_LOGIC_VECTOR(3 DOWNTO 0));
  end mux_2_1 ;
architecture mux_2a1 of mux_2_1 is
  begin
  process (x,y,s) is
    begin
    if (s ='0') then
      z <= x;
    else
      z <= y;
    end if;
  end process;
```

```
end mux_2a1;
```

5. Multiplexor 4 a 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity mux_4_1 is
  port(A,B,C,D : in STD_LOGIC_VECTOR(3 DOWNTO 0);
       S0,S1: in STD_LOGIC;
       Z: out STD_LOGIC_VECTOR(3 DOWNTO 0));
end mux_4_1 ;
architecture mux_4a1 of mux_4_1 is
 component mux_2_1
  port(x,y : in STD_LOGIC_VECTOR(3 DOWNTO 0);
     s: in STD_LOGIC;
     z: out STD_LOGIC_VECTOR(3 DOWNTO 0));
 end component;
 signal m1, m2: std_logic_vector(3 DOWNTO 0);
 begin
   c1: mux_2_1 port map(A,B,S0,m1);
   c2: mux_2_1 port map(C,D,S0,m2);
   c3: mux_2_1 port map(m1,m2,S1,Z);
end mux_4a1;
```

5.1 Test bench multiplexor 4 1

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

ENTITY prueba_mux IS
END prueba_mux;
ARCHITECTURE test5 OF prueba_mux IS
COMPONENT mux_4_1
PORT(A,B,C,D : in STD_LOGIC_VECTOR(3 DOWNTO 0);
   S0,S1: in STD_LOGIC;
   Z: out STD_LOGIC_VECTOR(3 DOWNTO 0));
END COMPONENT;
--SIGNAL a, b,salida :std_logic_vector(3 downto 0);
SIGNAL A, B,C, D, z : std_logic_VECTOR(3 DOWNTO 0);
SIGNAL S0, S1: STD_LOGIC;
BEGIN
   i: MUX_4_1 PORT MAP(A, B,C, D, S0, S1, z);
```

```
   A<="0000" AFTER 0 ns, "0010"AFTER 5 ns, "0010" AFTER
10 ns, "1001" AFTER 15 ns,"1001" AFTER 20 ns, "1100"
AFTER 25 ns, "1111" AFTER 30 ns;
   B<="0001" AFTER 0 ns, "1010"AFTER 5 ns, "1010" AFTER
10 ns, "0010" AFTER 15 ns,"0010" AFTER 20 ns, "1010"
AFTER 25 ns, "1111" AFTER 30 ns;
   C<="0010" AFTER 0 ns, "1000"AFTER 5 ns, "1100" AFTER
10 ns, "0110" AFTER 15 ns,"1110" AFTER 20 ns, "0110"
AFTER 25 ns, "0110" AFTER 30 ns;
   D<="0101" AFTER 0 ns, "1101"AFTER 5 ns, "1001" AFTER
10 ns, "0001" AFTER 15 ns,"1001" AFTER 20 ns, "0101"
AFTER 25 ns, "0101" AFTER 30 ns;
   S0<='0' AFTER 0 ns, '0'AFTER 5 ns, '1' AFTER 10 ns,
'0' AFTER 15 ns, '1' AFTER 20 ns, '0' AFTER 25 ns,'0'
AFTER 30 ns;
   S1<='0' AFTER 0 ns, '0'AFTER 5 ns, '1' AFTER 10 ns,
'1' AFTER 15 ns,'1' AFTER 20 ns, '0' AFTER 25 ns,'1'
AFTER 30 ns;
END test5;
```

ANEXO. 2

Código utilizado en Verilog

1. Diseño puertas lógicas

```
module puertas_logicas (a, b, sal_and, sal_or,
sal_nota, sal_xor);
input [3:0]a, b;
output [3:0]sal_and, sal_or, sal_nota, sal_xor;
assign sal_and = a & b;
assign sal_or = a | b;
assign sal_nota = ~a;
assign sal_xor = a ^ b;
endmodule
```

1.1 Test bench puertas lógicas

```
module prueba_puertas_tb;
wire [3:0]t_sal_and, t_sal_or, t_sal_nota, t_sal_xor;
reg [3:0]t_a, t_b;
puertas_logicas my_gate( .a(t_a), .b(t_b),
.sal_and(t_sal_and), .sal_or(t_sal_or),
.sal_nota(t_sal_nota), .sal_xor(t_sal_xor));
initial
begin
t_a = 4'b0000;
t_b = 4'b0100;
#5
t_a = 4'b0001;
t_b = 4'b0101;
#5
t_b = 4'b0110;
#5
t_a = 4'b0010;
#5
t_b = 4'b0111;
#5
t_a = 4'b0011;
#5
t_b = 4'b1000;
#5
t_a = 4'b0100;
#5
t_b = 4'b1001;
#5
```

```
t_a = 4'b0101;
#5
t_b = 4'b1010;
#5
t_a = 4'b0110;
#5
t_b = 4'b1011;
#5
t_a = 4'b0000;
t_b = 4'b1000;
end
endmodule
```

2. Diseño sumador

```
module sumador (a, b, y, acarreo);
input [3:0]a, b;
input acarreo;
output [4:0]y;
wire [4:0] suma;
assign y = a + b;
assign y = suma[3:0];
assign acarreo = suma[4];
endmodule;
```

2.1 Test bench sumador

```
module sumador_tb;
wire [3:0]t_y;
wire t_acarreo;
reg [3:0]t_a, t_b;
sumador my_gate( .a(t_a), .b(t_b), .y(t_y),
.acarreo(t_acarreo) );
initial
begin
t_a = 4'b0000;
t_b = 4'b0001;
#5
t_a = 4'b0010;
t_b = 4'b1010;
#5
t_a = 4'b1001;
t_b = 4'b0010;
#5
t_a = 4'b1100;
t_b = 4'b1010;
#5
```

```
t_a = 4'b1111;
t_b = 4'b1111;
end
endmodule
```

3. Diseño restador en complemento A2

```
module restador_a2 (a, b, y, acarreo);
input [3:0]a, b;
output reg [3:0]y;
output reg acarreo;
reg [4:0] suma;

always@(a or b)
  begin
  if (a >= b)
  begin
    assign suma = ((~b)+"0001") + a;
    assign y = suma[3:0];
    assign acarreo = "0";
  end
  else
  begin
    assign suma = ((~a)+"0001") + b;
    assign y = suma[3:0];
    assign acarreo = suma[4];
  end
  end
endmodule
```

3.1 Test bench de restador complemento A2

```
module restador_tb;
reg [3:0]t_a, t_b;
wire [3:0]t_y;
wire t_acarreo;
restador_a2 my_gate( .a(t_a), .b(t_b), .y(t_y),
.acarreo(t_acarreo) );
initial
begin
t_a = 4'b1000;
t_b = 4'b1010;
#5
t_a = 4'b1010;
t_b = 4'b1000;
#5
```

```
t_a = 4'b1001;
t_b = 4'b0010;
#5
t_a = 4'b1100;
t_b = 4'b1010;
#5
t_a = 4'b1111;
t_b = 4'b1111;
end
endmodule
```

4. Diseño multiplexor 2 a 1

```
module mux_2_1 (sel, a, b, y);
input [3:0]a, b;
input sel;
output reg [3:0]y;
always @(a or b or sel)
  if (sel == 0)
    assign y = a;
  else
    assign y = b;
endmodule
```

5. Diseño multiplexor 4 a 1

```
module mux_4_1(sel0,sel1,a,b,c,d,out);
     input [3:0] a,b,c,d;
     input sel0, sel1;
     output [3:0]out;

     wire [3:0]mux1,mux2;
     mux_2_1 mux_1(sel0,a,b,mux1);
     mux_2_1 mux_2(sel0,c,d,mux2);
     mux_2_1 mux_3(sel1,mux1,mux2,out);
endmodule
```

5.1 Test bench multiplexor 4 a 1

```
module mux41_tb;
reg [3:0]t_a, t_b, t_c, t_d;
reg t_sel0, t_sel1;
wire [3:0]t_out;
mux_4_1 my_gate(.sel0(t_sel0), .sel1(t_sel1),.a(t_a),
.b(t_b), .c(t_c), .d(t_d),  .out(t_out) );
initial
```

```
begin
t_sel0 = 1'b0;
t_sel1 = 1'b0;
t_a = 4'b0000;
t_b = 4'b0001;
t_c = 4'b0010;
t_d = 4'b0101;
#5
t_sel0 = 1'b0;
t_sel1 = 1'b0;
t_a = 4'b0010;
t_b = 4'b1010;
t_c = 4'b1000;
t_d = 4'b1101;
#5
t_sel0 = 1'b1;
t_sel1 = 1'b1;
t_a = 4'b0010;
t_b = 4'b1010;
t_c = 4'b1100;
t_d = 4'b1001;
#5
t_sel0 = 1'b0;
t_sel1 = 1'b1;
t_a = 4'b1001;
t_b = 4'b0010;
t_c = 4'b0110;
t_d = 4'b0001;
#5
t_sel0 = 1'b1;
t_sel1 = 1'b1;
t_a = 4'b1001;
t_b = 4'b0010;
t_c = 4'b1110;
t_d = 4'b1001;
#5
t_sel0 = 1'b0;
t_sel1 = 1'b0;
t_a = 4'b1100;
t_b = 4'b1010;
t_c = 4'b0110;
t_d = 4'b0101;
#5
t_sel0 = 1'b0;
t_sel1 = 1'b1;
t_a = 4'b1111;
```

```
t_b = 4'b1111;
t_c = 4'b0110;
t_d = 4'b0101;
end
endmodule
```